Application of Daedal in Huawei’s Base Station Scheduler
and Enlargement of Extended Data-Race-Free Regions through Conflict Isolation

Per Ekemark
Abstract

Daedal in Huawei’s Base Station Scheduler and Enlargement of xDRFs through Conflict Isolation

Per Ekemark

Part I:

In order to improve energy efficiency, decoupled access-execute based Daedal is applied to, and evaluated for, a part of Huawei’s base station software. Daedal saves energy by concentrating cache misses in a cache pre-load access phase, which can be run at a lower clock frequency, placed ahead of targeted code. The application of Daedal requires adaptations to work with the base station software’s existing compiler and to handle its complex code base. Between the sample targets, one is found to have fundamental compatibility issues with Daedal; the other is sped up by a factor 2.5 but despite using address analysis based feedback directed optimisation still experiences an overall slowdown due to a heavy access phase, giving it a best case normalised energy-delay product of 1.32. For future improvement, Daedal may need alternate prioritisation rules in order to produce a lighter access phase and the ability to handle more complex code, including function calls and rollback of visible memory modifications. Target software must also be accommodated to avoid hard-to-analyse program features.

Part II:

Several parallel programming languages enforce a data-race-free (DRF) memory model. Extended data-race-free (xDRF) analysis is a technique that can improve the reach of DRF semantics by combining multiple DRF regions, but this sometimes fails due to falsely identified memory conflicts. By amending the xDRF analysis with conflict isolation, DRF regions can be combined anyway while conflicts are marked and excluded from extended regions. With conflict isolation, the xDRF analysis is able to match a manually determined optimum and improve the performance of a state-of-the-art dual-mode cache coherency protocol where the original analysis falls short.
Contents

Abstract i

Contents ii
  List of Figures ......................................................... v
  List of Tables ........................................................ vii
  List of Listings ......................................................... vii

Preface viii

I Application of DAEDAL in Huawei’s Base Station Scheduler ix

Glossary x
  Abbreviations ............................................................ x
  Terms ................................................................. xi

1 Introduction 1
  1.1 Background of LTE base stations ................................. 2
  1.2 Background of DAEDAL and DAE ................................. 3
    1.2.1 Access phase generation ................................. 5
    1.2.2 Multi-versioning and indirection level ..................... 8
  1.3 Contribution ....................................................... 10

2 Method 11
  2.1 Finding DAEDAL targets ........................................... 12
    2.1.1 Ideal target properties ................................ 12
    2.1.2 Profiling the base station software ..................... 13
  2.2 Integrating DAEDAL ............................................... 13
    2.2.1 Processing selective code ................................ 14
    2.2.2 Adaptations and workarounds ............................. 16
  2.3 DAEDAL improvements ............................................. 17
## Contents

### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Base station timing pipeline</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>DAE concept</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>DAE loop transform</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>DAE execution model</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Access phase transformation</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Example of indirection level</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of profiling using the PMU</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Partial DAEDAL application</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>DAEDAL process components</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Placement of profiling calls</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Address collection</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Feedback prefetch filter</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Base station software overview</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Test bench for Layer 2</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Expert function call tree</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Target A, prefetch characteristics</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Target B, prefetch characteristics</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>Target A, normalised execution time</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Target B, normalised execution time</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>Task queue in target B</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Target B with loop progression, normalised execution time</td>
<td>39</td>
</tr>
<tr>
<td>3.10</td>
<td>Superfluous prefetches by type in target A</td>
<td>41</td>
</tr>
<tr>
<td>3.11</td>
<td>Target A, AAopt, normalised execution time</td>
<td>42</td>
</tr>
<tr>
<td>3.12</td>
<td>Target A, AAopt, LF, normalised execution time</td>
<td>43</td>
</tr>
<tr>
<td>3.13</td>
<td>Target A, AAopt, HF/LF, normalised execution time</td>
<td>44</td>
</tr>
<tr>
<td>3.14</td>
<td>Target A, AAopt, HF/LF, normalised EDP</td>
<td>45</td>
</tr>
<tr>
<td>B.1</td>
<td>Address collection events</td>
<td>55</td>
</tr>
<tr>
<td>C.1</td>
<td>Placement of profiling calls for whole target measurements</td>
<td>56</td>
</tr>
<tr>
<td>C.2</td>
<td>Target A, Initial (phase), norm. CAE (gran. 1)</td>
<td>59</td>
</tr>
<tr>
<td>C.3</td>
<td>Target A, Initial (phase), norm. CAE (same gran.)</td>
<td>59</td>
</tr>
<tr>
<td>C.4</td>
<td>Target A, Initial (whole), norm. CAE (gran. 1)</td>
<td>60</td>
</tr>
<tr>
<td>C.5</td>
<td>Target A, Initial (whole), norm. CAE (same gran.)</td>
<td>60</td>
</tr>
<tr>
<td>C.6</td>
<td>Original Target B, Initial (phase), norm. CAE (gran. 1)</td>
<td>61</td>
</tr>
<tr>
<td>C.7</td>
<td>Original Target B, Initial (phase), norm. CAE (same gran.)</td>
<td>61</td>
</tr>
<tr>
<td>C.8</td>
<td>Original Target B, Initial (whole), norm. CAE (gran. 1)</td>
<td>62</td>
</tr>
<tr>
<td>C.9</td>
<td>Original Target B, Initial (whole), norm. CAE (same gran.)</td>
<td>62</td>
</tr>
<tr>
<td>C.10</td>
<td>Target A, Inline (phase), norm. CAE (gran. 1)</td>
<td>63</td>
</tr>
<tr>
<td>C.11</td>
<td>Target A, Inline (phase), norm. CAE (same gran.)</td>
<td>63</td>
</tr>
</tbody>
</table>
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.12</td>
<td>Target A, Inline (whole), norm. CAE (gran. 1)</td>
<td>64</td>
</tr>
<tr>
<td>C.13</td>
<td>Target A, Inline (whole), norm. CAE (same gran.)</td>
<td>64</td>
</tr>
<tr>
<td>C.14</td>
<td>Original Target B, Inline (phase), norm. CAE (gran. 1)</td>
<td>65</td>
</tr>
<tr>
<td>C.15</td>
<td>Original Target B, Inline (phase), norm. CAE (same gran.)</td>
<td>65</td>
</tr>
<tr>
<td>C.16</td>
<td>Original Target B, Inline (whole), norm. CAE (gran. 1)</td>
<td>66</td>
</tr>
<tr>
<td>C.17</td>
<td>Original Target B, Inline (whole), norm. CAE (same gran.)</td>
<td>66</td>
</tr>
<tr>
<td>C.18</td>
<td>Target A, InlineSA (phase), norm. CAE (gran. 1)</td>
<td>67</td>
</tr>
<tr>
<td>C.19</td>
<td>Target A, InlineSA (phase), norm. CAE (same gran.)</td>
<td>67</td>
</tr>
<tr>
<td>C.20</td>
<td>Target A, InlineSA (whole), norm. CAE (gran. 1)</td>
<td>68</td>
</tr>
<tr>
<td>C.21</td>
<td>Target A, InlineSA (whole), norm. CAE (same gran.)</td>
<td>68</td>
</tr>
<tr>
<td>C.22</td>
<td>Target B, InlineSA (phase), norm. CAE (gran. 1)</td>
<td>69</td>
</tr>
<tr>
<td>C.23</td>
<td>Target B, InlineSA (phase), norm. CAE (same gran.)</td>
<td>69</td>
</tr>
<tr>
<td>C.24</td>
<td>Target B, InlineSA (whole), norm. CAE (gran. 1)</td>
<td>70</td>
</tr>
<tr>
<td>C.25</td>
<td>Target B, InlineSA (whole), norm. CAE (same gran.)</td>
<td>70</td>
</tr>
<tr>
<td>C.26</td>
<td>Target A, InlImpSA (phase), norm. CAE (gran. 1)</td>
<td>71</td>
</tr>
<tr>
<td>C.27</td>
<td>Target A, InlImpSA (phase), norm. CAE (same gran.)</td>
<td>71</td>
</tr>
<tr>
<td>C.28</td>
<td>Target A, InlImpSA (whole), norm. CAE (gran. 1)</td>
<td>72</td>
</tr>
<tr>
<td>C.29</td>
<td>Target A, InlImpSA (whole), norm. CAE (same gran.)</td>
<td>72</td>
</tr>
<tr>
<td>C.30</td>
<td>Target B, InlImpSA (phase), norm. CAE (gran. 1)</td>
<td>73</td>
</tr>
<tr>
<td>C.31</td>
<td>Target B, InlImpSA (phase), norm. CAE (same gran.)</td>
<td>73</td>
</tr>
<tr>
<td>C.32</td>
<td>Target B, InlImpSA (whole), norm. CAE (gran. 1)</td>
<td>74</td>
</tr>
<tr>
<td>C.33</td>
<td>Target B, InlImpSA (whole), norm. CAE (same gran.)</td>
<td>74</td>
</tr>
<tr>
<td>C.34</td>
<td>Target B, InlImpSA-LP (phase), norm. CAE (gran. 1)</td>
<td>75</td>
</tr>
<tr>
<td>C.35</td>
<td>Target B, InlImpSA-LP (phase), norm. CAE (same gran.)</td>
<td>75</td>
</tr>
<tr>
<td>C.36</td>
<td>Target B, InlImpSA-LP (whole), norm. CAE (gran. 1)</td>
<td>76</td>
</tr>
<tr>
<td>C.37</td>
<td>Target B, InlImpSA-LP (whole), norm. CAE (same gran.)</td>
<td>76</td>
</tr>
<tr>
<td>C.38</td>
<td>Target A, PrimProfopt HF (phase), norm. CAE (stage 1)</td>
<td>77</td>
</tr>
<tr>
<td>C.39</td>
<td>Target A, PrimProfopt HF (whole), norm. CAE (stage 1)</td>
<td>77</td>
</tr>
<tr>
<td>C.40</td>
<td>Target A, PrimProfopt LF (phase), norm. CAE (HF, stage 1)</td>
<td>78</td>
</tr>
<tr>
<td>C.41</td>
<td>Target A, PrimProfopt LF (whole), norm. CAE (HF, stage 1)</td>
<td>78</td>
</tr>
</tbody>
</table>

#### 7.1 Examples of situations allowing or disallowing xDRF formation
#### 7.2 Special examples of situations disallowing xDRF formation

#### 8.1 Difference between xDRF and xDRF-CI

#### 9.1 Non-enclave nDRF regions in code
#### 9.2 Non-enclave nDRF regions at runtime, normalised to Directory
#### 9.3 Network traffic, normalised to Directory
#### 9.4 Execution time, normalised to Directory
Contents

List of Tables

3.1 Top level target candidates ........................................ 30
3.2 DAEDAL targets .................................................... 30
9.1 Benchmark compilation time ...................................... 93

List of Listings

A.1 C code for profiling using the PMU ............................. 51
B.1 C code for collecting address usages ......................... 53
Preface

This thesis is comprised of two parts, one about DAEDAL and decoupled access-execute (DAE) applied in an industry setting at Huawei and a second smaller project about an improvement to extended data-race-free (xDRF) analysis using conflict isolation.

There are several reasons I include both projects in a thesis that would normally only focus on one. First, I have previously been involved in the development of both DAEDAL and xDRF analysis and have an interest in seeing both projects progress. Second, there is a potential synergy between the projects in that xDRF analysis could be a useful tool if applying DAEDAL (or a DAE based successor) to synchronised parallel code. However, this connection is not explored in this thesis. Third, the xDRF project provides a clearer connection to issues regarding parallelism, the main topic of my Master’s programme, than the DAEDAL project does. Finally, I needed something to work on (the xDRF analysis) while waiting for some practical details of the DAEDAL cooperation project between Uppsala University and Huawei to be sorted out.

While working on my thesis I have had help and support from many people.

I would like to thank Alexandra Jimborean and Stefanos Kaxiras for preparing and suggesting the cooperation project with Huawei to me. I would also like to thank them, as well as Alberto Ros and Kim-Ahn Tran at Uppsala University, for providing help and advice to interpret results and find solutions when I have faced problems.

I would like to thank Andreas Hansson, Tommy Deng, 张有志 (Zhang Youzhi), and 毕波 (Bi Bo) at Huawei for taking an interest in DAEDAL and making the cooperation project possible. I would particularly like to thank Andreas for working very closely with me and providing expert help and guiding me through Huawei’s base station software. I also owe thanks to several people at the Huawei’s Shanghai offices for providing assistance and useful information, not the least 朱小婷 (Zhu Xiaoting) who has been of great help when dealing with lab equipment. Furthermore, I appreciate the team at Huawei’s Kista office who have provided a welcoming and professional work environment.

Finally, I would like to thank family and friends for all encouragement and support they have given me when I have struggled to progress with the work on my thesis.

Per Ekemark,
Stockholm,
November 2018
Part I

Application of DAEDAL in Huawei’s Base Station Scheduler
Glossary

Abbreviations

**CAE** coupled access-execute. A description and denotation of normal code where memory accesses and computations are mixed, or coupled, in the instruction stream, as opposed to DAE.

**DAE** decoupled access-execute. A technique and denotation of code where memory accesses, in particular cache misses, are separated, or decoupled, from computations in the instruction stream.

**DVFS** dynamic voltage and frequency scaling. A power management technique where the voltage and clock frequency of a CPU are continually adjusted to match the current computational need or thermal limit.

**EDP** energy-delay product. A measure of energy efficiency computed by multiplying the energy and time required to execute a task. Useful as a trade-off measurement between the factors as a change in one often implies an opposite change in the other.

**IP** internet protocol. A protocol for transferring packets over a network. Resides in Layer 3 of the OSI model.

**IPC** instructions per cycle. A measure of throughput of instructions in a processor. The theoretical maximum is equal to the number of instructions the processor is capable of issue every cycle but the actual value is typically lower due to biased use of functional units and memory stalls.

**IR** intermediate representation. The storage format of a program used by a compiler while manipulating (e.g. optimising) code. In the context of LLVM this is an assembly-like representation with target specific details abstracted away. The IR can be stored on file in human a readable text format (.ll) or in a compact bitcode format (.bc).

**LTE** long-term evolution. A high-speed telecommunication standard, superseding previous 2G (GSM/EDGE) and 3G (UMTS/HSPA) standards.
Glossary

MPKI misses per kilo instruction. A measure of the frequency of cache misses in a program. An indicator of how memory or compute bound a program is or how well a program utilises the cache. Often described per 1000 (kilo) instructions but may occur with other or no prefix if it aids readability.

OSI open systems interconnection. The OSI model is a conceptual model describing standard communication functions in computer systems. The model divides the components of communication systems into abstraction layers, building on each other for increased complexity. The model is maintained under ISO/IEC 7498-1.

PMU performance monitor unit. A component of many modern processors with the task of monitoring and counting hardware events that occur in the CPU.

RPKI refills per kilo instruction. A measure of the frequency of cache refills in a program. Similar to MPKI but RPKI may be lower if a single refill can serve several misses.

RTOS real-time operating system. An operating system designed to run real-time applications with some guarantees on timing.

TTI transmission time interval. In telecommunication systems, the time required to transmit one encoded block of data over the radio link. In the LTE this is 1 ms.

UE user equipment. A term used in some telecommunication systems, notably LTE, to describe any end-user device connected to the network. Quintessentially this is a mobile phone but it could also be a laptop, car, remote sensor, or any other device connected to the network.

Terms

access phase The access part of a chunk where prefetches are inserted and as much as possible of the code that does not support the prefetches are removed. The access phase typically has to be side-effect free.

bitcode The compact binary format of LLVM IR stored in .bc files.

cache line The smallest unit of data a cache can replicate from higher levels in the memory hierarchy.

cell Used in telecommunication systems to describe the domain of a logical antenna.

chunk The inner loop created for a DAE targeted loop which can be adjusted to run for a maximum number of iterations according to granularity. Alternatively, the process of separating a loop into such inner and outer loops.

Daedal An add-on to LLVM that applies DAE optimisation to programs.
Glossary

**execute phase**  The execute part of a chunk that is a representation of the original code from before applying DAE.

**granularity**  A measure of the number of iterations a chunk is limited to. A high value, or coarse granularity, allows the chunk to run more iterations than with a low value, or fine granularity, before being forced to exit to the outer loop.

**indirection**  A load of an address to be used by another load.

**indirection level**  In the context of a prefetch or load, a measure of how many other loads are required to produce the address for the prefetch or load. In the context of an access phase, the maximum indirection level of any one prefetch in the access phase.

**Layer 2**  The data link layer of the OSI model which manages communication traffic with directly connected nodes. The implementation of Layer 2 is a central part in LTE base stations.

**LLVM**  A compiler suite with modular support.

**miss**  In the context of caches, a miss occurs when the processor attempts to access a piece of memory and finds it is not in the cache. This results in a lookup in the next level of the memory hierarchy, potentially the main memory.

**pass**  A modular compiler stage that can be used by the LLVM optimiser.

**phase**  Can refer to either access phase or execute phase.

**prefetch**  An operation in a processor moving data from main memory to the cache without assigning the data to a register.

