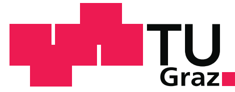


DI (FH) Andreas Rechberger

Model Assisted Automated HW/SW Partitioning

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner

Institute for Technical Informatics

Graz, February 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date

Signature

Abstract

The future of computing is heterogeneous. With technological trends to include processing capability into a large variety of devices, today's challenges in distributed computing are no longer solved on homogenous systems. Not only federations of devices are used to tackle various difficulties, also single devices nowadays provide a collection of diverse compute units.

This thesis presents a generalized abstraction of algorithms, suited to analyse its potential for distribution. These algorithms might emerge from any domain of software tasks, such as number crunching, machine learning, classical signal processing, or control theory. Such an abstract view does combine the elements of static and dynamic analysis of the algorithm, and deals with the concrete pattern of its computational profile. As counterpart to the model of the computation, a representation of a dedicated compute arrangement is proposed, in order to deal with the diverse nature of a complex of compute-nodes. This provides an approach to model heterogeneous systems. These systems can be a collection of embedded devices, like micro-controllers, but also a combination of cloud servers with edge computing devices. With these two models in place an estimation procedure is given to predict the performance profile of the hardware setup upon serving the defined algorithm. The performance profile does not only include the overall execution time, but also a detailed view on the data flow and node allocation. Such an allocation profile for example allows drawing conclusions on how to improve the hardware arrangement, or the algorithm, to more efficiently achieve its goal, based on concrete facts.

In this work a set of algorithms is analysed by the procedure given, and its distribution pattern is dissected in detail. Exemplarily the impact of various modifications and extensions, like adding additional compute nodes, to the hardware setup is performed. For these adaptions the impact on the allocation profile is provided, such that a concrete judgment on their impact can be made.

Based on the presented results, future research directions in the domain of heterogeneous and parallel distribution of algorithms are illustrated.

Zusammenfassung

Der technologische Trend zielt zurzeit eindeutig in Richtung verteilter und vernetzter Systeme, weshalb der Aufbau von Computersystemen immer komplexer wird. Dies erfordert eine Anpassung der Softwarealgorithmen, da in diesem Umfeld nicht mehr von einer homogenen oder monolithischen Rechnerarchitektur ausgegangen werden kann. Die an eine Software gestellten Aufgaben umfassen neben maschinellen Lernen, neuronalen Netzwerken und ähnlich rechenintensive Operationen auch klassische Signalverarbeitung oder regelungstechnische Aspekte. Um diese umfassenden und fachlich breit gestreuten Anforderungen mittels heterogener Systeme lösen zu können sind neue innovative Ansätze zur Verteilung und Aufteilung notwendig.

In dieser Dissertation wird eine verallgemeinerte Darstellung von Algorithmen vorgestellt, welche die Analyse der möglichen Verteilungen auf verschiedene Rechenknoten erleichtert. Diese basiert auf statischen wie auch dynamischen Aspekten. Zusätzlich dient eine flexible Repräsentation eines Verbundes von Rechenwerken als Gegenstück für die Verteilungsanalyse. Diese Rechenwerke können ein Verbund von eingebetteten Systemen, beispielsweise Mikrocontroller, aber auch eine Ansammlung von Edge- und Cloud-Servern sein.

Mit diesen zwei Modellen—jenes des Algorithmus und jenes der Hardware—ist es möglich, die Auslastung der Rechenwerke abzuschätzen. Dieses Auslastungsprofil beinhaltet, neben der globalen Sichtweise, auch eine genaue Abbildung der zeitabhängigen lokalen Ressourcenauslastung und ermöglicht somit eine fundierte, auf konkreten Daten basierende, Aussage über die Eignung eines Hardware Setups für einen definierten Algorithmus.

Die Ergebnisse dieser Arbeit zeigen eine detaillierte Aufschlüsselung einer Reihe von Algorithmen. Die durchgeführten Simulationen beziehen sich auf bekannte, und somit leicht abschätzbare, Hardwarestrukturen, welche im zweiten Schritt modifiziert werden. Die Auswirkungen dieser Modifikationen auf die Performance werden im Anschluss diskutiert. Abschließend werden, basierend auf den Ergebnissen dieser Arbeit, einige Anknüpfungspunkte für zukünftige Arbeiten in den Forschungsgebieten der heterogenen Rechenwerke und Parallelisierung aufgezeigt.

Contents

1 Introduction **1**
- 1.1 Background and Motivation 1
- 1.2 Research Focus 2
  - 1.2.1 Description of Algorithms 3
  - 1.2.2 Data flow analysis of C and C++ programs 3
- 1.3 Related Work 5
  - 1.3.1 Modelling of Algorithms 5
  - 1.3.2 Customized compute-nodes 6

2 Methodology **7**
- 2.1 Basic Concepts 7
- 2.2 Extracting Data Dependencies 9
  - 2.2.1 Integer Ranges 9
  - 2.2.2 Indirect Memory Access 10
  - 2.2.3 Call Graphs 12
- 2.3 Preparing the Code 13
  - 2.3.1 Program optimization 16
  - 2.3.2 Program optimization, vectorization 18
  - 2.3.3 Front End and Compiler Optimizations 21
  - 2.3.4 Φ-nodes 21
  - 2.3.5 Function calls 24
  - 2.3.6 Initialization and detecting input and output data 25
  - 2.3.7 Cross function Φ-nodes 26
  - 2.3.8 GEP-nodes 26
  - 2.3.9 Raising Abstraction Level 30
- 2.4 Representing Hardware 32
  - 2.4.1 Hardware Model 33
  - 2.4.2 Execution time estimation basics 35

3 Results **41**
- 3.1 Control and Data Flow Graphs 41
  - 3.1.1 Simplified Microbenchmark 41
  - 3.1.2 Arithmetic benchmark 46
- 3.2 Manual Performance Estimation 49
- 3.3 Automated Distribution Analysis 52
  - 3.3.1 Anomalies 62
  - 3.3.2 Randomization 64
  - 3.3.3 A Heterogeneous Example 65

4 Conclusion **74**
4.1 Future Work 77

List of Figures

2.1 Control Flow Graph for Listing 2.14 18
2.2 Example, sum elements of array 34
2.3 ARM Cortex-M (simplified) [RB18a] 35
2.4 Multi chip hardware model [RB18a] 36

3.1 Normalized estimates vector sum ($\sum_i^N a_i$)[RB18a] 43
3.2 Normalized estimates FIR iteration ($\sum_i^N a_i b_i$)[RB18a] 43
3.3 2x2 matrix vector multiply, fully unrolled 47
3.4 2x2 matrix vector multiply, no unrolling 48
3.5 Multi core architecture with 4 ALU cores 49
3.6 Multi core architecture, transfer node allocation 50
3.7 Performance 16x16 Matrix Vector Multiplication 52
3.8 16x16 matrix vector multiplication, computation cycle 104...107 54
3.9 16x16 matrix vector multiplication, computation cycle 124...127 55
3.10 16x16 matrix vector multiplication, computation cycle 1370...1373 56
3.11 16x16 matrix vector multiplication, instruction distribution . . . 58
3.12 Different HW models, instruction distribution 59
3.13 ALU Allocation (absolute) 59
3.14 ALU Allocation (adapted numbering) 60
3.15 Load average (100 cycles window) 61
3.16 SpeedUp (Monte Carlo Method) 65
3.17 $\hat{G}_4$ Hardware Model 67
3.18 SpeedUp of Heterogenous Hardware (Monte Carlo Method) . . . 67
3.19 Load average (100 cycles window) 69
3.20 $\hat{G}_4$ model, sequence time 28 to 31 70
3.21 $\hat{G}_4$ model, sequence time 32 to 35 71
3.22 $\hat{G}_4$ model, sequence time 36 to 40 72

4.1 Control and data flow graph (CDFG) of Listing 4.1 with $a = 10.0$ 75

List of Tables

3.1 Early Evaluation Models 44
3.2 Total Allocation 62
3.3 Average Speed Up 66

Listings

1.1 Simple parallelizable example 3
1.2 Non-trivial functions 3
2.1 Example with loop, initial source 7
2.2 Example with loop, optimised source 8
2.3 `foldl` C source 8
2.4 Ada integer range 9
2.5 C++ integer range 10
2.6 C++ integer range - LLVM assembly language code of Listing 2.5 10
2.7 Indirect Memory Access 11
2.8 Memory Access - Individual Variables 12
2.9 Memory Access - Individual Variables 12
2.10 Function Calls 13
2.11 C++ execution tracking LLVM assembly language [LLV18] code 14
2.12 First Level Tracking Function [RB18b] 15
2.13 Second Level Tracking Function 15
2.14 `foldl` original code 16
2.15 `foldl` modified code 17
2.16 Numeric example (with temporary computations) 18
2.17 Straight forward intermediate representation of Listing 2.16 . . . 19
2.18 Optimized intermediate representation of Listing 2.16 19
2.19 Adding two vectors, C++ code 20
2.20 Adding two vectors, C code 20
2.21 Adding two vectors, expected intermediate representation code . 20
2.22 Adding two vectors, LLVM assembly language code of Listing 2.20 22
2.23 Adding two vectors, C code with expressing alias free input . . . 23
2.24 Adding two vectors, LLVM assembly language code of Listing 2.23 23
2.25 Adding two vectors, C++ with compiler extension 24
2.26 Adding two vectors, x86 assembly code Listing 2.25 25
2.27 Φ-node example C++ source 27
2.28 Φ-node example LLVM assembly language source 28
2.29 Tracking `getelemntptr` instructions 29
2.30 `foldl` optimized representation 30
2.31 `pow` implementation 32
2.32 vector norm 32
2.33 vector norm, intermediate representation 39
3.1 array multiply accumulate (simplified FIR core) 41
3.2 `foldl` core, ARM Cortex-M4 assembly code 45
3.3 `foldl` core, LLVM assembly language code 45
3.4 Matrix Vector multiply, C++ code 46

3.5 Selection Logic for next operation 63
4.1 Branches in code 74

Glossary

Ahead-Of-Time compiler A compiler which fully compiles and links the program prior to executing it. It is usually called only once to generate the executable. VII

ARM Cortex-M4 A 32-bit micro-controller core developed by ARM Limited, primarily intended for the embedded market. Available with (Cortex-M4F) or without (Cortex-M4) floating point engine. V, 44, 45

ARM Limited A vendor of CPU cores. Most prominently the Cortex series.. VII

C6x Family of digital signal processor (DSP), developed by Texas Instruments Incorporated. 6, 35

global navigation satellite system A navigation system based on satellite supported positioning. Most prominent implementations are NAVSTAR Global Positioning System (GPS), Galileo, Globalnaya navigatsionnaya sputnikovaya sistema (GLONASS) and BeiDou Navigation Satellite System (BDS). 31, 32

intermediate representation A representation of the code in the way used by the compiler. This is the format used internally to hand over a module from one stage or optimization step to the next one. V, 3, 15–17, 19–21, 25, 30, 31, 33, 34, 39, 42, 44, 53

interpreter A program that executes the instructions of a source program step by step. It does not translate the program into native machine code prior to doing so, neither does convey interpretation data for future use. If it keeps a transformed version of the source instruction for performance reasons, it is called Just-In-Time compiler. VII, 13, 14

Just-In-Time compiler A compiler which performs the compilation of a module during run time. It is basically a functional merge between an Ahead-Of-Time compiler and an interpreter. It is common to invoke this compiler every time the program is launched, or if in-memory modifications of the program have been performed. VII, 15, 25, 27

LLVM assembly language The LLVM assembly language is the intermediate language used by the LLVM compiler infrastructure. It follows the single static assignment (SSA) paradigm. see. V, VIII, 9, 10, 13–16, 19, 21–24, 26, 28, 30, 42, 44–46, 53, 74, 76

LLVM project LLVM is a brand which covers all the projects and modules governed by the LLVM open source project. This includes the LLVM assembly language, the LLVM optimizer (compiler framework), clang (the C++ front end), and various other projects. 9, 13, 15, 16

non-vloatile memory A memory type which preserves data during power off. Common realizations are electrically erasable programmable read-only memory (EEPROM), Flash-EEPROM (FLASH) or ferroelectric random access memory (FRAM). 78

PlusCal An algorithm language for specification of algorithms [Lam09] whose intention is to replace pseudo code representations. Can be transpiled to TLA$^+$. 5

TensorFlow A software library for expressing algorithms [Aba+16]. While primarily used in the domain of machine learning, and in particular for the representation of artificial neuronal networks, it can be used to describe algorithms for wide variety of algorithms. 19

Texas Instruments Incorporated US based technology company focusing on semiconductors and electronics. VII

TLA$^+$ Temporal Logic of Actions. A formal specification language used for modelling algorithms above the code level [Lam03]. VIII, 5

Acronyms

3D three dimensional. 1

ALU arithmetic logic unit. 3, 18, 48–52, 57, 60, 62, 63, 65, 66, 68

ASIC application specific integrated circuit. 5

ASLR address space layout randomization. 63

AST abstract syntax tree. 3, 7

BDS BeiDou Navigation Satellite System. VII

BFS breadth first search. 64

CDFG control and data flow graph. III, 21, 24, 25, 32, 37, 42, 47, 64, 74–77

CFG control flow graph. 74

CPU central processing unit. 1–3, 6, 20, 21, 30, 33, 35, 37, 42, 48–53, 57, 60

DAG directed acyclic graph. 33, 74, 76

DFS depth first search. 64

DSP digital signal processor. VII, 2, 3, 35, 37

ECC error correcting code. 1

EEPROM electrically erasable programmable read-only memory. VIII, IX

FFT fast Fourier transform. 6

FIFO first in first out. 41, 64

FIR finite impulse response. 41

FLASH Flash-EEPROM. VIII

FPGA field programmable gate array. 2, 3, 5, 6, 18

FPU floating point processing unit. 33, 35

FRAM ferroelectric random access memory. VIII

GLONASS Globalnaya navigatsionnaya sputnikovaya sistema. VII

GPS NAVSTAR Global Positioning System. VII

GPU graphics processing unit. 2, 18

HLS high level synthesis. 5, 6, 78

IDE integrated development environment. 42

IO input/output. 1

IoT internet of things. 78

LIFO last in first out. 23, 64

MAC multiply accumulate. 31, 41, 46, 49, 51, 56, 77

MCU micro controller unit. 78

PC personal computer. 2

RAM random access memory. 46, 49, 51, 78

RISC reduced instruction set computer. 6, 33, 37, 41, 42

SSA single static assignment. VII, 9, 13, 21, 33

SSD solid state disc. 1

TTA transport triggered architecture. 6

VLIW very large instruction word. 6

Chapter 1

Introduction

1.1 Background and Motivation

The future of computing is heterogeneous. With the first computers operating mainly on a single central processing unit (CPU), in the last decades computer architectures have evolved from such single core architectures to multi-core arrangements.

Initially this trend was caused by hitting technical limits, in particularly the inability to simply further increase the clock speed of processors. This increase of clock speed from one generation of processors to the next has offered a constant improvement to software performance, without the need for adoption of the latter. This phenomenon has often been referred as “free lunch” in software development literature.

The new hardware paradigm however offered to pack multiple CPU cores into a single computer. This has proven to supply a more feasible approach to increase the computing power from a hardware point of view.

With no longer relying on the hardware improvements to introduce a performance gain in existing software the design methods and algorithms had to adapt to the newly offered hardware concurrency [SL05].

These newly developed paradigms for software development in general, and algorithm design in particular, focused on a homogeneous compute architecture. Several identical CPUs deployed in the proximity of a single device. A modern computer system however has to fulfil a large set of tasks, each of which with a possibly very different performance profile. For example the computational profile of implementing a file system consists mainly of traversing through data structures like trees and hash tables [Mat+07] as well as implementing caches to compensate for long input/output (IO) times. On the other hand rotating a three dimensional (3D) graphic is dominated by mathematical operations on matrices, respectively quaternions [DKL98].

In order to optimize such scenarios specialized computing hardware has been introduced. Such a specialized element was highly optimized for its intended use. Keeping the previous examples in mind these were implementing routines for solid state disc (SSD) drives and their wear levelling and algorithms for error correcting code (ECC) on the one hand, and video decoding or 3D algorithms in the other hand.

Naturally software and algorithm designers sought for ways of further

improving their code. This included using compute nodes intended for a particular usage pattern in realms which they have not been initially designed for. Probably the most prominent example is the utilization of graphic cards and their graphics processing units (GPUs) for machine learning applications.

Developing software for such a heterogeneous composition of computing power is a challenging task, as finding a suitable computational distribution is influenced by a lot of parameters. Such parameters do not only include the properties of a specific compute-node as such but also of the interconnect between them, and the way they influence each other. Data transfer and its dependencies can dominate the performance metrics.

A connatural problem often arises for basic signal processing applications. Developing various kinds of sensors can in an abstract way be seen as fitting a set of measurement data to a specific signal model. With higher demands on the quality of the sensors the signal model as well as the set of sampling data becomes larger and more complex. In [Rec+19] the algorithm used in a densitometer is described in more detail from its signal processing of view. Computationally the task of the sensor software is to determine a set of parameters according to Equations (1.1) and (1.2). For a reasonable performance this demands to fit a signal model with 12 parameters (for each of the curves m_0, m_1 and m_2 the four element vector $\mathbf{P}$ has to be estimated) to a set of over two million data points. A suitable arrangement of compute units (microcontrollers, field programmable gate arrays (FPGAs) or CPUs) is required to economically fulfil this task.

$$\mathbf{P} = \begin{bmatrix} A_0 & \alpha & \omega_d & \Phi_0 \end{bmatrix} \tag{1.1}$$

$$m(t, \mathbf{P}) = P_0 e^{-P_1 t} \sin(P_2 t + P_3) \tag{1.2}$$

When comparing a system with only one or even several micro-controllers, like ones based on the Arduino [Smi11] platform, with a GPU equipped personal computer (PC) the amount of data, as well as the number of operations per second differ by several orders of magnitude. However the basic principles of the problem are similar:

> *Problem statement:* How to decompose an algorithm, or software to suitably apply it on a set of compute nodes ?

1.2 Research Focus

This work shall address the process of partitioning the computation of a specified algorithm on multiple computing devices. These devices can cover general purpose processors, dedicated DSPs as well as specialized hardware circuitry like FPGAs.

The aim is to provide mechanisms and tools which allow an early evaluation of several architectural choices. The process shall be possible in a design stage where the concrete hardware-software architecture is not fully specified. As such, the—probably imprecise—results shall be accompanied by a confidence metric.

1.2.1 Description of Algorithms

In general the description (or specification) of mathematical algorithms can be done in several ways. While a textual description is easy to be understood by humans, such an approach tends to be hard to interpret by machines. As this is usually incomplete or ambiguous. A pure formal—mathematical [Lam03]—description on the other hand is clear without ambiguity and as such easy to interpret by computer programs. Nevertheless, despite successful applications [Bee08; Ver+11] such approaches have not yet established as widespread tooling within the industry.

1.2.2 Data flow analysis of C and C++ programs

The languages C++ and C are likely to be the most prominent languages used for embedded computing. Compilers are available for a wide range of CPU cores suited for small scale micro-controllers to large processing units as well as signal processors. Due to large variety of existing tools (compiler front ends) the analysis of the specified algorithm can be based on various abstraction levels. For example the abstract syntax tree (AST) or the intermediate representation language used by a particular compiler [Mer03; LA04b].

For separating the computation into multiple units it is required to decompose the data flow and data dependency and identify possible parallelism. It is also compulsory to identify the communication effort in between the separated computation units.

For elementary mathematical operations in a small scale context the dependency analysis (as well as the potential for utilizing parallel computing units) is obvious.

```
1 y = a + b;
2 x = a + c;
```

Listing 1.1: Simple parallelizable example

Given the example in Listing 1.1 it is trivial to deduce the possible computation given two independent adder units.

On this level of abstraction the deduced parallelism can be used to optimize the calculation using multifunction arithmetic logic units (ALUs) as found on most DSPs as well as data path duplication for hardware (e.g. FPGA) implementations. For splitting the computation in between multiple loosely coupled computing devices (multiple CPUs/DSPs or CPUs/FPGAs based architectures) it is required to identify larger sets of operations which can be transferred to another computing unit.

Given following example:

```
1 u = fft(a);
2 v = delaunay(x,y);
3 w = max(u);
```

Listing 1.2: Non-trivial functions

Assuming a code as depicted in Listing 1.2 it would be desired to conclude that the computation of w is suitably combined with the computation of the Fourier series, while the Delaunay triangulation can be done separately.

With having a method of specifying the algorithm, the capability to decompose and analyse it as well as sufficient data to estimate the relevant parameters of an arbitrarily chosen hardware-software architecture, the final goal is to provide a facility for automated or semi-automated optimization.

1.3 Related Work

This section presents a review on related literature and approaches in the domain of compiling data to heterogeneous systems, compiling for parallel systems and software distribution analysis. As there is a reasonable amount of research actively ongoing in all the these domains, this overview is not intended to be exhaustive. It is focused on research projects most relevant in the scope of this thesis.

1.3.1 Modelling of Algorithms

Algorithms, or more generally software, can be modelled on a large set of different abstraction levels, as well as languages. Commonly the "code level" is used to specify a concrete implementation of an algorithm.

The code in such a case is represented in a programming language like Ada [ISO12], C [ISO18], C++ with its recent advancements [ISO98; ISO11; ISO14; ISO17a], Erlang [AB20], Java [Gos+20], Python [RP18] or any other language chosen from the comprehensive arsenal of programming languages available.

Higher level modelling

When describing algorithms on a higher level of abstraction, also called "above code level", descriptive languages like TLA^+ [Lam03] or PlusCal [Lam09] can be used. They have been successfully utilized in various research approaches [Bee08; Ver+11; KK20] to verify an algorithm from a functional point of view. Different to this work such approaches focus on proving the algorithm to be correct, and keep the implementation aspect out of scope.

This work assumes the algorithm, however it is represented, to be correct by definition and deals with the implementation aspects. In particular on how to distribute the algorithm on a dedicated arrangement of computation nodes.

High level synthesis

Especially when verifying algorithms for correctness a high level description is beneficial. This simplifies methods of formal verification and model checking. All of these approaches, at least to a certain extent, lack the details to derive a concrete implementation from it. MATLAB models for example usually represent any data as vectors or matrices. Modelling a low pass filter is a simple matter of applying the entire input vector to the filter object or function. This not only improves the performance of the computation it also relieves the model from the details of the implementation. Exemplary such a detail might be the fact that filter processing happens sample by sample bound to a constant rate of input data, dictated by the properties of a data acquisition unit. Omitting this level of detail for the modelling results in a much better abstracted view, suited for high level treatment of the algorithm.

In order to generate an implementation, hence code, of such an abstract description high level synthesis (HLS) [GB08] is used. Most often this synthesis targets digital hardware respectively FPGAs or application specific integrated circuits (ASICs) [MS09; SW19]. Quite a decent amount of tooling [DD15;

Cad20; Sil20], also commercial ones, are available to perform such a high level synthesis. While not applicable for the generalized use case, a reasonable quality in comparison to handcrafted coding can be achieved [DDS18; GW20; Wad20].

In this work the synthesis step is not in the primary focus. Rather than actually implementing a given algorithm on a particular hardware setup, or synthesizing a suitable digital logic appropriate for an FPGA, we study whether a hardware setup fits for the algorithm. In particular the identification of bottlenecks is of interest for this work. In order to be of use for a general and complete software sequence, and not only a dedicated part of an algorithm (like computing the fast Fourier transform (FFT)) a certain amount accuracy is sacrificed. While such a sacrifice will restrict the results from being directly used for system synthesis, it allows a more general approach to the problem.

1.3.2 Customized compute-nodes

In order to bridge the gap between the hardware centric flows of HLS, and a software centric view it is an ongoing research approach to customize compute nodes based on the needs of the algorithm. Especially soft processor cores embedded into an FPGA offer interfaces to extend their functionality [Xil19; Int20a]. This extends the capabilities of a regular CPU core by dedicated instructions which utilize dedicated processing hardware. With a suitable choice in hardware extensions the computational performance of the CPU can be significantly improved for a given algorithm. Another approach is to modify the processor structure more fundamentally in order to fit it to the desired computation profile [Jää+17]. Such transport triggered architectures (TTAs) exhibit a data path centric view, and aim at arranging a suited set of compute nodes on a common data bus. In order to gain an advantage in computation time quite some effort is delegated to compile time. This allows the timing in between the various compute units to be specifically crafted for the hardware arrangement. In an ideal case the housekeeping part of the code (like loop overheads) can be completely eliminated. A TTA structure can be seen as going one step further on a very large instruction word (VLIW) core—like the C6x—which itself is to some extent an extension to a reduced instruction set computer (RISC). The ongoing research in context of the TTA-based Co-design Environment [Jää+17] is probably closest to this work. The research focus differs in the scope of the analysis. While [Jää+17] generally aims at the "in-chip" scope, like improving the code density via compression [MHJ20], or exploring new concepts for FPGA soft processor cores [Ter+20], this work is more in favour of a more abstract view. This reduces the level of detail on the processor internals, like a cycle accurate model of the memory access of the CPU, and in turn allows highlighting of architectures with a set of highly heterogeneous and arbitrarily coupled compute nodes. Today's compiler frameworks, and retargetable compilers are not sufficiently capable of handling a hardware setup which combines very different compute architectures as a single entity.

Chapter 2

Methodology

2.1 Basic Concepts

Prior to analysing the computational effort of an algorithm, it needs to be expressed in a suitable and conveniently machine-readable manner, that is basically expressing it in any desired programming language.

There are two fundamentally different approaches to quantify a piece of code, representing an algorithm. Those are the static and the dynamic code analysis.

Static analysis is mainly used for code inspection and white box software testing [Agh+13; KR19], often using information extracted by compilers [Ant+04]. Based on static analysis results the program flow (data flow) can be extracted. While variable allocation and memory consumption can also be retrieved with static analysis, this task becomes more and more complex. The liveness analysis of objects already requires a quite in-depth understanding of the source code and the semantics of the programming language. Such an analysis is much more effort than simply extracting the AST of the program.

Also handling loops is a challenge for symbolic execution [Har+10; Val14]. Providing an example for the code presented in Listing 2.1 the KLEE [CDE08] symbolic virtual machine would explore a total of 256 possible paths within the function shown. By the slight modification of only changing the type of the parameter `x` and variable `y` from `uint8_t` to `uint32_t` this exploration explodes into an insane amount of $2^{32} = 4294967296$ paths.

```c
1  #include <stdint.h>
2  uint8_t func(uint8_t x)
3  {
4      uint8_t i = 0;
5      while(x--)
6      {
7          ++i;
8      }
9      return i;
10 }
```

Listing 2.1: Example with loop, initial source

By running a simple optimization on the input the code in Listing 2.1 will

be transformed into the functionally equivalent representation of Listing 2.2. Basically all commonly used compilers [GNU20; Int20b; Mic20] succeed in appropriately optimizing this example. On a global scale the function itself is likely to be eliminated altogether, provided that inlining is not forcefully disabled.

```
1 #include <stdint.h>
2 uint8_t func(uint8_t x)
3 {
4     return x;
5 }
```

Listing 2.2: Example with loop, optimised source

This transformed function represents the property $f(x) = x$, which is obviously trivial to analyse from a computational (and data flow graph) point of view as well as for performing formal proofs.

In this work we assume the algorithm as such to be correct, hence the formal or non-formal proofs on correctness are not a primary target. This renders the possibility to utilize the large set of optimization algorithms to simplify the source code prior to the analysis, from which the previously demonstrated transformation being one of them.