**refill**  In the context of caches, a refill is the act where the processor moves a piece data from a higher to a lower level in the memory hierarchy, thus placing it closer to the processor. A refill is typically the resolution following one or more cache misses to addresses within a single refill quanta, e.g. a cache line.
Chapter 1

Introduction

Huawei is one of the world’s largest telecommunication companies, active in the whole spectrum of the field with phones for consumers and base stations and other network solutions for carriers. While on different ends of the spectrum, both sides provide the challenge of striking a balance between high performance—to enable demanding applications or serving high bandwidth to a large number of users—and low energy expenditure—to save battery life or meeting limits on power delivery and cooling in challenging locations. The path to improvement requires continual research into better energy efficiency.

Dynamic voltage and frequency scaling (DVFS) is a technique for improving energy efficiency by lowering voltage and frequency in a processor to save energy when full performance is not needed or cannot be utilised. However, DVFS is too slow to work on small time scales, such as when an instruction misses in cache and forces execution to stall while waiting for memory. DAEDAL [13] has been proposed as a tool to improve the reach of DVFS through decoupled access-execute (DAE). This technique generates from a targeted section of code an access phase containing chiefly memory instruction that pre-loads the cache in order to eliminate cache misses in an ensuing execute phase. These phases provide larger sections in the program with similar characteristics, allowing DVFS to save energy in the access phase where full performance cannot be used and run the execute phase efficiently at full speed.

Huawei have a large spread of base stations for the current generation LTE telecommunication network and are one of the leaders in the development of the next generation 5G network standard. With the new standard, the timing tolerances become tighter, performance requirements higher, all while on a power budget that must meet power availability and cooling capacity requirements for deployment in the field. DAEDAL is proposed as a part of the solution to achieve improved energy efficiency in Huawei’s base station software to allow for higher performance while maintaining the power budget.

In this project, DAEDAL is evaluated in a version of Huawei’s LTE base station software. DAEDAL is applied to a part of the scheduler in the software, where compatibility issues are investigated and performance is measured on base station hardware. While DAEDAL show some potential, there are outstanding issues that prevent improvements in energy efficiency for the targeted part of the scheduler.
Figure 1.1: **Base station timing pipeline.**
Timing diagram showing the path and timing of data in the base station pipeline. Data enters as IP traffic from the carrier managed core network and goes through the *downlink* via Layer 2 (manages repackaging and scheduling of data) and *Layer 1* (generates radio signal patterns) before arriving at clients. Return data takes the reverse path through the *uplink* where telemetry is also collected to serve as basis for new scheduling decisions.

### 1.1 Background of LTE base stations

In LTE networks, the base station serves as the bridge between the carrier core network and the client user equipment (UEs). The base station implements both Layer 2 and Layer 1 of the OSI model. Layer 2 keeps track of the connected UEs, repackages data from higher layers according to the link standard, and decides which UEs are allowed to transmit or receive when and on what frequency sub-bands. Layer 1 is responsible for modulating and transmitting the link packets from Layer 2 to the UEs over radio.

Figure 1.1 shows an overview of the processing tasks going on in an LTE base station. The process is divided into time slots or *transmission time intervals (TTIs)*, which is the time unit during which one UE can be serviced on any particular frequency sub-band. In LTE, each TTI is 1 ms long, a design choice influenced by several factors, including scalability, flexibility, network latency, and radio utilisation properties. Due to the recurring deadline, equipment must be designed to closely follow the schedule, not least the base station which coordinates the communication. A flow cycle (marked in red in figure 1.1) consists of eight TTIs and allows the base station two TTIs to (i) make a scheduling decision ($Sch$) based on pending data in downlink buffers, the data’s priority, and current radio conditions, (ii) pack data from the buffers into link packets ($RLC, MAC$) along with some control channel data, including a schedule for uplink transmission from the UEs, and (iii) hand the link packets to Layer 1 ($PHY$) for transmission. Next, the UEs are allowed four TTIs to receive the signal, do necessary processing, and transmit uplink data according to the schedule they received earlier. Finally, in the last two TTIs the base station receives the uplink transmission from the UEs (not necessarily the same that was active in the downlink transmission), checks acknowledgements and error corrections ($HARQ$) to tell if a retransmission is necessary, and starts to make a new uplink schedule for the next flow cycle. The whole flow cycle process is pipelined such that every step of the cycle is performed in every TTI.

Many aspects of the base station are standardised within the industry to ensure interoperability between equipment from different manufacturers. This is particularly true for visible time critical events and encoding and error correction formats. As a
result, some parts of the base station construction offer less opportunities for custom solutions than others. An exception is the scheduler which is largely vendor specific, where a good algorithm can set a manufacturer apart in terms of network performance.

Radio access is divided both in time, as TTIs, and frequency, on different sub-bands or resource groups. It is the schedulers job to assign these radio resources to different UEs. In each TTI, a decision has to be made where the scheduler allocates the available resource groups to one or more UEs (or none if there is no pending traffic). For the best network performance, the scheduling decision has to take into account quality of service guarantees and fairness for data waiting to be transmitted to ensure timely delivery and and the current radio conditions to minimise the risk of retransmission. It also has to strike a balance between being advanced enough to avoid predictable network errors and fast enough to deliver scheduling decisions every TTI.

Huawei’s scheduler handles different cells (approximately logical antennas) a base station is operating separately. The scheduling is split on a set number of processor cores. If a more complex scheduling algorithm would be needed or the scheduling would have to be done quicker (for example with a 125 µs TTI in future mobile network standards), it may be necessary to allocate more cores to the scheduling task. This could be a problem with regard to the power budget for the base station, as such, technology improving energy efficiency is important.

1.2 Background of DAEDAL and DAE

Dynamic voltage and frequency scaling (DVFS) is a useful tool for improving energy efficiency of processors. When full performance is not needed or cannot be used due to other bottlenecks, voltage and frequency can be lowered to decrease power consumption. Voltage has the bigger potential impact due to its quadratic relationship with power, however, modern processors already use voltages close to the operating threshold of silicon transistors. Thus, in an effect known as the *end of Dennard scaling* [3], the available voltage range has shrunk, making voltage scaling less useful than it once were. This puts more emphasis on the frequency scaling component of DVFS and its linear relationship to power.

One type of bottlenecks where DVFS can be useful is when memory access misses in cache, incurring a stall while the main memory services the request. DVFS is, however, too slow to react to individual stalls and is therefore only applicable in parts of a program where stalls are frequent. To improve the reach of DVFS, Koukos et al. [12] proposes to combine it with a technique called *decoupled access-execute (DAE)*. The idea is to partition a *coupled access-execute (CAE)* program, where (potentially stall inducing) memory accesses and other computations are interleaved, into segments and preface each of them with a stripped down copy of themselves, mainly containing memory accesses and address calculations, in order to pre-load the cache. Each segment is split into two phases: a memory bound *access phase* incurring numerous stalls while the cache is pre-loaded; and a compute bound *execute phase* rarely experiencing stalls as most memory accesses will hit in cache. The phases exposes larger consecutive parts of the program
Chapter 1. Introduction

Figure 1.2: **DAE concept.**

The example shows a situation in CAE code (top three charts) where a number of loads intermingled with computations are missing in cache and causing miss penalties and stalls. Three loads with stalls are visible in full but only the first (leftmost) is drawn. The three CAE charts are showing the behaviour of the program under different clock frequencies. The DAE chart (bottom) is showing the same example under decoupled execution. First an access phase with only the loads at a low frequency, pre-loading the cache, then an execute phase with the full program segment at a high frequency, running faster as the loads now hit in cache.

where DVFS can be used to lower frequency for access and increase it for execute.

Figure 1.2 visualises the issue. In the standard CAE (or coupled execution) mode (top three charts) there are three loads (only the first on the left is drawn in detail) causing miss penalties as they miss in cache. Some of this miss penalty can be used to perform computations that do not depend on the result of the load, but when there are no more independent instructions in the instruction buffer a stall is generated. Depending on how many independent instructions there are, the stall can be longer (centre) or shorter (right). During the stalls, power is spent unnecessarily, waiting. Using DVFS to set an intermediate frequency instead of the highest will slow down computation but not (or only negligibly) the miss penalties, closing some of the stalls (right) but, due to the variable stall length, only shorten others (left and centre). This will lower power consumption, due to the lower frequency (and voltage), without lowering performance, assuming all computations are still performed during miss penalties, but some power is still wasted during the remaining stalls. Going further to the lowest frequency, there is a risk of having computations overshoot the miss penalties. This will lower the power consumption even further but also incur a performance loss.

The idea of DAE (bottom chart) is to run the program segment twice, first stripped down to only loads and necessary address calculations, then again with all instructions, loads and computations. The initial access phase fetches data to the cache while using a low frequency. Stalls may still occur if there are internal data dependencies or the memory queue gets full, but power will only be spent in vain at a low level. The following execute phase is run at a more power intensive high frequency but will finish faster as the load instructions will hit in cache.

As a concept, DAE was introduced in 1982 by Smith [15] as an architecture with
separate functional units for access and execution, each taking a separate stream of instructions, with the purpose of improving performance over single stream instruction systems. In other words, this is a type of an asymmetric multiprocessing system. As discussed, Koukos et al. [12] take a different approach with the DAE concept by running the access and execute phases sequentially in combination with DVFS to improve energy efficiency. This work is continued by Jimborean et al. [7] by automating the previously manual access phase generation, with different approaches for affine and general code. Ekemark [4] expands on the automatic access phase generation for general code by adding a tuning option making it possible adjust how aggressively data is pre-loaded into the cache. By adjusting the maximum allowed depth of indirection for loads in the access phase, multiple differently tuned versions are generated. Conceptually, the best version can be determined and selected at runtime. With some further tuning this line of work ultimately results in DAEDAL [13] which is the basis for this project.

The rest of this background gives a basic description of how the access phase generation is performed in DAEDAL (section 1.2.1) and how multiple versions are generated based on indirection level (section 1.2.2). Further details can be found in the works of Ekemark [4] and Koukos et al. [13].

1.2.1 Access phase generation

While DAE as a concept is applicable on a whole program, in practice it is more useful to target hot loops. This way only a small part of the program code have to be analysed and partially duplicated in order to have a large impact on the execution of the program. For this reason, DAEDAL is designed to generate access phases for manually targeted loops only.

When a loop is targeted for DAE transformation by DAEDAL, there are three distinctly recognisable alterations taking place. (i) The loop is segmented into several chunks. This is done by enclosing the body of the targeted loop into an inner loop. Besides terminating on the same condition as the outer loop, the inner also counts the number of iterations it has performed and quits to the outer loop once it reaches a set value, known as the granularity. The point of dividing the loop into chunks is to have a way of adjusting the amount of loaded data to fit in the cache, typically L1, in order to have an access phase as large as possible without cache collisions. (ii) The inner loop is duplicated to run twice in sequence. The former copy is the prototype for the access phase and the later becomes the execute phase. To avoid issues with running code twice, the final access phase must be non-modifying outside of its own scope. (iii) Finally, the prototype access phase is transformed by stripping away instructions not necessary for filling the cache. Primarily, this involves isolating loads and the instructions they depend on and removing the rest. Figure 1.3 illustrates the difference between the original and DAE transformed versions of a targeted loop, as if the changes were applied directly on C source code.

The execution model of a DAE transformed loop is illustrated in figure 1.4. When a target loop is entered, the access phase starts by running the first set of iterations according to the set granularity. Since the access phase is non-modifying (outside of its
Chapter 1. Introduction

Loops targeted by DAE are subjected to three significant forms of transformations: (i) The loop is segmented into chunks formed by an inner loop layer that runs for a maximum of \textit{granularity} iterations; (ii) the inner loop is duplicated with each copy representing the access and execute phase, respectively; and (iii) the leading copy is transformed to produce the access phase by removing code unrelated to memory access.

During execution, a DAE transformed loop starts by running iterations in the access phase. Once the number of iterations set by \textit{granularity} has been run (3 in this example), the equivalent iterations are run in the execute phase. The same procedure is repeated for every chunk until \textit{cond} does not hold. Because the inner loops are also controlled by \textit{cond}, the last chunk may (as in this example) run for fewer iterations.
Chapter 1. Introduction

Figure 1.5: Access phase transformation. The access phase transformation consists of four main steps: (i) The loads in the original body are identified, (ii) the dependencies of the loads are isolated (may include address computations and other loads), (iii) the top level loads in the dependency chains are replaced by prefetches, and (iv) finally, superfluous instructions are removed to form the body of the access phase.

own scope) it will not change the state of the program, the only practical side effect is a prepared cache. This allows the execute phase to run the equivalent iterations with the full, unmodified code as if the access phase did not exist, while missing in cache significantly less. The access and execute phases keeps alternating in the remaining chunks. As the inner loops in the phases also depend on the original loop condition $cond$, the phases may finish early in the last chunk if the change in $cond$ does not align with the granularity.

The core part of the DAE transformation is the conversion of the original loop body to the body of the access phase. This conversion process has four steps, which are outlined in figure 1.5. (i) First, load instructions targeting memory in the non-local scope are identified, loads of local variables do not have to be considered since those variables will die when the access phase ends. The identified loads are the only instructions that are desirable to run in order to pre-load the cache, however, some of the loads require other instructions to compute its address. (ii) Thus, the dependencies of the loads are isolated, these may include general computations or other loads. The later is an example of an indirection. (iii) Then, the top-level loads, that no other instructions depend on, are replaced with prefetch instructions. This has to be done since the loads that nothing depends on are typically optimised out by dead code elimination later in the process, unlike prefetches which are not touched by later optimisation. The prefetches instruct the memory system to fetch referenced memory locations to cache; as a bonus, no CPU register has to be used as the data is not going to be used in the access.
phase. (iv) Finally, superfluous instructions are removed. Any instructions that are not prefetches or dependencies thereof are not needed and deleted, yielding the body of the access phase.

This description of the transformation process is slightly simplified. For example, the description only considers data dependencies while control dependencies, or control flow, must be considered as well. This is done by locating all branch instructions and their data dependencies and allowing them to stay in the access phase as well. Should a load replaced with a prefetch be a dependency of a branch, both the load and the prefetch are kept in the access phase. Any parts of the control flow that is not needed is removed by dead code eliminate in later optimisation steps.

Another detail not covered by the above description is dependencies on aliasing store instructions. If alias analysis determines that there are any stores ahead of a load writing to the same address, that store needs to be in the access phase as well. However, since the access phase needs to be non-modifying outside of its local scope, no stores to memory outside the scope of the access phase can be in it. Should a contradiction arise, DAEDAL disqualifies the targeted loop.

Other details that deserve a mention are:

- DAEDAL is implemented using LLVM [14] and its compiler pass framework. Transformations are performed on the LLVM intermediate representation (IR).
- Loops must be manually target by placing the command `#pragma clang loop vectorize_width(1337)` directly ahead of it in the source code.
- Interprocedural analysis is not supported, thus function calls must be inlined in the target.
- Duplication of the segmented inner loop is performed by extracting the inner loop to its own function, making a copy of the extracted function and adding another call to the copy, performing the access phase transformation on the copy called first, and inlining the calls to the extracted functions again.

1.2.2 Multi-versioning and indirection level

Depending on the target program, pre-loading every relevant memory location may not be the best tactic. Some loads may, for example, require a lot of computations to find the address for or may depend on a voluminous amount of other loads. Whether it is beneficial to prefetch any specific load is a function of the cost of having it and its dependencies (considering any overlap between different loads) in the access phase and the benefit of avoiding a cache miss in the execute phase. Finding the value of this function is, however, a difficult task, especially at compile time when cache behaviour is unknown. Instead, heuristic rules are needed that can give a reasonable valuation of every load.

To limit the reach of the access phase, DAEDAL uses a heuristic rule based on indirections. Each load in the access phase is given a score, known as indirection level, which is the number of other loads the one being evaluated has a data dependency on. Figure 1.6 gives an example of how the indirection level is determined for a statement in C.
Chapter 1. Introduction

\[ x = a[*b+c->d->e[*f]]; \]

\[ t1 = *b; \]
\[ t2 = c->d; \]
\[ t3 = t2->e; \]
\[ t4 = *f; \]
\[ t5 = t3[t4]; \]
\[ x = a[t1+t5]; \]

**Figure 1.6:** Example of indirection level.

The single C statement (top left) loads a value to \( x \), which requires several indirect loads. The indirection level of the load of \( x \) is equal to the number of indirect loads. By separating the statement to multiple with one load per line (top right), it is easier to count. Before the load of \( x \), five other loads are required, giving it an indirection level of 5. Representing the statement as an abstract syntax tree (bottom) makes it even easier to count indirections. (Immediate values (global variables, field names, etc.) in grey, address calculations in red, and loads in blue with corresponding separated statement above left and indirection level above right.)

In order to provide flexibility, Daedal accepts a setting for which indirection levels to allow in the access phase. Any loads with an indirection level above the setting are considered superfluous and removed with the other leftover instructions in step (iv) of the access phase transformation. Thus, setting a high limit will likely include more loads and address calculations, leading to a heavy access phase, while setting a low limit will only include loads with lower complexity and fewer calculations, leading to a light access phase. A heavy access phase will thoroughly pre-load the cache but can duplicate a lot of code, while a light access phase may run quicker but risks targeting only loads that the hardware prefetcher can deal with already.