The example given in Listing 2.1 is, for the sake of the argument, quite artificial. Nevertheless, it resembles quite closely a real life scenario. With only a minor modification (Listing 2.3 and Equation (2.1)) the code will implement a function usually called `foldl` (left fold) [Hut99].

$$s = \sum_{i=0}^{N-1} a[i] \tag{2.1}$$

```
1  #include <stddef.h>
2  #define N 3
3  int TestFunction(const int array[N])
4  {
5      int sum = 0;
6      for(size_t i=0;i<N;++i)
7      {
8          sum += array[i];
9      }
10     return sum;
11 }
```

Listing 2.3: `foldl` C source

Some programming languages have build-in support for this operation (like Haskell [Pey02]), or implement them via library functions (`std::accumulate` in C++).

As stated in [LA04a; RB18b] within compilers it is common practice to separate the language front end, the optimization stages and the code generation. Using a dedicated front end module simplifies the handling of multiple

programming languages. This can cover of a group of similar languages, as it is the case for C, C++ and Objective-C, or a set of different dialects of the same language as it is the case for C++ [ISO98; ISO11; ISO14; ISO17a].

As soon as the various front ends use a common interface for their later stages (GIMPLE, RTL and GENERIC as used in GCC [Mer03], or LLVM assembly language [LA04b] as used in by LLVM project) a common tooling can be used for implementing the optimizations and other tasks.

Intermediate representations for imperative languages often have SSA form [App98]. Such a representation is beneficial for several optimization techniques like constant propagation, variable range analysis and dead code analysis.

For these tasks it does no longer matter whether the front end was parsing a commonly known language (like C, C++ or Java), or generated the intermediate code based on less prominent ones like Cobol, Ada, Mercury, Pascal or Modula-2.

2.2 Extracting Data Dependencies

The concept of symbolic execution is a well known technique, whose basic principles are known since several decades [Kin76]. It has proven to be an invaluable tool for software testing and formal proofs. However, it is less suited for determining concrete examples for the computational effort. One of the reasons for this is the fact that programming languages usually have a very limited set of numerical representations. Path explosion is an abiding companion of symbolic execution [Xu+18].

2.2.1 Integer Ranges

While there are languages which intrinsically offer methods to constrain integer values, most of them do this only in a quite broad range. Following we briefly introduce the integer types used in some programming languages.

Ada [ISO12] which was specifically designed for embedded and real time tasks does allow specific constraints on the possible value range an integer variable can hold (refer Listing 2.4).

```
1  type Custom_Number is range -10 ..  244;
```

Listing 2.4: Ada integer range

C++ [ISO17a] on the other hand has a very limited type system for integers. It does support fixed size integers (like `uint8_t` for an 8-bit unsigned integer), but only on a byte granularity (8, 16, 32, 64 bits). Also, the numerical range is bound to the size. Declaring an 8-bit integer variable whose value ranges from example -10 to 244 is not possible. Integers are obliged to follow the twos complement notation and value range. With some trickery it is of course possible to add more flexibility with respect to the bit size. Executing the code in Listing 2.5 would yield the value of zero rather than 16 which one might

expect (refer Listing 2.6). This is simply caused by the 4 bit limitation given in line 5 of the code, and the corresponding unsigned integer overflow and wrap-around rules. Nevertheless, arbitrary ranges—not limited to the power of two—would require using a dedicated coding or library.

```
1  #include <stdint.h>
2
3  struct CustomInt_t
4  {
5      uint8_t value : 4;
6  };
7
8  int main()
9  {
10     CustomInt_t i;
11     i.value = 15;
12     ++i.value;
13     return i.value;
14 }
```

Listing 2.5: C++ integer range

```
1  %struct.CustomInt_t = type { i8 }
2
3  define i32 @main()
4  {
5      ret i32 0
6  }
```

Listing 2.6: C++ integer range - LLVM assembly language code of Listing 2.5

Haskell [Pey02] even supports to define arbitrary precision integers. These variables have an unbounded range from a definition point of view. While an implementation will experience limitations, those are basically contributed to the finite size of the computer's memory, not by the language or runtime framework. Even on a small computer the limitations caused by memory exhausting will exceed any reasonable limit for a numerical analysis.

Therefore, determining the boundaries for concrete values of integer variables based on the type information only, delivers poor results in the general case. Except for Ada only a very generous upper/lower bound can be derived. As integers are the preferably choice for the data type encoding the number of loop iterations this bound is usually not sufficient to measurably reduce the exploration space.

2.2.2 Indirect Memory Access

If indirect memory access is used, a similar problem does arise. Exemplary on a code as shown in Listing 2.7 concrete execution and symbolic execution or static analysis will produce different conclusions with respect to data dependency.

```
1  #include <array>
2  using A_t = std::array<int, 5>;
3  A_t a = { 1,5,7,9,38 };
4
5  int extr(A_t& src, int offs)
6  {
7      return src[offs];
8  }
9
10 int func(int offs)
11 {
12     return extr(a, offs);
13 }
```

Listing 2.7: Indirect Memory Access

In order to simplify this example, the out of bound check for the memory access is not considered, and it is further assumed that no invalid memory access can occur. That is parameter `offs` is limited to the interval $[0, 4[$.

Both, static and dynamic analysis, can conclude on a data dependency of function `func` to access a single element of the global array `a`. Symbolic execution in accordance with a static analysis can deduce a data read dependency to the array. Without the knowledge of the offset parameter `offs` however the dependency is either on the entire array, or requires forking five analysis paths, one for each possible element. A single concrete execution on the other hand is able to isolate the individual element accessed, but will be unable to create the four forks for the remaining data flow dependencies.

When performing a data flow analysis, whose target is to support distributed and parallel execution, it is most suited to be able to slice vectorized data into its individual elements. By considering the individual array elements as independent entities, they can also be allocated to different memory units in a distributed computation environment. Whether the algorithm as such is formulated using array notation, or by using five individual variables (refer Listing 2.8) should yield identical results. Obviously such a coding would be less maintainable from a source code point of view.

Again the examples given above are somehow artificial. But by generalizing the statements given it can be claimed that compilers and intermediate representations handle aggregate data types quite similar to arrays. For an analysis and distribution investigation it is desirable to decompose aggregate data structures (as shown in Listing 2.9) into their individual elements.

In case the analysis succeeds in abstracting a concrete memory layout, it is no longer relevant whether the data is stored on consecutive memory locations or not. Once the elements of an array can be treated independently of each other they can also have an individual data type, hence be aggregate types. For describing an algorithm and coding it in a programming language these two concepts are fundamentally different. For the dependency analysis they are however much more similar.

```
int a1 = 1;
int a2 = 5;
int a3 = 7;
int a4 = 9;
int a5 = 38;

int func(int offs)
{
    switch(offs)
    {
        case 0: return a1;
        case 1: return a2;
        case 2: return a3;
        case 3: return a4;
        case 4: return a5;
    }
    return 0;
}
```

Listing 2.8: Memory Access - Individual Variables

```
struct Product_t
{
    int     ID;
    double  price;
};

int get_ID(const Product_t& p)
{
    return p.ID;
}

double get_price(const Product_t& p)
{
    return p.price;
}
```

Listing 2.9: Memory Access - Individual Variables

2.2.3 Call Graphs

In addition to the statements required to express the desired algorithm, each programming language has syntactical elements not related to functionality. Declaring functions or objects is not strictly necessary in a lot of cases. Nevertheless, for a clean and maintainable as well as testable implementation they are of paramount importance. They are designed to enable a modular and maintainable code base.

The Listing 2.10 highlights two aspects of this difference. The implementation of the `TestFunction` only serves as syntactic sugar, hiding the internal name of `f4`. It is basically an empty function body, which does nothing more as to serve as alias for `f4`. This is iteratively true for `f3` and `f2`.

Any reasonably configured optimizer will eliminate this hierarchy during its function inlining processing. Also, for a data graph dependency the function

```
int f1( int a1, int b1 ) { return a1 + b1; }
int f2( int a2, int b2 ) { return f1(a2,b2); }
int f3( int a3, int b3 ) { return f2(a3,b3); }
int f4( int a4, int b4 ) { return f3(a4,b4); }

int TestFunction( int a, int b )
{
    return f4(a,b);
}

int main()
{
   return TestFunction(1,2);
}
```

Listing 2.10: Function Calls

calls shall not be reflected as dedicated nodes. Even if the call hierarchy is not inlined by the compiler all the operations performed on function entry, function exit and call invocations do not contribute to the complexity analysis of the algorithm.

Whether the concrete hardware does perform stack adjustment operations, or demands the function operands to be copied into dedicated hardware registers is an implementation detail, and should not be relevant.

2.3 Preparing the Code

Based on the arguments given in the previous sections, this work uses the intermediate representation of the compiler as base for its analysis. As the LLVM project [Lat08] framework has been chosen for front end processing, the intermediate language is the LLVM assembly language [LLV18]. In addition, the analysis is based on a concrete execution of the code. In principle the concepts described in the next sections can be applied to any SSA based representation. However not all aspects of the LLVM assembly language are present in other intermediate languages, like the ones used by the GCC compiler framework [Mer03] or other compilers. The adoptions required to fit into other compiler frameworks are not discussed in this work.

The simplest approach when analysing a concrete execution of the code, is to launch the program via an interpreter. The step by step execution is a trivial base to record all the operations performed by the algorithm. However, such an approach does not deliver a high performance with respect to the execution speed of the analysis.

A more sophisticated approach can be derived based on the nature of the LLVM assembly language itself. Within the LLVM assembly language a function is composed of elementary program units. Each unit can start with an arbitrary instruction, but is allowed only to end with a limited set of terminating instructions. The terminating instructions can for example be a branch instruction, or a return instruction. Such an elementary unit is called `BasicBlock`.

In case the execution enters such a basic block all its instructions must be executed. When conditional branching is required, the branch instruction will terminate the basic block. For each of the conditions of this branch (true or false, respectively branch taken or not taken) a dedicated basic block will be generated. This reflects a significant difference to a front end language like C++. While there are "terminating instructions" in C++ (for example the `return` statement) it is not guaranteed that all code lines within a scope are actually executed. Especially the presence of exceptions do significantly alter the call tree and program flow. Within the LLVM assembly language and language run-time these are represented by a set of dedicated function calls (e.g. `__cxa_allocate_exception`, `__cxa_throw` [Fre17], and the `unreachable` [LLV18] instruction).

The analysis of the execution can therefore process all the instructions of a basic block at once, and is no longer obliged to switch between executing the algorithm, and updating the data flow data on an instruction level granularity, as it would be the case when executing the code by an interpreter. The analysis engine does only require the sequence of basic blocks for its task. If the code of the algorithm to be tested is modified as such that upon entry in each basic block a dedicated callback function is invoked, the required trace data can be made easily available to an external module.

The basic sequence for the data and control flow extraction therefore is a five-step procedure, as listed below:

1. load the code to be analysed into memory
2. modify each basic block, to invoke the trace function upon entry
3. compile the modified module to native code
4. natively execute the module
5. post process the generated tracking data

Performing in memory modification of the loaded module does require inserting instructions into the program, in particular a function call. This function call is required to convey sufficient information to uniquely identify the basic block. The simplest approach in doing so is to assign an arbitrary but unique identifier to each basic block, and modify the code such that each basic block invokes the external tracking function upon its entry. This is depicted in Listing 2.11, with the unique identifier set to the value of 42.

```
1  define i32 @main()
2  {
3      call void @TrackBasicBlock( i32 42 )
4      ret i32 0
5  }
```

Listing 2.11: C++ execution tracking LLVM assembly language [LLV18] code

In order to decouple the analysis code base and the algorithm, it is desirable not to share type information between them. Such a type information for

example can be the type of the analysis object, or the types used within the LLVM project code base. An example is the `llvm::Instruction` class used to implement all LLVM assembly language specific aspects of instructions. This class is usually not present in an intermediate representation of an algorithm.

Aiming to simplify the instrumentation code generator the tracking function is implemented in a two-step approach. The function call inserted into the module operates only with build in types, namely three integers. They represent the address of a second level function within the Just-In-Time compiler's instance, a reference to the tracking instance and the first instruction of the block to be executed [RB18b]. Their basic concept ins shown in Listing 2.12 and Listing 2.13.

```
1  #include <stdint.h>
2  namespace CodeTracker
3  {
4      class  CodeTracker_t;
5  }
6  namespace llvm
7  {
8      class Instruction;
9  }
10
11 typedef int (*Trampoline)(CodeTracker::CodeTracker_t*, llvm::
       ↪ Instruction*);
12
13 extern "C"
14 {
15   void TrackBasicBlock_SpringBoard(uint64_t FctPtr, uint64_t me,
       ↪ uint64_t InstrcPtr)
16   {
17     Trampoline trampoline = reinterpret_cast<Trampoline>(FctPtr);
18     trampoline(reinterpret_cast<CodeTracker::CodeTracker_t*>(me),
19                reinterpret_cast<llvm::Instruction*>(InstrcPtr));
20   }
21 }
```

Listing 2.12: First Level Tracking Function [RB18b]

```
1  void Track_Trampoline(CodeTracker::CodeTracker_t* me, llvm::
       ↪ Instruction* I)
2  {
3      try
4      {
5          me->Track(I);
6      }
7      catch(...)
8      {
9          // ...
10     }
11 }
```

Listing 2.13: Second Level Tracking Function

Using this separation allows the first level (the spring board) to use incomplete types. I can cast its arguments to pointers to forward declared types. This redeems its compilation unit from including any external headers, like the ones of the LLVM project code base. In addition, does this technique of conveying pointers by storing their address values camouflage the types from the module to be instrumented. Finally, forcing a C level calling convention simplifies the implementation of the instrumentation code. In particular this is by ensuring the much simpler C name mangling rules being applied for the spring board function. With the C++ name mangling scheme the function signature would be encoded in the symbol. This would have yield something like `@_Z27TrackBasicBlock_SpringBoardmmm`, depending on the compiler used.

Using the example of the `foldl` function introduced previously (Listing 2.3), the original and modified intermediate representation code is demonstrated in Listing 2.14 and Listing 2.15.

```
1  ; Function Attrs: norecurse nounwind readonly uwtable
2  define dso_local i32 @TestFunction(i32* nocapture readonly)
3  {
4    br label %3
5
6  2:                                                   ; preds = %3
7    ret i32 %8
8
9  3:                                                   ; preds = %3, %1
10   %4 = phi i64 [ 0, %1 ], [ %9, %3 ]
11   %5 = phi i32 [ 0, %1 ], [ %8, %3 ]
12   %6 = getelementptr inbounds i32, i32* %0, i64 %4
13   %7 = load i32, i32* %6, align 4
14   %8 = add nsw i32 %7, %5
15   %9 = add nuw nsw i64 %4, 1
16   %10 = icmp eq i64 %9, 3
17   br i1 %10, label %2, label %3
18 }
```

Listing 2.14: `foldl` original code

The function contains 3 basic blocks. The function entry (an unnamed block), the loop exit “`2:`”, and the loop body “`3:`” graphically visualized in Figure 2.1. Each of which is equipped with a call to the spring board function. This basically conveys the information “this block is now being executed”, as shown in lines 9, 13 and 19 of Listing 2.15. For reasons of better readability the 64 bit address values in these function calls have been simplified to three to four digit decimal numbers in the exemplary code.

2.3.1 Program optimization

In this work we rely on the fact that the algorithm representation, respectively the program code under analysis, has experienced reasonable optimizations. A simple translation of the front end source code (e.g. the C++ files) into the LLVM assembly language intermediate representation, without running optimization algorithms (a debug build), will deliver quite underwhelming results.

```
1   ; Function Attrs: norecurse nounwind readonly uwtable
2   define dso_local i32 @TestFunction(i32* nocapture readonly)
3   {
4     call void @TrackBasicBlock_SpringBoard(i64 140, i64 516, i64
        ↪ 5167)
5     br label %3
6
7   2:                                              ; preds = %3
8     call void @TrackBasicBlock_SpringBoard(i64 140, i64 516, i64
        ↪ 3240)
9     ret i32 %8
10
11  3:                                              ; preds = %3, %1
12    %4 = phi i64 [ 0, %1 ], [ %9, %3 ]
13    %5 = phi i32 [ 0, %1 ], [ %8, %3 ]
14    call void @TrackBasicBlock_SpringBoard(i64 140, i64 516, i64
        ↪ 9328)
15    %6 = getelementptr inbounds i32, i32* %0, i64 %4
16    %7 = load i32, i32* %6, align 4
17    %8 = add nsw i32 %7, %5
18    %9 = add nuw nsw i64 %4, 1
19    %10 = icmp eq i64 %9, 3
20    br i1 %10, label %2, label %3
21  }
```

Listing 2.15: `foldl` modified code

Not only will the generated intermediate representation show artefacts of the implementation which are not related to the algorithm itself, it will also expose lots of boiler plate data introduced by the language front end designed to support later optimization steps. Also, the functionality for target lowering of the code, that is generating native assembly for the desired execution machine, relies on metadata and constructs of the previous processing stages.

An example for the first type of artefacts is given in Listings 2.9 and 2.10. Neither the function calls, nor the fact that within the code the product data is represented in an aggregate data structure do contribute to an understanding of the computational profile of this algorithm. Some of such artefacts can be dissolved by a smart tracking engine, while others cannot. Listings 2.16 to 2.18 demonstrate an optimization, which is assumed to be beyond the capability of a data flow analysis.

Under the assumption that the underlying hardware can do the basic integer arithmetic with identical performance, it is trivial to transform the expression "$a + a + a + a + b + c$" into "$4a + b + c$", and as such reducing five operations to three. Whether the "multiply with four" operation is implemented by the shift instruction or with a multiply instruction depends on the specifics on the hardware. In the majority of cases the shift operation will be more beneficial, and for this reason the intermediate representation prefers this implementation.

Although this transformation is quite obvious it necessitates that the engine performing this transformation has a concrete understanding on the operations as such. In this case it is not only required to know an addition and its mathematical properties, it also requires to know the multiply (or shift) operations

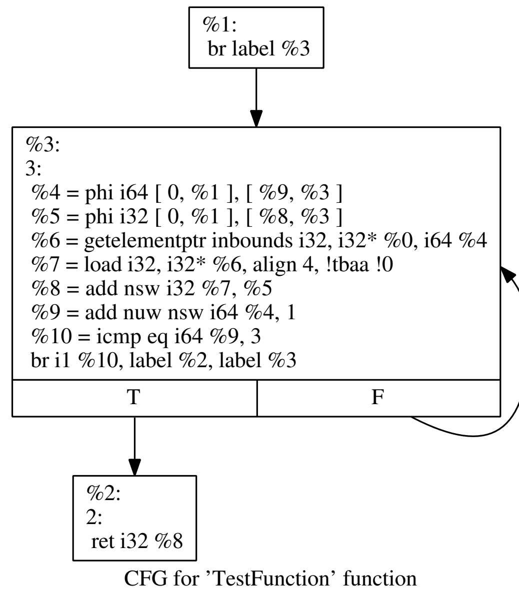


Figure 2.1: Control Flow Graph for Listing 2.14

```
1  int func(int a, int b, int c)
2  {
3      int t1 = a + a;
4      int t2 = a + a;
5      int t3 = t1 + t2;
6      return t3 + b + c;
7  }
```

Listing 2.16: Numeric example (with temporary computations)

and their properties. Additionally, a set of rules is required to describe their mathematical relationship. Without this replacing a sequence of operations (`add`) with a completely different one (`mul`) and still preserving an identical result is not possible.

In a scenario where the dependencies of operations, and their applicability to certain nodes on a distributed hardware shall be analysed this might not be the case. Such a task demands a clear understanding on which data is required for an operation (its inputs), which data is generated (its outputs), and the time required to do so. As such for a common integer ALU, which executes each binary integer operation in the same amount of time, there is no need to distinguish `add`, `sub` or `mul` instructions.

2.3.2 Program optimization, vectorization

Vectorizing loops and arithmetic operations has been a research topic for several decades. A work published in 2011 has made claim "that despite all the work done in vectorization in the last 40 years $45 - 71\%$ of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers we evaluated" [Mal+11]. Especially due to the advancement of GPUs and FPGAs a large amount of computing devices with massively parallel

```
1  ; ModuleID = 'opt.bc'
2  source_filename = "..\5CTempVariableArtefact.cpp"
3  target datalayout = "e-m:w-i64:64-f80:128-n8:16:32:64-S128"
4  target triple = "x86_64-pc-windows-msvc19.22.27905"
5
6  ; Function Attrs: noinline nounwind uwtable
7  define dso_local i32 @func(i32, i32, i32)
8  {
9    %4 = add nsw i32 %0, %0
10   %5 = add nsw i32 %0, %0
11   %6 = add nsw i32 %4, %5
12   %7 = add nsw i32 %6, %1
13   %8 = add nsw i32 %7, %2
14   ret i32 %8
15 }
16
17 !llvm.module.flags = !{!0, !1, !2, !3}
18 !llvm.ident = !{!4}
19
20 !0 = !{i32 2, !"CodeView", i32 1}
21 !1 = !{i32 2, !"Debug␣Info␣Version", i32 3}
22 !2 = !{i32 1, !"wchar_size", i32 2}
23 !3 = !{i32 7, !"PIC␣Level", i32 2}
24 !4 = !{!"clang␣version␣9.0.1␣"}
```

Listing 2.17: Straight forward intermediate representation of Listing 2.16

```
1  ; Function Attrs: noinline norecurse nounwind readnone uwtable
2  define dso_local i32 @func(i32, i32, i32) local_unnamed_addr
3  {
4    %4 = shl i32 %0, 2
5    %5 = add nsw i32 %4, %1
6    %6 = add nsw i32 %5, %2
7    ret i32 %6
8  }
```

Listing 2.18: Optimized intermediate representation of Listing 2.16

computing capability have entered the mass market. For almost any platform, may it be embedded or not, such a structure demands the vectorization still to be an active research topic [Men+19; Ama20]. Optimization algorithms are based on a widespread portfolio [LA00; Haj+20]. But the LLVM assembly language is not always a perfect fit to represent high level type information with respect to graphs. Other representations are under research for more abstracted tasks, like representing TensorFlow graphs [Lat+20] or hardware descriptions [Sch+20].

Especially for a C++ or C front end the extraction of a vectorized representation exhibits more peculiarities than immediately obvious.

For two different implementations of the simple algorithm of element wise adding two 128 element vectors (Listings 2.19 and 2.20), the expectation is to get an ideal coding within the intermediate representation. The expected coding is presented in Listing 2.21. Neither shall the two different implementations produce different output, nor do we expect any overhead added to the LLVM

assembly language. However, with a default configuration none of the compilers available today will generate such an intermediate representation. Even when explicitly instructed only to consider the case for 128 elements (by defining $N = 128$ as compile-time constant). This is due to the fact that compilers use a dedicated metric to identify the vectorization width. Usually this selects values between 4 and 16. Therefore, the compiler has to generate a loop or a sequence of instructions to perform the 128 additions in packs of 4 to 16 values. The chosen value depends on the intended hardware architecture. This is not because such an representation is not valid in the intermediate representation, it is more attributed to the fact that current compilers are designed for regular CPUs. Those usually do not employ a capability to handle such an amount of parallelization, therefore generating such code a does not make sense.

```
1 #include <algorithm>
2 #include <functional>
3
4 template<typename T, unsigned N>
5 void  TestFunction(const T (& a)[N], const T (& b)[N], T (& c)[N])
6 {
7     using namespace std;
8     transform(cbegin(a), cend(a), cbegin(b), begin(c), plus<>{});
9 }
```

Listing 2.19: Adding two vectors, C++ code

```
1 #define N   128
2 void foo(const float * a, const float *  b, float *c)
3 {
4     for(int idx=0;idx<N;++idx)
5         c[idx] = a[idx] + b[idx];
6 }
```

Listing 2.20: Adding two vectors, C code

```
1 %5 = load <128 x float>, <128 x float>* %a, align 4
2 %7 = load <128 x float>, <128 x float>* %b, align 4
3 %8 = fadd <128 x float> %5, %7
4 store <128 x float> %8, <128 x float>* %c, align 4
```

Listing 2.21: Adding two vectors, expected intermediate representation code

Forcing the compiler to use a factor of 128 for the vectorization still does only partly deliver the expected code (Listing 2.22, lines 13-21). Doing so requires a dedicated set of command line options, in the case of the clang compiler they are "`-x c -O3 -emit-llvm -mllvm -force-vector-width=128 -g0`". This time the boiler plate code (initial basic block, and the block following label `22:` in line 24) is caused by a property of the front end language. Neither the C code (Listing 2.20) nor the C++ coding (Listing 2.19) express any guarantee with respect to aliasing (Listing 2.22). The two pointers `a` and `c` might very well

point to the same object, similar as the arrays `b` and `c` might be the same object. From a language point of view the compiler is not allowed to assume those memory regions to be distinct. This mandates the run-time checks performed upon function entry.

In the C language the desired constraint can be expressed with the `restrict` keyword (Listings 2.23 and 2.24). The same holds true when reverting to the GCC compilers `__restrict__` (Listings 2.25 and 2.26) in C++. The latter keyword however is not standardized.

Summarizing the aspects of vectorization it is evident that if relying on a regular compiler it cannot be guaranteed that full vectorization is always expressed within the LLVM assembly language. The reasons given are the inherent property of having a CPU like execution unit in mind, and the potential inability of the front end to express the vectorization properly. For a hardware structure which has the capability to actually execute in an embarrassingly parallel [Fos95] manner additional handling is required. For the cases given a desired result would be to extract a data flow graph which represents the embarrassingly parallel nature of the vector addition. Independent of whether the intermediate representation represents the operation in a single vectorized instruction, or a sequence of 128 sequential instructions, or any combination of which.

2.3.3 Front End and Compiler Optimizations

The primary input to the analysis engine is based on the abstraction level of the compilers intermediate representation. The implementation however is based on a dedicated programming language, C++ in the examples given. As discussed in the previous sections the concept relies on a decent set of optimization to produce reasonable results. In the following sections some compiler optimizations as well as peculiarities of the LLVM assembly language, and their impact to the analysis concept are discussed.

In this context the generation of the CDFG competes with the vectorization optimization passes of the compiler. As the vectorization capabilities of the front end are usually limited to a certain width (usually less than 4) there is not much lost by inhibiting the vector optimization in the front end.

Throughout this work the front end and optimization configuration has been chosen as such that the vectorization passes are disabled.

2.3.4 Φ-nodes

The SSA notation is generated from the program source, used to represent the code internally to the compiler. Paired with the control flow graph it has established itself as suitable data structure for compilers. It allows efficient implementation of various operations of paramount importance to compilers in general, and optimization algorithms in particular [Cyt+91]. For a discussion on one of such optimizations, the constant propagation, refer [WZ91].

Φ-nodes are a special instruction within SSA notations. The Φ-instruction is a typed instruction, with an arbitrary long list of input tuples. Each input tuple is composed of a reference to a variable and a referenced basic block.

```
1  define dso_local void @foo(float* %0, float* %1, float* %2) {
2    %4 = getelementptr float, float* %2, i64 128
3    %5 = getelementptr float, float* %0, i64 128
4    %6 = getelementptr float, float* %1, i64 128
5    %7 = icmp ugt float* %5, %2
6    %8 = icmp ugt float* %4, %0
7    %9 = and i1 %7, %8
8    %10 = icmp ugt float* %6, %2
9    %11 = icmp ugt float* %4, %1
10   %12 = and i1 %10, %11
11   %13 = or i1 %9, %12
12   br i1 %13, label %22, label %14
13 14:                                              ; preds = %3
14   %15 = bitcast float* %0 to <128 x float>*
15   %16 = load <128 x float>, <128 x float>* %15, align 4
16   %17 = bitcast float* %1 to <128 x float>*
17   %18 = load <128 x float>, <128 x float>* %17, align 4
18   %19 = fadd <128 x float> %16, %18
19   %20 = bitcast float* %2 to <128 x float>*
20   store <128 x float> %19, <128 x float>* %20, align 4
21   br label %21
22 21:                                              ; preds = %22, %14
23   ret void
24 22:                                              ; preds = %3, %22
25   %23 = phi i64 [ %51, %22 ], [ 0, %3 ]
26   %24 = getelementptr inbounds float, float* %0, i64 %23
27   %25 = load float, float* %24, align 4
28   %26 = getelementptr inbounds float, float* %1, i64 %23
29   %27 = load float, float* %26, align 4
30   %28 = fadd float %25, %27
31   %29 = getelementptr inbounds float, float* %2, i64 %23
32   store float %28, float* %29, align 4
33   %30 = or i64 %23, 1
34   %31 = getelementptr inbounds float, float* %0, i64 %30
35   %32 = load float, float* %31, align 4
36   %33 = getelementptr inbounds float, float* %1, i64 %30
37   %34 = load float, float* %33, align 4
38   %35 = fadd float %32, %34
39   %36 = getelementptr inbounds float, float* %2, i64 %30
40   store float %35, float* %36, align 4
41   %37 = or i64 %23, 2
42   %38 = getelementptr inbounds float, float* %0, i64 %37
43   %39 = load float, float* %38, align 4
44   %40 = getelementptr inbounds float, float* %1, i64 %37
45   %41 = load float, float* %40, align 4
46   %42 = fadd float %39, %41
47   %43 = getelementptr inbounds float, float* %2, i64 %37
48   store float %42, float* %43, align 4
49   %44 = or i64 %23, 3
50   %45 = getelementptr inbounds float, float* %0, i64 %44
51   %46 = load float, float* %45, align 4
52   %47 = getelementptr inbounds float, float* %1, i64 %44
53   %48 = load float, float* %47, align 4
54   %49 = fadd float %46, %48
55   %50 = getelementptr inbounds float, float* %2, i64 %44
56   store float %49, float* %50, align 4
57   %51 = add nuw nsw i64 %23, 4
58   %52 = icmp eq i64 %51, 128
59   br i1 %52, label %21, label %22
60 }
```

Listing 2.22: Adding two vectors, LLVM assembly language code of Listing 2.20

```
1  #define N   128
2
3  void foo(const float * restrict a, const float * restrict b, float
       ↪  * restrict c)
4  {
5      for(int idx=0;idx<N;++idx)
6          c[idx] = a[idx] + b[idx];
7  }
```

Listing 2.23: Adding two vectors, C code with expressing alias free input

```
1  define dso_local void @foo(float* noalias nocapture readonly %0,
       ↪ float* noalias nocapture readonly %1, float* noalias
       ↪ nocapture %2)
2  {
3    %4 = bitcast float* %0 to <128 x float>*
4    %5 = load <128 x float>, <128 x float>* %4, align 4, !tbaa !2
5    %6 = bitcast float* %1 to <128 x float>*
6    %7 = load <128 x float>, <128 x float>* %6, align 4, !tbaa !2
7    %8 = fadd <128 x float> %5, %7
8    %9 = bitcast float* %2 to <128 x float>*
9    store <128 x float> %8, <128 x float>* %9, align 4, !tbaa !2
10   ret void
11 }
```

Listing 2.24: Adding two vectors, LLVM assembly language code of Listing 2.23

All variables in this list of tuples must have the same type. At runtime the Φ-node represents one of the listed values. The selection which of those is chosen is dictated by the program flow. The basic block which was executed previously to encountering the Φ-node is the basis of this selection. For this reason Φ-nodes by definition are only allowed at the very beginning of a basic block.

Reverting again to code example in Listing 2.14 lines 10 and 11 give an example for the usage of such a Φ-node. Variable `%4` represents the loop counter. Its value is the constant zero in case the loop body is approached via the function entry (the basic block in line 4, by convention of the LLVM assembly language implicitly named `%1`), and the value of variable `%9` when approached by the loop body (`%3`). This models a variable which is initialized to zero, and incremented in every loop iteration. The very same concept is applied to the variable pair `%5, %8`, whose represent the sum for the fold operation.

Resolving Φ- nodes to their corresponding concrete value (or variable) is not a particularly complicated task. It does only require to keep track of the previously executed basic block. A single function can be composed of an arbitrary amount of basic blocks, whose in turn can be executed in any order and repetition. Therefore, such a tracking is more complex than to memorize the previous instruction (program counter). It has to follow a call stack (last in first out (LIFO) queue) like concept, but on the abstraction level of basic blocks, rather than functions. For the sake of completeness it should be noted that the Φ-node selection can yield another Φ-node, and as such demanding the resolving to work recursively until a regular variable is encountered.

```
1  #include <algorithm>
2  #include <functional>
3  #include <memory>
4  using namespace std;
5
6  template<typename T, unsigned N>
7  void  TestFunction( const T  (& __restrict__ a)[N], const T (&
       ↪ __restrict__ b)[N], T (& c)[N])
8  {
9       transform(cbegin(a), cend(a), cbegin(b), begin(c), plus<>{});
10 }
11
12 using T = int;
13 constexpr int N = 128;
14
15 template void TestFunction(const T (& a)[N], const T (& b)[N], T
       ↪ (& c)[N]);
16
17 void  foo( const T  * __restrict__ a, const T * __restrict__ b, T*
       ↪  c)
18 {
19      transform(a, a+N, b, c, plus<>{});
20 }
```

Listing 2.25: Adding two vectors, C++ with compiler extension

2.3.5 Function calls

The LLVM assembly language allows representing code segments in separate functions, a feature which almost all programming languages support. Dealing with them is a necessity for the data flow extraction engine. Depending on the specific settings configured for the language front end, a large set of functions can be eliminated by the help of inlining. However, a modern compiler usually does not provide guarantees whether a function is inlined or not. In particularly it can be safely assumed that in every module there will be at least one function call left. The reason for the compilers not to give such guarantees is to keep their manoeuvrability with respect to program optimization as high as possible. Further, flattening a full program into a single function is usually not practical.

As the function calls cannot be completely eliminated, they require some logic within the data and control flow tracking engine. In order to properly merge the data and CDFG generated within a sub function into the parent scope, its root and leaf nodes need to be linked to the global scope. Basically the principle relies on the input and output arguments of a function to be mapped to the local variables in the callers scope. Similarly, the return value of the function is to be linked to the corresponding variable at the caller side. For the tracking it is of minor importance whether the variables exposed from the module have the form of a function return code, or are syntactically modelled by call be reference [ISO17b] semantic in the language front end.

```
1  void TestFunction<int, 128u>(int const (&) [128u], int const (&)
     ↪ [128u], int (&) [128u]):
2          xor     eax, eax
3  .L2:
4          movdqu  xmm0, XMMWORD PTR [rsi+rax]
5          movdqu  xmm1, XMMWORD PTR [rdi+rax]
6          paddd   xmm0, xmm1
7          movups  XMMWORD PTR [rdx+rax], xmm0
8          add     rax, 16
9          cmp     rax, 512
10         jne     .L2
11         ret
12 foo(int const*, int const*, int*):
13         xor     eax, eax
14 .L6:
15         movdqu  xmm0, XMMWORD PTR [rsi+rax]
16         movdqu  xmm1, XMMWORD PTR [rdi+rax]
17         paddd   xmm0, xmm1
18         movups  XMMWORD PTR [rdx+rax], xmm0
19         add     rax, 16
20         cmp     rax, 512
21         jne     .L6
22         ret
```

Listing 2.26: Adding two vectors, x86 assembly code Listing 2.25

2.3.6 Initialization and detecting input and output data

For any concrete execution a reasonable set of input data is required to produce a suitable computational profile upon investigating the algorithm. For this reason it is beneficial to exclude a certain initialization procedure from the analysis. In absence of a Just-In-Time compiler and on a hardware model which does not apply data dependant costs to arithmetic operations the initialization has minor impact to the computational profile. A matrix vector multiplication can, without the loss of generality, be decomposed in its intermediate representation with its input being run-time initialized to "all zero" values. The concrete execution will yield a vector with each of elements to be zero, but the amount of multiplications and additions performed will be a suitable representation of the computational effort. Once the procedure becomes more complex, and for example includes a floating point division, the simple "all zero" initialization might no longer deliver a representative operation pattern. The early program abort due to a divide by zero exception will render incorrect results. A similar effect can arise if the hardware model employs a data dependant execution time. This can for example mimic an optimization that $a \cdot b = a$ if $b = 1$. This is a common implementation in software based floating point arithmetic (refer Listing 2.31 for a similar optimization).

In order to compose an executable program the algorithm therefore will not be represented by the program entry (e.g. the `main` function), but rather by a function invoked at some point within the execution flow. This has two consequences. First there is an arbitrary long sequence of code which shall not contribute to the CDFG. Reading data from a file, or generating random or pseudo-random data are examples of which. This aspect can easily be

handled by not instrumenting those functions with the various `Track_` functions described in Section 2.3 and Listing 2.11. The second aspect is more subtle. The initialization routines are likely to initialize various variables, which as such from a language specification point of view, must not have a read only property. They cannot be constant. From an algorithm point of view they might however be constant, in case they represent a constant input. In an equation like $c = 4b$, the variable "b" as well as the literal "4" are constant within the context of the expression. However, in case b is to be initialized by some code outside the scope of the analysis, it cannot be represented as a constant variable. For an analysis this reflects the fact that if a variable is flagged as constant within the LLVM assembly language it implies to be read only. However, it cannot be concluded that any non-constant variable is actually written. This information has to be extracted from the execution tracing.

2.3.7 Cross function Φ-nodes

The argument tracing will not only need to resolve function arguments to its corresponding parent scope variables, it also has to do this in the awareness of Φ-nodes. In a first order Φ-node resolving, the local variable can resolve to a function argument. In this case the resolver has to continue its mapping in the parent scope of the caller. On this second level the argument on the caller side can also be represented by Φ-node.

An example of such a coding is given in Listings 2.27 and 2.28. Resolving the variable `%1` in Listing 2.28 line 4 causes a sequence of back tracking events.

1. Resolve `%1` on local function scope
   By convention the arguments of the function are numbered from `%0` onwards. `%1` hence is the second argument of the function `?add@@YAHHH@Z`.

2. Resolve the argument via the call stack information
   Within the callers scope (line 24), the second argument is extracted, yielding variable `%8`.

3. As `%8` is an indirect access, the back tracking has to continue
   The load instruction (pointer indirection) of variable `%8` yields the Φ-instruction in line 18 (`%.0`).

4. Φ-node transformation
   The Φ-node in question is within basic block "`5:`", while the load instruction is in basic block "`7:`". To correctly resolve the variable `%.01`, knowledge of the basic block from which block "`5:`", not block "`7:`" was entered in its last execution is required.

By this example it is obvious that the basic block tracking needs to reflect the stack based scoping as mentioned in Section 2.3.4.

2.3.8 GEP-nodes

When dealing with data arrays, or aggregate data structures in general, another aspect of the intermediate language has to be taken into account. As soon as a

```
1  constexpr int N = 5;
2
3  int add(int a, int b)
4  {
5      return a+b;
6  }
7
8  int func(int (&a)[N])
9  {
10     int s = 0;
11     for(int v : a)
12     {
13         s = add(s,v);
14     }
15     return s;
16 }
```

Listing 2.27: Φ-node example C++ source

computational operation is applied to data stored in an array, it is required to explicitly reference a single entry within the aggregate set.

Doing this causes memory access to happen via the pointer arithmetic like scheme of the `getelemptr` instruction. This closely resembles the index operator of the C language.

Digging into the details of this index operator (and as such the `getelemptr`) reveals that for those not only an arbitrary number of arguments is required, but also that resolving the corresponding element of the aggregate structure requires the runtime values of the arguments. The analysis concept described so far does perform a dynamic analysis of which blocks are to be traced (and their order of execution), backed up by a purely static analysis of the block content itself. The operand data for the array indexing is only available as reference to a variable or the instruction computing it but, not the actual value it has been assigned when executing the array indexing.

In order to make this values available for the control and data flow extraction engine the dynamic part of the instrumentation requires some extension. Most prominently the static processing of the elementary block has to be interrupted upon reaching an instruction which requires the actual values of its operands. Doing so demands the instrumentation mechanism to be changed. Rather than simply inserting an informative callback ("this block is now executed") at the beginning of each elementary block and recursively doing so for all subroutines, the memory access instructions need to be handled. Differently than inserting a simple informative callback the extended tracing function is required to additionally convey the actual values of the arguments.

Listing 2.29 demonstrates the instrumentation required for such an extended data tracking. In line 14 as described above the regular basic block tracking is done. The argument data for the `TrackBasicBlock_SpringBoard` functions consists of references to the internal data structures of the Just-In-Time compiler only. This information does only relate to the source of the algorithm under analysis, they do not convey any information on concrete values of variables. As such this tracking component is restricted to perform static analysis, on the

```
1  ; Function Attrs: noinline nounwind uwtable
2  define dso_local i32 @"?add@@YAHHH@Z"(i32, i32)
3  {
4    %3 = add nsw i32 %0, %1
5    ret i32 %3
6  }
7
8  ; Function Attrs: noinline nounwind uwtable
9  define dso_local i32 @"?func@@YAHAEAY04H@Z"([5 x i32]*
       ↪ dereferenceable(20))
10 {
11   %2 = getelementptr inbounds [5 x i32], [5 x i32]* %0, i64 0, i64
        ↪ 0
12   %3 = getelementptr inbounds [5 x i32], [5 x i32]* %0, i64 0, i64
        ↪ 0
13   %4 = getelementptr inbounds i32, i32* %3, i64 5
14   br label %5
15
16 5:                                                ; preds = %10, %1
17   %.01 = phi i32 [ 0, %1 ], [ %9, %10 ]
18   %.0 = phi i32* [ %2, %1 ], [ %11, %10 ]
19   %6 = icmp ne i32* %.0, %4
20   br i1 %6, label %7, label %12
21
22 7:                                                ; preds = %5
23   %8 = load i32, i32* %.0, align 4
24   %9 = call i32 @"?add@@YAHHH@Z"(i32 %.01, i32 %8)
25   br label %10
26
27 10:                                               ; preds = %7
28   %11 = getelementptr inbounds i32, i32* %.0, i32 1
29   br label %5
30
31 12:                                               ; preds = %5
32   ret i32 %.01
33 }
```

Listing 2.28: Φ-node example LLVM assembly language source

abstraction level of a single basic block.

The static analysis has to stop processing the `BasicBlock` ad interim once approaching the `getelementptr` in Listing 2.29 line 17. This is caused by its incapacity to properly track the argument denoting the array index (the last argument of the instruction, `%4`). To perform a proper slicing of the array, and providing a data flow graph with decomposed elements, the concrete value of variable `%4` is required. The newly introduced tracking function `Track_GEP_i64` steps into this role, providing the essential data. In addition to the similar references to the code base, as used in the `TrackBasicBlock_SpringBoard` invocations, its last argument holds the index information for the slicing (variable `%4` in this case).

Once the concrete values have been feed into the tracking engine, the static instruction sweep of the basic block will continue. By convention of the implementation this is implemented by an additional call to the already existing `TrackBasicBlock_SpringBoard` function. While not obvious in this example

```
1  define dso_local i32 @TestFunction(i32*)
2  {
3    call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 6032)
4    br label %2
5
6  2:                                               ; preds = %10,
     ↪ %1
7    %3 = phi i32 [ 0, %1 ], [ %9, %10 ]
8    %4 = phi i64 [ 0, %1 ], [ %11, %10 ]
9    call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 6328)
10   %5 = icmp ult i64 %4, 3
11   br i1 %5, label %6, label %12
12
13 6:                                               ; preds = %2
14   call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 3728)
15   call void @Track_GEP_i64(i64 9357, i64 3624, i64 3728, i64 %4)
16   call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 3728)
17   %7 = getelementptr inbounds i32, i32* %0, i64 %4
18   %8 = load i32, i32* %7, align 4
19   %9 = add nsw i32 %3, %8
20   br label %10
21
22 10:                                              ; preds = %6
23   call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 176)
24   %11 = add i64 %4, 1
25   br label %2
26
27 12:                                              ; preds = %2
28   call void @TrackBasicBlock_SpringBoard(i64 1147, i64 3624, i64
       ↪ 7208)
29   ret i32 %3
30 }
```

Listing 2.29: Tracking `getelemntptr` instructions

the `getelementptr` instructions, contrary to for example the Φ-instructions, can be arbitrarily distributed within the basic block. For this reason the separation of concrete tracking and static sweeps is beneficial from an implementation point of view. Conceptually lines 14, 15 and 16 in Listing 2.29 could have been merged into a single call.

In addition to giving an example of the array slicing capability, Listing 2.29 demonstrates two other aspects. Firstly it can be observed that the arrangement of basic blocks differs from Listings 2.14 and 2.15. Nevertheless, both implementations reflect the same algorithm. The arrangement of basic blocks is heavily influenced by the compiler options, most notably the configured optimization levels.

For the examples given the performed optimization steps have mostly been individually chosen. The focus is on providing applicative examples, not primarily generating the most efficient code. An optimized (but not vectorized)

version of the `foldl` operation on three elements is given in Listing 2.30.

Generalizing the findings we can conclude that there is no guarantee on the order of operations, or a particular arrangement on basic blocks. While the compiler is obliged to produce "correct" code, the "as-if" rule [ISO17a] offers a large degree of freedom in the concrete implementation. This is the base of all compiler optimizations [God17; Tan+20].

```
1  define dso_local i32 @TestFunction(i32*)
2  {
3    call void @TrackControl_SpringBoard(i64 40952, i64 7032, i8 1)
4    call void @TrackBasicBlock_SpringBoard(i64 1147, i64 7032, i64
       ↪ 6488)
5    %2 = load i32, i32* %0
6    call void @Track_GEP_i64(i64 9357, i64 7032, i64 1376, i64 1)
7    call void @TrackBasicBlock_SpringBoard(i64 1147, i64 7032, i64
       ↪ 1376)
8    %3 = getelementptr inbounds i32, i32* %0, i64 1
9    %4 = load i32, i32* %3
10   %5 = add nsw i32 %4, %2
11   call void @Track_GEP_i64(i64 9357, i64 7032, i64 6896, i64 2)
12   call void @TrackBasicBlock_SpringBoard(i64 1147, i64 7032, i64
       ↪ 6896)
13   %6 = getelementptr inbounds i32, i32* %0, i64 2
14   %7 = load i32, i32* %6
15   %8 = add nsw i32 %7, %5
16   ret i32 %8
17 }
```

Listing 2.30: `foldl` optimized representation

2.3.9 Raising Abstraction Level

One of the greatest drawback of evaluating the algorithm on LLVM assembly language scope its very low level view of things. The LLVM assembly language does to a large extend reflect the instructions of a general purpose CPU. While it does have an abstract view on data types and memory from an operation point of view, it boils down to basic assembly level operations. Generating native machine code out of LLVM assembly language is therefore a task with acceptably small effort. The target specific back-end needs to perform register allocation and stack assignment to variables, but can map most intermediate representation statements directly to their corresponding machine instruction.

For data flow and sequencing this has shown to be very effective abstraction. Quite commonly although it is beneficial to raise the abstraction level for certain computations. This can for example be the case for functions whose computational profile is well known. In such a case an optimal assignment of the individual operations to the compute-nodes is either well known, or a dedicated library is used for this purpose. In both cases it usually does not provide additional insights if these functions are dissolved, and included in the analysis. Alternatively it might be desirable to run an investigation on a partially implemented algorithm. A partially implemented algorithm is a piece of code where not all the dependencies are known to the compiler. For

some functions only their signature (inputs and outputs) are known, and only a mock-up implementation is available.

Such functional blocks (function calls) will be threaded as unresolved external entities. As long as they do not generate data whose concrete value is needed (like the array indices shown in Listing 2.29 line 17) this will not affect the computational analysis. It will—in the general case—consider the unresolved function call as regular operation, with its function arguments as inputs, and the functions return code as output.

Listing 2.32 demonstrates the computation of the vector L2 norm, according to Equation (2.2).

$$x = \sqrt{\sum_{i=0}^{N} |a_i|^2} \tag{2.2}$$

Assuming the hardware in question is capable of providing the `sqrt` instruction, but neither has support for a hardware loop nor floating point squaring, the above mechanisms suites the desired functionality. The computation of the square root shall be handled as "external call", in this simple case with a single input, and a single output. The remaining part of the algorithm shall be optimized at full extend, and handled accordingly. This includes the likely mapping according to Equations (2.3) and (2.4), as well as decomposing the C++ lambda expression within loop caused by the function `std::accumulate`.

$$\forall x \in \mathbb{R} : |x|^2 = x^2 \tag{2.3}$$

$$x^2 = x \cdot x \tag{2.4}$$

Listing 2.33 depicts the corresponding intermediate representation, demonstrating the desired behaviour. The function call to the square root computation (line 37) is simply integrated into the code as external function call. The analysis engine can threat it as single input single output operation. The lambda function (Listing 2.32 line 10) with its `std::pow` invocation is fully dissolved. Its multiply accumulate (MAC) operation is reflected in Listing 2.33 lines 24-26.

In particular the optimization of the `pow` function relies on an interplay of compiler optimization and library implementation. With squaring being quite common, and the transformation according to Equation (2.4) usually being beneficial on most hardware platforms, the usual implementation of the `pow` function (Listing 2.31) reflects this property. Inlining and constant propagation will be in charge of reducing the source level boiler plate code into a single `fmul float %10, %10` instruction (Listing 2.33 lines 25). And common mathematical optimization for the float data type (Equation (2.3)) can eliminate the need for the `abs` function.

This can also be done on a much different scope for example on a sparsely distributed system with a web server and a resource limited interface node [Tan+17; Rec+18]. For this case running a geo-location algorithm (global navigation satellite system data to physical address lookup) is performed. In this scenario only the server part is capable of running the lookup in full detail, that is solving the point in polygon problem for a large set of complex polygons.

```
1 inline float pow(float mantissa, int exponent)
2 {
3     if (exponent == 2)
4     {
5         return mantissa * mantissa;
6     }
7
8     return powf(mantissa, static_cast<float>(exponent));
9 }
```

Listing 2.31: pow implementation

```
1  #include <numeric>
2  #include <cmath>
3  #include <iterator>
4
5  template<typename T, size_t N>
6  T TestFunction(const T(& a)[N])
7  {
8      using namespace std;
9      return sqrt(accumulate(begin(a), end(a),T(0),
10          [](T sum, T next){ return sum + pow(abs(next),2); }));
11 }
```

Listing 2.32: vector norm

Such polygons occur when mapping a latitude and longitude data obtained by global navigation satellite system to a district with a certain area. As the district boundaries are usually of highly irregular shape, numerous points are required to correctly describe those boundaries as polygons. A trade-off has to be found on either transmitting all or just parts of the large polygon set (database) to the interface node, or running all computations on the server.

2.4 Representing Hardware

Once the control and data flow has been extracted, a suitable metric is required to express a statement whether a dedicated arrangement of compute-nodes and storage-nodes is suited for this particular algorithm. Such a metric can have quite widespread criteria, depending on the design goal of the hardware arrangement.

While the execution time is certainly not the only criteria applied, in most of the cases it will however be of some relevance. In order to estimate the amount of time required by a certain "distributed device" to execute an algorithm the CDFG needs to be mapped onto this hardware arrangement. For each of the operations to be performed an execution-node within the hardware representation needs to be chosen. Such a selection requires to consider the data, time and transfer dependencies. For the sake of the argument we assume that only binary functions are available, in particular no ternary add is available. For this assumption the data dependency of a simple 3 element add operation (Figures 2.2a and 2.2b) is expressed in the data flow graph (Figure 2.2c). It

depicts the second `add` operation (`add (5 | 7)`) to have a data dependency on the first one (`add(3 | 4)`). Within an execution mapping these two operations are bound to be executed sequentially. They still can be distributed on different execution-nodes. Whether this is beneficial from a performance point of view depends on the layout of the hardware's storage units, mainly whether an additional data transfer is required in such a case.

Another common metric for a dedicated hardware mapping is the utilization of compute- and storage units. Not only is this a good indicator whether the selected hardware arrangement fits to the desired algorithm, such a metric is relevant for other aspects as well. Having lots of unused hardware units indicates a waste of resources, likely causing inefficient energy usage.

Such an under utilized hardware can be caused by a limitation of the algorithm itself. Namely, a sequential algorithm being applied to a hardware unit capable of parallel computation [Amd67; Gus88]. It can also indicate a constraint in the hardware arrangement, like a bottleneck for the data transfer. [WWP09] introduced the term "operational density" to discuss multi processor architectures and their performance bounds with respect to operations and memory bandwidth.

2.4.1 Hardware Model

With representing a hardware arrangement as graph, utilization of all the graph based algorithms is possible within the estimation and mapping process. A data flow graph can be expected to be a directed acyclic graph (DAG), as it is generated by the SSA based intermediate representation. A graph representing a hardware composition will not have this property.

Its nodes are suitably composed of two different types. Nodes which can store data (storage-nodes), and nodes capable of servicing various operations (compute-nodes). The membership to these two groups is a metadata property of each node. Whether a membership to one of the two groups is however mutually exclusive depends on the abstraction level of the modelling. The edge weights will reflect the transfer cost applied to values passing in between nodes. When this property is expressed in transfer time, the usual set of path finding and shortest path algorithms [Bel58; Dij59] can be employed to estimate the computational cost of a specific operation on a desired node.

In a quite low level style of hardware modelling, a CPU core and its attached storage units can have quite different transfer cost. Likely the CPU to register bank transfer is the cheapest, probably even a zero cost data exchange. A non-volatile storage like a flash memory might require several CPU cycles to be accessed. Also, the estimator for the data and computation distribution will need to honour the fact that a flash memory is not suited for temporary data due to its limited write cycles. Using a directed graph, with the storage-node for the flash memory being connected in one direction only, can be used to reflect this property. Any attempt to find a path to store a value within the flash is doomed to fail in such a case.