To find the best setting for the indirection level limit, it is necessary to try multiple versions and choose the one that performs best. For a fully automated approach, multiple versions of the access phase would have to be added to a program and the best be determined and selected at runtime. This feature is, however, not available in Daedal at this time. Instead, multi-versioning have to be applied by compiling multiple copies of the target program, each with a different setting for the indirection level limit. The multiple versions of the program then have to be tested individually to find the best trade-off between heavy and light access phases.
Chapter 1. Introduction

1.3 Contribution

The main contribution of this project is to apply DAEDAL to a part of the scheduler in the Layer 2 component of Huawei’s base station software and evaluate to what degree—if any—DAEDAL can improve energy efficiency. This involves the high level goals of

- investigating the code of the Layer 2 scheduler to find suitable sample targets,
- integrating DAEDAL into the compilation process of the base station software,
- investigating and solving compatibility issues, primarily using workarounds,
- measuring performance and estimating changes in energy efficiency compared to the original characteristics, and
- looking for optimisation opportunities to improve the results.

In the line of carrying out these goals, a number of practical secondary tasks also have to be completed:

- Creating or customising support for measuring performance using the ARM performance monitor unit (PMU),
- adapting DAEDAL to work on only a part of a program rather than a compiling a full project,
- adding support in DAEDAL to trace memory accesses in targeted code,
- creating a tool to analyse the memory access trace, and
- performing minor general adjustments to DAEDAL.

To clarify, the project does not involve any major changes to how DAEDAL works. The product of this project should also be considered a prototype, partly because the application of DAEDAL is not exhaustive, partly because DAEDAL itself is in a prototype stage as it depends on unavailable software and hardware pertaining to rapid control of DVFS.

There is one fundamental difference between the base station software and the benchmarks DAEDAL has been developed and tried against previously: The base station software is in principle a program that operates in an infinite loop with certain constraints, i.e. TTI boundaries, while previous benchmarks have been finite without any particular deadlines to meet. While a never ending loop should not be an issue in itself, deadlines may be, depending on how their size compares to chunks.
Chapter 2

Method

In principle, applying DAEDAL is a simple procedure: place the command

```
#pragma clang loop vectorize_width(1337)
```

ahead of each loop that is to be targeted and compile. However, in practice there are usually additional details to take into consideration. In order to target any loops at all, the appropriate ones have to be identified which may require some performance analysis. During compilation, values for granularity and indirection level have to be selected, and in order to find the optimum, several variants have to be explored.¹ In addition, DAEDAL has certain limitations and requirements on the target code in order to work properly. Any issues that arise as a result of this have to be dealt with as they appear, either by switching target loops or by changing the loop to abide by DAEDAL’s limitations. Finally, the target program, Huawei’s base station software, does not fit the compilation template provided by DAEDAL. Thus, a custom solution is required to interface DAEDAL with the existing build system of Huawei’s base station software.

To apply DAEDAL to the Huawei base station software, it is necessary to go through all the above steps. To begin with, a technique to find suitable target loops has to be developed and used in order to find susceptible parts of the software (section 2.1). This requires a way to measure performance and a test case to serve as a benchmark. Once target loops have been found and marked with the targeting command, the next step is to construct a compilation procedure for DAEDAL that can be incorporated into the existing build system for the base station software (section 2.2). The general idea is to adapt the DAEDAL compilation procedure to work on parts of programs (i.e. objects) rather than complete applications. This way, only relevant parts of the base station build system have to be altered to accommodate DAEDAL, the rest can stay intact. Apart from the integration, DAEDAL itself also receives some minor improvements (section 2.3).

After the application of DAEDAL and initial performance evaluation (section 3.3) it is apparent that there are some issues. The problems are investigated by analysing the

¹In a recent thesis, Jin [8] explores ways to automate the selection of granularity and indirection level through feedback compilation.
memory access behaviour (section 2.4) which leads to certain workarounds (section 3.4) and optimisations (section 2.5).

Finally, to evaluate the performance of the targeted code it is important to not only look at execution time. The point of DAEDAL is to improve energy efficiency, therefore a metric which reflect this characteristic is needed. For this purpose, energy-delay product (EDP) is a useful measure (section 2.6).

2.1 Finding DAEDAL targets

Finding targets to apply decoupled access-execute (DAE) to, using DAEDAL, involves finding a loops with suitable properties. Since the point of DAEDAL is to improve energy efficiency in code that performs sub-optimally in that regard, DAEDAL should ideally be applied to code with certain characteristics that can be improved by DAE. Section 2.1.1 describes these traits and why they are needed for DAEDAL to work well.

To determine the characteristics of a loop, some sort of profiling tool is needed. For this purpose, performance counters are used to collect runtime statistics. Section 2.1.2 describes how this works.

2.1.1 Ideal target properties

There are three major properties to look for in prospective target loops for DAEDAL: (i) how hot they are, (ii) how often they miss in cache, and (iii) how big the data sets they operate on are.

First, the targets contribution to the overall runtime should be as big as possible. This is to spread the benefits of DAE to as much of the runtime as possible. At the same time it is desirable to keep the code size of the target as low as possible to limit analysis overhead and, since DAEDAL duplicates some of the code when creating access phases, code size increase. Hot loops usually show both of these properties and therefore have potential to be good targets.

Second, targets should have potential for improvement. Since DAEDAL works by essentially concentrating cache misses to the access phase, there must be a problem with frequent misses in cache to begin with. To find out whether this is the case, it is of interest to measure and investigate instructions per cycle (IPC) and misses per kilo instruction (MPKI). Generally, the lower the IPC and the higher the MPKI, the more there is to work with.

Finally, the size of the data sets the targets use should be considered. As the idea of DAEDAL is to fill the L1 cache in the access phase and use it in the execute phase, the data set of the whole target should be big enough to fill the cache, potentially several times over, while the data set of the smallest divisible unit of the target, i.e. one loop iteration, should be small enough not to overfill it. As long as these properties are fulfilled, granularity can be used to adjust the data set of each chunk to fit the cache.
Chapter 2. Method

2.1.2 Profiling the base station software

The base station software runs on specialised hardware based around a cluster of ARM Cortex-A72 [2] processors (see section 3.1.1 for more information on the platform). There are a few options for profiling but what has been working best is a feature in Huawei’s runtime environment which allows access to the performance monitor unit (PMU) available in ARM processors.

The PMU can be set up to count the occurrences of up to six different event types for each core individually. Most relevant here is to count elapsed clock cycles, executed instructions, and cache refills (approximately cache misses).

While the Huawei software already implements way of using the PMU, some of the details are not good enough for the profiling need at hand. Therefore, two new functions, `PmuGetStartAcc` and `PmuGetEndAcc`, are implemented (with inspiration from pre-existing functions). The chief difference is that the new functions access the PMU directly instead of through function calls, accumulate results if they are used more than once (e.g. in a loop), and can separate the results between different monitored code regions. A simplified version of the source code implementation is available in appendix A.

These new functions work by placing them in source code around the function or loop that should be monitored. `PmuGetStartAcc` starts the PMU counters while `PmuGetEndAcc` stops the counters and adds the result to an accumulated total score. An index supplied to the latter function determine which score bank to accumulate the results in such that results from different code regions can be told apart. Figure 2.1 shows how the functions would be placed in source code to monitor different code regions.

To find targets with appropriate properties to be DAEDAL targets, the components of a high level function is profiled. Based on the results (high execution time, low IPC, etc.) interesting parts are iteratively investigated on a finer scale until loops with a reasonable scope are found. See section 3.2 for more specific details on the target selection.

```c
void profile_me() {
    PmuGetStartAcc();
    int space = setup();
    PmuGetEndAcc(0);
    PmuGetStartAcc();
    while (space > 0) {
        ...
    }
    PmuGetEndAcc(1);
}
```

Figure 2.1: Example of profiling using the PMU. The functions `PmuGetStartAcc` and `PmuGetEndAcc` (orange background) placed to monitor the first function call with index 0 and the loop with index 1.

2.2 Integrating DAEDAL

DAEDAL already has a working compilation procedure, which is designed to compile entire projects. While the provided mould can fit smaller projects well, bigger ones with specialised build environments may require a custom solution. In the case of Huawei’s base station software, it is a large project with many files, modules, and switches set at compile time. In addition, it uses a custom version of GCC instead of LLVM which
Chapter 2. Method

DAEDAL relies on. To fully switch to LLVM and DAEDAL would be difficult and error prone, instead a simpler custom solution is employed.

In this project, DAEDAL is only applied to a part of the base station software. Thus, instead of merging the pre-existing build system with DAEDAL’s entirely, DAEDAL’s compilation procedure is reorganised to be able to work on on parts of programs and applied only to the relevant parts of the base station scheduler (section 2.2.1).

The usage of DAEDAL for the base station software is not free from problems. Major issues are the presence of function calls and writes to globally visible memory. These require a few workarounds to make the project compile at all (section 2.2.2).

2.2.1 Processing selective code

Instead of using the DAEDAL compiler for the entire base station software, the DAEDAL compiler is altered to work on parts of programs, i.e. from source code to objects, rather than to executable. The build environment of the base station software is then modified to accommodate the DAEDAL compiler for selected parts of the program. The rest of the base station software is compiled using the pre-existing build environment and linked with the object generated by the DAEDAL compiler.

The illustration in figure 2.2 provides an overview of the compilation procedure where only relevant parts are compiled with LLVM and DAEDAL. The source code files are split into two categories depending on whether they are related to any DAEDAL targets. The files that are not related are passed through the compile and link steps as usual. The files that are related are redirected through the alternate compilation procedure. They are first passed to LLVM’s compiler, clang, to produce bitcode in LLVM’s intermediate representation (IR). The bitcode files are combined into a single module with llvm-link. The DAEDAL tool then applies its optimisations to the module and passes the result to clang a second time in order to produce an object file. The object is linked with the rest of the code in the standard linker to produce the final executable file.

The DAEDAL process, depicted in figure 2.3, is further subdivided into smaller steps. Most of the steps are implemented as compiler passes in LLVM’s modular compiler
Figure 2.3: **DAEDAL process components.**

Shows the flow of intermediate files in the DAEDAL process. Lines represent files, width their approximate size. Grey lines are alternate or optional paths. The custom pre- and post-process blocks are set apart as they are for tailored code external to DAEDAL. **Address collection** is set apart as it is a configurable optional step. **DAE** is set apart as it is the core step of DAEDAL, with **CAE** a reference alternative.

framework. Some of them are inherited from the existing DAEDAL implementation with no or light modification while others are entirely new. The steps break down as follows:

**Custom pre-process [new]** Performs compatibility patches to resolve issues that DAEDAL normally cannot handle. Primarily this means inlining functions into the target code since DAEDAL does not support inter-procedural analysis. This step is tailored to the selected targets. Further information in section 2.2.2.

**Mark** Translates the pragma command inserted in source code to mark targets into other identifiers that are easier to keep track of while processing the code.

**Chunk** Partitions the targeted loops into outer and inner loops. Both loops exit on the original condition but the inner also exits when a counter reaches a limit. A template for the limit constant is emitted in a separate file to be customised and integrated later.

**Extract** Separates the inner loops of the targeted code into separate functions.

**DAE/CAE** The DAE step is the core of DAEDAL and inserts the access phases created with the guide of supplied settings, indirection level being the most important, and plants calls that record performance statistics. The actions of the DAE step are described in figure 1.5 with accompanying text on pages 7 and 8 in section 1.2.1.

Should the build process be configured as such, the CAE step can be used instead, which only inserts calls for performance recording without creating any access phases. The point of the coupled access-execute (CAE) version is to act as a baseline against the DAE version when comparing performance.
Chapter 2. Method

These steps are also responsible for inserting performance measurement code. DAEDAL already has some support for measurements on ARM. However, to offer more details and flexibility, the same functions described in section 2.1.2 are used instead. Calls to the measurement functions are inserted before and after each phase in an equivalent fashion to what is shown in figure 2.4. A unique index is supplied for each target and phase to get separate results.

**Custom post-process [new]** Serves a similar purpose as the pre-process: to allow any custom tailored comparability fixes. More specifically, this step facilitates part of a fix for an issue found in one of the targeted loops. More information in section 3.4.

**Inline phase** The anti-step of the extraction where the DAE optimised inner loop functions (the access and execute phases) are re-inserted into their outer loops.

**O3** Performs a standard round of optimisations (the -O3 preset in LLVM) to remove dead code and clean up artefacts from earlier steps.

**Address collection (optional) [new]** If enabled in configuration, inserts calls to record uses of memory addresses in code targeted by DAEDAL. More information in section 2.4.1.

**Granularity** The final step reads the target granularity from the configuration, fills in the template emitted by the chunk step, and links the constants within with the rest of the targeted code.

### 2.2.2 Adaptations and workarounds

When first trying to compile the base station software, with DAEDAL integrated and a couple of target loops selected, a few issues arose which prevented successful compilation. There are two problems that are identified as the root cause: (i) function calls and (ii) store instructions in the dependency paths for prefetches or control flow instructions. These features are not allowed in the access phase. To allow compilation, the problems are remedied using workarounds tailored to the selected targets. However, the solutions do imply some caveats.
Chapter 2. Method

The first issue, regarding function calls, is rooted in DAEDAL’s lack of interprocedural support. The simplest way of tackling this issue is to inline any functions called from the access phase. This is done by adding the custom pre-process compiler pass which instructs LLVM to inline functions directly and indirectly called by the targets. The functions to be inlined are identified manually and collected in a list coded into the pre-process pass. However, inlining is not always an option (e.g. calls by function pointer, recursive functions, functions not processed by DAEDAL, etc.). In these cases, the called functions are investigated to see whether they are free from writes to global scope and if they are, the calls are whitelisted to tell DAEDAL to accept them. Fortunately, for the selected targets all non-inlinable functions are free from writes or only have acceptable writes (i.e. output to an error or trace log, which is not a problem in practice if it is expected). Drawbacks with these approaches are increased code size in the binary for the inlined functions and unprocessed code (performing the full work as in the execute phase) in the whitelisted functions.

The second issue, about store instructions, stems from a conflict between the required non-modifying property of the access phase and stores to globally visible memory (possibly being used to convey information within or between loop iterations). Since the access phase must be non-modifying, all stores to global memory have to be removed from it. If any such stores are part of the dependencies for a prefetch or control flow instruction, removal may cause unintended behaviour. Normally, DAEDAL would reject targets where this situation occurs, but to allow the selected targets to compile, this restriction is disabled. This alteration can cause said unintended behaviour, but as the alias analysis used to determine the dependency link to the stores often yields inconclusive results (which must conservatively be regarded as positive), the exclusion of the stores may still pass without issues. More robust solutions to this problem have been explored before using transactional memory [20, 13].

2.3 DAEDAL improvements

While not a main objective of this project, there have been some opportunities to make some general improvements of DAEDAL. Most of these are of a “convenience of use” character but one particularly improves the code DAEDAL outputs.

As explained in section 2.2.1, DAEDAL operates on its targets by extracting the inner loop in each target to a function, from which the access phase is built. Later, the access and execute phases are re-inlined into the outer part of the targeted loops. As the compiler extracts the loops, it has to ensure that necessary state can be transferred from the call site to the extracted functions. The extraction mechanisms arranges this by adding code that saves the state to a structure on the caller side and code that loads from the structure on the callee side. When the extracted functions are re-inlined again, the state transfer is no longer needed and can for the most part be removed by general optimisation (i.e. in the O3 step). However, when the DAE step inserts prefetch instructions targeting the addresses of the state transferral memory operations, the general optimiser is not as successful in removing the redundant operations. To eliminate this potential
source of inefficiency, DAEDAL is altered to recognise the transferral operations and refrain from attaching prefetches to them, thus enabling the general optimiser to remove the transferral operations.

Other changes to DAEDAL include (i) functionality to mark instructions with LLVM metadata to enable alternate behaviour in the DAE pass (useful for workarounds in sections 2.2.2 and 3.4), (ii) additional metadata to keep track of the relationships between loads and prefetches outside of the DAE pass (useful for address analysis in section 2.4), (iii) addition of command line switches to simplify control of features in the DAE pass, and (iv) improved accounting and report of processed and rejected prefetches.

2.4 Analysing memory accesses

From initial performance tests (section 3.3) it turns out that the results indicate issues with overhead and inefficacy. To find out what could be causing some of the issues, it is of interest to more closely analyse the programs behaviour. Since DAEDAL is supposed to affect the strategy of memory access, it is natural to look closer at how the program behaves with regard to accessing the memory. For this, some tools are developed to record (section 2.4.1) and analyse (section 2.4.2) the memory access patterns of particularly the access phase but also the execute phase.

There are two concrete benefits to doing this analysis. The first is the ability to investigate possible causes behind inefficacy (section 3.4). The other is that the analysis results can be used to remove superfluous prefetches (section 2.5).

2.4.1 Address collection

The first step of analysing memory accesses is to collect a transcript of executed memory operations and their target addresses. This is done by modifying the program during compilation such that it will keep a record of memory operations it performs and print a list of them. The collection mechanism relies on two major components: (i) a routine that can save a memory address and some metadata when called and (ii) a way to trigger the collection routine at every memory operation.