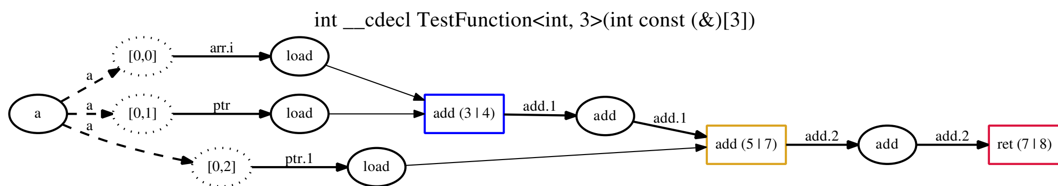
A simplified example of a RISC based CPU architecture is depicted in Figure 2.3. All nodes but one (the `CPU/floating point processing unit (FPU)`) represent the various storage options for variables in a basic micro-

```
1 #include <numeric>
2 #include <iterator>
3
4 template<typename T, unsigned N>
5 T TestFunction(const T (& a)[N])
6 {
7   using namespace std;
8   return accumulate(begin(a), end(a), T(0));
9 }
```

(a) C++ source

```
1  ; Function Attrs: nounwind uwtable
2  define linkonce_odr dso_local i32 @"??
      ↪ $TestFunction@H$02@@YAHAEAY02$$CBH@Z"([3 x i32]*
      ↪ dereferenceable(12) %a)
3  {
4   entry:
5   %arr = getelementptr inbounds [3 x i32], [3 x i32]* %a, i64 0,
       ↪ i64 0
6   %0 = load i32, i32* %arr, align 4
7   %ptr = getelementptr inbounds [3 x i32], [3 x i32]* %a, i64 0,
       ↪ i64 1
8   %1 = load i32, i32* %ptr, align 4
9   %add.1 = add nsw i32 %1, %0
10  %ptr.1 = getelementptr inbounds [3 x i32], [3 x i32]* %a, i64 0,
       ↪ i64 2
11  %2 = load i32, i32* %ptr.1, align 4
12  %add.2 = add nsw i32 %2, %add.1
13  ret %add.2
14 }
```

(b) intermediate representation source



(c) Control and data flow graph

Figure 2.2: Example, sum elements of array

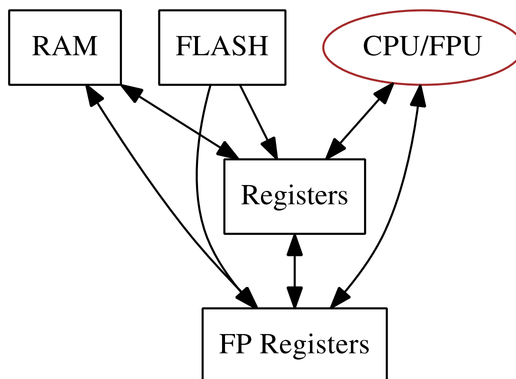


Figure 2.3: ARM Cortex-M (simplified) [RB18a]

controller entity. Similar with two exceptions all connections are bidirectional. Only the flash connections are unidirectional. It cannot be written in this model. However, it might quite well suite as storage for constants.

In order to model a storage-node which can hold variables prior to executing the algorithm to be analysed a "materialization" property is introduced. Any storage-node might declare itself to have this property, indicating values can be applied to it without the execution of the instructions actually executing a store operation.

The model in Figure 2.3 still is quite simple as the `CPU/FPU` node can access both register banks, the regular one and the floating point one. Also there is no distinction between integer and floating point arithmetic. More complex system descriptions, as shown in Figure 2.4, may separate the integer and floating point units. For this model the `CPU` core cannot directly access the `FPU` cores registers, and vice versa. Data exchange in between those two would utilize the `RAM` storage-node. Transferring a constant from the `ROM` to the `FPU` node is a quite expensive task. The shortest and (if excluding insentient ones) only path is via the `Registers` node to the `RAM` node, and then via the `FP Registers` to the `FPU`.

The DSP system in Figure 2.4 resembles a simplified representation of a C6x micro-controller. It has two register banks (`Register A, Register B`) two set of identical compute units (`L, M, S, D`). Each of which has a slightly different set of capabilities [Ins02]. The `M` unit for example is unable to generate constants, while the `S` unit is less capable with respect to multiply operations.

2.4.2 Execution time estimation basics

For properly modelling the behaviour of such arrangements the data transfer model has to be extended. Transferring data alongside the graph is no longer just a matter of consuming time (specified by the edge weight), but might also allocate compute-nodes to do so. In order to express such behaviours in the hardware graph an additional metadata property is required to be assigned to each edge. This property declares a set of compute units required to perform a data transfer alongside this edge.

After—presumably—identifying the shortest path by traditional algorithms [Bel58; Dij59] the cost model requires adaption. In case a transfer utilizes

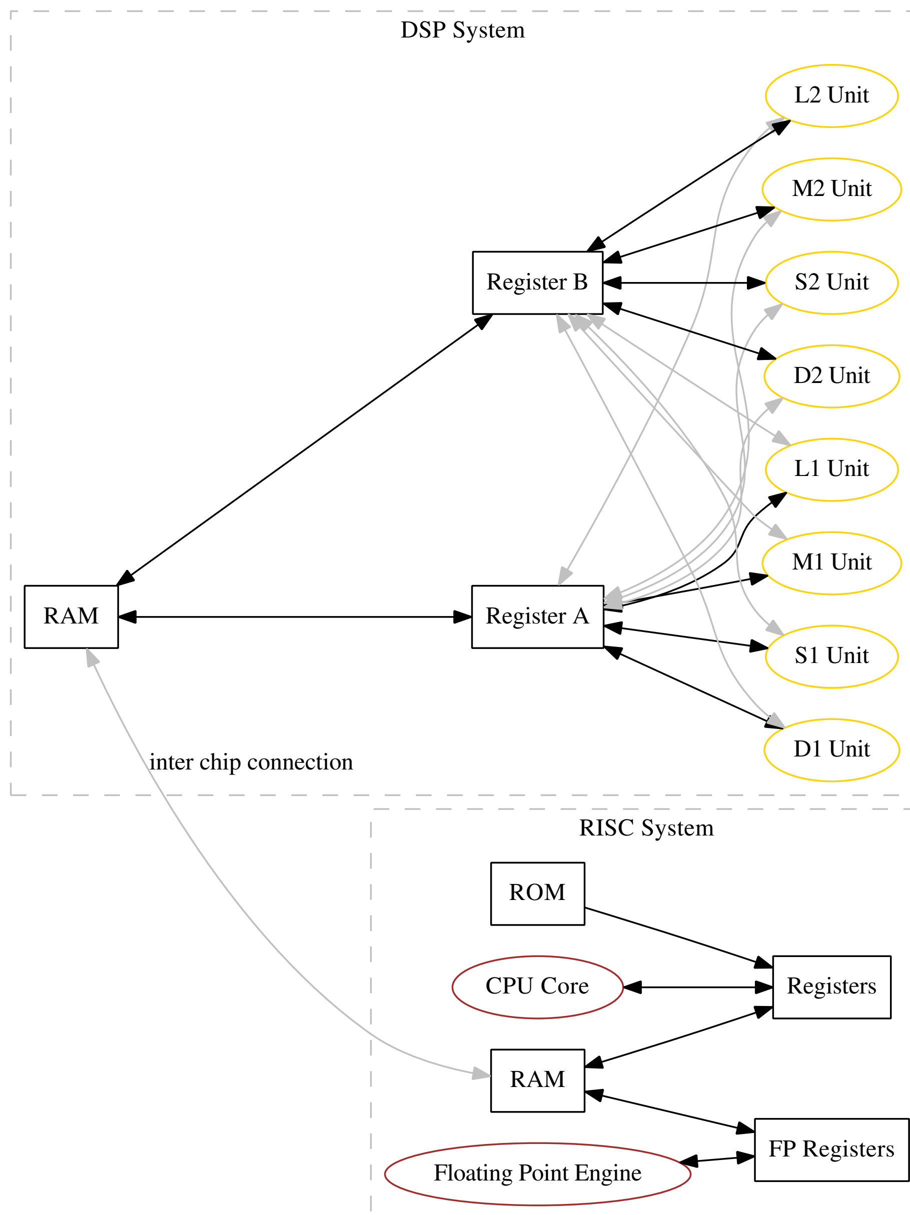


Figure 2.4: Multi chip hardware model [RB18a]

an edge whose metadata property indicates the involvement of another node during this transfer, the allocation of this node has to be taken into account. If the node is already busy at the time consumed by the desired transfer, the data transfer cannot happen. In such a case additional delays need to be introduced, to reflect the congestion of the required resources.

Consequently, each node needs to keep track on its allocation profile. Considering the parallel nature of a generic allocation algorithm, the assignments are not guaranteed to be monotonic in time. A node might as well be requested to contribute to an activity prior to its last recorded execution. Its allocation tracking needs to properly handle such scenarios.

Facing this set of constraints, with respect to transferring data to compute units, is a major issue in allocating a computation onto a distributed hardware. While the aspects change whether the hardware represents a sparse distribution (with mostly large edge weights) or a dense allocation (with lots of "cheap" connections), the fundamental problem of transfer allocation stays the same.

For the example given above there are multiple potential bottlenecks. For the RISC system the data transfer from `ROM` to `RAM` is likely to allocate the CPU node. As both paths would require the CPU node to be available during the transfer with such a constraint the paths `ROM` to `Registers` and `Registers` to `RAM` cannot be utilized simultaneously. Despite the independent edges retrieving some values from both of these to memories would demand sequential scheduling. This constraint cannot be extracted out of the hardware graph as it is depicted in Figure 2.4. This is simply because the metadata of the corresponding edges is not shown. Another constraint will materialize within the DSP unit and the data transfer to and from its local `RAM` storage.

On a very basic scale the distribution algorithm has to perform following tasks.

1. Select an operation from the CDFG which shall be "executed".
   Any operation within the graph whose dependencies (inputs) have already been computed (are available) can be selected.

2. Select a suitable compute-node to perform the operation.
   In order to be considered as suitable a node has to expose following properties.

   - It can in principle execute the desired operation.
   - All the inputs can be transferred to the compute-node.
     This mandates the have at least one existing path from their current storage-node to the compute-node in question.
   - The storage-nodes alongside this path have free storage slots to hold the variable in transit.
   - If the operation is generating a result (is "non-void"), there shall be a storage-node with a sufficient number of free slots, to hold the result value.

   The latter two properties might mandate an additional operation in the next step.

3. Optionally rearrange the stored variables in order to free space in the involved storage-nodes. Either by moving them to other nodes (spilling) or by overwriting them.

4. Estimate the total cost of the operation.
   This includes the execution time, the time required for all the data transfers, and possibly all delay slots introduced due to resource congestion.

```
1  ; Function Attrs: noinline nounwind uwtable
2  define linkonce_odr dso_local float @"??
       ↪ $TestFunction@M$0CA@@@YAMAEAY0CA@$$CBM@Z"([32 x float]*
       ↪ dereferenceable(128))
3  {
4    call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 928)
5    %2 = alloca %0, align 1
6    call void @Track_GEP_i64i64(i64 4077, i64 744, i64 584, i64 0,
         ↪ i64 32)
7    call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 584)
8    %3 = getelementptr inbounds [32 x float], [32 x float]* %0, i64
         ↪ 0, i64 32
9    call void @Track_GEP_i64i64(i64 4077, i64 744, i64 792, i64 0,
         ↪ i64 0)
10   call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 792)
11   %4 = getelementptr inbounds [32 x float], [32 x float]* %0, i64
         ↪ 0, i64 0
12   %5 = bitcast %0* %2 to i8*
13   call void @llvm.lifetime.start.p0i8(i64 1, i8* %5)
14   call void @Track_GEP_i64i32(i64 4077, i64 744, i64 8, i64 0, i32
         ↪  0)
15   call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64 8)
16   %6 = getelementptr inbounds %0, %0* %2, i64 0, i32 0
17   store i8 undef, i8* %6, align 1
18   br label %7
19
20 7:                                                ; preds = %7, %1
21   %8 = phi float [ %12, %7 ], [ 0.000000e+00, %1 ]
22   %9 = phi float* [ %13, %7 ], [ %4, %1 ]
23   call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 5320)
24   %10 = load float, float* %9, align 4
25   %11 = fmul float %10, %10
26   %12 = fadd float %11, %8
27   call void @Track_GEP_i64(i64 4077, i64 744, i64 6640, i64 1)
28   call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 6640)
29   %13 = getelementptr inbounds float, float* %9, i64 1
30   %14 = icmp eq float* %13, %3
31   br i1 %14, label %15, label %7
32
33 15:                                               ; preds = %7
34   call void @TrackBasicBlock_SpringBoard(i64 867, i64 744, i64
         ↪ 2744)
35   %16 = bitcast %0* %2 to i8*
36   call void @llvm.lifetime.end.p0i8(i64 1, i8* %16)
37   %17 = tail call float @sqrtf(float %12)
38   ret float %17
39 }
```

Listing 2.33: vector norm, intermediate representation

Declaration of Sources:
Chapter 2 was based on and reuses material from the following sources, previously published by the author:

[RB18a] Andreas Rechberger and Eugen Brenner. "Generalized Execution Time Estimation". In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–4.

[RB18b] Andreas Rechberger and Eugen Brenner. "Partitioning of Algorithms for Distributed Computation". In: *Embedded World 2018 - Proceedings*. Embedded World Conference, Mar. 2018.

References to these sources are not always made explicit.

Chapter 3

Results

3.1 Control and Data Flow Graphs

3.1.1 Simplified Microbenchmark

For a first evaluation a simple hardware model, according to Figure 2.3 is used. It reflects a simple RISC core with floating point capability. Two very basic routines are used in this initial evaluation. At first this involves adding a few numbers (Figure 2.2a, Equation (2.1)), and later the element wise MAC of two arrays (Listing 3.1, Equations (3.1) and (3.2)).

$$y = \sum_{i=0}^{N-1} c[i] \cdot x[i] \tag{3.1}$$

$$r = r + (b \cdot c) \tag{3.2}$$

$$y[n] = \sum_{i=0}^{N-1} c[i] \cdot x[i - N] \tag{3.3}$$

The MAC routine is the simplified form of the core routine of an finite impulse response (FIR) filter (Equation (3.3)). The input data is represented by argument `a`, while `b` would correspond to the filter coefficients. The first in first out (FIFO) nature of the input array, required to perform filtering of a complete input vector, is omitted in this example.

```
1 template<typename T, unsigned N>
2 T TestFunction(const T (& a)[N], const T (& b)[N])
3 {
4     T r = T(0);
5     for(auto idx=0;idx<N;++idx)
6         r += a[idx] * b[idx];
7     return r;
8 }
```

Listing 3.1: array multiply accumulate (simplified FIR core)

These routines are quite primitive, and although self-contained as such, are rather useless on their own. However, their expected runtime and compute profile is well known, which qualifies them for a first comparison.

In addition to the model based estimation a very basic model-less algorithm is used. The model-less algorithm does not reflect any data move operation, and simply counts the number of compute operations within the CDFG. By definition such an estimation can only produce reasonable results under a dedicated set of boundary conditions.

- The estimation is only valid for a hardware arrangement with a single compute-node.
- The compute nodes capability closely matches the instruction set of the LLVM assembly language.
- There is no significant limitation on the data access.

Except for the register pressure, and the need for spilling these assumptions hold for the model referenced in Figure 2.3. For the model based estimation some parametrization within the model has been applied. Almost all instructions within the intermediate representation are assumed to map to a single cycle instruction of the RISC core. Only a subset of the more complex floating point arithmetic functions (like for example the floating point square root (`VSQRT`)) is assigned a higher cycle cost. A compute-node will therefore require several cycles to server these instructions. The second adoption is on the data load and store operations. Locality of the data, and as such the assignment to registers is a dominant concern when mapping the intermediate representation to target specific assembly [ASU06, Chaper 11]. The spilling operations [FH92] will degrade the execution performance. Finding an optimal allocation of variables within the registers is an ongoing research topic. In addition, there are plenty of algorithms [AG01; QP08; Len12] available to perform this tasks. Any compiler is required to implement at least one of these algorithms, when generating the machine code. For this early study an overly simplified approach is used. We assign a fixed amount of cycles to be consumed by any spilling (respectively load/store) operation, rather than modelling the register allocation. This fixed cost and the number of registers are parametrized to generate a set of reference models. For the full parameter sweep a set of 66 models is used in this early evaluation. Their details are shown in Table 3.1. Most of the early estimation results have already been presented in a previous publication [RB18a], whereas in this work some more details on the interpretation are given.

For reference both algorithms have been executed on a hardware board [ST 18] with the CPU cycle count as figure of merit for the execution time. The build-in cycle counter of the ARM core was used to count the number of CPU cycles. Its data was manually extracted via an integrated development environment (IDE)[1]. As development environment VisualGDB [Vis20], with the GCC [GNU20] was used for compiling the code. For such small code snippets all recent compiler versions produce identical assembly output.

The results are shown in Figures 3.1 and 3.2. The execution time estimates in these figures are normalized to cycle count measured by the hardware reference prototype.

[1] `https://visualgdb.com/tutorials/arm/chronometer`

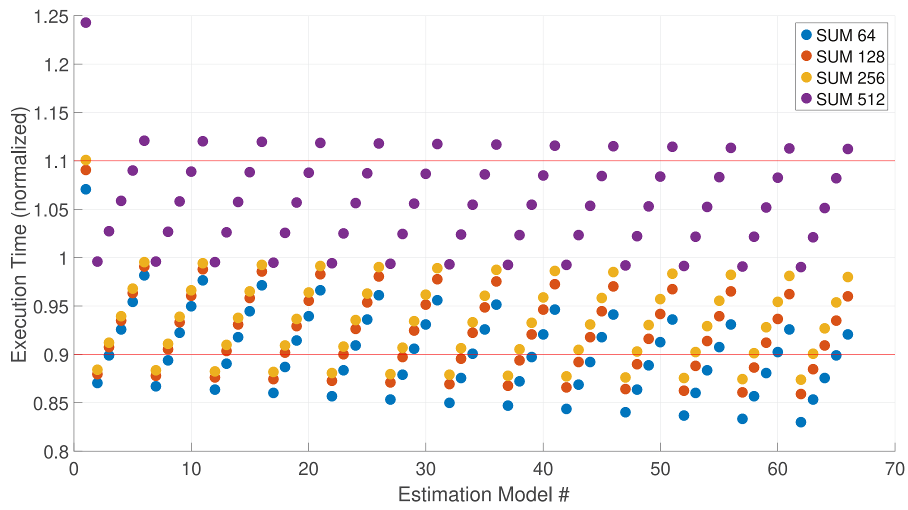


Figure 3.1: Normalized estimates vector sum ($\sum_i^N a_i$)[RB18a]

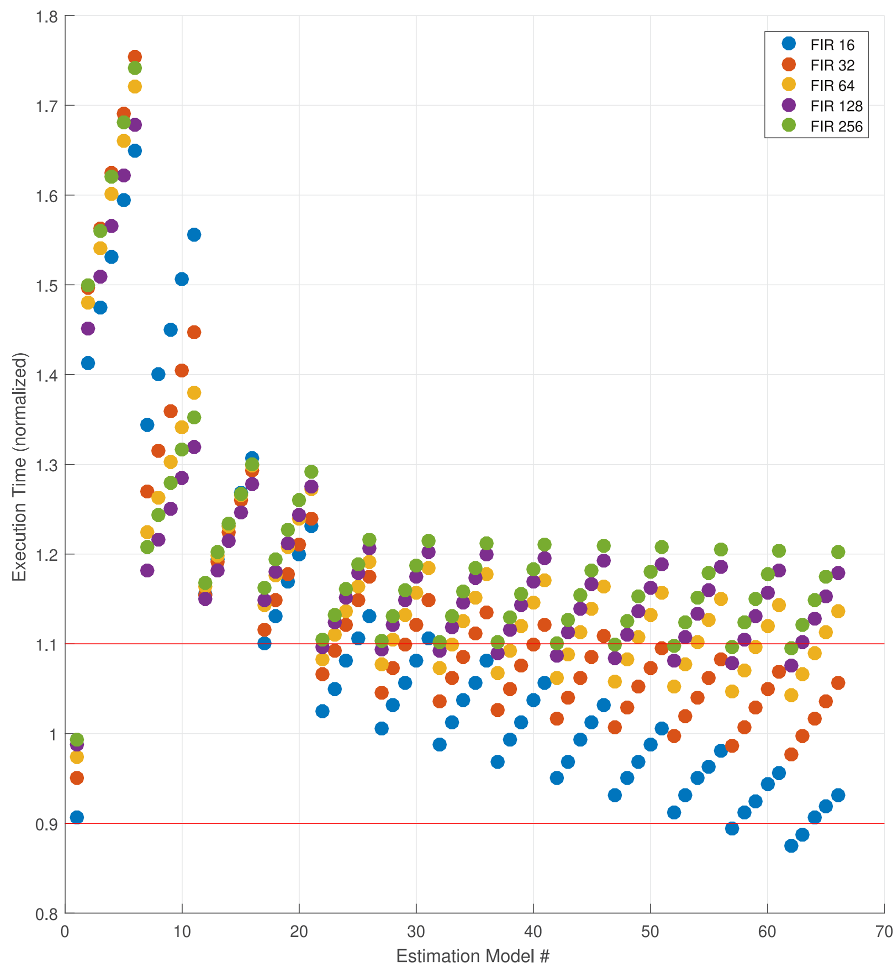


Figure 3.2: Normalized estimates FIR iteration ($\sum_i^N a_i b_i$)[RB18a]

Table 3.1: Early Evaluation Models

Model No.	No. of Registers	Spilling Cost
1	n/a	n/a
2	3	2.0
3	3	2.25
4	3	2.5
5	3	2.75
6	3	3.0
7	4	2.0
8	4	2.25
9	4	2.5
10	4	2.75
11	4	3.0
12	5	2.0
...		
65	15	2.75
66	15	3.0

For the algorithm performing a simple accumulation (Figure 2.2a) there is quite a good match between the estimate and the true value (real hardware model). Increasing the load store delay has a rather linear impact on the execution time estimation. For a simple fold operation this is expected, as its memory access profile is straight forward. Every element in the input array needs to be accessed exactly once, and the access is performed in strictly increasing indexing. With a model not reflecting any cache behaviour the access pattern, whether its continuous increasing, decreasing or random cannot impact the estimation. There are no cache lines to hit, or burst accesses to be modelled. There is also almost zero impact on the number of registers in the hardware model. As a rule of thumb a larger number of registers offers better performance, as the increase locality of the data and virtually act as cache. But for a basic `foldl` operation a total of three registers is already sufficient (loop counter, accumulator and zero flag). As such it is quite apparent that increasing the register count even further will not improve the performance, as the implementation is unable to utilize the additional registers.

Modelling on such a low level does reveal quite a large degree of hardware subtleties. The ARM Cortex-M4 for example has pre and post increment load operations. This allows the compiler to generate a four instruction loop for the fold core (`ldr, cmp, add, beq` as shown in Listing 3.2). The intermediate representation does not have this ability, and such is obliged to revert to a five instruction sequence (`load, add, add, cmp, br` as shown in Listing 3.3). For the second `add` instruction clang has decided to a particular LLVM assembly language idiom rather than adding a pointer (or index).

It uses a combination of a Φ-node together with a `getelementptr` instruction referencing the second element. The indexing starts at zero, hence index one references the second element. Subsequently, accessing the second element of an array (pointer) based on the previous second element advances through the

input array. An alternative representation would have been incrementing the index by one and using the increased index for the `getelementptr` instruction.

```
1  ;template<typename T, unsigned N> T TestFunction(const T (& a)[N])
2     4:   2300            movs    r3, #0
3  ;       __init = _GLIBCXX_MOVE_IF_20(__init) + *__first;
4     6:   f850 2b04       ldr.w   r2, [r0], #4
5  ;     for (; __first != __last; ++__first)
6     a:   4288            cmp     r0, r1
7  ;       __init = _GLIBCXX_MOVE_IF_20(__init) + *__first;
8     c:   4413            add     r3, r2
9  ;     for (; __first != __last; ++__first)
10    e:   d1fa            bne.n   6 <
         ↪ _Z12TestFunctionIiLj40EET_RAT0__KS0_+0x6>
11 ;{
12 ;  using namespace std;
13 ;  return accumulate(begin(a), end(a), T(0));
14 ;}
15   10:   4618            mov     r0, r3
16   12:   4770            bx      lr
```

Listing 3.2: `foldl` core, ARM Cortex-M4 assembly code

```
1  for.body.i.i:                  ; preds = %entry, %for.body.i.i
2    %_UFirst.010.i.i = phi i32* [ %incdec.ptr.i.i, %for.body.i.i ],
       ↪ [ %arraydecay.i, %entry ]
3    %_Val.addr.09.i.i = phi i32 [ %add.i.i.i, %for.body.i.i ], [ 0,
       ↪ %entry ]
4    %0 = load i32, i32* %_UFirst.010.i.i, align 4, !tbaa !8
5    %add.i.i.i = add nsw i32 %0, %_Val.addr.09.i.i
6    %incdec.ptr.i.i = getelementptr inbounds i32, i32* %_UFirst.010.
       ↪ i.i, i64 1
7    %cmp.i.i = icmp eq i32* %incdec.ptr.i.i, %add.ptr.i
8    br i1 %cmp.i.i, label %"??$accumulate@PEBHH@std@@YAHQEBH0H@Z.
       ↪ exit", label %for.body.i.i
```

Listing 3.3: `foldl` core, LLVM assembly language code

This single miss-modelling bias causes a constant error within each loop iteration, as otherwise both model and reference have an identical instruction set. This systematic estimation error is however "accidentally" compensated by the cost assigned to the load/store operation. The cost model assigns two cycles for a load operation, while the real hardware consumes only one cycle. As the load instruction is also executed once per loop iteration, these two errors compensate each other.

The FIR algorithm, in contrast to the `foldl` operation, experiences a strong dependency on the number of registers within the model. This is also a quite expected behaviour. The reference hardware (ARM Cortex-M4) has 16 general purpose registers, and while not all of them can be freely used (like the stack pointer or link register) an estimation is damned to be quite bad if restricting the hardware model to three registers. Even assigning very low spilling costs (like model index 2) cannot compensate for such a modelling flaw. A 40...50 %

increase compared to reality is experienced. As soon as reaching the critical limit of seven registers (index 22 and onwards) the estimation results flatten. This is again an expected behaviour. Both data vectors are initially located in FLASH, respectively random access memory (RAM) and the performance does no longer increase as soon as sufficient registers for the loop counter, memory pointers and accumulate variable are available.

The notable exception is the 16 tap FIR algorithm which continues to speed up with making further registers available. This is likely to be caused by the fact that the coefficient vector is small enough to fit almost entirely into the register bank.

In total the model has a slight bias, representing its tendency to estimate a bit too pessimistic. A plausible reason for this bias might be the fact that the model (and the LLVM assembly language) does not have a representation of a MAC instruction, while the reference hardware does. This enforces the estimator to run two consecutive operations (multiply and accumulate), while the reference does merge these two operations into a single one (MLA [ARM10]).

Summarizing for a quite simplified modelling approach $\pm 10\%$ accuracy can be achieved. To reach this accuracy some tuning of the model parameters is required. This however is a micro benchmarking comparison, and such a tuning has a reasonable potential to fulfil Knuth's "premature optimization is the root of all evil" [Knu74] claim.

3.1.2 Arithmetic benchmark

For a second benchmark various hardware arrangements are evaluated with respect to their performance for a matrix vector multiplication (Equation (3.4) and Listing 3.4). C style indexing (zero based) is used for these examples and equations.

$$r = \begin{pmatrix} m_{0,0} & m_{0,1} & \cdots & m_{0,n-1} \\ m_{1,0} & m_{1,1} & \cdots & m_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{m-1,0} & m_{m-1,1} & \cdots & m_{m-1,n-1} \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} \tag{3.4}$$

```
1  template<typename T, size_t N, size_t M>
2  void TestFunction(const T(&m)[M][N], const T(&v)[N], T(&r)[M])
3  {
4      for (size_t i = 0; i < M; i++)
5      {
6          T tmp = T(0);
7          for (size_t j = 0; j < N; j++)
8          {
9              tmp += m[i][j] * v[j];
10         }
11         r[i] = tmp;
12     }
13 }
```

Listing 3.4: Matrix Vector multiply, C++ code

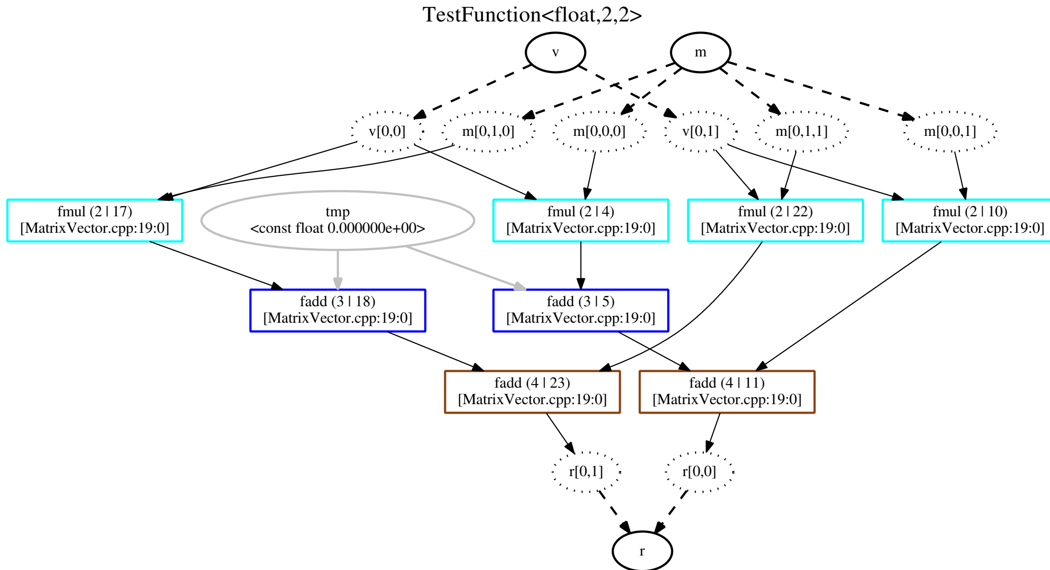


Figure 3.3: 2x2 matrix vector multiply, fully unrolled

The extracted CDFG for such a small program depends quite heavily on the optimizations performed by the compiler. Due to the fact that the matrix size is known at compile time, the optimizer is very likely to fully unroll the nested loop. Especially if the matrix size is very small. For the sake of the argument, and to still have a suitable sized graph for the visualization the CDFG of a 2x2 matrix is depicted in Figure 3.3. Each rectangular node in this picture indicates a compute operation which needs to be performed, while the elliptic nodes are used to represent variables. Dashed lines indicate the array slicing as described in Section 2.3.8. The "meta nodes" indicating the variables m, v and r serve only representational proposes. For the analysis only the individual array and matrix elements are of interest. The numbers given next to the operations name indicate the rank and linear execution count. The rank is an indicator for the ability to schedule the instruction with respect to others. A rank of zero indicates any operation whose dependencies are fulfilled at the beginning. As such they only depend on initially available input. A rank indicator of one indicates an operation whose depends on at least one rank zero operation. A rank indicator of two reflects an operation whose depends at least on one rank one statement and so forth. The linear execution count is the time stamp assigned to this operation when executing the program on a hardware setup with a single compute-node. Both numbers are only indicative and the automated scheduling is not required to oblige to them. Obviously all control structures, like branches, have been eliminated in this example, as both of the loops have been fully unrolled by the optimizer.

Figure 3.4 demonstrates the same code as shown in Figure 3.3 but with forcefully forbidding loop unrolling. This causes the loop overhead (counting and conditional branching) to survive the optimization steps, and as such be represented in the CDFG.

The hardware models used for this evaluation are composed of a general purpose compute-node, and a set of dedicated arithmetic cores, as exemplarily depicted in Figure 3.5. Some properties are common for all the hardware

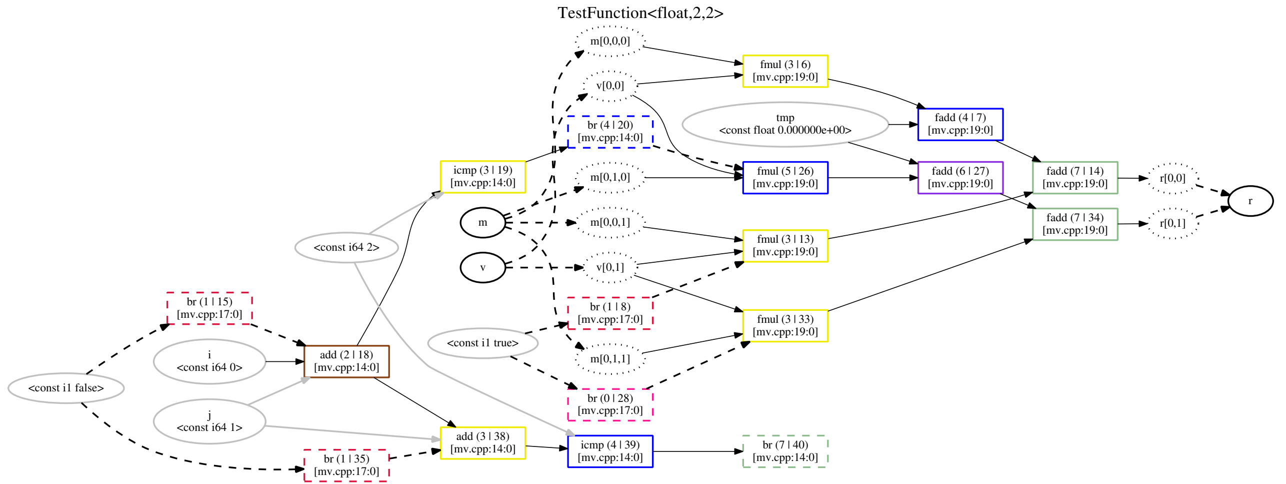


Figure 3.4: 2x2 matrix vector multiply, no unrolling

arrangements evaluated.

- All input data (the matrix `m` and the vector `v`) is initially located in the `RAM (main)` node.
- The complete output data (the vector `r`) shall finally be stored in the `RAM (main)` node.
- The general purpose compute core (CPU) is the only ode which can access this storage.
- The `ALU` nodes on the other hand are much more capable with respect to arithmetic processing. In particular the can perform floating point operations much faster than the general purpose node.
- Any data transfer over these edges requires utilization of a dedicated compute-node.

Obviously a balance has to be found to either transfer the data to the `ALU` nodes, perform a fast execution, and copy the data all the way back to the `RAM (main)`, or run the computation with higher cost on the general purpose core. The basic fact that increasing the amount of compute nodes will not linearly decrease the runtime is well known [Amd67; Gus88]. The problem however remains to identify the threshold at which additional compute units do not speed up the computation any more, or are only inadequately utilized. Answering this questions does not only depend on the computational profile of the algorithm, but also on the properties of the model. Most dominantly the transfer cost over the edges, or the ratio between the `CPU` floating point cost and the transfer time. The node allocation for such a transfer, and the parametrizable cost (`tacc`) are depicted in Figure 3.6. The model used demands the general purpose core (`CPU`) to transfer data to and from the main RAM (`RAM (main)`) and the ALU exchange node (`ARAM (shared)`). Each ALU in turn can access this node, and transfer its data to its individual register storage (e.g. `Regs #1`). The transfer cost is variable (`tacc`), but generally high. In order to prevent excessive spilling cost, a dedicated storage-node (like `RAM #1`)

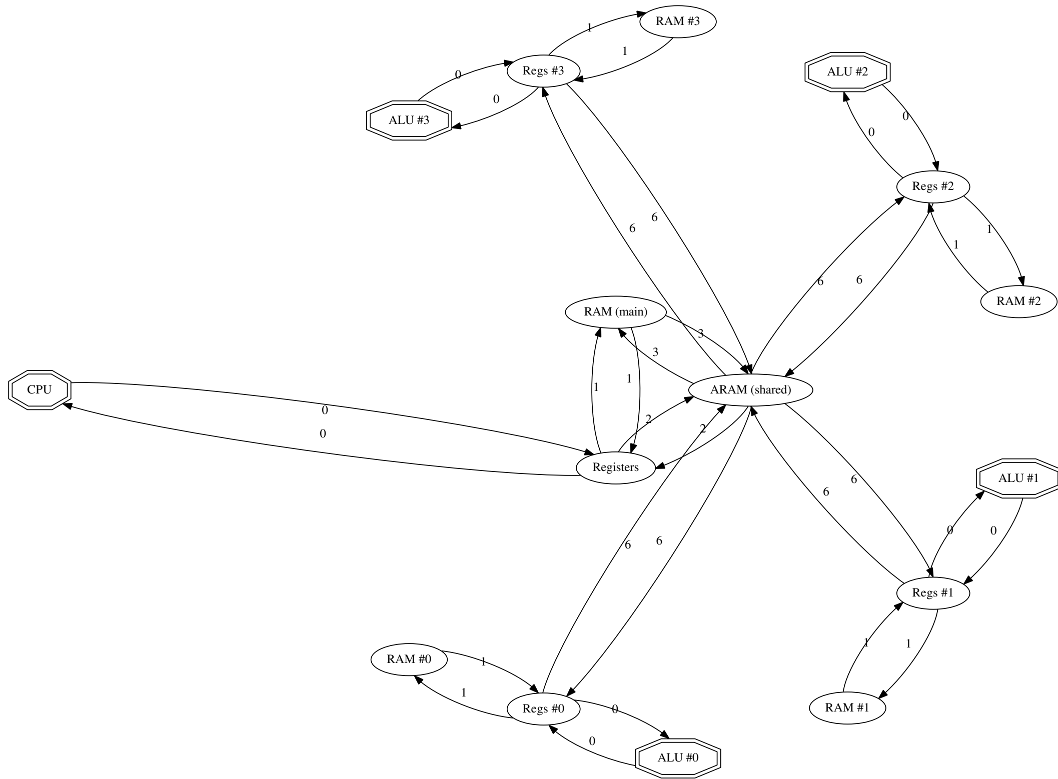


Figure 3.5: Multi core architecture with 4 ALU cores

is associated with each ALU. Figure 3.6 depicts the transfer realms in different colours. All ALU units can access the shared storage in parallel, but at high cost. With more ALUs this cost is likely to increase due to bus arbitration.

From an implementation point of view such a design can be implemented in a straight forward synchronous design, with a global clock. The shared RAM entity (`ARAM (shared)`) might be a dual ported memory. Using a dedicated port for the CPU side access offers a short access timing. The multi-user port for the various ALU cores requires several cycles to be accessed, likely to increase with extending the model with additional ALU units.

3.2 Manual Performance Estimation

A matrix vector multiply has a well known computational profile. Multiplying a $m \times n$ matrix with a vector requires exactly $m \cdot n \cdot n = m \cdot n^2$ MAC operations. Assuming all computation to be done by ALU nodes, the data transfer requirements are quite predicable as well. The matrix, as well as the vector has to be transferred to the ALU cores, hence into the shared memory. Once the result is complete the result vector `r` demands transfer into the main RAM. So a simple straight forward algorithm would use the general purpose CPU node to transfer the matrix `m` and vector `v` to the shared storage-node, wait for the ALU nodes to finish the computation, and copy the result back to the main RAM.

Following the notation given by Knuth [Knu76] its computational effort is given by Equation (3.5). All computations which are served by the CPU are

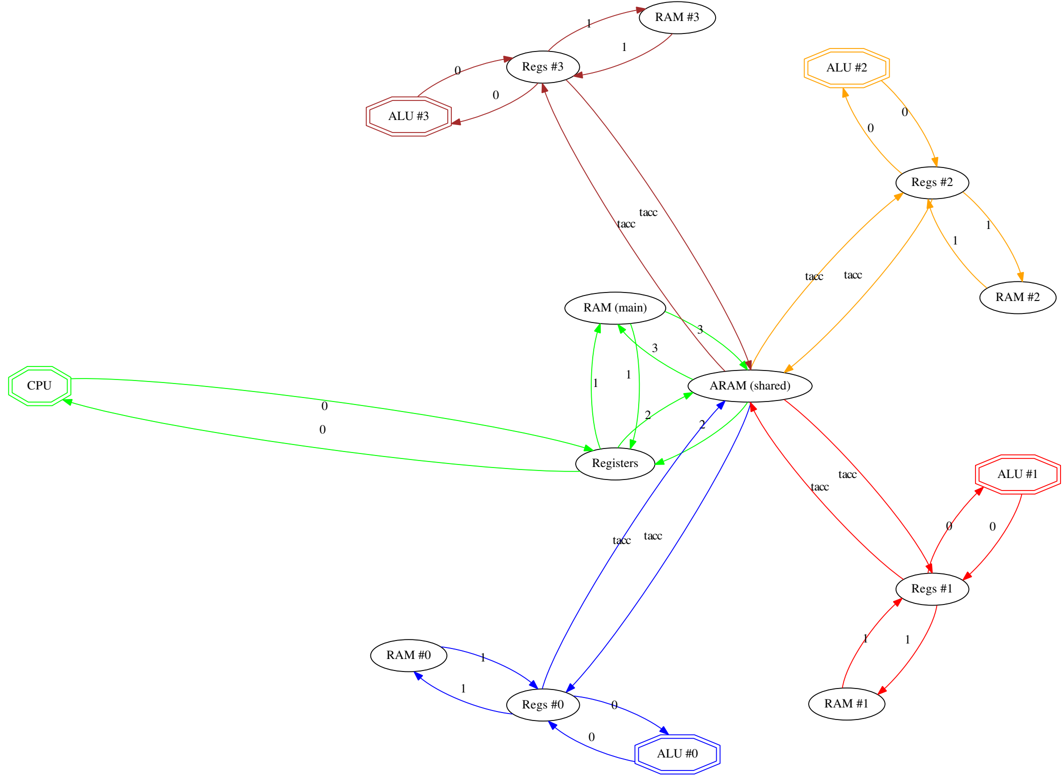


Figure 3.6: Multi core architecture, transfer node allocation

pure transfer operations, as in our scenario all mathematical operations are done by the ALU units.

$$f_{CPU}(n, m) = \Omega(m \cdot n + m + n) \tag{3.5}$$

$$m \cdot n \gg n \wedge m \cdot n \gg m \tag{3.6}$$

$$\hat{f}_{CPU}(n, m) = \Omega(m \cdot n) \tag{3.7}$$

The lower computation bound can be simplified with reasonable accuracy to Equation (3.7), when assuming m and n to be sufficiently large, such that Equation (3.6) holds.

For the corner cases $n = 1$, respectively $m = 1$ the computation degrades to a row vector column/vector operation, and Equation (3.5) collapses to Equation (3.7) and further to $f(m, 1) = \Omega(m)$ and $f(1, n) = \Omega(n)$.

With only a single node (the CPU) being responsible for these actions no option or distribution exists to fasten the computation. For the sake of simplifying this argument and without proof any housekeeping computations (like loop counters), done by the CPU are considered as not relevant for the result and ignored. Either due to the known matrix size the loops are unrolled by the compiler, or the loop overhead happens in parallel to the ALU movements. For the mathematical operations the manual prediction becomes a little more subtle. In order to provide suitable boundary conditions, following additional assumptions are made:

- The vector size n and the number of ALU (c) cores follow Equation (3.8),

that is the vector size is a multiple of the number of ALUs.

$$\exists i \in \mathbb{N}^+ : i \cdot c = v \tag{3.8}$$

- The RAM module attached close to the ALU cores are sufficiently large. In particular, they are large enough to hold the entire vector `v`, and all spilling data.

The following assumptions do not claim any formal guarantee on the optimal distribution, but for an intuitive estimation a few conclusions might be expected:

- The MAC operation for a single matrix line with the vector is performed entirely by a single ALU. This allows to conclude that no intermediate results are shared between the ALU nodes.
- Spilling is only performed into the dedicated RAM nodes close to the ALU. No spilling data is to be (expensively) transferred to the `ARAM (shared)` node.

Under these assumptions the intuitive estimation for a four ALU hardware with a two cycle access for `tacc` performing a 16x16 matrix vector multiplication boils down to following:
To transfer the matrix and the vector to the shared RAM accessible by compute-nodes (ALUs) $3(m \cdot n + n + m) = 864$ cycles are required.

Each ALU in turn will be in charge of 4 matrix rows. This will require $tacc \cdot n = 32$ cycles to fetch the vector data and $4 \cdot tacc \cdot m = 128$ cycles to fetch all the rows.

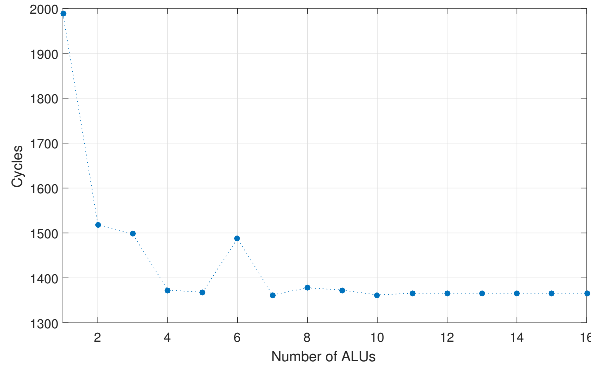
Assuming a single cycle performance for `fmul` and `fadd` we will end up in $(m \cdot 4) \cdot 2 = 128$ cycles for the computation itself. Because all ALU cores shall run in parallel, it is sufficient to consider the execution time of one core. For the result we need to transfer only one value per matrix line, hence another $4 \cdot tacc = 8$ cycles.

The CPU than will need another $3 \cdot 16$ cycles to transfer the data from the shared RAM to the main RAM.

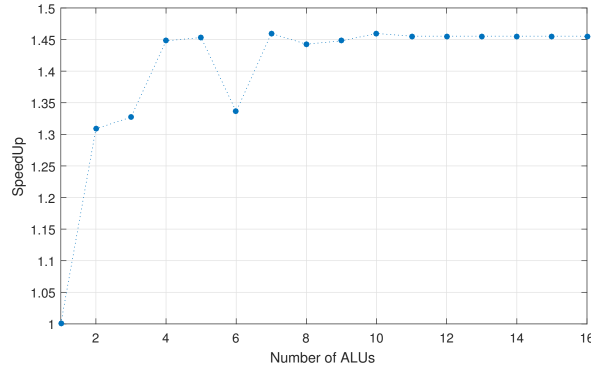
The crude estimate is $t_{CPU} = 864$ and $t_{ALU} = 160 + 128 + 128 + 8 + 16 = 440$ cycles. Assuming full parallelism the optimistically formulated lower bound is given by equation Equation (3.9) with C denoting some small constant to attribute for the delayed start of the ALUs. They cannot start until at least some required data has arrived at the node "`ARAM (shared)`".

$$t_{min} > \min(t_{CPU}, t_{ALU} + C) \tag{3.9}$$

However, the hardware model does only supply eight registers near the ALU core. This is insufficient to store the entire vector `v` within the register bank. Estimating the spilling cost is a tedious and error prone process.



(a) Required compute cycles



(b) Achieved speed up

Figure 3.7: Performance 16x16 Matrix Vector Multiplication

3.3 Automated Distribution Analysis

In order to eliminate the flaws in such a manual performance estimation an automated approach is desired. With expressing the hardware model as directed and weighted graph a generalized algorithm for the estimation can be used.

Based on the hardware model as depicted in Figure 3.6 the number of cycles and respectively the speed up to run the 16x16 matrix vector multiplication follows Figures 3.7a and 3.7b.

It is evident that the manual estimation was quite inaccurate, and far to optimistic. As it did not take the limited space in the register bank into account, this was expected.

In general the speed up shown in Figure 3.7b shows a habitual behaviour. A matrix vector multiplication is well suited for parallelization, and as such adding more ALU cores will improve the performance. However, the hardware is not able to suitably utilize more than four ALU cores in this scenario. As all data has to be prepared via the CPU core it materializes as bottleneck for the parallelization. This is especially true as the data transfer within the domain of the ALU cores is faster. With $tacc = 2$, they are faster with their memory access compared to the three cycles required for the CPU to access the shared RAM.

A graphical investigation of the distribution pattern as generated by the automated distribution algorithm is depicted in Figure 3.8. These figures demonstrate the activity of the computation hardware at the various steps within the computation cycle. A computation cycle is equivalent to a certain

fixed time slice. It is normalized to the assumed clock period of the fastest node within the arrangement.

Within these graphs an octagonal node represents a compute-node, that is one which is suited to serve instructions. A node might be capable of serving any possible instruction or just a subset. The number shown below the node identifier (a unique name) represents the number of instructions the node has already served.

A node shown with an elliptical shape represents a storage-node. It cannot process instructions but store a certain amount of values. Different to compute nodes the number printed below the nodes name does not represent the number of executed instructions, but the amount of variables stored within the node. While the intermediate representation is a strictly typed language the storage-nodes are not. For reasons of simplification the implementation only models their storage capacity via a number of value slots. There is no formal dependency between the number of bytes allocated within this memory, and the number of variables currently stored. A variable of type `double` consumes a single slot, as well does a variable which represents a single bit (`bool` in C++ or `i1` in LLVM assembly language).

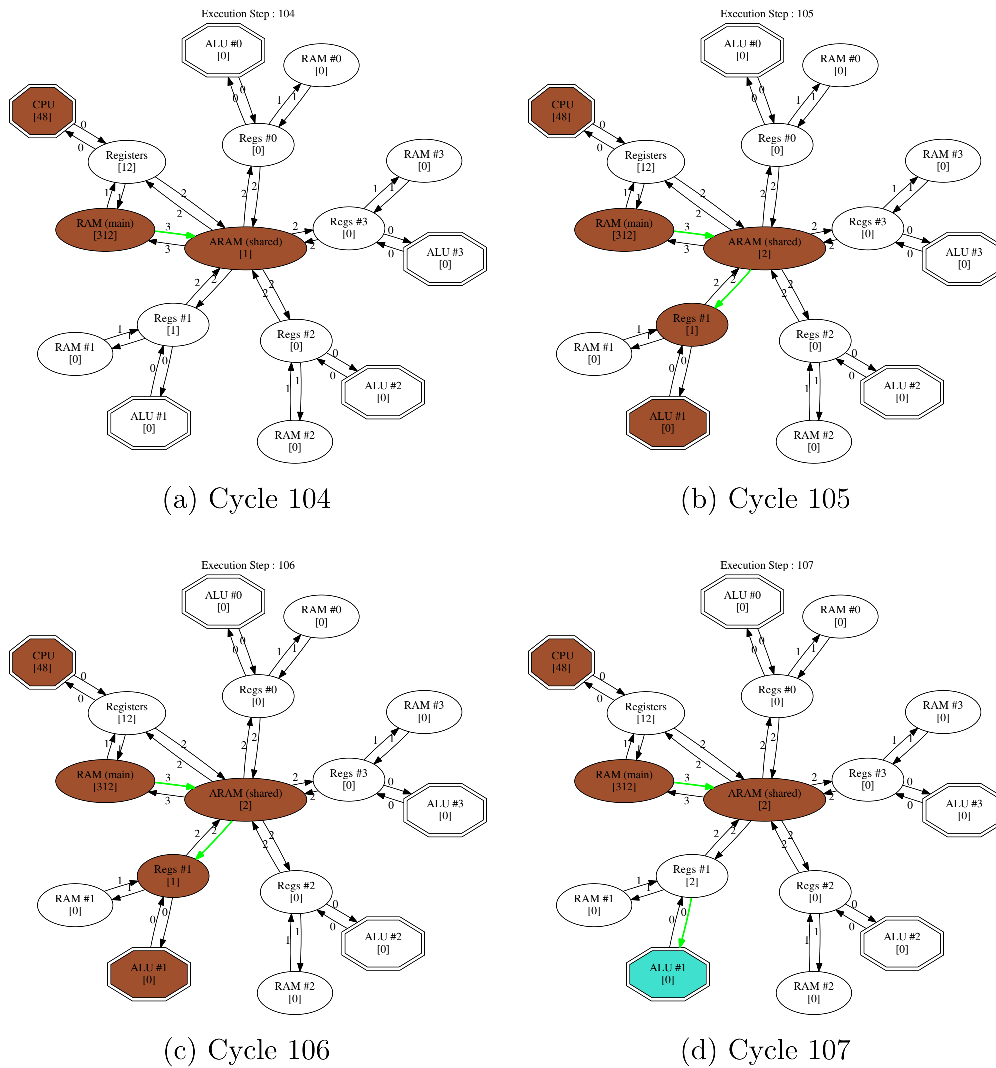
Coloured nodes and edges represent and activity, while black nodes and connections are inactive at the specific time point. Nodes coloured in brown (colour code `#A0522D` "sienna" [Fe01]) declare an active data transfer. This involves the originating storage-node, as well as the target storage-node. Optionally a data transfer might also occupy a computational node (like the CPU). In such a case the corresponding node is coloured as well.

Occupying a dedicated compute-node for data transfer operations is only relevant for the nodes allocation and availability for other actions. It does not increment the counter monitoring the number of instructions served by this node. The edges allocated by the transfer are marked in green (colour code `#00FF00` [Fe01]).

Compute nodes whose are filled with cyan (colour code `#40E0D0` "turquoise" [Fe01]) are currently serving an instruction.

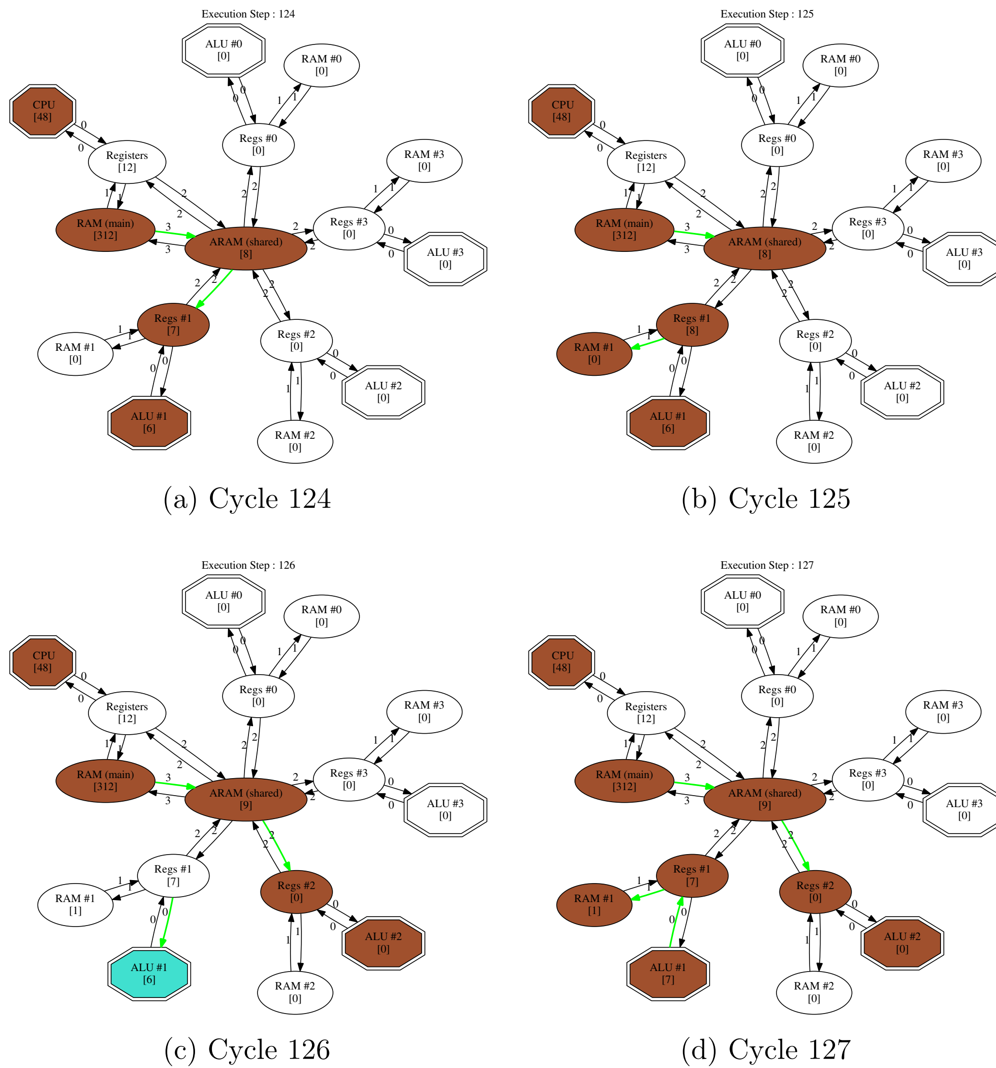
The cycle 104 (Figure 3.8a) represents the cycle at which the CPU node is finalizing the transfer of the second operand of the first `fmul` operation (`v[15]`). As it can be seen by the variable counter in `Regs #1` the first operand (`m[15,15]`) has already been placed in the desired target node in previous computation cycles. Immediately after finalizing this transfer (Figure 3.8b) the operand is rerouted to the `Regs #1` node in order to fulfil the needs of running the `fmul` operation on node `ALU #1` in the computation cycles to follow. In parallel the CPU continues to load data into the `ARAM (shared)` node in order to prepare for future arithmetic operations on the floating point capable nodes `ALU #0` to `ALU #3`. Once the two required variables have been transferred to the desired storage-node (`Regs #1`) the attached computation (`ALU #1`) node executes the instruction, fetching both operands from `Regs #1` (Figure 3.8d). In this case this is the only place suited for the operands.