The collection routine is simple in its construction; it is a function, call it AddrRec, that is implemented in source code and linked with the rest of the program. AddrRec accepts a memory address, as well as metadata about what type of operation the address comes from (e.g. prefetch or load), a location ID that identifies which instruction in code caused the operation, and a source load ID which correlates prefetches with the loads they replace (in case both are left in the access phase), and stores the information to a structure that is printed later by another function. A reference implementation of AddrRec is available in appendix B.

In order to trigger the collection function at the right time with the right arguments, a new LLVM pass is created (described as the address collection step in section 2.2.1). The address collection pass walks through the access and execute phases and inserts calls to AddrRec before each memory operation with the same address as argument along
Chapter 2. Method

Figure 2.5: **Address collection.**

Address collection is facilitated by inserting calls to the collection function, `AddrRec`, before every memory operation. As arguments, `AddrRec(id, loc, type, ptr)` accepts a source load ID (`id`), a location ID (`loc`), the type of operation (`type`), and the pointer address (`ptr`).

with metadata. The transformation is visualised in figure 2.5.

To make it easier to interpret the log of memory accesses, records of other events are also added to the transcript. This includes when target loops, access and execute phases, as well as individual loop iterations are entered and exited. It is, however, inconvenient to find these breakpoints, as well as the correlation between loads and prefetches, in the address collection pass itself. Instead, earlier passes in compilation procedure are modified to insert annotations with metadata in the LLVM IR while the relevant information is more readily available.

### 2.4.2 Address analysis

The second step of analysing memory accesses is to use the transcript, obtained when running the program with calls to the collection function inserted, to generate a summary or highlight various patterns or properties. To this end, a program to sort through the records in the log of memory operations is created. It is constructed with several different analysis profiles, each able to present certain properties in a unique way. The main focus for most of the profiles is to determine whether prefetch instructions are useful. What the individual profiles do and what they can be used for, is described more precisely in the following text.

**Invariable** One interesting property about prefetches in the access phase is whether they are doing something new each time they are called in the target loop. The *invariable* analysis answers this by scanning the log of events and keeping track of the history of the target address for each prefetch instruction between loop iterations and chunk cycles. The analysis displays for each prefetch in which iterations it was executed and whether
Chapter 2. Method

it accessed a new (for that specific prefetch) address or if it had been visited before in a previous chunk, iteration, or even in the same iteration (in case of a nested loop). The analysis profile can investigate loads in a similar manner and pair them with their relative prefetches using the source load ID.

Should a prefetch be found to always access the same address, the prefetch can be considered unnecessary since that target address is expected to be maintained in the cache by the execute phase, with at most one initial miss. The reasoning behind this assumption is that the granularity is set such that the workload of each chunk fits in the cache.

**Execprof** The execprof analysis is similar to the invariable analysis except that it focuses on the execute phase. Naturally, it investigates loads rather than prefetches, and it can observe stores as well.

This profile mainly serves as a comparison baseline for the invariable profile. Such a comparison is useful when trying to spot unexpected differences in behaviour between the access and execute phases.

**Unused** As suggested, the unused analysis checks for each prefetch how many of the addresses it fetched were actually used in the execute phase. If the addresses targeted by a prefetch are not used in the execute phase, then the prefetch must be superfluous.

Another possibility if any unused prefetched are found is that the access phase behaves differently from the execute phase, which could mean that there are some bug that requires further investigation.

**Stores** The stores profile gives insight into potential read-write conflicts. The analysis simply reveals any prefetches or loads in the access phase that uses the same address as a store in the execute phase. This analysis can flag legitimate cases (e.g. when the write occurs after the load and the address changes between iterations) which means further investigation is required to determine actual errors.

**Redundant** A trivially redundant prefetch is one that remains in the access phase alongside the load it is based on, and is thus not necessary since the load is doing the same job. Daedal can already detect and remove such prefetches if the load remains in the access phase because another prefetch depends on it. With the source load IDs in place it would also be simple to create a compiler pass, to run after general optimisation, that removes redundant prefetches if their loads are kept in the access phase because of dependencies from control flow instructions.

The redundant analysis lists occurrences of executed prefetches and loads with the same source load ID.

**Overlap** The overlap analysis groups prefetches and loads based on the addresses they use. Each group represents a cache line and a number of following lines. This emulates the fetch quantum of the memory system. The cache line size and how many lines that
makes up the fetch quantum are configurable. The purpose of the overlap analysis is to identify prefetches that are implicitly made superfluous by other (potentially logically unrelated) loads or prefetches.

2.5 Feedback prefetch filtering

While constructing the access phase, DAEDAL’s DAE pass inserts prefetches alongside global loads at or below the indirection level limit. Some of these prefetches may be superfluous in the sense that the data they are supposed to fetch to cache is already there or will be fetched by some other instruction in the access phase. Since prefetches in the instruction stream necessarily add at least some overhead as they are decoded, removing superfluous prefetches should improve the performance of the access phase without affecting the impact on the execute phase.

One class of these superfluous prefetches are trivially redundant; they coexist in the access phase with the loads they are created from. Some of these redundant prefetches are removed by the DAE pass, the rest could, thanks to the introduction of source load IDs as metadata in the LLVM IR code, be removed by a hypothetical filter pass after optimisation with dead code elimination.

Other superfluous prefetches are more difficult to find statically and may require runtime analysis to spot. For this purpose, the address analysis from section 2.4 can be used. With the feedback from the analysis, a procedure to recompile the program, with superfluous prefetches removed, can be constructed (section 2.5.1). In order to know precisely which prefetches to remove, various analysis profiles are used to produce a list of sacrificial instructions (section 2.5.2).

2.5.1 Feedback filter

To produce an improved version of the executable, based on analysis feedback, the program must first be compiled in the usual manner, as described in section 2.2, with the address collection step enabled. The executable must then be run with an appropriate test case to produce a transcript of performed memory operations. The transcript is then analysed and processed to produce a list of source load IDs for the prefetches that are superfluous and can be removed. Using a copy of the IR code output from the O3 step (i.e. just before the address collection calls are added), a filter procedure removes the prefetch instructions with IDs listed in the output from the analysis. The filtered code is fed back into the optimisation step (to exploit any new optimisation opportunities following the removal), after which the compilation procedure continues as usual until a new executable is produced. This procedure is illustrated in figure 2.6. The program can be incrementally improved by repeating the address analysis (with different settings) and filter more prefetches from a copy of an already enhanced version.

When running the executable to generate the address collection transcript it is important that the address collection step has been used to insert the address collection calls. However, these calls inevitably add an overhead that will throw any performance
Chapter 2. Method

From figure 2.3

Figure 2.6: Feedback prefetch filter.
The flow of intermediate files in the process of filtering superfluous prefetches with analysis feedback. The executable generated as shown in figures 2.2 and 2.3 is run to collect a transcript of accessed addresses which are analysed to provide feedback for filtering the previous output of the O3 step. The filtered code is used to generate an optimised executable. The feedback optimisation loop is repeatable.

measurements off. Thus, in order to compare the versions before and after feedback filtering, a second variant of the executables, without address collection calls, have to be used to measure performance for the different versions.

2.5.2 Filter rules

During the address analysis where the list of source load IDs to remove is produced, four analysis profiles are interesting to use: invariable, unused, redundant, and overlap. While all of these profiles have the ability to point out superfluous prefetches, the unusual profile is the odd one out as it does not identify any prefetches of that category to remove. Thus, only the other three make any practical difference for the feedback optimisation.

As for the other profiles, they each identify superfluous prefetches according their descriptions in section 2.4.2. The invariable profile identifies which executed prefetches repeatedly access the same address. The source load IDs of prefetches that repeat addresses between chunks or iterations are added to the filter list. The redundant profile identifies prefetches that are executed alongside their corresponding load instructions and add their source load IDs to the filter list. Finally, the overlap profile determines which cache lines that are accessed during execution by which instructions and in what order. This analysis is set to assume that a cache miss (i.e. the first observed access) fetches (i.e. accesses) the missed and the next cache line (by address). The source load IDs of prefetch instructions that access a cache line second to another instruction in the same iteration are added to the filter list.

The generation of incrementally improved versions of the program is split into three
Chapter 2. Method

stages where each successive variant builds on the code and analysis feedback of the previous. Stage 1 is simply the initial version compiled without any feedback. Stage 2 removes prefetches based on the invariable analysis profile (and in principle the unused profile as well but as mentioned it does not make any difference). Stage 3 removes prefetches based on the redundant and overlap profiles. The feedback filtering is split on two rounds in order get insight into intermediate behaviour with the intention to see if the invariable based filter opens up any new optimisation opportunities. To preserve as many potentially relevant prefetches as possible in stage 2, the compilation procedure for stage 1 is altered slightly to prevent the DAE pass from removing any trivially redundant prefetches it would normally detect. These prefetches are removed using the redundant based filter when compiling stage 3.

2.6 Calculation of energy-delay product

When considering performance characteristics of an application, an important and relatively easy to measure quantity is execution time, or delay. Another relevant metric is power dissipation or energy consumption, which is what DAE is primarily meant to improve. As Gonzalez and Horowitz [5] observe, it is easy to improve energy performance at the expense of increased delay. As such, it is desirable to consider both components to ensure that an improvement in one area is not just shifting negative aspects to the other. Energy-delay product (EDP) is a metric that does this by multiplying energy consumption and time (or a time based performance score as Gonzalez and Horowitz use) in order to weigh them against each other.

To compute the EDP of an application, a measurement for both energy and time is needed. While time can easily be obtained (in the form of a clock cycle count), energy measurements typically require specialised equipment which are not available in this project. The next best thing is to estimate the energy based on execution time and additional information about the processor. Furthermore, since a normalised EDP is more interesting than an absolute, there are additional simplifying assumptions that can be made when comparing to a baseline.

Calculating the normalised EDP, \( EDP_{\text{Norm}} \), for DAE results with CAE as baseline is done by

\[
EDP_{\text{Norm}} = \frac{EDP_{\text{DAE}}}{EDP_{\text{CAE}}} = \frac{(E^L_A + E^H_E) \cdot (T^L_A + T^H_E)}{E^H_{CAE} \cdot T^H_{CAE}} = \left( \frac{E^L_A + E^H_E}{E^H_{CAE}} \right) \cdot \left( \frac{T^L_A + T^H_E}{T^H_{CAE}} \right)
\]

(2.1)

The \( EDP_{\text{Norm}} \) thus depends on the energy, \( E \), and time, \( T \), measurements from CAE and the execute phase (subscript \( E \)) of DAE at high (superscript \( H \)) frequency and access phase (subscript \( A \)) at low (superscript \( L \)) frequency.

In this project, neither energy nor time measurements are directly available. Instead, it is necessary to describe these parameters using the clock cycle count, \( c \), which is
available. For time this is simple as it only requires scaling by the clock frequency, \( f \).

\[
T = \frac{c}{f}
\] (2.2)

Thus, the time component of \( EDP_{Norm} \) can be written as a function of \( c \) and \( f \).

\[
\frac{T_L^A + T_E^H}{T_{CAE}^H} = \frac{c_A/L}{f_L} + \frac{c_E^H}{f_H} = \frac{f_H}{f_L} \cdot \frac{c_A^L + c_E^H}{c_{CAE}^H}
\] (2.3)

The energy component requires a few more steps. Energy is the product of time, \( T \) and power, \( P \).

\[
E = T \cdot P
\] (2.4)

Here the formula for time, equation (2.2), can be reused. For power it is good to be aware that there are two components, static and dynamic.

\[
P_{total} = P_{static} + P_{dynamic}
\] (2.5)

The static power represent the part of the overall power drawn by the processor at all times, while the dynamic power represent the energy that is switched when executing instructions. Static power is by no means a negligible part of the power calculation, but as it is the dynamic component that DAE can affect, only that part is taken into consideration in this project.

The dynamic power is described by

\[
P = P_{dynamic} = \alpha CV^2 f
\] (2.6)

where \( \alpha \) is the activity factor, representing the functional load the executed program is putting on the processor; \( C \) is the capacitance of the processor’s transistor logic, together with the activity factor this represents the effective capacitance that is charged and discharged on average each clock cycle by a particular load; \( V \) is the voltage, the square of which multiplied with the previous factors represents the energy dissipated when charging and discharging the effective capacitance each clock cycle; and \( f \) is the frequency, the reciprocal of the time each clock cycle takes. [6]

Energy can now be described by

\[
E = T \cdot P = \frac{c}{f} \cdot \alpha CV^2 f = \alpha CV^2 c
\] (2.7)

then the energy part of \( EDP_{Norm} \) can be described as

\[
\frac{E_L^A + E_E^H}{E_{CAE}^H} = \frac{\alpha_A^L C (V_L)^2 c_A^L + \alpha_E^H C (V_H)^2 c_E^H}{\alpha_{CAE}^H C (V_H)^2 c_{CAE}^H} = \frac{\alpha_A^L}{\alpha_{CAE}^H} \cdot \left(\frac{V_L}{V_H}\right)^2 \cdot \frac{c_A^L}{c_{CAE}^H} + \frac{\alpha_E^H}{\alpha_{CAE}^H} \cdot c_E^H
\] (2.8)

Note that the capacitance cancels out as it is only hardware dependent and does not change between experiments in the this project.
Chapter 2. Method

Finally, the full expression for $EDP_{Norm}$ becomes

$$EDP_{Norm} = \frac{\alpha_L^L \cdot \left( \frac{V_L}{V_H} \right)^2 \cdot c_A^L + \alpha_H^H \cdot c_A^H}{c_{CAE}^H} \cdot \frac{c_E^L \cdot c_A^L + c_E^H}{c_{CAE}^H}$$

(2.9)

Besides the clock cycle counts, which are measured, there are a number of ratios in this expression. The ratios refer to the differences in frequency, voltage, and activity factor between the access and execute phases and the CAE version.

The frequency and voltage ratios are known as they are part of the execution configuration (section 3.1.1). The ratios for activity factor, however, are hard to reason about without actual power measurements or an accurate power model. In previous DAE work [4, 13] a power model developed by Koukos et al. [12] is used to estimate the activity factors\textsuperscript{2}. The shape of this model suggests that the activity factor is affected by IPC in a linear fashion, however, that is for a completely different processor architecture. It is possible there is a similar relationship between activity factor and IPC for the hardware used in this project, but to get any meaningful activity factor ratios a new power model would be required.

\textsuperscript{2}To be precise, previous work estimates the effective capacitance, the product of capacitance and activity factor.
Chapter 3

Evaluation

The evaluation of the application of DAEDAL in Huawei’s base station software roughly takes place in two distinct phases. The initial goal of the project is to integrate DAEDAL such that a few selected target loops can be tried under DAE optimisation. The first phase of the evaluation thus focuses in part on measuring performance of individual components of the base station software to find suitable target loops (section 3.2) and in part on observing the effects on performance DAEDAL has on the target loops (section 3.3).

Following initial results, focus is given to development of solutions to fix issues with the integration of DAEDAL and improve upon the results through optimisation. Thus, the second phase of the evaluation focuses on the description and evaluation of an issue in one of the selected target loops (section 3.4), using feedback optimisation techniques to improve the results of another target (section 3.5), and considering the combined impact of both speed and energy performance by estimating EDP (section 3.6).

All aspects of the evaluation requires a way to run the base station software in a consistent way such that measurement results for different tuning options and modifications can be compared. The testing setup used to collect result data is described in section 3.1.

3.1 Experimental setup

The base station software is designed to run on specialised hardware, described in section 3.1.1 Information about relevant software is available in section 3.1.2. The base station software has some support for running test scenarios, how this works and is set up is described in section 3.1.3. Finally, the high level procedure of compiling, deploying, and gathering test results is described in section 3.1.4.

3.1.1 System properties

The target device is an embedded system made by HiSilicon for use in Huawei’s telecommunication systems. The specific design is equipped with a cluster of ARM Cortex-A72 [2] multi-core processors. The processors can run at several in a range of frequencies. In this project, a high frequency and a low at about $2/3$ of the high are used.
The core voltage is the same for both frequencies at 0.8 V. Each core has a 3-way L1 instruction cache of 48 KiB and a 2-way L1 data cache of 32 KiB, each A72 unit has a 16-way L2 cache, and the entire cluster has a shared L3 cache. All cache is coherent.

3.1.2 Software and versions

The base station software is built using GCC 4.7.1, configured for armeb, with some Huawei extensions as part of their SDK. For the parts of the software being passed through DAEDAL, LLVM 3.8.1 is used to compile and run the DAEDAL passes. (LLVM itself is built using standard GCC 5.4.0.) The build machine is running Ubuntu 16.04.

The target machine is running a Huawei developed real-time operating system (RTOS) based on the LINUX kernel. The base station software is executed under a Huawei developed runtime environment.

3.1.3 Test case

As already described in section 1.1, the main purpose of the base station and its software (overview shown in figure 3.1) is to receive IP packets from the core network, repack into a format suitable for radio transmission, and pass the new packets of data to the physical layer where it is encoded and relayed to the radio hardware for transmission to user equipment (UEs) (downlink). A similar procedure is performed in reverse for data transmitted from the UEs to the core network (uplink). In the middle sits the scheduler which uses metadata about the network traffic and telemetry information from the UEs to decide how to order and prioritise different data streams with the goal of using the limited radio resource as efficiently as possible.