Figure 3.9 depicts the computational sequences 124 to 127. In cycle 125 (Figure 3.9b) the register pressure on `Regs #1` becomes too high. As there is no more space available in the register bank to store the result of the instruction, there is need of moving one of the intermediate values to an alternate location.



(a) Cycle 104
(b) Cycle 105
(c) Cycle 106
(d) Cycle 107

Figure 3.8: 16x16 matrix vector multiplication, computation cycle 104...107



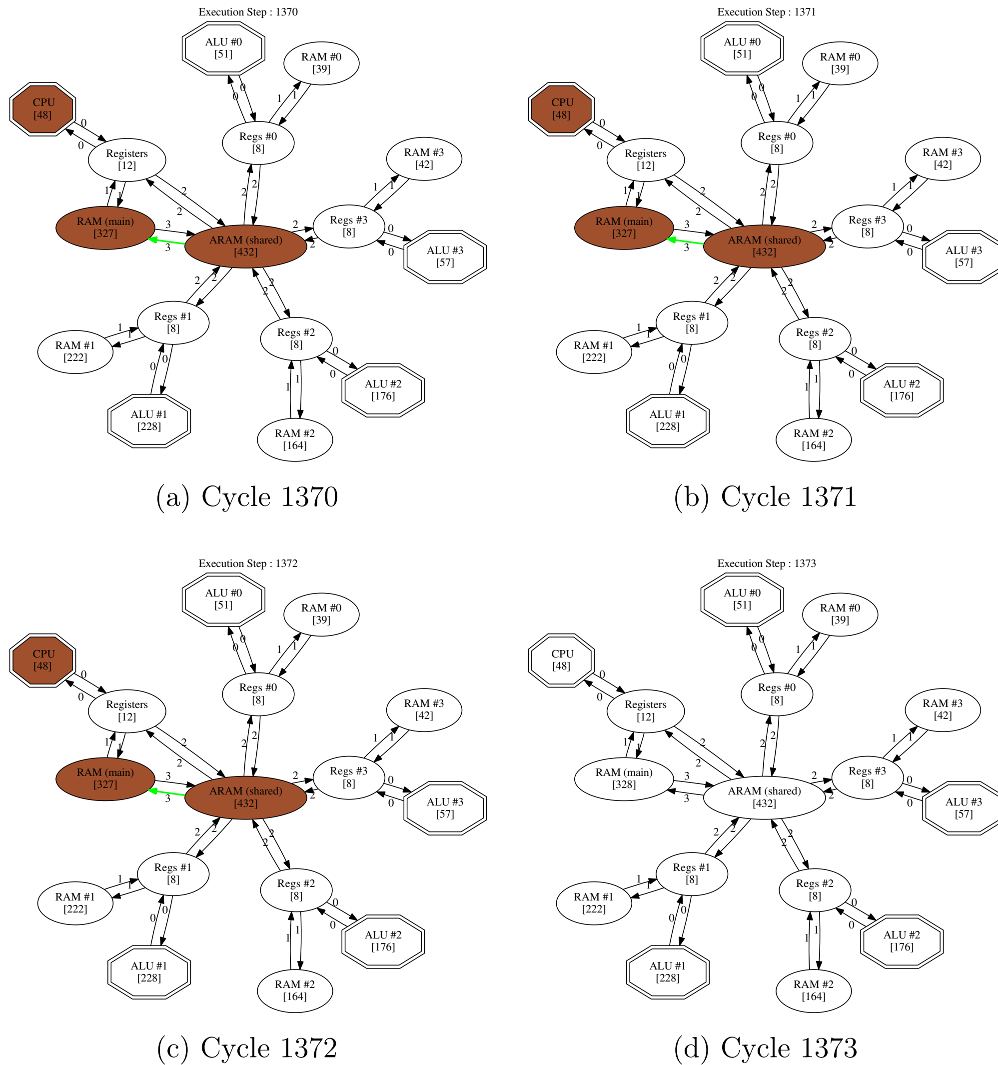
(a) Cycle 124

(b) Cycle 125

(c) Cycle 126

(d) Cycle 127

Figure 3.9: 16x16 matrix vector multiplication, computation cycle 124...127



(a) Cycle 1370

(b) Cycle 1371

(c) Cycle 1372

(d) Cycle 1373

Figure 3.10: 16x16 matrix vector multiplication, computation cycle 1370...1373

This is called spilling [FH92]. For the depicted hardware structure the `RAM #1` node is the most appropriate node to do so. It has the lowest cost to store and restore a value.

In parallel the `ALU #2` (Figures 3.9c and 3.9d) node is in the stage of becoming ready for processing its first instruction, by loading the required operands.

The final steps of the computation sequence are shown in Figure 3.10. The last data is transferred to the `RAM (main)` node to fulfil the finalization criteria as stated in Section 3.1.2. From the node counters it is apparent that not all the compute-nodes have been utilized to an equal amount. As the data transfer to the floating point capable nodes, via the `ARM (shared)` nodes acts as bottleneck, this hardware arrangement cannot provide a sufficient amount to data bandwidth to fully occupy all computation nodes.

Multiplying the 16x16 matrix with an 16x1 vector requires 256 MAC operations. As the `ALU #x` nodes do not support this as a single instruction, its has to be emulated by a sequence of two operations (`fmul`, `fadd`). As such the compute-nodes (`ALU #0`...`ALU #15`) are expected to serve 512 instructions.

Figure 3.11b depicts the total number of instructions executed by the CPU and ALU cores of the different hardware model variants. For this test a set of 16 hardware models has been used, equipped with respectively one to sixteen ALU cores according to Figure 3.12. Despite the fact that the general purpose computation node (CPU) is highly unsuited to perform such floating point operations a few of them are assigned to it. This is simply an indication that the data transfer rate to the computation nodes is faster than their computational capability.

The CPU node can push a new value into the ARAM (shared) node every third cycle. The computation nodes can extract those values every second cycle, hence faster. But at least a third cycle is consumed by the operation itself. Including the spilling efforts this allows the CPU to expensively pre-process the data prior to moving it to the ARAM (shared).

In the example given in Figure 3.11 the computational performance difference is 80:1. That is the CPU node requires 80 cycles to perform a floating point operation, while the dedicated floating point ALUs can server such an instruction in a single cycle. As long as there are less that three floating point nodes within the hardware arrangement, the data transfer bandwidth outperforms the computation capability. The CPU can push more data to the ALU nodes that those are capable of processing.

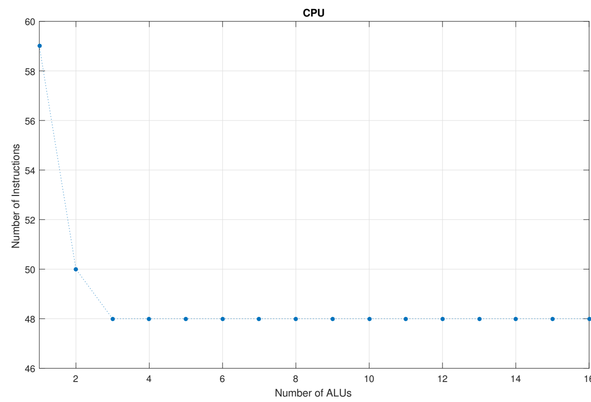
With at least three floating point units the processing capability exceeds the threshold, and using the slow CPU to run some of the fmul instructions is no longer beneficial.

For the hardware models (Figure 3.12) used in this example all floating point nodes are equal. Each of which has the same memory bandwidth to the shared memory node, as well as operates as single cycle compute-node. Figures 3.12a to 3.12d and 3.13 depict the node allocation exactly as chosen by the automated distribution algorithm. On Figure 3.12b for example it can be seen that ALU #1 (351 instructions) is used more heavily than ALU #0 (only 159 instructions).

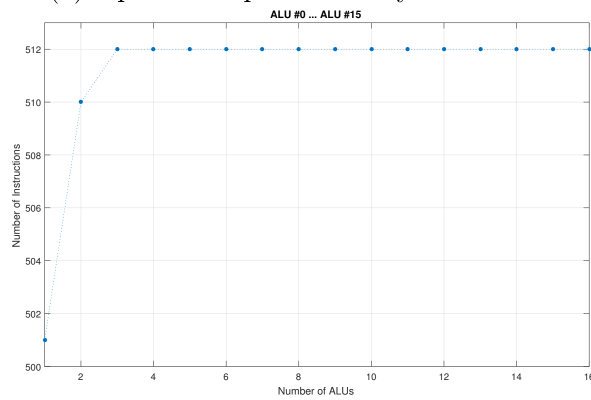
Figure 3.13 provides a clear evidence that the computational load is not evenly distributed among the available ALU cores. This visualization however is not suited for a proper demonstration of the distribution aspect. For reasons given above for this hardware arrangement we can arbitrarily reshuffle their numbering after allocating the instructions, without influencing the result.

Similar to Figure 3.13, Figure 3.14 depicts the amount of operations performed by the various nodes. For a better visualization of this fact however, the numbering of the ALU nodes has been rearranged in this graph. In Figure 3.14 ALU #0 always represents the floating point node with the highest number of instructions executed, and subsequent numbering is used for ALU nodes with decreasing allocation. ALU #15 will therefore always represent the least used node. This depicts with much more clarity that the allocation of added ALU units is quite low. Starting with one ALU the distribution algorithm will obviously allocate all of the instructions not handled by the CPU to it. Adding another unit will achieve a 70:30 split, hence the second unit will only take care of one third of the instructions, or computed the other way round do only half as much work as the primary node.

Even if an infinite amount of floating point compute-nodes would be added,

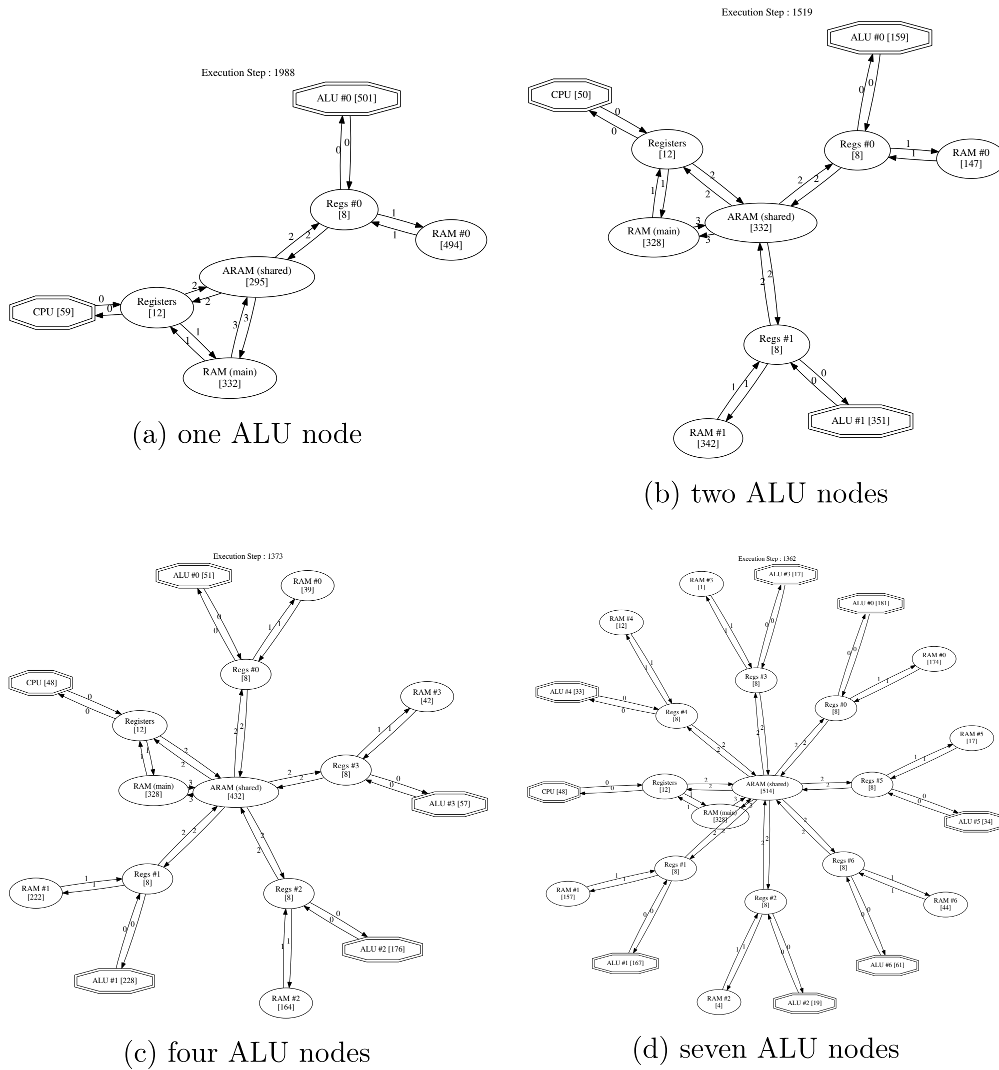


(a) Operations performed by CPU node



(b) Operations performed by ALU node

Figure 3.11: 16x16 matrix vector multiplication, instruction distribution



(a) one ALU node

(b) two ALU nodes

(c) four ALU nodes

(d) seven ALU nodes

Figure 3.12: Different HW models, instruction distribution

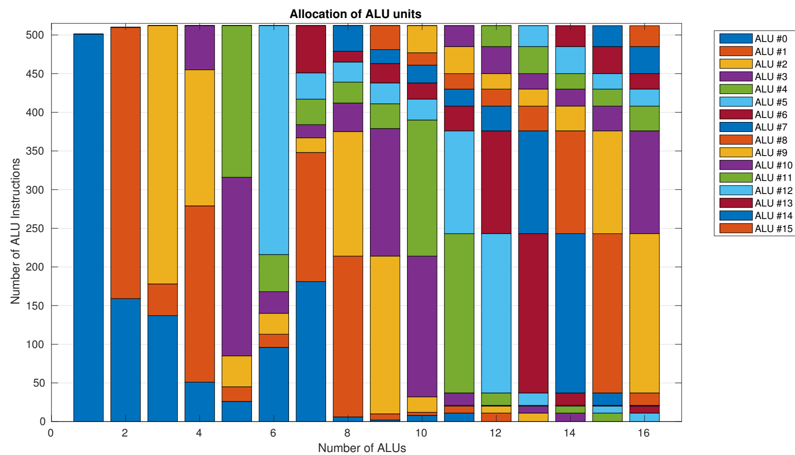


Figure 3.13: ALU Allocation (absolute)

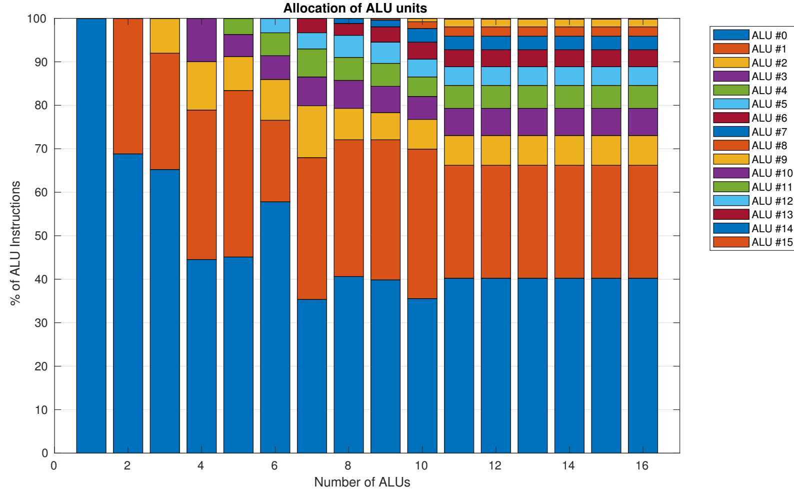


Figure 3.14: ALU Allocation (adapted numbering)

almost 70 percent of the workload will be served by only two units. Even if compute performance (speed up) is more important than hardware efficiency, respectively allocation, for this arrangement (Figure 3.12) four compute units are sufficient (Figure 3.7b).

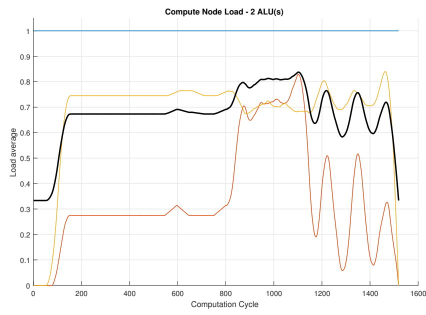
In addition to the overall utilization, the allocation of the nodes as function of the time (or cycle count) is a relevant metric for the efficiency of the hardware, when executing a dedicated algorithm. As all the previous figures this metric is highly dependent on the computational and memory profile of the algorithm.

Figure 3.15 depicts the normalized load of each compute-node within the hardware models with two (Figure 3.15a) to seven (Figure 3.15f) ALU units. Within a single cycle a node can only be busy (hence 100 percent loaded), or idle (0 percent load). The load average is computed by post processing this impulse train with a zero phase moving average filter ([OS09]).

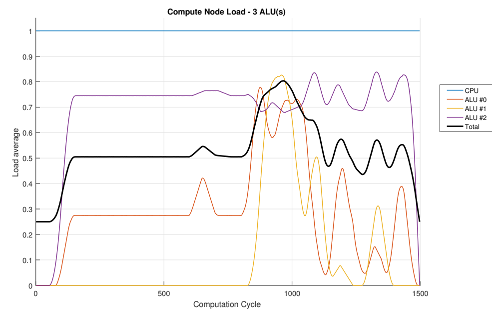
Due to the nature of the hardware arrangement only the CPU node can begin to operate from the very beginning, the ALU nodes are obliged to idle until at least some data transfer to the `ARAM (shared)` has finished. For the initial data transfer, the parameters (the matrix m, and the vector v) need to be transferred to the ALU nodes. After the computation the result vector has to be stored in the `RAM (main)` node. During this phase the CPU is the primary node in charge, and limits the overall allocation. This is visible in the black line of figure 3.15, showing a drop at the beginning and the end of the sequence.

Table 3.2 shows the overall allocation for the entire algorithm. In Equation (3.10) $l(i)$ denotes the binary load indication, hence the unfiltered raw data of Figure 3.15, t_{max} is the runtime of the sequence for this particular hardware platform.

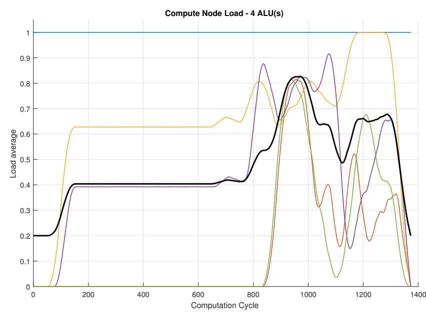
$$\hat{l} = \sum_{i=0}^{t_{max}} \frac{l(i)}{t_{max}} \tag{3.10}$$



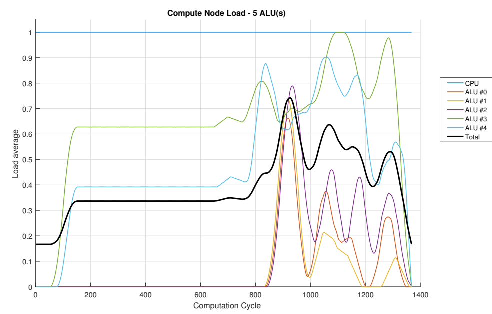
(a) two ALU nodes



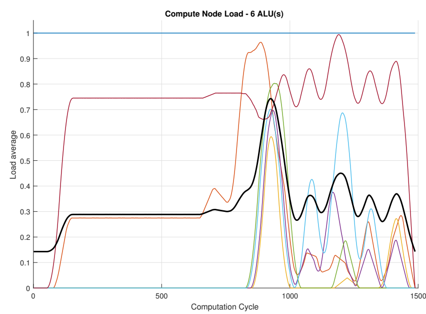
(b) three ALU nodes



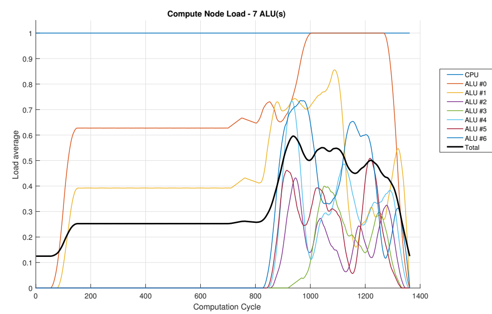
(c) four ALU nodes



(d) five ALU nodes



(e) six ALU nodes



(f) seven ALU nodes

Figure 3.15: Load average (100 cycles window)

Table 3.2: Total Allocation

No of ALUs	**System Load, $\hat{l}$**
1	0.8468
2	0.6743
3	0.5254
4	0.4811
5	0.3984
6	0.3251
7	0.3253
8	0.2831
9	0.2596
10	0.2390
11	0.2164
12	0.1997
13	0.1855
14	0.1731
15	0.1623
16	0.1527

3.3.1 Anomalies

Although it is expected that adding additional compute nodes will not always increase the performance, we expect from a suitable distribution algorithm to at least not degrade performance with additional compute power. For the scenario considered in the previous section this can be formally expressed. Figure 3.12 depicts some hardware models. Equation (3.11) denotes the graph representing a model with i ALU nodes. V_i is the set of vertices (compute or storage-nodes), while E_i being the set of vertices and edges. As additional ALU nodes will only extend the graph we can rely on the fact that Equation (3.12) holds.

$$HW_i = G(V_i, E_i) \tag{3.11}$$

$$G(V_{i+1}, E_{i+1}) \subset G(V_i, E_i) \tag{3.12}$$

$$S(G_{i+1}) \leq S(G_i) \tag{3.13}$$

Following this it is trivial to conclude that any valid solution for the distribution problem for graph G_i is also valid for graph $G_j \forall j > i$. This can be simply achieved by not using the new vertices and edges introduced by G_j. Under the assumption that newly introduced nodes and edges shall improve the performance metric of the system it is unlikely the optimal solution, but it proves that G_j cannot perform worse than G_i with an optimal distribution algorithm (Equation (3.13)). The speed up S is therefore bound to be monotonically increasing (Equation (3.13)). A distribution algorithm which fails to deliver this performance metric is considered as non-optimal.

As the goal of a distribution algorithm as adumbrated in Section 2.4.2 is to find an optimal distribution, it seems obvious to expect such a behaviour. Newly added resources whose fail to contribute to a better result shall simply

not be used. However, Figure 3.7 clearly indicates the contrary. The hardware equipped with six ALU nodes (G_6) performs worse than its subsets G_5 and G_4, violating the assumption above.

A careful comparison on the node allocation of the G_5 and G_6 case yields a difference in the first step of the algorithm, the selection of the next instruction from the set of "ready" instructions. An instruction is considered "ready" if all its inputs have been computed, hence its preconditions are fulfilled. The description on 2.4.2 ensures that for an selected instruction the optimal node is selected. It however does not make any claim how to choose the next instruction for scheduling in case the ready set contains more than one operation.

Non determinism

While not immediately obvious, the implementation of the distribution algorithm described above has a nondeterministic property. Listing 3.5 depicts the relevant code snippets. The `get_next_operation` function implements a deterministic operation retrieval, based on the first element in the `std::set`. A `set` is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. [ISO98; ISO11; ISO14; ISO17a]. The graph itself uses `boost::listS` as container to represent the vertex list in its internal structures. With such a setting the type `VertexId` is of type pointer to `OperationProperties_t` [SLL01]. This obliges the `std::set` to sort the elements based on the address in the memory where the elements are stored. For this implementation all of them are allocated on the heap.

```
1  namespace DFG
2  {
3   struct OperationProperties_t;
4   typedef boost::adjacency_list<boost::listS, boost::listS, boost::
        ↪ bidirectionalS, OperationProperties_t, DataProperties_t,
        ↪ GraphProperties_t> OperationsMap_t;
5
6   typedef OperationsMap_t::vertex_descriptor VertexId;
7  }
8
9  std::set<DFG::VertexId> m_ready_to_compute_nodes;
10
11 virtual std::optional<DFG::VertexId> get_next_operation() const
       ↪ noexcept
12 {
13     if (all_nodes_computed())
14     {
15         return {};
16     }
17     return *m_ready_to_compute_nodes.begin();
18 }
```

Listing 3.5: Selection Logic for next operation

Basically all modern operating systems implement some sort of address space layout randomization (ASLR) as part of their security functionality

[Sno+13]. While the effectiveness varies on the nature of the attack [Sha+04; Mee10; EPA16] the effect in our case is that the sort order of the container holding the "ready to compute operations" is effectively random.

Depending on the extracted CDFG, the retrieval order will influence the performance of the distribution algorithm. The latter claim can easily be demonstrated by following the example in Listing 3.1. The set of "ready to compute" operations will initially contain all the multiplication instructions. Servicing each of this instruction potentially adds a new operation to the set, the `fadd` instruction of the adder tree. If the selection strategy would now be FIFO based the distribution analysis will first schedule all multiply operations prior to any `add` operation. This basically creates a temporary array of size `N`, which in a second step will be computed identically to the `foldl` example given in Figure 2.2a, Equation (2.1). If on the other hand any newly available `fadd` operation is scheduled immediately for execution, hence following a LIFO scheme, only a single temporary has to be stored, the accumulator `r`. It is trivial to conclude that on a system with a single compute-node the latter approach does not generate any spilling demand, and by this will outperform the FIFO strategy.

Next to the random approach described above there are several strategies possible to select the next instruction. A non-exhaustive enumeration includes:

- Select the next operation based on the depth first search (DFS) [Tar72; Sed92] principle.
  This tends to increase the locality of the data, and reduce the register pressure. As the next operation chosen will at least depend on the output of the current one spilling is less likely.

- A selection based on the breadth first search (BFS) approach [Sed92; Hol99].
  This way favours to keep the liveness of intermediate data short, by prioritizing scheduling operations which "use" already available data over the ones which utilize recently created one.

- Following a LIFO approach.
  This expresses the desire to process operations which became recently "ready" as fast as possible. Again the basic idea in mind is to reduce spilling efforts.

None of the approaches can guarantee an optimal result for a generalized algorithm on a generalized graph. Using the DFS or LIFO approach might be more beneficial for the matrix multiply algorithm. The primary argument for this is the fact that this approaches will implicitly fuse the multiply and accumulate operation and as such avoid the spilling of multiplication results. A full featured liveness analysis however is a topic whose would require more sophisticated algorithms [ASU06].

3.3.2 Randomization

With not having a generically suitable strategy for instruction selection this optimization can also be done with Monte Carlo methods [Eck87]. This

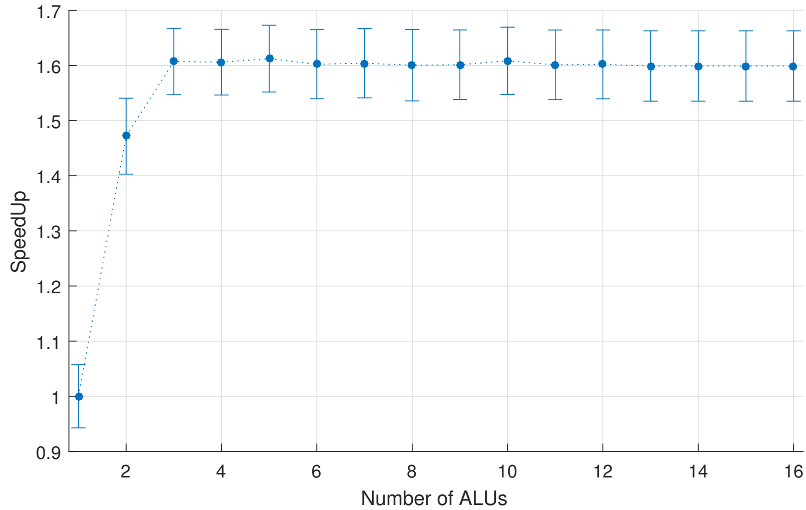


Figure 3.16: SpeedUp (Monte Carlo Method)

approach basically means selecting the next instruction randomly from the pool of ready instructions, and repeat the entire simulation reasonably often. By this the unknown but presumably deterministic behaviour of the system is revealed.

Figure 3.16 and Table 3.3 depict the results of such a Monte Carlo run. For this picture 49 simulations have been performed. With a reasonable confidence it can be concluded that more than three ALU nodes do not provide any significant benefit. Also, the performance increase with going from the G_2 model to the G_3 model is already quite small with an increase of less than 10 percent. This is a slightly different picture than that of a single run (Figure 3.7b) which claims the maximum speed up to be achieved with four or more floating point units (G_4). Also, the average speed up has increased from 1.45 to 1.61 in the Monte Carlo run. All models have a relatively stable deviation of the achieved speed up of approximately 0.063.

Repeating the Monte Carlo experiment with another 34 simulations yields a very similar picture. The maximum speed up becomes flat from G_3 onwards, but at a slightly lower level ($\approx 1.53 \pm 0.075$, rather than $\approx 1.6 \pm 0.063$). In absolute terms the second run was slightly worse in performance. On average, it required 38 cycles more to run the entire test sequence. Presumably making only a small number of 34 runs for the Monte Carlo trial is insufficient for an optimal result.

In general, it can be observed that running performance evaluations on such small test programs is susceptible to effects of micro benchmarking. Transferring the result of such a small scale benchmarking to a larger algorithm should only be done with caution.

3.3.3 A Heterogeneous Example

As a second example a less homogeneous hardware is used. Its basic architecture is similar to the one presented in the Equation (3.11) and Figure 3.12 of the previous section, but this time the ALU units do not have equal capabilities. Their access performance to the `ARAM (shared)` node as well as the computation

Table 3.3: Average Speed Up

No of ALUs	Speed up (S)	σ_S
1	1.0000	0.1043
2	1.3716	0.1092
3	1.5250	0.0689
4	1.5333	0.0689
5	1.5248	0.0699
6	1.5282	0.0697
7	1.5305	0.0708
8	1.5250	0.0716
9	1.5277	0.0696
10	1.5238	0.0702
11	1.5204	0.0707
12	1.5205	0.0706
13	1.5183	0.0711
14	1.5183	0.0711
15	1.5183	0.0711
16	1.5183	0.0711

time of their instructions have been altered. The access time for all the models $\hat{G}$ are adjusted according to Equations (3.14) and (3.15). The number of cycles to process a floating point instruction is given by Equation (3.16). In total again sixteen models ($\hat{G}_1 ... \hat{G}_{16}$) are used.

$$t_{acc_i} = C - i \tag{3.14}$$

$$C = t_{acc} + i_{max} \tag{3.15}$$

$$t_{fp_i} = i + 1 \tag{3.16}$$

This effectively renders a hardware model whose employs capable compute nodes, but connects them inefficiently. As an example the $\hat{G}_4$ model as depicted in Figure 3.17 has its most powerful compute unit (`ALU #0`) connected via a four cycle transfer delay to its primary data source (`ARAM (shared)`). It can perform any floating point operation within two cycles, but when including the fetching delay of a single parameter this results in a 6 cycle demand for servicing the request.

With the exception of the `CPU` node, the `ALU #3` and its compute time of five cycles is the slowest node in this arrangement. However, due to its fast access time of a single cycle, its single parameter fetch and compute timing is identical to the one of `ALU #0`.

The speed up achieved by the various hardware models, compared to a single ALU model is shown in Figure 3.18. For these models ($\hat{G}_i$) an argument similar to Equation (3.12) no longer holds due to the condition given by Equation (3.17). As such, Equation (3.18) no longer mandates the speed up (S) to be monotonically increasing.

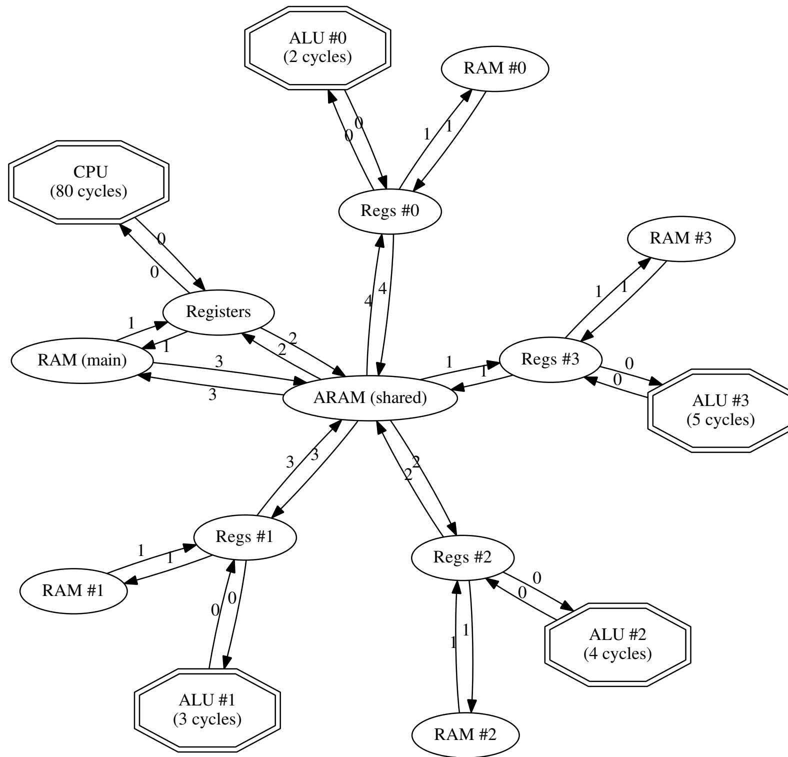


Figure 3.17: $\hat{G}_4$ Hardware Model

$$\hat{G}_i \nsubseteq \hat{G}_{i+1} \tag{3.17}$$

$$S(\hat{G}_{i+1}) \ngeq S(\hat{G}_i) \tag{3.18}$$

Figure 3.19 depicts the allocation of the individual nodes for the models $\hat{G}_1$ (Figure 3.19a) to $\hat{G}_6$ (Figure 3.19f). With more compute nodes the load average starts to oscillate. This is true for the overall load, as well as for the load metric of the individual nodes. This is an indication for the presence of bottlenecks. The particular algorithm (the matrix vector multiplication) used for these examples is not prone to computation deadlocks. But the data transfer

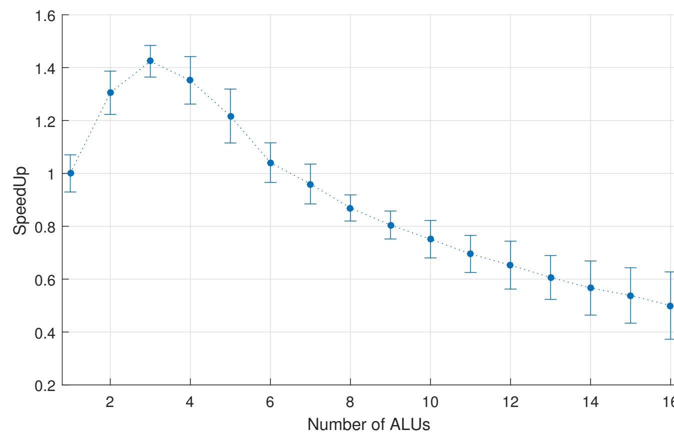
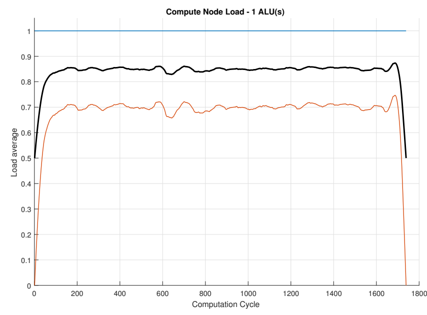


Figure 3.18: SpeedUp of Heterogenous Hardware (Monte Carlo Method)

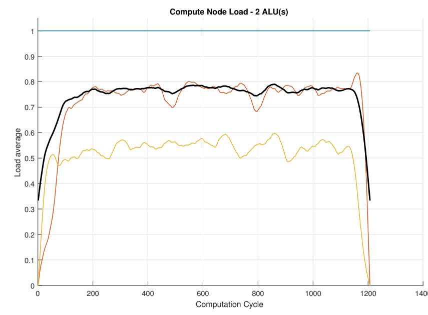
of intermediate values in between the ALU units can potentially severely impact the performance. Such a transfer is shown in Figures 3.20 to 3.22. Stripping all operations not relevant for the argument the sequence is as follows:

- Cycle 28
  `ALU #2` initiates an `fadd` instruction
- Cycle 31
  `ALU #2` finalized the `fadd` operation, and starts transferring the result to the `ARAM (shared)` node
- Cycle 33
  The transfer has finished, and `ALU #0` starts fetching the value
- Cycle 37
  The transfer has finished, and `ALU #0` starts fetching the second parameter (computed in cycle 17 (not shown) on `ALU #3`)
- Cycle 41 (not shown)
  The two operands (from `ALU #3` and `ALU #2`) are available at `ALU #0` and can be processed.

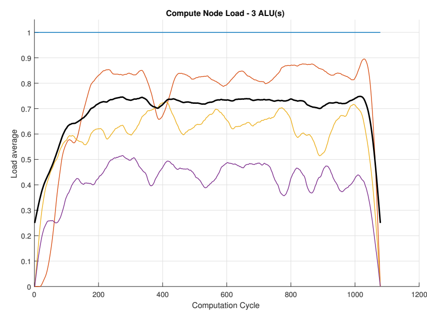
This simple snippet from the execution history of the nodes demonstrates the impact of assigning an operation to an specific ALU. Not only the servicing ALU is utilized, but also additional nodes can be obliged to assist. In this case this is demanded by the hardware arrangement, as only the originating node can transfer its intermediate data from its register set to the shared memory location.



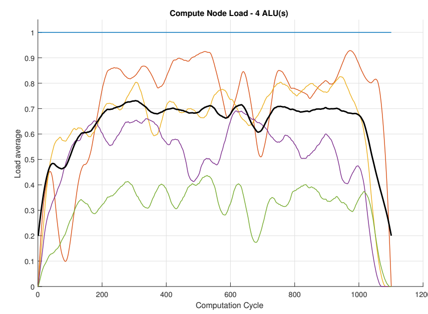
(a) one ALU node



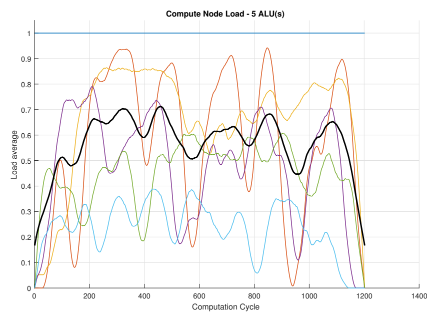
(b) two ALU nodes



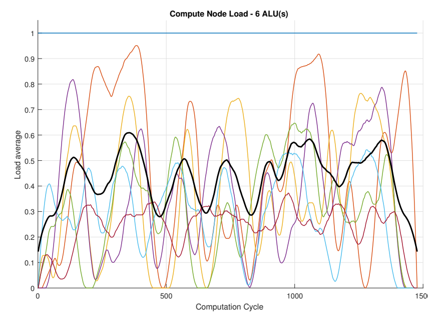
(c) three ALU nodes



(d) four ALU nodes

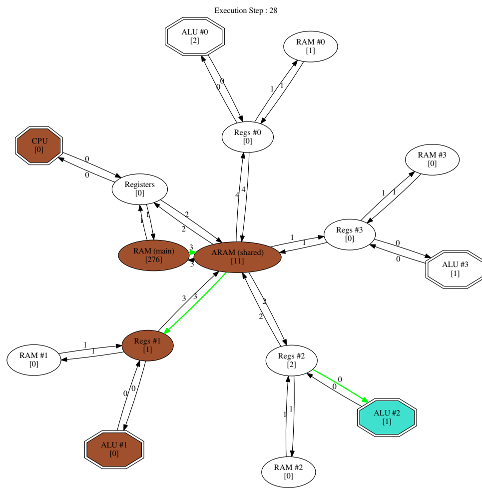


(e) five ALU nodes

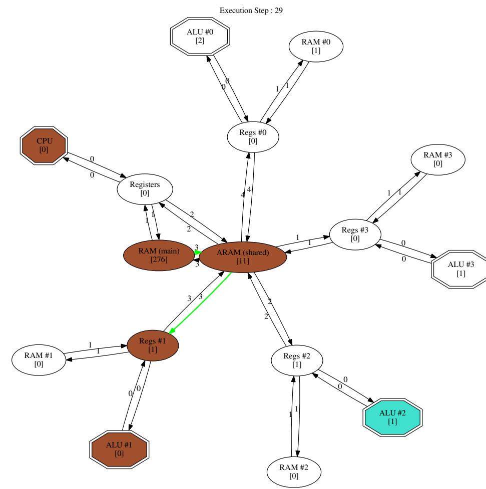


(f) six ALU nodes

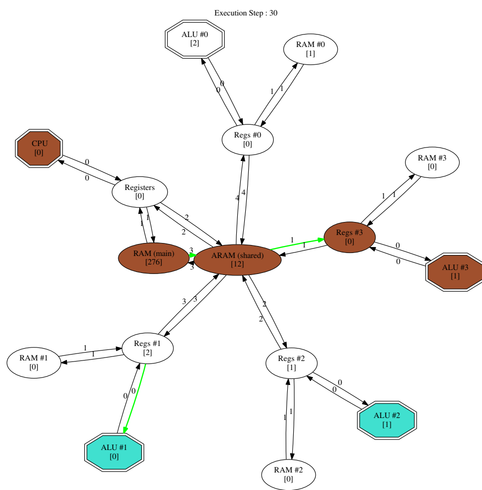
Figure 3.19: Load average (100 cycles window)



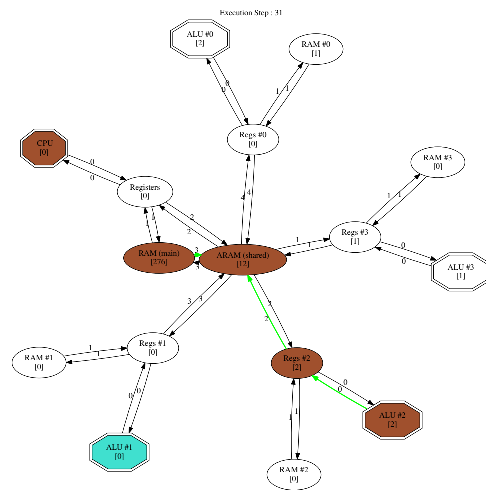
(a) $\hat{G}_4$ model, sequence time 28



(b) $\hat{G}_4$ model, sequence time 29

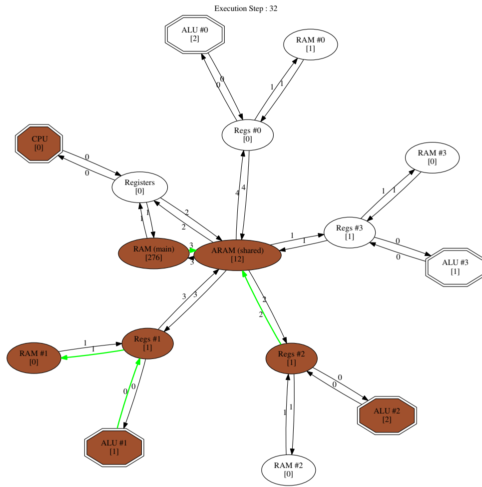


(c) $\hat{G}_4$ model, sequence time 30

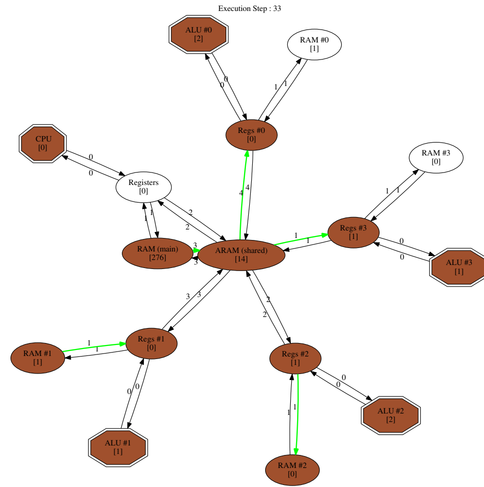


(d) $\hat{G}_4$ model, sequence time 31

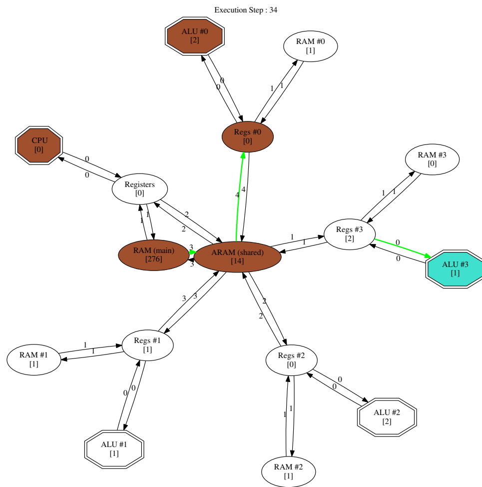
Figure 3.20: $\hat{G}_4$ model, sequence time 28 to 31



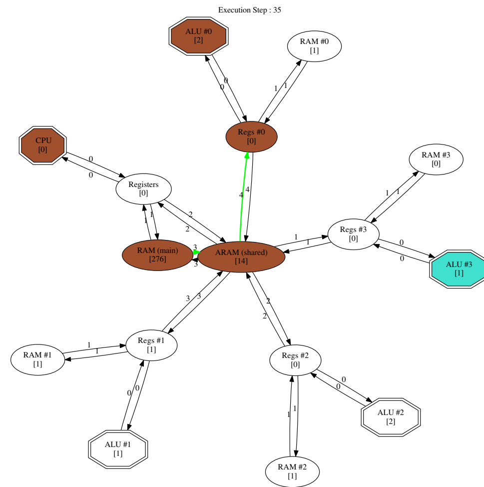
(a) $\hat{G}_4$ model, sequence time 32



(b) $\hat{G}_4$ model, sequence time 33

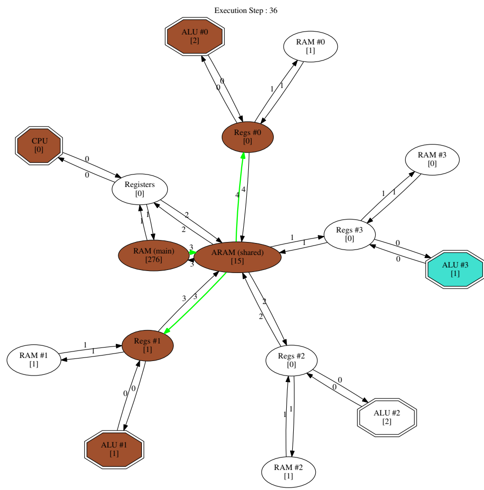


(c) $\hat{G}_4$ model, sequence time 34

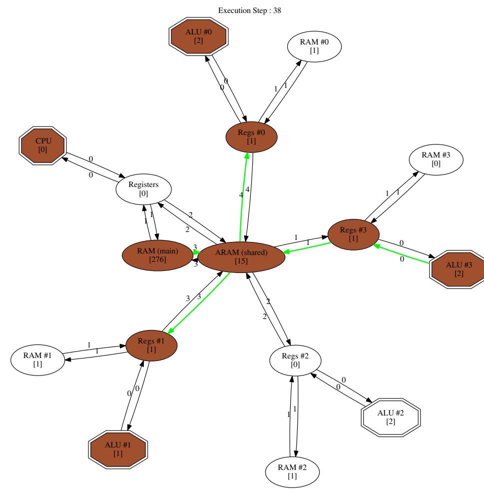


(d) $\hat{G}_4$ model, sequence time 35

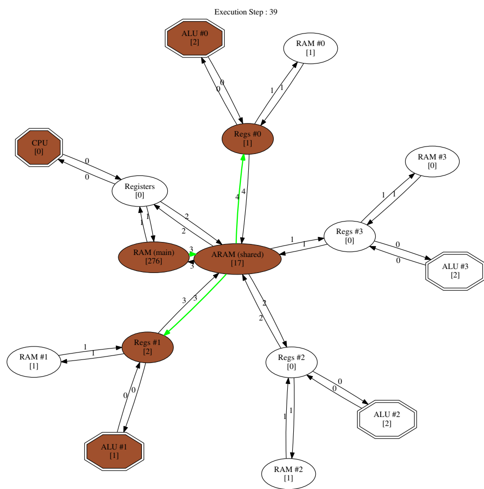
Figure 3.21: $\hat{G}_4$ model, sequence time 32 to 35



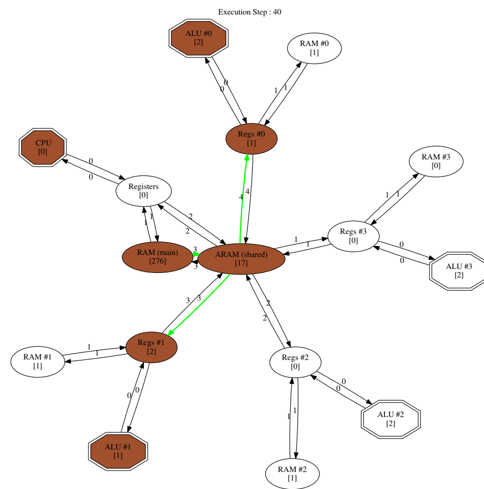
(a) $\hat{G}_4$ model, sequence time 36



(b) $\hat{G}_4$ model, sequence time 38



(c) $\hat{G}_4$ model, sequence time 39



(d) $\hat{G}_4$ model, sequence time 40

Figure 3.22: $\hat{G}_4$ model, sequence time 36 to 40

Declaration of Sources:
Chapter 3 was based on and reuses material from the following sources, previously published by the author:

[RB18a] Andreas Rechberger and Eugen Brenner. “Generalized Execution Time Estimation”. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–4.

[RB18b] Andreas Rechberger and Eugen Brenner. “Partitioning of Algorithms for Distributed Computation”. In: *Embedded World 2018 - Proceedings*. Embedded World Conference, Mar. 2018.

References to these sources are not always made explicit.