For testing new features in the base station software there are tools built for it that simulates the environment the system would normally operate in. This test bench (see figure 3.2) emulates the core network interface by sending packets into the Layer 2 downlink. The scheduled and repackaged data intended for the physical layer is intercepted by an emulated analogue that returns the data to the uplink as if the UEs would send back the packets they receive. When the Layer 2 uplink hands over packets intended for the core network, the test bench collects them and checks that all sent packages are received.

The main component in setting up a test case is to specify how the test bench should behave. The test bench exposes settings for the number of cells to simulate, the number of UEs each cell should serve, the number of data carriers each UEs should use, the bit rate and packet length to use when generating data for each carrier, and the test duration in transmission time intervals (TTIs) (one millisecond each). In addition, it is also possible to specify when during the test to conduct performance measurements. Regarding carriers, the base station software supports multiple data carriers using different throughput guarantees, however, the test bench is set up to only use the standard ‘non-guaranteed’ carrier for all traffic. Also note that the test duration is the time during which data is generated and injected into the system by the test bench; it takes additional TTIs for all packets to finish their journey.
Chapter 3. Evaluation

Figure 3.1: **Base station software overview.**
Block diagram overview of the base station software. IP data is received and passed through the Layer 2 to be scheduled and packed. Layer 1, or the physical layer, encodes the data and passes it to antenna hardware where it is transmitted to UEs. Data received from UEs is decoded by Layer 1 and handed to the Layer 2 uplink where it is unpacked and passed on as IP traffic.

Figure 3.2: **Test bench for Layer 2.**
The test bench (red) generates data and submits it to the Layer 2 downlink, emulates the physical layer with UEs by returning received data back to the Layer 2 uplink, and receives data from the Layer 2 uplink to check and count all packets that arrive as they complete their journey.
Chapter 3. Evaluation

When carrying out tests, a single combination of parameters have been used to produce the results:

- 12 cells
- 200 UEs per cell
- 1 data carrier per UE
- 16 kbit/s per carrier
- 200B per packet
- 30 TTI test duration
- measurement window between TTIs 20 to 30 (exclusive)

The scheduler of the base station software is configured to run on multiple of the system’s cores, other parts of the software are mapped across remaining resources.

The test configuration is compiled with expert help in order to achieve a balanced scenario. Some variation of the parameters have also been investigated, though without significant impact to the measured part of the code. More extreme changes have not been possible due to limitations in the test bench.

3.1.4 Testing procedure

When performing a set of tests, the base station software is first compiled by the build system that has been modified to accommodate DAEDAL (section 2.2) which produces a runnable software library. This library is bundled with Huawei’s runtime environment and RTOS in a system image intended for the embedded target device. This is automated by a script that supplies DAEDAL settings, invokes the base station software build system, and generates a system image for the target device. To build an entire suite of tests, this is further automated by a script that repeats the procedure for every desired combination of DAEDAL parameters. The system image versions can then be packaged and transferred to a lab computer connected to the target device.

Once on the lab computer, an automation script is run which opens a communication channel to the embedded target device. The lab computer instructs the device to load a system image and boot into it, then issues commands to start a benchmark test. This is repeated for all compiled images in the batch. A record of the communication channel, which contains relevant result data, is saved such that it can be brought back outside of the lab environment.

As a final step, the communications log is parsed to extract raw data which is then processed to produce convenient output containing raw and derived data points that can be presented in tabular or graphical form.

3.2 Target selection

The search for targets starts in an expert designated function which is a major component of the downlink scheduler. This function is called once for each cell and TTI during a benchmark test.

In the first round of profiling, which involves individually measuring performance on the dozen of functions that the expert function is divided into, it is found that 97.7% of the time is spent in only three of them. A breakdown of the these three candidate functions is shown in table 3.1.
Chapter 3. Evaluation

Table 3.1: Top level target candidates. The characteristics of the biggest top level functions in the expert designated function. Time is presented as a percentage of the time spent in the expert function. Load RPKI refers to refills for the L1 cache caused by load operations.

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Time (%)</th>
<th>IPC</th>
<th>Load RPKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidate A</td>
<td>18.2</td>
<td>confidential</td>
<td>confidential</td>
</tr>
<tr>
<td>Candidate B</td>
<td>39.8</td>
<td>confidential</td>
<td>confidential</td>
</tr>
<tr>
<td>Candidate C</td>
<td>39.7</td>
<td>confidential</td>
<td>confidential</td>
</tr>
<tr>
<td>Other</td>
<td>2.34</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 3.2: Daedal targets. The characteristics of the target loops. Time is presented as a percentage of the time spent in the expert function. Load RPKI refers to refills for the L1 cache caused by load operations. Iterations denotes the average number of iterations the target loop executes each time it is entered (target B always executes exactly 9 iterations).

<table>
<thead>
<tr>
<th>Target</th>
<th>Time (%)</th>
<th>IPC</th>
<th>Load RPKI</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target A</td>
<td>16.3</td>
<td>confidential</td>
<td>confidential</td>
<td>9.96</td>
</tr>
<tr>
<td>Target B</td>
<td>39.0</td>
<td>confidential</td>
<td>confidential</td>
<td>9.00</td>
</tr>
</tbody>
</table>

As explained in section 2.1.1, a desirable target should have a long duration for greater impact as well as low IPC and high MPKI for a bigger potential improvement following optimisation. However, the ARM PMU does not report cache misses, only the similar metric cache refills. Thus, due to availability, refills per kilo instruction (RPKI) is used in place of MPKI. The difference is that the former could have lower values as the refill metric does not include multiple cache misses that are served by the same cache refill.

While candidates B and C have longer execution time, which, would imply a wider impact on the system if targeted for optimisation, they also have high IPC and low RPKI which leaves less room for improvement. The opposite conditions apply for candidate A.

Since the primary goal of this evaluation is to try the efficiency of Daedal, it is more interesting to target code that is likely to benefit from optimisation. With this in mind, candidate A becomes the most interesting function with the runner up B as a secondary option. However, neither of these candidates are directly suitable as targets since they are function calls rather than loops. The final targets have to be loops which can be divided into chunks that operate on datasets that will fit in the cache.

The contents of candidates A and B are profiled further in greater detail. A suitable loop is found directly in candidate A while a few steps down the call tree is required before finding a target in candidate B. The characteristics of the target loops are shown in table 3.2. The call tree describing the relationship between targets, candidates, and expert function in the system is presented in figure 3.3.

To avoid a surprise later on it is worth noting now that although target B has been selected for experimentation it is later disqualified because of compatibility issues (see
Chapter 3. Evaluation

The call relationship between the expert function, candidates, and targets. Observe that the targets are always loops in the parent function. This depiction of the call tree only shows functions on the relevant path, the full tree expands further in other branches.

3.3 Integration performance

With DAEDAL integrated (section 2.2), an experimental procedure in order (section 3.1), and a couple of sample targets (section 3.2), it is possible to start investigating DAEDAL’s effect on performance. To begin with, the optimisation regarding re-inlining of the phases, mentioned in section 2.3, is examined (section 3.3.1). Following this, the available ranges of granularities and indirection levels are inspected with regard to the sample targets in order to find a set of DAE settings with which to measure performance (section 3.3.2). Finally, performance results are collected and reviewed across different granularities and indirection levels (section 3.3.3).

3.3.1 Inlining improvement

After initial integration of DAEDAL, one of the first improvements made is the suppression of prefetching of loads produced in the process of extracting a target loop to a function (section 2.3). The effect of the suppression is that the general optimiser is more successful in cancelling out the generated loads in the extracted function against corresponding writes at the call site once the phases are re-inlined.

These virtual loads are detected at an early stage in the DAE pass and are subsequently removed from the list of loads considered for prefetching. For target A the DAE pass reports 25 virtual loads which is 14% of the loads initially considered for prefetching. The corresponding figure for target B is 100 virtual loads which is 8.7% of all considered. These numbers are independent from granularity and indirection level and show up again later in figures 3.4 and 3.5.

Looking at the concrete effect the prefetch suppression has on the code after general optimisation, there is a clear difference with and without suppression. Without prefetch suppression, the general optimiser leaves 23 of the 25 virtual loads in the access phase and 2 of the 25 in the execute phase for target A. For target B 62–63 (depending on
indirection level) of the 100 virtual loads in the access phase and as much as 99 in the
execute phase are left in the code. For both of the targets the general optimiser does
a little better in the access phase and a little worse in the execute phase at indirection
level 0. There are never any issues (all virtual loads are removed) with the CAE baseline
as no prefetches are ever inserted. With prefetch suppression, the general optimiser
manages to remove all virtual load from both targets at all indirection levels.

With the change to suppress prefetching of virtual loads, the algorithm calculating
indirection level is also altered. The new variant does not count the virtual loads toward
the indirection metric. For this reason it is difficult to make fair comparison between
the two revisions. As a general observation, however, the performance improves by up to
about 5%. In target A the improvement mostly affects the access phase while in target B
both phases benefit. For more details on the performance difference, see appendix C or
more specifically C.3 and C.4.

3.3.2 Choosing DAE settings

The main DAE settings of concern is the granularity and indirection level. How to tune
these properties is not obvious and multiple variants usually have to be investigated
before anything optimal can be found. In parallel with this work, Jin [8] has investigated
the possibilities of automating the search for optimal settings. For this project, however,
a range of settings are chosen manually to be compared.

The granularity determines the number of iterations to run in the phases of each
chunk in a target loop. A good granularity adjusts the chunk length such that the access
phase fills the cache with new data as close to the capacity as possible. Too short and
there will be overhead due to unnecessary frequent switching between phases, too long
and there will be overhead due to collision evictions of data before it can be used in the
execute phase.

An analysis of how much memory is accessed in each iteration of the target loop can
give a hint toward a good granularity. However, the choice of indirection level can affect
how much data the access phase actually fetches to the cache and thus what the best
granularity is. For the targets at hand it is known (from table 3.2 on page 30) that neither
of the selected target loops typically exceed 10 iterations. This only leaves a few viable
options for the granularity setting, allowing investigation of most without performing a
memory analysis first.

For further investigations, the granularities 1, 2, 3, 5, 10, and 20 are used. 1 and 10
because they represent the extremes of having a chunk encompass only a single iteration
and the whole target, respectively. 20 to verify the chunk borders work as expected, this
granularity should behave identically to that of 10. 5 as it splits a target between two
chunks (6–9 would as well but not as evenly). Finally, 2 and 3 to have a few more options
for comparison.

When selecting indirection level the goal is, as described in section 1.2.2, to affect
the amount of code included in the access phase to strike a balance between prefetching
and repetition of computations. What the best level is depends on the characteristics
of the target program. The use of pointers and indirection as well as how beneficial the
memory access pattern is for the hardware prefetcher are examples of factors affecting how well different indirection level settings will do.

To select settings to test, it is useful to start with an overview over the available options. The DAE pass produces an output with statistics of the number of processed loads and how many prefetches were inserted or disqualified for various reasons. By running the DAE pass with increasing indirection level settings, starting with 0 (allowing only prefetches with statically known addresses), until the pass reports that all prefetch opportunities have been exhausted (no prefetches are overlooked on account of the imposed indirection level limit), the level of which varies between different targets, it is possible to collect the outputs and generate a summary from which more interesting indirection levels can be identified.

Figures 3.4 and 3.5 visualises the feedback from the DAE pass across different indirection level settings for targets A and B, respectively. Prefetches are put in one of four categories: virtual if they are disqualified from the access phase for prefetching a virtual load (section 3.3.1); indirection limit if they are disqualified because the prefetch’s indirection level exceeds the setting in use; duplicate if they are disqualified because another prefetch with a trivially identical address has already been inserted (a pre-existing feature in Daedal); or inserted if they pass through the other filters and really are inserted into the access phase. These four categories sum to processed which should be, and is, constant across all indirection level settings for each target. Once all prefetches have been inserted, the DAE pass removes all redundant prefetches that target loads other prefetches depend on. Subtracting the redundant prefetches from the inserted yields an actual number left in the access phase.

The figures also have an alternating shading where a shift indicates a change in any of the plotted values. Following the definition of indirection level (section 1.2.2), a load (or prefetch) with indirection level $x$ does not imply the existence of another with indirection level $x - 1$. For this reason there are occasionally several sequential indirection levels with no change in the reported statistics. Within these groups each indirection level setting produces the same output code.

As performance are measured for both targets at the same time, it is desirable to select a set of indirection levels that are of interest for both. For this reason, levels 0–5, 8, 14, 18, 20, 22, and 27 are used in further investigations. For target A, indirection levels 0–5 cover the interesting settings where many prefetches transition from being over the limit to being included in the access phase. The remaining unique levels are covered by either 8 or 14 and one of the other remaining chosen ones. In target B there are a bit too many distinct indirection levels to test them all. Once again, levels 0–5 covers some of the interesting settings and 8 gives a sample where the changes flatten out. Target B also exhibits significant changes at higher indirection levels, the remaining chosen levels cover this section sparsely.

### 3.3.3 Performance results

With a set of selected granularities and indirection levels, the different versions are compiled and sent to the test system to measure their performance. Figures 3.6 and 3.7
Chapter 3. Evaluation

Figure 3.4: **Target A, prefetch characteristics.**
Shows for different indirection levels the number of processed prefetches inserted into the access phase or disqualified for being either virtual, duplicates, or exceeding the indirection limit. After a second pass of analysis, some prefetches are deemed redundant and removed, leaving an actual number of prefetches. Processed and virtual are independent of indirection level (constant values to the right). Shifting background shading indicate change in any value.

Figure 3.5: **Target B, prefetch characteristics.**
For details, see caption of figure 3.4.
Chapter 3. Evaluation

Figure 3.6: Target A, normalised execution time.
Note: Indirection levels 8 and 14 are code wise equivalent as indicated in figure 3.4, as are levels 18, 20, 22, and 27.

visualises the measured execution time (based on cycle count), normalised to the CAE version at granularity 1, for targets A and B, respectively. Before normalisation, the counts consist of the average from four iterations of the same experiment.

Starting with target A (figure 3.6), there are a couple of things to observe in the chart. First, the introduction of the access phase has given a huge impact on the execute phase. That the execute phase can complete in only 40\% of the time compared to CAE is evidence that the access phase successfully pre-loads useful data to the cache. The backside is that the access phase itself is rather large as it executes in about 80\% of the CAE time, giving a total slowdown of around 20–25\%. The best case is indirection level 2 and granularity 2 where the access and execute phases take 81\% and 38\%, respectively, of the baseline time for a total slowdown of 19\%. This means that in order to gain anything by using Daedal, it is necessary to run the access phase at a lower frequency in order to have any chance at improved energy efficiency. This is discussed further in section 3.6.

The second thing to highlight is that neither granularity nor indirection level appear to impact the execution time to any greater degree. Granularity 2 is often beating the alternatives, likewise indirection level 2 is typically better than the others, but in both cases the margins are very slight. The low variability of the DAE tuning options have different explanations for each dimension.

The rationale behind adjusting the indirection level is to provide a varying set of prefetch instructions that the compiler’s optimiser will not remove. Since the access phase is side effect free, the expectation is that the compiler should be able to eliminate
Chapter 3. Evaluation

most of the remaining code that the prefetches are not dependent on. In the ideal case, the access phase with the highest available indirection level is expected to be as heavy as necessary, i.e. essentially a copy of the execute phase with only the parts not concerning accesses to the global state removed. For the lowest indirection level little more than the relevant prefetch instructions and any supporting control flow instructions are expected, leading to a much lighter access phase. Target A does not present these ideal characteristics. At the highest indirection level the code size of the access phase is about 80% of the execute phase, which is acceptable for a heavy access phase. However, at a low indirection level the code size of the access phase is still about 70% of the execute phase, which is a bit too much, in particular when considering that the prefetch instructions only make up about 5% of the access phase. Clearly, the indirection level options provide some variability in the code size, but only a small one.

There are two prime suspects for why this low variability behaviour is showing: (i) Calls to whitelisted functions could be dependencies of control flow instructions. If this is the case, the dead code elimination optimisation, which is responsible for removing redundant control flow, may not be able to remove a whitelisted call even if the control flow instruction is eliminated. If the arguments to the call depend on loads, the access phase can end up heavier than intended. To solve, the dead code elimination optimiser has to be provided enough information to determine that the call is access only and safe to remove. (ii) Low indirection level prefetches could reside behind control flow instructions. In this case, the dead code elimination optimiser cannot remove such control flow instructions because of the prefetches. If in turn the control flow instructions depend on loads, the access phase can, again, end up heavier than intended. For this, a possible remedy would be to change the definition of indirection level to also include control flow dependencies in addition the data dependencies the indirection level is already based on. Neither of these paths of investigation will, however, be explored further in this project.

Regarding granularity, the expectation is that as it becomes coarser (more iterations in each chunk) the overhead cost associated with switching phase will decrease for both access and execute. However, at some point the granularity may become coarse enough to cause some data loaded by the access phase to be evicted before it can be used in the execute phase. This would negatively affect the execute phase but not have much impact on the access phase. For the case at hand, these effects do show but only to a small degree. While the variations are at the edge of what could be considered noise, the pattern is too persistent between indirection levels to be a coincidence.

Continuing with target B (figure 3.7), the situation is quite different. The execute phase is only modestly affected by the access phase and the access phase itself is fairly short but grows sharply in execution time at certain granularities. Similarly to target A, indirection level have little to no effect on the performance.

There is one thing that is very important to notice above anything else, which is that the access phase at granularity 20 takes significantly longer time to run than the equivalent at granularity 10. However, since it is known (from table 3.2 on page 30) that target B never executes more than 10 iterations, granularities 10 and 20 should be equivalent and have the same execution time. As this is not the case, the probable
Chapter 3. Evaluation

3.4 Incompatibility issues

The shape of the performance graph for target B (figure 3.7), in particular the jump from granularity 10 to 20, is peculiar. Since the number of available iterations is entirely encompassed in a single chunk at granularity 10, granularity 20 should perform very similarly since the phases in the chunk should terminate when the available iterations are exhausted, well before the iteration limit of the chunk is reached. Yet, at granularity 20 the access phase runs for about 50% longer than at granularity 10.

A possible reason for the unexpected behaviour is that the mechanism that normally ends the loop in the original code or execute phase does not work as intended in the access phase. If the exit condition of the loop in some way depends on modifications to the global state, this would be a plausible explanation and a complication of ignoring the warning for conflicting stores.

Figure 3.7: Target B, normalised execution time.
Chapter 3. Evaluation

Priority queue

Elements Priority list

Nodes Head

Figure 3.8: Task queue in target B.

In target B, each loop iteration performs a task. Tasks are held as elements in an array of a priority queue structure. The task order is determined by a priority list which is a linked list with indices into the array of the queue structure.

To find out, a manual review of the source code of the target loop is conducted. A suspect for the issues is found in the task queue for the loop. The structure of the queue is shown in figure 3.8. The queue contains an array of elements, each describing a task to be performed during an iteration. There is also a priority list in connection with the queue structure. It is implemented by a linked list where the nodes carries the index of a task item in the main queue structure and the ordering of the list represents the priority of the tasks.

The problem is that as the loop works through the tasks, the pointer to it is unlinked from the priority list. Because the queue resides in the global state the stores involved in the pop are removed from the access phase. This means that for a single access phase instance, only one task item is processed repeatedly in all iterations.

A general solution to the issue would be preferable, however, this would be rather difficult to implement directly into DAEDAL and is outside the scope of this project. Instead, a patch to fix the problem is deployed for investigative purposes. The patch is implemented as changes to the source code of target B in the base station software. It works by stepping past a number of nodes in the priority list when loading a task in the access phase. For each iteration one more item is skipped (i.e. none for the first iteration, one for the second, etc.). When the access phase starts, the number of nodes to skip is reset as the execute phase would already have popped the items it actually worked through. As this workaround also requires writes to global state, the DAE pass in DAEDAL is modified to ignore store instructions as long as they are tagged with a specific piece of metadata which is added to the relevant instructions in the custom pre-process step (figure 2.3 on page 15). The custom post-process step ensures that the workaround is only performed in the access phase by setting constants introduced in the source code.

38
differently depending on if they are used in the access or execute phase.

With the fix in place and target B able to progress through the loop, the performance can be measured again. As the results in figure 3.9 show, the change has altered the performance characteristics. The execution time of the access phase still grows with coarser granularity, but even sharper than before. In exchange, the improvement in the execute phase also increases with granularity. On the whole, however, the changes cancel out, making the combined result similar to before. This shows that the fix helps with pre-loading more relevant data, however, the difference between granularity 10 and 20 is still present which means that the problem the fix is supposed to solve is still present.

To dig deeper and find the issue, the address analysis tool introduced in section 2.4 is used. The `invariable` and `execprof` profiles of the tool are particularly helpful. By comparing the memory access patterns of the access and execute phases, it becomes clear that the execute phase always (at least under the test case being used) performs eight full iterations and exits early on the ninth. The access phase, however, does not stop at the equivalent point, instead it continues to run iterations until the limit placed by the granularity is reached.

Through correlation between analysis output, LLVM IR code, and base station software source code, using location and source load IDs, the reason the execute phase exits while the access phase continues is found to be related to a conditional that checks whether the scheduling quota for the current TTI is full. Normally this mechanism works by taking a scheduling decision during an iteration and allocating a resource in the global state. If in the next iteration there are no more scheduling resources, the loop exits, otherwise, a new item is processed and a new scheduling decision is taken and resource allocated. Since the access phase is not allowed to modify global state, the scheduling
quota never fills up and the loop continues until the granularity limit is reached.

Another discovery stemming from the use of the address analysis tool is that the access phase is performing significantly fewer memory accesses than the execute phase. Generally, this would be fine at a low indirection level, while at a high, almost all memory access are expected to be accounted for. This is a strong indication that there are more undiscovered issues with target B. Instead of trying to patch all of them, focus is redirected on improving the performance of target A.

3.5 Address analysis optimisation

While target A sees a large performance improvement in the execute phase under DAEDAL, the access phase negates the speed gain and even causes a slowdown. A limited slowdown is acceptable as long as the energy efficiency improves in the exchange. As it stands, target A has a total slowdown of around 20–25% as seen in figure 3.6 on page 35. This is relatively high, making it worth looking for an improvement.

Using the address analysis tool (introduced in section 2.4) to investigate prefetches, it turns out that many of them are ineffective. Different tool profiles reveal different kinds of superfluous prefetches that are not doing any useful work: repeated prefetching from the same address (invariable), prefetches coexisting with their source loads (redundant), and prefetching of already cached cache lines (overlap). As prefetches at the very least add some overhead as they are decoded from the instruction stream, removing superfluous ones should improve the performance of the access phase without affecting the execute phase.

To investigate the performance of target A when run without superfluous prefetches, the address analysis tool is used to inform optimisation of the access phase through the feedback procedure described in section 2.5. This procedure generates three stages of optimisation where each successive version is improved based on the behaviour of the previous.

Stage 1 is the original version with all inserted prefetches (orange line in figure 3.4 on page 34) in the code. To the left in figure 3.10, the analysis results for stage 1 are shown. For all indirection levels, roughly half of the inserted prefetches are ever executed (orange), the rest reside in control flow paths not taken in the test case. Each of the three analysis profiles identify a portion of the executed prefetches as superfluous. Collectively, they account for all executed prefetches, but only the invariable profile is considered when removing prefetches to produce stage 2. For all but the lowest indirection levels this is about half of the executed prefetches.

The analysis results for stage 2 is shown to the right in figure 3.10. The idea of only removing a portion of the superfluous prefetches to give the optimiser the opportunity to convert some of the remaining superfluous prefetches to useful ones did not work. Again, the two remaining analysis profiles collectively deem all executed prefetches superfluous. Thus, stage 3 is created without any of the previously executed prefetches in the code.

Consequently, no prefetches are executed in stage 3, however, the access phase still performs essentially all of the loads as the earlier stages execute. This means that even
Chapter 3. Evaluation

Figure 3.10: **Superfluous prefetches by type in target A.** Shows the number of unique prefetches (by source load ID) that are executed (orange) and identified as superfluous in the access phase of target A. The different superfluous categories partially overlap. The analysis results from *stage 1* and *stage 2* are shown to the left and right, respectively. *Stage 2* does not have the *invariable* category.

*Stage 3* can perform the task of pre-loading the cache, using only loads instead of prefetch instructions.

Figure 3.11 shows the performance results when executing the different optimisation stages. To limit the number of versions, only granularity 2 is investigated since it performed marginally better than the others in the original results. Also worth noting is that the CAE version is the same for all stages since the optimisation only touches prefetches.

As can be seen in the plot, the use of the address analysis feedback optimisation has been a success, despite no actual prefetch instructions being executed in the end. The performance of the execute phase is not affected by the increasingly aggressive optimisation stages of the access phase. At the same time, the access phase is about 10% faster in *stage 3* than in *stage 1*. At *stage 3* there is now only a slowdown of around 10–15%. The best case is indirection level 0 at *stage 3* where the access and execute phases take 74% and 38%, respectively, of the baseline time for a total slowdown of 12%.

### 3.6 EDP comparison

Until now, all performance results have been presented in terms of execution time. This is a useful metric to use in order to understand the behaviour of the measured program. However, the primary goal of DAEDAL is to improve energy efficiency using DAE and dynamic voltage and frequency scaling (DVFS). To evaluate this, energy-delay product (EDP) is a useful metric as it incorporates both time and energy consumption.

Measuring EDP is a somewhat more convoluted process than simply measuring time. As the name suggests, the energy-delay product also requires a measurement of energy
Chapter 3. Evaluation

Figure 3.11: Target A, AAopt, normalised execution time.

consumption to be collected. However, in this project no hardware for measuring energy is available; instead, energy is estimated based on execution time. The downside of this is that some assumptions are needed which hurt accuracy. Refer to section 2.6 for details on the calculation of EDP.

The execution model of DAEDAL is to use DVFS to save energy in the access phase by running it at a low frequency and finish the execute phase quickly by running it at a high. This model requires fast per-core DVFS where frequency, and voltage if possible, can be changed between the access and execute phases. However, the test hardware in this project is not equipped with these specific DVFS capabilities nor does DAEDAL come with any implementation of its DVFS based execution model. Thus, in order to estimate EDP results as if the intended execution model could be used, the EDP is computed using the already presented results for the execute phase at high frequency (figure 3.11) and measurements from a second round of tests at low frequency for the access phase (figure 3.12).

Figure 3.12 shows the performance results when running the same experiments from section 3.5 with a lower frequency, $2/3$ of the high. To make the plot comparable to the high frequency equivalent in figure 3.11, the results are rescaled to the same time unit (taking the different cycle times at different frequencies into account) and normalised against CAE at high frequency.

There are no big surprises when comparing the high and low frequency results. At a glance, the shape of the results are very similar, the execute phase performs stably with little regard to indirection level or optimisation stage, and the access phase generally improves with advancing optimisation stage and takes somewhat longer with increasing
indirection level. The differences only becomes apparent when paying closer attention to the time scale. At a normalised time of approximately 0.6, the execute phase takes about 50% longer to run at low frequency compared to the high. As the low frequency is about $\frac{2}{3}$ of the high, this closely matches the expectation for a compute bound piece of code, which the execute phase is expected to be when paired with an access phase that thoroughly pre-fills the cache.

The access phase itself, however, only takes about 25% longer at low frequency. This indicates that the access phase, as desired, fills more of the time waiting for a memory fetch with computations to offset some of the performance loss due to the lower core frequency. Some performance loss is expected for the access phase when lowering the frequency, owing to any necessary compute bound code segments, but how big the loss is depends on the ratio between, and interleaving of, memory and compute bound code sections. It is already established that there are some issues with generating a light access phase for target A (section 3.3.3). Given solutions to the issues, it is possible that a lighter, more memory bound access phase could be created, especially for the lowest indirection levels. Such an access phase could have a lower performance loss when lowering the frequency. However, a lighter access phase might also have a weaker impact on the execute phase.

Ultimately, neither the high nor the low frequency results are independently representative of the intended execution model for DAEDAL. To emulate results under that model, figure 3.13 shows combined measurements for target A by using execute phase
results from experiments at high frequency (figure 3.11) and access phase results from experiments at low (figure 3.12). The combined results show that there is a performance overhead in terms of execution time upwards of 50% compared to CAE, with the best case (indirection level 0, stage 3) at an approximate 30% overhead. In general, and in the context of DAEDAL results in previous work [4, 13], this is a rather big overhead which is not indicative of good energy efficiency figures.

The energy efficiency is evaluated using EDP as described in section 2.6. As a reminder, the normalised EDP is computed using the formula

$$EDP_{Norm} = \frac{\alpha_L^T \cdot \left(\frac{V_L}{V_H}\right)^2 \cdot \frac{c_L^L}{c_{CAE}} + \alpha_H^H \cdot \frac{c_L^H}{c_{CAE}} \cdot \frac{f_H}{f_L} \cdot \frac{c_L^L}{c_{CAE}} + \frac{c_H^E}{c_{CAE}}}{c_{CAE}^H} \quad (2.9 \text{ revisited})$$

In this equation, the cycle counts are known as they are measured. The frequency ratio, $f_H/f_L$, can be computed to be 1.54 using the configured frequencies. The voltage ratio, $V_L/V_H$, is 1 since the voltage setting is the same for both high and low frequency. The activity factor ratios, $\alpha_L^T/\alpha_{CAE}^T$ and $\alpha_H^H/\alpha_{CAE}^H$, could, as explained at the end of section 2.6, be derived from IPC given an appropriate power model. However, such a power model is unavailable for the particular test hardware. Therefore, these ratios are assumed to be 1 lacking better options.

Figure 3.14 shows the EDP based on the data from figure 3.13 under the assumption that the unknown activity factor ratios are set to 1. As expected from the large overhead
in execution time, there is also an overhead compared to the CAE baseline in EDP. In the best case (indirection level 0, stage 3) the overhead is 32\%. As a result, it is clear that more work would be needed to optimise DAEDAL for the target in order to make it beat the energy efficiency of the baseline.

To be thorough, a little discussion about what influence the unknown activity factor ratios can have is appropriate. In previous work [12, 4, 13], the power model used to approximate the activity factor, $\alpha$, is based on a linear polynomial as a function of IPC where the first degree coefficient is on the order of a tenth of the constant.

\[
\alpha \approx a \cdot \text{IPC} + b,
\]

\[
10a \sim b \implies \alpha \propto \text{IPC} + 10
\]

This model is made for a completely different processor architecture (Intel Sandy Bridge) and cannot be expected to be accurate for the ARM platform used in this project. Nevertheless, assuming that the relationship between activity factor and IPC is similar for the ARM platform, it is possible to get a hint of the error that is introduced by ignoring the activity factors when computing the EDP.

By using the IPC for CAE at high frequency, $IPC_{CAE}^H$, as well as IPCs for the access and execute phases, $IPC_A^L$ and $IPC_E^H$, from stage 3, which is the best performing and thus most important optimisation stage, the activity factor ratios become

\[
\frac{\alpha_A^L}{\alpha_{CAE}^H} \approx \frac{IPC_A^L + 10}{IPC_{CAE}^H + 10} \approx 1.00
\]

\[
\frac{\alpha_E^H}{\alpha_{CAE}^H} \approx \frac{IPC_E^H + 10}{IPC_{CAE}^H + 10} \approx 1.07
\]
Chapter 3. Evaluation

This means that there is no particular change in the energy estimate for the access phase but a 7% increase for the execute phase. Using these numbers instead in equation (2.9), the best case EDP increases by 2.7% from 1.32 to 1.35.

Another avenue for future consideration is to increase the frequency for the execute phase to one higher than what is used in this project. The ARM chip can possibly run at a 40% higher frequency with a voltage of 0.9 V. If the execute phase for target A would follow the trend set by the results in figures 3.11 and 3.12 linearly, the execute phase could finish 30% earlier. However, given the size of the access phase this would still not beat the execution time of the CAE baseline. Furthermore, due to the higher voltage, this would also bring a 25% increase in power draw for the execute phase. Another consideration in regard to the voltage increase would be whether voltage regulators could keep up. The length of a single phase is on the order of one to a few microseconds, which is typically too quick for common off-chip voltage regulators [11] keep up with. Without a solution for this, the 25% increase in power would also affect the access phase, since the higher voltage would have to be used all the time to sustain the higher frequency in the execute phase.
Chapter 4

Related Work

An early concept of DAE was introduced by Smith [15] in 1982. He suggests an architecture consisting of separate functional units for access and execute which uses different instruction streams but collaborate by sending data between each other. The benefit of this asymmetric multiprocessor system is improved performance achieved through the doubled instruction issue and the out-of-order nature of being able to overlap memory operations and calculations.

The idea of performing memory operations on separate processor cores has also been explored more recently. Kamruzzaman et al. [9] implements DAE to accelerate a single threaded program running on a multi-core processor by executing prefetch helper threads on unused cores. The idea is to generate helper threads, that only performs prefetching and necessary calculations, and divide a program into chunks such that the helper threads can run a number of chunks ahead of the main thread which between each chunk migrates to the next core with a pre-loaded cache. With this technique they achieve improve performance and, despite higher power consumption, lower energy usage, owing to the sped up execution.

DAEDAL’s history has its early roots in work on fine scale DVFS. Keramidas et al. [10] approaches the problem by developing interval-based models to predict program behaviour in order to generate DVFS policies that continuously adapt to optimal settings for the current program interval. They later continue this work [16] by implementing the models for the Linux kernel and testing them on real hardware, which yields good results for memory-bound workloads.

Koukos et al. [12] brings the fine scale DVFS together with DAE. The motivation is to expand the applicability beyond heavily memory-bound programs by separating tightly coupled memory operations and arithmetic computations into access and execute phases with distinct characteristics and that are large enough to take advantage of DVFS. Following positive results, Jimborean et al. [7] create models for automating the separation into access and execute phases. These include a polyhedral model for affine code and a general model for other code. The automatic separation achieves results on par with the manual.
Chapter 4. Related Work

In his Bachelor’s thesis, Ekemark [4] extends the general DAE model for automatic separation of access and execute phases with options for how complex the access phase is allowed to be in terms of memory indirections. The benefit of having differently generated versions is that the most beneficial variant can be chosen for each program that the optimisation is applied to. A weakness of the general separation model is that it cannot handle store instructions involving the global scope that are critical to functionality. Zacharopoulos [20] explores in his Master’s thesis a way to handle this weakness using hardware transactional memory. While the efficiency results of this approach are underwhelming, it exemplifies a way to handle a prohibitive issue with the general DAE separation model. After some iteration, improvement, and extended testing, these works are presented in a paper by Koukos et al. [13], representing the definitive version of Daedal.