Chapter 4

Conclusion

In the previous chapters it has been shown that a CDFG is a suitable representation for an automated distribution of an algorithm on a heterogeneous hardware arrangement. It can be shown with simple examples that reverting to the data flow only is insufficient. If for example the code shown in Listing 4.1 is invoked with the parameter $a = 10.0$, the loop will be executed a single time. This is determined during run-time by two conditional branch operations (`br` in combination with `fcmp` caused by Listing 4.1 line 7). There is no data dependency between the update of the result b and the value a used to determine the loop condition. However the `fmul` operation depends on the branch in an "execute after" relation. This is depicted in Figure 4.1 with the dashed lines.

```
1  template<typename T>
2  T TestFunction(T a, T b) __attribute((noinline));
3
4  template<typename T>
5  T TestFunction(T a, T b)
6  {
7      while(a>7.53)
8      {
9          b *= 1.1;
10         a -= 10;
11     }
12     return b;
13 }
```

Listing 4.1: Branches in code

The combined CDFG represents a graph, more specifically a DAG. Its common notation is given by either a tuple or triple notation as given in Equations (4.1) and (4.2) [Küh96; Lee99; Cor+01; BW10]. With restricting the set V to be a finite set of operations this implies the execution time to be finite as well. Dealing with an infinite amount of operations does not make sense in this scenario. It would reflect an algorithm which cannot be computed in finite time. This definition is based on the operations of executing the sequence which specify the algorithm, not the operations to define the algorithm. Such a definition (in source form like LLVM assembly language for example) might look different. The control flow graph (CFG) of Listing 4.1 would only depict a single `fmul` node, independently of how often this instruction is actually

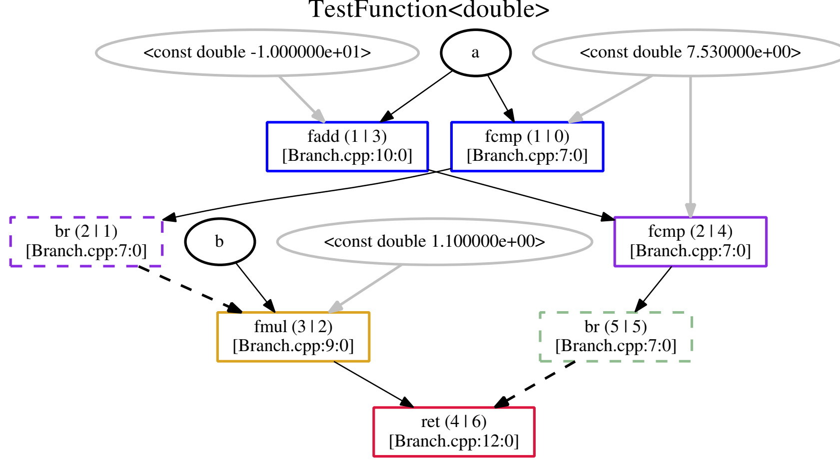


Figure 4.1: CDFG of Listing 4.1 with $a = 10.0$

executed. For the distribution algorithm to work each executed operation needs to be uniquely assigned to a vertex.

$$CDFG_{tuple} = G(V, E) \tag{4.1}$$

$$CDFG_{triple} = G(V, E, \phi) \tag{4.2}$$

$$E = E_d \cup E_c \tag{4.3}$$

$$V = V_c \cup V_d \tag{4.4}$$

The CDFG does contain two sets of vertices and edges. A vertex can either represent an operation or an input variable respectively a constant. These nodes are represented by two disjoint sets. V_c the set of compute-nodes and V_d the set of data-nodes (Equation (4.4)). Similar the set of edges E contains the edges representing a data flow (E_d) and edges declaring a control flow dependency (E_c). Using this distinction simplifies the realization of the distribution algorithm. From a graph point of view this distinction is of no relevance. Whether an edge emerges from a value-node, or from a compute-node does not influence the properties of the graph. The destination node simply handles them as "input dependency". For the distribution algorithm we need to assign certain properties to the set of compute vertices V_c of the CDFG. Equation (4.5) defines the function `op` which retrieves the operation assigned to such a vertex. The set of valid operations depends on the abstraction level of the analysis. In general O (Equation (4.6)) is a set of arbitrary operations. A compute operation however cannot be nullary. Data vertices are nullary, and are not associated to a specific operation. Further, operations cannot be assumed to be commutative. Therefore each data dependency edge needs a dedicated property to specify an index $idx(e)$ as given by Equation (4.8), its ordinal number. With this property the input arguments form an ordered set which uniquely maps all the data flow edges of any compute vertex to the arguments of the n-ary function represented by $op(v)$. This ordering is not shown in any of the depicted graphs in this work. For all the figures shown, the vertices and edges are arranged based on layout demands only. Equation (4.7) shows the minimal operation set for the example given in Figure 4.1. It is a

subset of all the LLVM assembly language instructions available. A distribution algorithm has to comply with the constraints given by these dependencies (topological ordering of the graph), but within these bounds can choose any execution order desired.

$$
\begin{aligned}
op : V_c &\rightarrow O && (4.5)\\
O &= \{\texttt{Operation}_\texttt{1}, \texttt{Operation}_\texttt{2}, ..., \texttt{Operation}_\texttt{n}\} && (4.6)\\
O_{concrete} &= \{\texttt{fmul}, \texttt{fcmp}, \texttt{fadd}, \texttt{br}, \texttt{ret}\} && (4.7)\\
idx : E_d &\rightarrow \mathbb{N}_0 && (4.8)
\end{aligned}
$$

Complementary to the generalized CDFG representation of an algorithm, a dedicated compute arrangements (the hardware) can be described by means of graphs as well. The hardware is expressed as directed graph (Equation (4.10)). This allows to model storage-nodes which can only be read, as well as for example reflecting different access times for retrieving and depositing data from storage. This however is not a strict necessity. If such a behaviour is not required for a particular hardware model, its graph can also be undirected. Neither is the hardware graph required to be free of loops, nor is this the case in most of the scenarios. This distinguishes the hardware graph from the CDFG, which as previously mentioned obeys the rules of a DAG. From a modelling perspective similar to the CDFG it can be beneficial to distinguish the storage-nodes $\tilde{V_S}$ and the execution-nodes $\tilde{V_E}$ as two disjoint sets (Equation (4.9)).

$$
\begin{aligned}
\tilde{V} &= \tilde{V}_S \cup \tilde{V}_E && (4.9)\\
HW &= G(\tilde{V}, \tilde{E}, \tilde{\phi}) && (4.10)
\end{aligned}
$$

For the generic distribution algorithm some auxiliary methods are required. These for example declare the properties of an edge. In the case of the "edge weight" this denotes the time quanta required to transfer an object across a particular connection. Within the examples given in the previous sections this time quanta was restricted to be an integer. This was only for reasons of a simpler demonstration. The edge weight can be any real number, but is required to be positive, including zero as given in Equation (4.11). For the sequence given in Section 2.4.2 additionally the two functions `cost` and `ops` (Equations (4.12) and (4.13)) are required. They reflect the execution cost of an operation $o \in O$ when served by node $v \in \tilde{V_E}$, and the set of operations which can be served by the node in general. S in this Equation (4.13) denotes the current state of the hardware arrangement HW. It reflects the location of all variables, and the executions already scheduled on the various execution-nodes. This function therefore does not deliver a static mapping on the execution time of an operation on a node. It does consider the current state and delivers the time t at which the function finalizes, if executed on node v with the precondition S. Equation (4.14) declares one of the conditions such that an algorithm can be mapped to a specific hardware. With allowing the state S to influence the computation capability of the hardware setup (Equation (4.13)) this becomes a complex condition. While obviously for a proper execution

at least one node must be capable of serving an instruction within the set $O_{concrete}$ there are additional conditions to be fulfilled. For example that any data required for an operation must have a path from its current location to the selected compute-node. These properties are dependent on the state S. Within this work this precondition was not formally verified upfront, but during run-time of the distribution algorithm.

$$w : \tilde{E} \rightarrow \{x \mid x \in \mathbb{R} \land x \geq 0\} \tag{4.11}$$

$$ops : \tilde{V}_E \rightarrow \{o \mid o \in O\} \tag{4.12}$$

$$cost : \{HW, \tilde{V}_E, O, S\} \rightarrow \mathbb{R} \tag{4.13}$$

$$\forall o \in O_{concrete}\ \exists ops(v \in \tilde{V}_E) \neq \emptyset \tag{4.14}$$

Section 3.3 showed the performance and distribution pattern of such an algorithm when mapped to a set of heterogeneous hardware arrangements by a simple distribution algorithm.

4.1 Future Work

In Section 3.3.1 the impact of the evaluation order of the instructions within the distribution algorithm has been briefly discussed. It intrinsically implies that for an automated distribution two disjoint sets need to be managed. The one holding all operations not yet scheduled but with all their dependencies fulfilled (the “ready to compute” set) and the one for the remaining instructions (the “not ready” set). Whenever a node has finished servicing an instruction, it potentially transfers an arbitrary amount of operations from the latter set to the first one. Whenever the cardinality of the “ready to compute” set is larger than one, the order of evaluation impacts the final result (Section 3.3.1). Therefore, in addition to selecting the optimal compute-node for an operation, the optimal order in which this selection takes place has to be chosen. This has similarities with the problem of scheduling jobs with precedence constraints. Variants of this have been researched since several decades [QK90], but are also subject to more recent publications. For example [Ben+20] discusses the scenario when dealing with errors in the execution, and as such the need to repeat a failed task. These research is however more abstract than this work, as considering rigid and moldable [Leu+04] tasks and not operations. In this work an operation can not be sliced and partitioned as moldable tasks can, but also does not have the strict performance characteristics of rigid tasks. Its performance metric varies among other things on the scheduling decisions of other (also independent) operations.

Another open issue arises whenever the representation of the operations of an algorithm does not entirely match the capabilities of the hardware. That is Equation (4.12) does not exist, but instead an alternative set of operations O' is available. An example of such a scenario is the MAC operation as discussed in Section 3.1.1. For a more precise estimation the instruction set O has to be transformed to O', including all the required modifications to the CDFG. This is similar to the common problem of modern compilers when lowering

their intermediate representation to target specific instructions. This process is called instruction selection, and discussed in various research on compiler and compiler optimizations [FFY01; EKS03].

Possible future extension to this work can also lift the restriction on a static hardware model. In this work the hardware connection properties are considered to be static. The transfer cost function (Equation (4.11)) is invariant on the state S introduced in Equation (4.13). By this the path finding is static, and can be computed a priori by means of well established algorithms [Bel58; Dij59]. When working with a hardware model which features different transfer cost depending on its state S, these algorithms need to be extended. Such an extension is proposed in [SG18]. Such models can reflect hardware arrangements whose connections interfere with each other. This is one approach to model a setup which has a difference between the logical view, and the physical realization. This might for example reflect a set of storage-nodes, which are connected via a shared bus interface. A micro controller unit (MCU) by this approach can express that fact that its external RAM as well as its non-vloatile memory, is connected via a shared hardware connection, the external memory interface. Internally or logically however they reflect to independent storage-nodes, and are used by compute-nodes accordingly.

Assuming a computationally efficient realization of the automated distribution is available, one can also extend the topic to a slightly modified research question. Rather than answering on the efficiency of one or multiple dedicated hardware arrangements for a particular algorithm we could ask for the most efficient hardware setup. This is basically building a particular compute arrangement specifically suited for this algorithm. With a large variety on the restrictions applied this topic is covered by several approaches in the area of HLS [MS09; Jää+17; SW19]. Especially the ability to connect different hardware models by adding edges in-between their disjoint graphs looks promising. This allows rather than synthesizing the entire hardware during the distribution analysis to revert to combining existing build blocks. For engineering purposes such an approach offers a quite promising field of application. With the emergence of internet of things (IoT) and publications in the domain of edge computing [Nez+20; Wan+21] a collection of independent but connected nodes whose can be utilized to serve a common goal becomes an interesting field of research.

Bibliography

[Bel58] Richard Bellman. “On a routing problem”. In: *Quart. Appl. Math* 16.16 (1958), pp. 87–90.

[Dij59] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[Amd67] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

[Tar72] Robert Tarjan. “Depth first search and linear graph algorithms”. In: *SIAM Journal on Computing* 1.2 (1972). DOI: 10.1137/0201010.

[Knu74] Donald E. Knuth. “Structured Programming with Go to Statements”. In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[Kin76] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[Knu76] Donald E. Knuth. “Big Omicron and Big Omega and Big Theta”. In: *SIGACT News* 8.2 (Apr. 1976), pp. 18–24. ISSN: 0163-5700. DOI: 10.1145/1008328.1008329.

[Eck87] Roger Eckhardt. “Stan ulam, john von neumann, and the monte carlo method”. In: *Los Alamos Science* 15 (1987), pp. 131–136.

[Gus88] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[QK90] Qingzhou Wang and Kam Hoi Cheng. “Parallel time complexity of a heuristic algorithm for the k-center problem with usage weights”. In: *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990.* Dec. 1990, pp. 254–257. DOI: 10.1109/SPDP.1990.143543.

[Cyt+91] Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. "Constant propagation with conditional branches". In: *ACM Transactions on Programming Languages and Systems* 13 (1991), pp. 291–299.

[FH92] Christopher W. Fraser and David R. Hanson. "Simple register spilling in a retargetable compiler". In: *Software: Practice and Experience* 22.1 (1992), pp. 85–99. DOI: 10.1002/spe.4380220105.

[Sed92] R. Sedgewick. *Algorithmen in C++*. Addison-Wesley, 1992. ISBN: 9783893194629.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. C++ In-Depth Series. Addison-Wesley, 1995. ISBN: 978-0-201-57594-1.

[Küh96] Dietmar Kühl. "Design patterns for the implementation of graph algorithms". MA thesis. Technische Universität Berlin, 1996. URL: http://www.dietmar-kuehl.de/generic-graph-algorithms.pdf (visited on 12/28/2020).

[App98] Andrew W. Appel. "SSA is Functional Programming". In: *ACM SIGPLAN Notices* 33.4 (1998), pp. 17–20.

[DKL98] Erik B. Dam, Martin Koch, and Martin Lillholm. *Quaternions, interpolation and animation*. Tech. rep. University of Copenhagen, 1998.

[ISO98] ISO/IEC JTC 1/SC 22. *ISO International Standard ISO/IEC 14882:1998 – Programming Language C++*. ISO. Oct. 1998.

[Hol99] Jason J. Holdsworth. *The Nature of Breadth-First Search*. 1999.

[Hut99] Graham Hutton. "A Tutorial on the Universality and Expressiveness of Fold". In: *J. Funct. Program.* 9.4 (July 1999), pp. 355–372. ISSN: 0956-7968. DOI: 10.1017/S0956796899003500.

[Lee99] Lie-Quan Lee. "The High Performance Generic Graph Component Library". MA thesis. Department of Computer Science and Engineering, University of Notre Dame, 1999.

[LA00] Samuel Larsen and Saman Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 145–156. ISBN: 1581131992. DOI: 10.1145/349299.349320.

[AG01] Andrew W. Appel and Lal George. "Optimal Spilling for CISC Machines with Few Registers". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 243–253. ISBN: 1581134142. DOI: 10.1145/378795.378854.

[Cor+01] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

[FFY01] Paolo Faraboschi, Joseph Fisher, and Cliff Young. "Instruction scheduling for instruction level parallel processors". In: *Proceedings of the IEEE* 89 (Dec. 2001), pp. 1638–1659. DOI: `10.1109/5.964443`.

[Fe01] Jon Ferraiolo and ed. *Scalable Vector Graphics (SVG) 1.0 Specification*. 2001. URL: `https://www.w3.org/TR/SVG10/` (visited on 12/28/2012).

[SLL01] J.G. Siek, L.Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. C++ In-Depth Series. Pearson Education, 2001. ISBN: 9780321601612.

[Ins02] Texas Instrumentes. *TMS320DM642 Technical Overview*. English. TI. Sept. 2002. 43 pp. URL: `https://www.ti.com/lit/ug/spru615/spru615.pdf` (visited on 12/28/2020).

[Pey02] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Vol. 13. Cambridge University Press, Jan. 2002. ISBN: 978-0521826143.

[EKS03] Erik Eckstein, Oliver König, and Bernhard Scholz. "Code Instruction Selection Based on SSA-Graphs". In: vol. 2826/2003. Sept. 2003, pp. 49–65. ISBN: 978-3-540-20145-8. DOI: `10.1007/978-3-540-39920-9_5`.

[Lam03] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2003. ISBN: 978-0-321-14306-8. URL: `https://lamport.azurewebsites.net/tla/book.html` (visited on 12/28/2020).

[Mer03] Jason Merrill. "Generic and gimple: A new tree representation for entire functions". In: *In Proceedings of the 2003 GCC Summi*. 2003.

[Ant+04] G. Antoniol et al. "Compiler hacking for source code analysis". In: *Software Quality Journal* 12 (2004), p. 2004.

[LA04a] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[LA04b] Chris Lattner and Vikram Adve. "The LLVM Compiler Framework and Infrastructure Tutorial". In: *LCPC'04 Mini Workshop on Compiler Research Infrastructures*. West Lafayette, Indiana, Sept. 2004.

[Leu+04] Joseph Y-T. Leung et al. *Handbook of scheduling. Algorithms, Models and Performance Analysis*. Ed. by Joseph Y-T. Leung. 2004. ISBN: 9781584883975. URL: `https://b-ok.org/book/1022695/3e78d0` (visited on 12/21/2020).

[Sha+04] Hovav Shacham et al. "On the Effectiveness of Address-Space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: Association for Computing Machinery, 2004, pp. 298–307. ISBN: 1581139616. DOI: `10.1145/1030083.1030124`.

[SL05] Herb Sutter and Jim Larus. "Software and the Concurrency Revolution". In: *ACM Queue* (Sept. 2005), pp. 54–62. URL: https://www.microsoft.com/en-us/research/publication/software-and-the-concurrency-revolution/ (visited on 12/28/2020).

[ASU06] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0-321-48681-1.

[Mat+07] Avantika Mathur et al. "The New ext4 filesystem: current status and future plans". In: Jan. 2007. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf (visited on 12/28/2020).

[Bee08] Robert Beers. "Pre-RTL Formal Verification: An Intel Experience". In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: Association for Computing Machinery, 2008, pp. 806–811. ISBN: 9781605581156. DOI: 10.1145/1391469.1391675.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf (visited on 12/28/2020).

[GB08] Rajesh Gupta and Forrest Brewer. "High-Level Synthesis: A Retrospective". In: *High-Level Synthesis: From Algorithm to Digital Circuit* (Jan. 2008). DOI: 10.1007/978-1-4020-8588-8_2.

[Lat08] Chris Lattner. "LLVM and Clang: Next Generation Compiler Technology". In: *The BSD Conference*. 2008.

[QP08] Fernando Magno Quintão Pereira and Jens Palsberg. "Register Allocation by Puzzle Solving". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 216–226. ISSN: 0362-1340. DOI: 10.1145/1379022.1375609.

[Lam09] Leslie Lamport. "The PlusCal Algorithm Language". In: *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.* (Jan. 2009). URL: https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/ (visited on 12/28/2020).

[MS09] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 18–25. ISSN: 1558-1918. DOI: 10.1109/MDT.2009.83.

[OS09] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0131988425.

[WWP09] Samuel Webb Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[ARM10] ARM, ed. *Cortex-M4, Technical Reference Manual*. ARM Limited. Mar. 2010. URL: https://static.docs.arm.com/ddi0439/b/DDI0439B_cortex_m4_r0p0_trm.pdf (visited on 07/26/2020).

[BW10] Edward A. Bender and S. Gill Williamson. *Lists, Decisions and Graphs*. University of California, Sept. 2010. URL: http://cseweb.ucsd.edu/~gill/BWLectSite/Resources/LDGbookCOV.pdf (visited on 12/28/2020).

[Har+10] William R. Harris et al. "Program analysis via satisfiability modulo path programs". In: *IN: POPL*. 2010, pp. 71–82.

[Mee10] Haroon Meer. *Memory Corruption Attacks The (almost) Complete History*. 2010.

[ISO11] ISO/IEC JTC 1/SC 22. *ISO International Standard ISO/IEC 14882:2011 – Programming Language C++*. ISO. Oct. 2011.

[Mal+11] Saeed Maleki et al. "An evaluation of vectorizing compilers". English (US). In: *Proceedings - 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011*. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT. 20th International Conference on Parallel Architectures and Compilation Techniques, PACT 2011 ; Conference date: 10-10-2011 Through 14-10-2011. Dec. 2011, pp. 372–382. ISBN: 9780769545660. DOI: 10.1109/PACT.2011.68.

[Smi11] Alan G. Smith. *Introduction to Arduino - A Piece of Cake*. CreateSpace Independent Publishing Platform, 2011. ISBN: 978-1-463-69834-8. URL: http://www.introtoarduino.com (visited on 12/28/2020).

[Ver+11] Eric Verhulst et al. *Formal Development of a Network-Centric RTOS*. Springer, Boston, MA, Jan. 2011. ISBN: 978-1-4419-9736-4. DOI: 10.1007/978-1-4419-9736-4.

[ISO12] ISO/IEC JTC 1/SC 22. *ISO International Standard ISO/IEC 8652:2012 Information technology — Programming languages — Ada*. ISO. Dec. 2012.

[Len12] Christian Lengauer. "Polly—Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22 (Dec. 2012). DOI: 10.1142/S0129626412500107.

[Agh+13] Ishwari Aghav et al. "Automated static data flow analysis". In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. July 2013, pp. 1–4. DOI: 10.1109/ICCCNT.2013.6726670.

[Sno+13] K. Z. Snow et al. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[ISO14] ISO/IEC JTC 1/SC 22. *ISO International Standard ISO/IEC 14882:2014 – Programming Language C++*. ISO. Dec. 2014.

[Val14] Celina Gomes do Val. "Conflict-Driven Symbolic Execution : How to Learn to Get Better". MA thesis. University of British Columbia, 2014. DOI: http://dx.doi.org/10.14288/1.0165906.

[DD15] Michael Dossis and Georgios Dimitriou. "Hardware Synthesis of High-Level C Constructs". In: *Proceedings of the 19th Panhellenic Conference on Informatics*. PCI '15. Athens, Greece: Association for Computing Machinery, 2015, pp. 83–85. ISBN: 9781450335515. DOI: 10.1145/2801948.2802029.

[Aba+16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[EPA16] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783743.

[Fre17] Free Software Foundation. *The C++ Runtime Library (libstdc++)*. Sept. 13, 2017. URL: https://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/ (visited on 12/25/2020).

[God17] Matt Godbolt. "What Has My Compiler Done for Me Lately - Unbolting the Compiler's Lid". In: *Cppcon 2017 KeyNotes*. CppCon. 2017. URL: https://github.com/CppCon/CppCon2017/tree/master/Keynotes (visited on 12/28/2020).

[ISO17a] ISO/IEC JTC 1/SC 22. *ISO International Standard ISO/IEC 14882:2017 – Programming Language C++*. ISO. Dec. 2017.

[ISO17b] ISO/IEC JTC 1/SC 7. *ISO/IEC/IEEE International Standard - Systems and software engineering -- Vocabulary*. ISO. 2017.

[Jää+17] Pekka Jääskeläinen et al. "HW/SW Co-design Toolset for Customization of Exposed Datapath Processors". In: *Computing Platforms for Software-Defined Radio*. Ed. by Waqar Hussain et al. Springer International Publishing, 2017, pp. 147–164. ISBN: 978-3-319-49679-5. DOI: 10.1007/978-3-319-49679-5_8.

[Tan+17] Satyanarayana Tani et al. "Application of crowdsourced hail data and damage information for hail risk assessment in the province of Styria, Austria". In: *Application of crowdsourced hail data and damage information for hail risk assessment in the province of Styria, Austria*. EGU 2017, IE2.1/NH9.19, Apr. 2017. URL: http://meetingorganizer.copernicus.org/EGU2017/EGU2017-6822.pdf (visited on 12/28/2020).

[DDS18] Georgios Dimitriou, Michael Dossis, and Georgios Stamoulis. "Operation Dependencies in Loop Pipelining for High-Level Synthesis". In: Sept. 2018, pp. 1–6. DOI: 10.23919/SEEDA-CECNSM.2018.8544930.

[ISO18] ISO/IEC JTC 1/SC 22. *ISO International Standard 9899:2018 Information technology — Programming languages — C*. ISO. June 2018.

[LLV18] LLVM Project. *LLVM Language Reference*. 2018. URL: https://llvm.org/docs/LangRef.html (visited on 01/03/2018).

[RB18a] Andreas Rechberger and Eugen Brenner. "Generalized Execution Time Estimation". In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, pp. 1–4.

[RB18b] Andreas Rechberger and Eugen Brenner. "Partitioning of Algorithms for Distributed Computation". In: *Embedded World 2018 - Proceedings*. Embedded World Conference, Mar. 2018.

[Rec+18] Andreas Rechberger et al. "HeDi - Hagelereignis Dateninterface". deutsch. Lange Nacht der Forschung 2018. Apr. 2018. URL: https://hedi.tugraz.at (visited on 12/28/2020).

[RP18] Guido Van Rossum and Python Development Team. *The Python Language Reference*. 12th Media Services, 2018. ISBN: 1680921614.

[ST 18] ST Microelectronics. *Discovery kit with STM32F407VG MCU*. 2018. URL: http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/stm32f4discovery.html (visited on 04/09/2018).

[SG18] Sunita and Deepak Garg. "Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue". In: *Journal of King Saud University - Computer and Information Sciences* (2018). ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.03.003.

[Xu+18] H. Xu et al. "Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs". In: *IEEE Transactions on Dependable and Secure Computing* (2018), pp. 1–1.

[KR19] S. Kim and J. Ryou. "Source Code Analysis for Static Prediction of Dynamic Memory Usage". In: *2019 International Conference on Platform Technology and Service (PlatCon)*. Jan. 2019, pp. 1–4. DOI: 10.1109/PlatCon.2019.8669417.

[Men+19] Charith Mendis et al. "Compiler Auto-Vectorization with Imitation Learning". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 14625–14635. URL: http://papers.nips.cc/paper/9604-compiler-auto-vectorization-with-imitation-learning.pdf (visited on 12/28/2020).

[Rec+19] Andreas Rechberger et al. "High Precision Vibration-Type Densitometers Based on Pulsed Excitation Measurements". In: *Sensors* 19.7 (2019). ISSN: 1424-8220. DOI: 10.3390/s19071627. URL: https://www.mdpi.com/1424-8220/19/7/1627 (visited on 12/28/2020).

[SW19] B. C. Schafer and Z. Wang. "High-Level Synthesis Design Space Exploration: Past, Present and Future". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), pp. 1–1. ISSN: 1937-4151. DOI: 10.1109/TCAD.2019.2943570.

[Xil19] Xilinx, ed. *MicroBlaze Processor Reference Guide*. Xilinx Inc. Mar. 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug984-vivado-microblaze-ref.pdf (visited on 12/20/2020).

[AB20] Ericsson AB. *Erlang Reference Manual User's Guide*. 2020. URL: https://erlang.org/doc/reference_manual/users_guide.html (visited on 12/17/2020).

[Ama20] Saman Amarasinghe. "Compiler 2.0: Using Machine Learning to Modernize Compiler Technology". In: *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '20. London, United Kingdom: Association for Computing Machinery, June 2020, pp. 1–2. ISBN: 9781450370943. DOI: 10.1145/3372799.3397167.

[Ben+20] A. Benoit et al. "Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms". In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2020, pp. 81–91. DOI: 10.1109/CLUSTER49012.2020.00018.

[Cad20] Cadence. *Stratus High-Level Synthesis*. Aug. 28, 2020. URL: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf (visited on 12/20/2020).

[GW20] Stephane Gauthier and Zubair Wadood. *High-Level Synthesis: Can it outperform hand-coded HDL?* Tech. rep. June 2020. URL: https://www.silexica.com/wp-content/uploads/High-level-Synthesis-Can-it-outperform-hand-coded-HDL.pdf (visited on 12/20/2020).

[GNU20] GNU Project. *GCC, the GNU Compiler Collection*. 2020. URL: https://gcc.gnu.org/ (visited on 06/23/2020).

[Gos+20] James Gosling et al. *The Java® Language Specification*. 2020. URL: https://docs.oracle.com/javase/specs/jls/se15/html/index.html (visited on 12/17/2020).

[Haj+20] Ameer Haj-Ali et al. "NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 242–255. ISBN: 9781450370479. DOI: 10.1145/3368826.3377928.

[Int20a] Intel, ed. *Nios II Custom Instruction User Guide*. Intel Corporation. Apr. 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf (visited on 12/20/2020).

[Int20b] Intel Corporation. *Intel® C++ Compiler*. 2020. URL: https://software.intel.com/en-us/forums/intel-c-compiler (visited on 06/23/2020).

[KK20] Y. Kim and M. Kang. "Formal Verification of SDN-Based Firewalls by Using TLA+". In: *IEEE Access* 8 (2020), pp. 52100–52112. DOI: 10.1109/ACCESS.2020.2979894.

[Lat+20] Chris Lattner et al. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[Mic20] Microsoft. *Visual Studio 2019*. 2020. URL: https://visualstudio.microsoft.com/vs (visited on 06/23/2020).

[MHJ20] Joonas Multanen, Kari Hepola, and Pekka Jääskeläinen. "Programmable Dictionary Code Compression for Instruction Stream Energy Efficiency". In: Dec. 2020.

[Nez+20] Zeinab Nezami et al. *Decentralized Edge-to-Cloud Load-balancing: Service Placement for the Internet of Things*. 2020. arXiv: 2005.00270 [cs.DC].

[Sch+20] Fabian Schuiki et al. *LLHD: A Multi-level Intermediate Representation for Hardware Description Languages*. 2020. arXiv: 2004.03494 [cs.PL].

[Sil20] Silexia. *SLX FPGA*. 2020. URL: https://www.silexica.com/wp-content/uploads/2020/03/SLX-FPGA-20.1.pdf (visited on 12/20/2020).

[Tan+20] Jialiang Tan et al. "What Every Scientific Programmer Should Know about Compiler Optimizations?" In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392754.

[Ter+20] Kati Tervo et al. "TTA-SIMD Soft Core Processors". In: Aug. 2020, pp. 79–84. DOI: 10.1109/FPL50879.2020.00023.

[Vis20] Visual GDB. *VisualGDB Serious Cross-Platform for Visual Studio!* 2020. URL: https://visualgdb.com (visited on 07/26/2020).

[Wad20] Zubair Wadood. *Adaptive Radar Beamformer: An HLS Optimization Case Study With SLX*. Tech. rep. 2020. URL: https://www.silexica.com/wp-content/uploads/Adaptive-Beamformer-An-HLS-Optimization-Case-Study-with-SLX-FPGA.pdf (visited on 12/20/2020).

[Wan+21] J. Wang et al. "Fast Adaptive Task Offloading in Edge Computing Based on Meta Reinforcement Learning". In: *IEEE Transactions on Parallel and Distributed Systems* 32.1 (Jan. 2021), pp. 242–253. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3014896.