With inspiration from both Daedal and Kamruzzaman et al.’s work, Weber [19] applies, in his Master’s thesis, DAE on ARM’s big.LITTLE architecture. The idea is to use access and execute phases and run the access on the architecture’s small power efficient cluster and migrating to the large performance cores for the execute. The results are promising but suffer from a large synchronisation overhead. Weber’s work also bear a symbolic similarity to Smith’s model of using asymmetric processors to perform fetch and computation.

Daedal is also used in Jin’s Master’s thesis [8] where she employs feedback directed compilation to tune granularity and indirection level in an automatic iterative process.

The DAE technique of Daedal has also inspired Clairvoyance [17] where Tran et al. perform a software out-of-order transformation to create small access and execute phases. A key difference from Daedal, which uses cache as a storage area for prefetched data, is that Clairvoyance reorders, rather than duplicates, accesses and holds fetched data in registers. This technique is especially powerful on hardware with no or limited out-of-order capabilities.

Tran et al. also publish SWOOP [18] as a follow-up to Clairvoyance. The principal difference is that SWOOP adapts the number of access phases it runs before switching to execute phases as a reaction to long latency cache misses. This is a dynamic form of granularity akin to the static form in Daedal.
Chapter 5

Conclusion

In this project, the scheduler of Huawei’s base station software is investigated through inspection and profiling to find suitable targets for DAEDAL. DAEDAL is in turn adapted to function as a part of a bigger process and incorporated into the build system of the base station software in order to apply DAE optimisation on the targets. However, this requires application of certain manual workarounds and weakening of some features that ensures correct behaviour. Following initial performance measurements, improvements are made by implementing an address analysis procedure and applying it to iteratively optimise the DAE transformation using dynamic feedback information. Finally, energy efficiency is investigated by calculating the energy-delay product (EDP) with an estimated energy consumption.

While neither of the investigated targets see an improvement from the use of DAEDAL, one of them show some promise. Despite having a best case total slowdown of 19%, caused by a heavy access phase, the execute phase of the target exhibits compute bound behaviour and its execution time is reduced to only 38% of the reference. Through feedback optimisation, the large impact of the access phase is reduced by about 10% to give a new best case total slowdown of 12%. This is, however, not enough to achieve an improved energy efficiency as the access phase takes approximately 25% longer at 2/3 frequency, giving a best case normalised EDP of 1.32 under DAEDAL’s intended DVFS based execution model.

Since the results indicate a degradation, they cannot be used to motivate development of base station hardware solutions that support the DVFS capabilities required to implement DAEDAL’s execution model. Better results could be achieved by investigating if other parts of the base station software work better with DAEDAL or by continuing to work with the targets in the scheduler and find solutions to the issues that ail them. For the latter, an interesting avenue would be to examine the proposed reasons for the low variation between indirection levels. This could be because of whitelisted function calls or low indirection prefetches implicitly forcing a higher than intended indirection level as a result of their interplay with control flow instructions and dead code elimination. Aside from these issues, there are other barriers to practical use of DAEDAL. Both manual inlining and ignoring unsafe removal of certain store instructions are required to
Chapter 5. Conclusion

compile the investigated targets. Some form of interprocedural analysis, perhaps generating access versions of called functions, and stronger analysis or dynamic approach (e.g. transactional memory) may be needed to handle these issues. Another improvement that could be made to DAEDAL is enhanced static filtering of redundant prefetches, which has been made possible by the identifiers attached to prefetches and loads in the course of performing address analysis.

The application of DAEDAL in Huawei’s base station software has provided a series of challenges for the optimisation technique. The novel difference for DAEDAL, that optimisation takes place in an infinitely repeating loop with deadlines to keep, has not, however, appeared as an issue since the targeted code fit well inside a single iteration of the main loop. Instead, the major issues stem from code complexity DAEDAL is not equipped to handle, which is a possible contributor for the less than ideal results. Even so, the large reduction of execution time for the execute phase indicates some potential, but the issues causing the low variability of the access phase across many optimisation settings makes it unclear whether the impact would hold under a lighter access phase. Further investigation is required to determine if DAEDAL can be adjusted to produce a lighter access phase for the target at hand, whether that will improve overall energy efficiency, and if similar results translates to other parts of the base station software.
## Appendix A

### Profiling with the PMU

Profiling the base station software is done by placing calls to the functions `PmuGetStartAcc` and `PmuGetEndAcc` around the code to be investigated. Listing A.1 shows a simplified implementation of the profiling functions.

#### Listing A.1: C code for profiling using the PMU.

Profiling is done by placing calls to `PmuGetStartAcc` and `PmuGetEndAcc` at the beginning and end, respectively, of code to be profiled. `PmuGetEndAcc` accepts an index in order to allow profiling of multiple (non-overlapping) code regions. Each sample is accumulated for each region into six variables, one for each PMU counter, together with a counter for the number of samples made. Functionality to enable the PMU, set the `CounterEvent_x` variables, configure and control the PMU counters, and switch the `PmuCollection` flag is assumed to be implemented elsewhere.

```c
#include <stdlib.h>
#include <stdint.h>

#include "pmu_tools.h"
/* pmu_tools.h provides:
 * SetEventSource(CounterEvent, Counter, InitialValue)
 * EnableAllCounters() // Start counters in PMU
 * DisableAllCounters() // Stop counters in PMU
 * uint32_t CounterEvent_0, CounterEvent_1, ..., CounterEvent_5 */

#include "pmu_profiling.h"
/* pmu_profiling.h provides:
 * LOC_IDX_CNT
 * uint32_t PmuEventData[LOC_IDX_CNT][8]
 * uint32_t PmuCollection // 0: Collection disabled, 1: Collection enabled */

#define DeclareCounter(c)  
    uint32_t CounterValue_##c = 0;  
    static __inline__ uint32_t GetCounterValue##c(void) { 
        uint32_t val; 
        asm volatile("mrc p15, 0, %0, c14, c8, #c : =r"(val));  
        return val;  
    }

DeclareCounter(0); DeclareCounter(1); DeclareCounter(2);  
DeclareCounter(3); DeclareCounter(4); DeclareCounter(5);  
#define GetCounterValue(c) (GetCounterValue##c())
```

51
Appendix A. Profiling with the PMU

```c
void PmuGetStartAcc(void)
{
    if (PmuCollection != 1) return; // Check if data is to be collected

    // SetEventSource(CounterEvent, Counter, InitialValue)
    #define SetEvent(c) SetEventSource(CounterEvent_##c, c, 0);
    SetEvent(0); SetEvent(1); SetEvent(2); SetEvent(3); SetEvent(4); SetEvent(5);
    EnableAllCounters(); // Start counters in PMU

    #define InitValue(c) CounterValue_##c = GetCounterValue(c);
    InitValue(0); InitValue(1); InitValue(2); InitValue(3); InitValue(4); InitValue(5);
}

void PmuGetEndAcc(uint32_t locIndex)
{
    if (PmuCollection != 1) return; // Check if data is to be collected

    #define AccValue(c) PmuEventData[locIndex][c] += 
                       GetCounterValue(c) - CounterValue_##c;
    AccValue(0); AccValue(1); AccValue(2); AccValue(3); AccValue(4); AccValue(5);
    DisableAllCounters(); // Stop counters in PMU

    PmuEventData[locIndex][6] += 1; // Increase invocation count
}
```
Appendix B

Address Collection

Listing B.1 contains a reference implementation of AddrRec, the function used to collect and record the occurrences of memory operations. AddrRec accepts four arguments: \textit{id}, an identifier used to correlate certain records; \textit{loc}, a serial number that uniquely identifies a specific call to AddrRec; \textit{type}, a predefined constant representing which kind of event that is occurring; and \textit{ptr}, which is the pointer address used by the memory operation, if applicable. When called, AddrRec will store the supplied argument in the next available slot in the array addresses. In case AddrRec may be called from different threads, the function \textit{current_vcpu} should return an index unique to that thread to avoid data races. Furthermore, the condition on line 37 can be replaced by an adequate expression if collection should only be active in some of the cases where AddrRec may be called.

Listing B.1: C code for collecting address usages.
Collection is done by calling AddrRec with relevant arguments to add to the log of events, addresses. Calls to AddrRec must be inserted with proper arguments where collection is desired. To complete the program, the node count on line 31 should be adjusted to accommodate the expected number of events, the test on line 37 should be replaced with a condition for when collection is to be active, and an output function, that saves the addresses array to file or similar, has to be implemented.

```c
#include <stdlib.h>
#include <stdint.h>

enum AddrType {
    AddrType_StartA = 0, // Start...
    AddrType_EndA = 1,   // ...and end access phase
    AddrType_LLoad = 2,  // Typical element in access phase
    AddrType_GLoad = 3,  // '' '' '' '' ''
    AddrType_Pref = 4,   // '' '' '' '' ''
    AddrType_StartT = 5, // Start...
    AddrType_EndT = 6,   // ...and end a target
    AddrType_StartI = 7, // Start...
   AddrType_EndI = 8,   // ...and end an iteration
    AddrType_StartE = 9, // Start...
    AddrType_EndE = 10, // ...and end execute phase
    AddrType_ELoad = 11, // Load in execute phase
    AddrType_EStore = 12 // Store in execute phase
};
typedef enum AddrType AddrType_t;
```
Appendix B. Address Collection

```c
struct AddrNode {
    uint32_t id; // Shared identifier between related events.
    uint32_t loc; // Unique identifier for the specific source of the event.
    AddrType_t type; // Event type identifier.
    uint8_t *ptr; // Pointer address payload, if applicable.
};
typedef struct AddrNode AddrNode_t;

// Provides current_vcpu() and NUM_VCPUS
#include "vcpu_info.h"

#define ADDRESS_NODES (50000)
AddrNode_t addresses[NUM_VCPUS][ADDRESS_NODES];
size_t nextFreeNode[NUM_VCPUS] = {0};

void AddrRec(uint32_t id, uint32_t loc, AddrType_t type, uint8_t *ptr) {
    if (1) { // Replace with adequate test to determine if collection is active.
        if (nextFreeNode[current_vcpu()] < ADDRESS_NODES) {
            AddrNode_t *addr = &addresses[current_vcpu()][nextFreeNode[current_vcpu()]+1];
            addr->id = id;
            addr->loc = loc;
            addr->type = type;
            addr->ptr = ptr;
        }
    }
}

// Convenience functions with type already set and ptr as NULL where it is irrelevant.
#define AddrPtr(type) void Addr##type(uint32_t id, uint32_t loc, AddrType_t* type, uint8_t *ptr) {
#define AddrEvt(type) void Addr##type(uint32_t id, uint32_t loc) {
#define AddrRec(type) void AddrRec(uint32_t id, loc, AddrType_t*, ptr);

AddrEvt(StartA); AddrEvt(StartT); AddrEvt(StartI); AddrEvt(StartE);
AddrEvt(EndA); AddrEvt(EndT); AddrEvt(EndI); AddrEvt(EndE);
AddrPtr(LLoad); AddrPtr(GLoad); AddrPtr(Pref); AddrPtr(ELoad); AddrPtr(EStore);
```

The `type` argument accepts one of the constants in the enum AddrType. The types Pref, GLoad, LLoad, ELoad, and EStore all represent memory operations where `ptr` takes the pointer address for the operation. Pref and GLoad represent prefetches and loads in the access phase and are tagged with an identifier in the `id` argument. A prefetch and a load share the same identifier if, and only if, they represent the same load (i.e. the prefetch is supposed to replace a load that the control flow depends on). LLoad (for ‘local load’) also occurs in the access phase for loads that have not been considered to be prefetched. The ELoad and EStore types represent loads and stores in the execute phase. LLoad, Eload, and EStore all use the reserved value 0 for the `id` argument.

The eight remaining types, Start and End T/A/E/I, represent the events of entering and exiting a target loop, an access phase, an execute phase, and an individual loop iteration, respectively. None of these types use the `ptr` argument, while `id` is unique for each target and unique for each phase. The `id` is irrelevant for iterations but is by conversion the same as for the target loop they are in. Also, the EndI type is typically
Appendix B. Address Collection

Figure B.1: Address collection events.
Apart from the collection of addresses from memory operations in the access and execute phases, some boundary events are recorded as well. The calls to the collection functions (in green) are inserted at compile time to generate markers in the address collection log where a target loop, an access phase, or an execute phase starts or ends, as well as when new iterations in the main loop (in the phases) begins.

not used in practice as the start of a new iteration or the end of a phase implies the event.

Additionally, each of the types have a specialised convenience call where only id and loc have to be supplied, as well as ptr for the memory operation types.

The use of AddrRec has been shown for memory operations in figure 2.5 on page 19. The placement of the event type calls is illustrated in figure B.1.
Appendix C

Extended Results

This appendix contains more measurement results from the experiments of the evaluation. The interesting data is already presented in the main part of the report, the following sections merely contains some variations for full disclosure. The variations include an alternate normalisation method, an additional measurement mode, and a few earlier versions with issues that is not or barely mentioned in the main report.

In each of the following sections, results for a specific stage of the experimentation are presented. Each of the applicable targets in the sections are generally presented using four charts. Aside for the normalisation method used in the main report, where all results are normalised to CAE at granularity 1, there is also a variant where each granularity is normalised separately. Usually there is little to no difference but in the cases there are the later variant masks some information that could be attributed to noise generated when passing chunk borders. These two charts are doubled up by an additional measurement mode where the targets have been monitored as a whole instead of measuring the access and execute phases separately. For this mode, measurement functions are inserted manually before and after the target loop in source code, positioning them as in figure C.1, rather than around phases automatically by the DAE pass during compilation, as shown in figure 2.4 on page 16. This mode eliminates any noise generated by frequently starting and stopping measurement but also monitors slightly more code as the switch between chunks is accounted for. The time difference between the measurement variants is typically small with a couple percent higher values for the whole target measurements at fine granularities; at coarser granularities the difference practically vanishes.

```
PmuGetStartAcc();
while (cond) {
    for (i = 0; cond && i < granularity; ++i) {
        Access
    }
    for (i = 0; cond && i < granularity; ++i) {
        Execute
    }
}
PmuGetEndAcc(t);
```

Figure C.1: Placement of profiling calls for whole target measurements.

Calls to the measurement functions (orange background) are placed before and after each target and results are collected under a unique index.
Appendix C. Extended Results

The next few pages describe the experiment stages for which performance results are collected and presented in the following sections.

**Initial**  section C.1, page 59
This is the first full set of results collected across a range of granularities and indirection levels. There are a number of differences between this version and the InlImpSA version, which is the one presented in the main part of the report, that are successively altered in the versions in between.

- In this version, inlining of the phases is inconsistent. Only the execute phase is inlined in the DAE version. The access phase is not inlined nor is the execute phase in CAE.

- Another difference is that prefetches for transferral or virtual loads, as mentioned in sections 2.3 and 3.3.1, are not avoided. This is changed in the InlImpSA version.

- A slightly different test case is used in this version. This only affects target B which is therefore referred to as original target B until the change in the InlineSA version.

**Inline**  section C.2, page 63
As the inconsistency with the inlining was spotted, it was altered for this updated version. The inlining boosts speed by 5% to 10% for the affected components, but more importantly, the inlining makes comparison between phases more trustworthy. Performance relative to CAE appear a bit worse in this version but different granularities and indirection levels respond differently.

**InlineSA**  section C.3, page 67
While investigating behaviour of in particular target B it appeared that the more interesting parts of the code was skipped and scheduling was essentially not performed. With expert help it was determined that the test bench delivered an odd value at one point which was altered with a change in the code. In the new variant of the test case used in this version Scheduling is Accepted.

The change in test case also change which loop under candidate B is best to target. Target B as visualised in the call tree in figure 3.3 on page 31 is the one used in this version and onward. The original target B used previously is actually a loop one function further down the call tree, the call to which is made inside the new target B.

**InlImpSA**  section C.4, page 71
This version implements the Improvement to the Inlining procedure mentioned in sections 2.3 and 3.3.1, where prefetching is avoided for the virtual loads responsible for state transferral between the extracted target functions and their original location. The benefit is that these virtual loads now can be eliminated by optimisation once the target functions are re-inlined. This is the version for which results are presented in section 3.3.
An issue introduced with this version is that *prefetch hoisting* is enabled. This places prefetches as early as possible in the code, as soon as the addresses used are available or computed, instead of close to the original loads. This is in principle a good thing but the specific implementation can move prefetches past conditional boundaries which can cause two types of problems and is the reason the feature is disabled by default. (i) Prefetches moved past conditional boundaries can fetch data that is never used in the execute phase. This can be detected by the *unused* address analysis profile and under this version a small number of prefetches do fit this category. (ii) Another issue is that the moved prefetches could reference inaccessible memory. In these cases the ARM PLD (prefetch) instruction could cause an asynchronous data abort [1, p. E2-2775] which depending on the program’s ability to handle the exception may lead to an error state. However, no issues related to this have been observed.

The prefetch hoisting feature is turned off again in the PrimProfopt version.

**InlImpSA – Loop Progression**  *section C.5, page 75*

The loop progression variant deals with the issues discussed in section 3.4. This version implements the workaround to allow more useful work to be done in target B. Since this does not affect target A, only results for target B are shown for this version.

**PrimProfopt**  *section C.6, page 77*

Finally, this version represents the results collected for the Profile or feedback based optimisation for the Primary target, i.e. target A, and EDP comparison discussed in sections 2.5, 3.5 and 3.6. The results for this version differ in style compared to the others. There are results available from both high (section C.6.1) and low (section C.6.2) frequency experiments. Also, only granularity 2 is investigated but instead three different stages of optimisation are compared. All results are normalised against CAE at granularity 2 and stage 1. Furthermore, even low frequency results are normalised against the high frequency baseline to allow comparison.

In this version, the prefetch hoisting, introduced in InlImpSA, is also turned off. This means that there are no longer any unused prefetches for the *unused* address analysis profile to detect.
Appendix C. Extended Results

C.1 Initial

![Figure C.2: Target A, Initial (phase), norm. CAE (gran. 1).](image1)

![Figure C.3: Target A, Initial (phase), norm. CAE (same gran.).](image2)

59
### Target A

**Initial (whole target)**

Normalised to CAE (granularity 1)

![Bar Chart](image)

Figure C.4: Target A, Initial (whole), norm. CAE (gran. 1).

**Initial (whole target)**

Normalised to CAE (same granularity)

![Bar Chart](image)

Figure C.5: Target A, Initial (whole), norm. CAE (same gran.).

---

60
Figure C.6: Original Target B, Initial (phase), norm. CAE (gran. 1).

Figure C.7: Original Target B, Initial (phase), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.8: Original Target B, Initial (whole), norm. CAE (gran. 1).

Figure C.9: Original Target B, Initial (whole), norm. CAE (same gran.).
Appendix C. Extended Results

C.2 Inline

Figure C.10: Target A, Inline (phase), norm. CAE (gran. 1).

Figure C.11: Target A, Inline (phase), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.12: Target A, Inline (whole), norm. CAE (gran. 1).

Figure C.13: Target A, Inline (whole), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.14: **Original Target B, Inline (phase), norm. CAE (gran. 1).**

Figure C.15: **Original Target B, Inline (phase), norm. CAE (same gran.).**
Appendix C. Extended Results

Figure C.16: Original Target B, Inline (whole), norm. CAE (gran. 1).

Figure C.17: Original Target B, Inline (whole), norm. CAE (same gran.).
Appendix C. Extended Results

C.3 InlineSA

Figure C.18: Target A, InlineSA (phase), norm. CAE (gran. 1).

Figure C.19: Target A, InlineSA (phase), norm. CAE (same gran.).
Appendix C. Extended Results

Target A

InlineSA (whole target)
Normalised to CAE (granularity 1)

Figure C.20: Target A, InlineSA (whole), norm. CAE (gran. 1).

Target A

InlineSA (whole target)
Normalised to CAE (same granularity)

Figure C.21: Target A, InlineSA (whole), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.22: Target B, InlineSA (phase), norm. CAE (gran. 1).

Figure C.23: Target B, InlineSA (phase), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.24: Target B, InlineSA (whole), norm. CAE (gran. 1).

Figure C.25: Target B, InlineSA (whole), norm. CAE (same gran.).
Appendix C. Extended Results

C.4 InlImpSA

![Graph](image)

**Figure C.26:** Target A, InlImpSA (phase), norm. CAE (gran. 1).

![Graph](image)

**Figure C.27:** Target A, InlImpSA (phase), norm. CAE (same gran.).
Appendix C. Extended Results

Figure C.28: Target A, InlImpSA (whole), norm. CAE (gran. 1).

Figure C.29: Target A, InlImpSA (whole), norm. CAE (same gran. ).
Appendix C. Extended Results

Figure C.30: Target B, InlImpSA (phase), norm. CAE (gran. 1).

Figure C.31: Target B, InlImpSA (phase), norm. CAE (same gran.).
Appendix C. Extended Results

![Diagram](image)

Figure C.32: Target B, InlImpSA (whole), norm. CAE (gran. 1).

![Diagram](image)

Figure C.33: Target B, InlImpSA (whole), norm. CAE (same gran.).
Appendix C. Extended Results

C.5 InlImpSA – Loop Progression

![Figure C.34: Target B, InlImpSA-LP (phase), norm. CAE (gran. 1).](image)

![Figure C.35: Target B, InlImpSA-LP (phase), norm. CAE (same gran.).](image)
Appendix C. Extended Results

Figure C.36: Target B, InlImpSA-LP (whole), norm. CAE (gran. 1).

Figure C.37: Target B, InlImpSA-LP (whole), norm. CAE (same gran.).
Appendix C. Extended Results

C.6 PrimProfopt

C.6.1 High Frequency

Figure C.38: Target A, PrimProfopt HF (phase), norm. CAE (stage 1).

Figure C.39: Target A, PrimProfopt HF (whole), norm. CAE (stage 1).
Appendix C. Extended Results

C.6.2 Low Frequency

Figure C.40: Target A, PrimProfopt LF (phase), norm. CAE (HF, stage 1).

Figure C.41: Target A, PrimProfopt LF (whole), norm. CAE (HF, stage 1).
References


References


Part II

Enlargement of Extended Data-Race-Free Regions through Conflict Isolation
Glossary

**CFG** control-flow-graph. A graph, typically built by a compiler, representing paths control can follow in a program. The graph describes which parts of a program can be executed in what order.

**CS** critical section. A protected piece of code in a parallel program only a single thread can gain access to at a time. Typically protected with a lock.

**DFS** depth-first-search. A graph search order where each neighbour is recursively explored completely before continuing with the next.

**DRF** data-race-free. The concept of a program being free from any possibility of a race condition. In the context of a region, a continuous piece of code where no races can occur due to design or use of synchronisation.

**iDRF** isolated data-race-free. In the context of a region, an instruction isolated from an xDRF-CI region to allow an otherwise impermissible extension.

**nDRF** non-data-race-free. In the context of a region, a critical section or other synchronisation construct separating DRF regions. Called *enclave* if an xDRF region can be extended across it, otherwise *non-enclave*.

**xDRF** extended data-race-free. The analysis of a program to find situations where properties of DRF regions can be extended across synchronisations (i.e. nDRF regions). In the context of a region, a collection of continuous DRF regions that can be extended across the separating (enclave) nDRF regions.

**xDRF-CI** extended data-race-free with conflict isolation. The extension of xDRF analysis, allowing otherwise impermissible extension by isolating memory conflicts in iDRF regions. In the context of a region, an xDRF region with additional extensions following the use of iDRF regions.
Chapter 6

Introduction

Since hitting the power wall in processor development, parallel computing has been instrumental in further increasing performance of applications. A challenge of this paradigm is the increased complexity associated with developing parallel programs, which stems from the need of coordinating concurrent code. Typically, coordination is achieved through communication using shared memory. Among languages for parallel programming there are different approaches to what kind of abstraction for coordination to provide the programmer.

The mainstream languages C and C++, among others, provide access to a shared-memory model with a well-defined consistency model that specifies when modifications become visible to other threads in the program. To provide guarantees, the model requires programs to be data-race-free (DRF), meaning that modifying accesses to a memory location by different threads must be performed with special atomic operations or be separated in time by a synchronisation mechanism such as a lock or a barrier. If data races are present the behaviour of the program is undefined [25, 26].

In applications adhering to the DRF rules, synchronisation divides the program code into synchronisation-free or DRF regions, i.e. continuous code outside of locks or between barriers, etc. This division enables compilers and tools to reason about code in synchronisation-free regions as if they were sequential code and to provide optimisations and analyses as such.

Previous work [34, 28] introduced the notion of extended data-race-free (xDRF) regions. This is the amalgamation of multiple DRF regions into one region, which preserves similar semantics as the components. The extension relies on analysis between adjacent DRF regions divided by a synchronisation. If the analysis determines that it is possible, an xDRF region can be formed across the synchronisation point and include the DRF regions on either side (though not the synchronisation point itself). Chapter 7 describes in more detail what xDRF is and how DRF extension is performed. The benefit of having larger regions with DRF semantics is that the same analyses applicable to regular DRF regions can be applied to a longer section of code, giving the possibility of improved optimisation or analysis outcomes.

An application for xDRF region delimitation, which was used as evaluation in previ-
ous work [28], is as a guide for a state-of-the-art, dual-mode cache coherence protocol, SPEL++ [34, 35]. In this protocol, coherence traffic is suspended for the duration of an xDRF region and only performed at the end when multiple coherence operations can be performed at once.

Another possible use of xDRF region delineation is as a complement to decoupled access-execute (DAE) techniques, such as DAEDAL [30] and CLAIRVOYANCE [40], when applied on parallel applications. The larger code areas allowing code motion as in single threaded applications could potentially allow application of DAE techniques on a wider set of targets.

The approach to extending DRF regions taken in the previous work [28] has some issues with suboptimal analysis, compared to that of an expert performing the delineation manually. The core of the problem is that the analysis of adjacent DRF regions, separated by a synchronisation point, can report false conflicts as a result of the analysis being conservative. A false conflict forces a break in the extension of an xDRF region, which is not desired.

This work proposes an alternate way of dealing with reported conflicts between DRF regions, allowing extension across synchronisation points regardless of the presence of conflicts. The principal change is that points of conflict preventing DRF extension are isolated by viewing them as synchronisation points and extending the DRF regions across them as well, thus allowing the creation of even larger extended data-race-free regions with conflict isolation (xDRF-CI). Chapter 8 describes in more detail how the xDRF analysis is changed in order to produce larger xDRF-CI regions. This proposal to utilise conflict isolation has already been presented in a recent paper [27].
Chapter 7

Background
Extended Data-Race-Free Regions

In order to clarify what extended data-race-free (xDRF) regions are, it is helpful to first be clear on what data-race-free (DRF) regions are in the first place as well as being familiar with some related concept (section 7.1). With knowledge of the basics, the concept of xDRF regions is approached in the form of examples (section 7.2), then an overview of their construction is described (section 7.3). For deeper knowledge into practical details when building xDRF regions and proofs for their properties, previously published papers are recommended [28, 27].

7.1 Concepts of DRF and xDRF regions

Parallel code where multiple threads operate on the same data must be synchronised in order to avoid conflicts and to make a program data-race-free. Fundamentally, synchronisation is facilitated by atomic operations but normally a library providing operations for common synchronisation patterns, such as pthreads, is used instead.

To avoid memory conflicts or data races, which occur when two threads concurrently access the same memory location and at least one of the accesses is a modification, a common synchronisation mechanism to use is a lock. A lock grants exclusive execution access to code protected by it—a critical section (CS)—to a single thread at the time. As long as all accesses to some shared memory resource are performed in critical sections protected by the same lock, no conflict will occur for that memory resource. Another common synchronisation pattern is the use of barriers and signal-waits. Barriers are useful when multiple threads are running the same code, to ensure that no thread start execution of the code after the barrier until all threads finish the execution of code before it. Similarly, signals and waits are useful when threads execute different code, to ensure that one thread will finish execution of the code before a signal ahead of another thread starting the execution of code after a wait. Both barriers and signal-waits are synchronisation points that provide a happens-before relationship in the code they occur in. In contrast, locks and critical sections provide a one-at-the-time semantic.
Synchronisation points and critical sections segment a program into a number of synchronisation-free regions, the consecutive pieces of code between synchronisation operations. In a DRF program, the synchronisation-free regions are data-race-free (DRF) regions in the sense that within and between them there is no risk of a data race as threads either operate on different memory locations or are separated in time, and in some aspects they can be treated as sequential code. To contrast, the rest of the code, i.e. the critical sections and synchronisation points, are here denoted as non-data-race-free (nDRF) regions; the name only reflects the fact that they are separate from DRF regions, they do not imply any data races.

Extended data-race-free (xDRF) regions consist of multiple DRF regions (or in the trivial case just a single) extended across nDRF regions. There are two equivalent ways of defining xDRF regions: (i) the collection of DRF that can be joined by extending over nDRF regions (this is what has already been alluded to) or (ii) the collection of DRF regions between nDRF regions that cannot be extended across. Which is more useful depends on application, but the latter is what is primarily used in the analysis to delimit the xDRF regions (section 7.3). As it is important to make a difference between nDRF regions that can and cannot be extended across, they are denoted as enclave nDRF regions if extension is possible and non-enclave nDRF regions if extension is not possible.

7.2 When xDRF regions are possible

The rationale behind creating xDRF regions in the first place is to allow larger sections of code to be reasoned about in a sequential-like manner. The possibility to perform extension of two DRF regions, split by an nDRF region, hinges on the case being that the DRF regions do not require a happens-before relation to separate them. To illustrate in what situations it is and is not possible to form an xDRF region across a synchronisation point, figures 7.1 and 7.2 provide a series of examples. Forming an xDRF region is possible in (a) and (e) but not in (b), (c), (d), and (f); (g) is a special case.

In the first example, (a), two threads are running the same code where an nDRF region, containing a lock on $L$ protects a CS where the shared variable $counter$ is incremented, splits a modification to $a[i]$ (assume different $i$ for each thread) and read of $b[i]$ into different DRF regions. Assuming the arrays $a$ and $b$ refer to different memory locations, there is no need for a happens-before relationship between the DRF regions and they can be extended into a single xDRF region. As it is extended across, the nDRF region between the DRF regions is enclave. The nDRF region is, however, not a part of the formed xDRF region and does not share in the xDRF region’s sequential-like semantics.

The next example, (b), shows two threads running the same code where a barrier as an nDRF region separates two DRF regions containing a modification and a read of $x[i]$. In the case an $i$ used by one thread in the first DRF region can be used by another in the second, the happens-before relationship provided by the barrier is required. Therefore, the DRF regions cannot be extended, the barrier is a non-enclave nDRF region.
Figure 7.1: Examples of situations allowing or disallowing xDRF formation.
Examples of critical sections and synchronisation points, or nDRF regions (marked in bold with indented intermediate rows), interacting between different threads (in (a) and (b) both threads run the same code). Conflicts preventing xDRF formation are marked in red. xDRF regions that can expand across an intermediate enclave nDRF region are shaded in colour.

In example (c) the two threads are running different code. Each code snippet has two DRF regions divided by an nDRF region. As the 'before' DRF region in $Th_0$ modifies $x$ while the 'after' DRF region in $Th_1$ reads it, a happens-before relationship is required between these DRF regions. The signal and wait provides this relationship by making $Th_1$ wait before reading $x$ until $Th_0$ signals it is done modifying it. In this case DRF extension is not possible.

Example (d) is similar to (c) as it has the same happens-before requirement between the uses of $x$. However, here the potential conflict is prevented by setting the shared variable $flag$ while holding lock $L$ in $Th_0$, copying $flag$ to a thread local variable in $Th_1$, also while holding lock $L$, and finally only performing the read of $x$ if $flag$ was set, indicating $Th_0$ was done modifying it. The lock and flag essentially emulates the function of a signal-wait pair. Since the happens-before relationship is required, DRF extension is not possible but this example shows a situation where extension is not possible across CS type nDRF regions.

Example (e) uses the same lock and flag construction of a signal-wait as in (d). However, here the data dependency requiring the happens-before behaviour is gone ($Th_1$ does not touch $x$). Therefore, it is possible to extend the DRF regions into a pair of xDRF regions, one each for the code of the separate threads. The caveat here is that while code in an xDRF region in general can be moved across enclave nDRF regions, dependence rules must take the code in critical sections into account. The $print$ in the code snippet after the CS in the code for $Th_1$ cannot be moved ahead of the nDRF region since it, through control flow, depends on the if clause, which in turn has a data dependency on the $local$ variable in the critical section.
Chapter 7. Background: Extended Data-Race-Free Regions

\[ T_{h_0}: T_{h_1}: T_{h_2}: T_{h_0}: T_{h_1}: \]

\[ x = 1 \quad \ldots \quad \ldots \]

signal \( A \) wait \( A \) \ldots lock \( A \) lock \( B \)

\ldots signal \( B \) wait \( B \) \ldots \ldots \ldots

\ldots print(\( x \)) \ldots unlock \( A \) unlock \( B \)

\ldots \ldots

(f) \quad \quad \quad \quad (g)

Figure 7.2: Special examples of situations disallowing xDRF formation.

Special examples of nDRF regions (marked in bold) interacting between different threads. In (f) the conflict (marked in red) preventing xDRF formation transitively affects several threads. In (g) there is no interaction since the threads synchronise on different variables.

Example (f) shows that happens-before relationships, that must be respected when considering xDRF formation, can occur between threads that do not directly synchronise with each other. In this case the relationship is enforced transitively, first through a signal-wait carried by resource \( A \) between the modifying thread, \( T_{h_0} \), and the intermediary thread, \( T_{h_1} \), then through a signal-wait carried by resource \( B \) between \( T_{h_1} \) and the reading thread, \( T_{h_2} \).

Finally, example (g) shows that nDRF regions using different resources to carry out their synchronisation do not have to be correlated against each other when considering xDRF formation. Whether forming xDRF regions is possible would depend on the results of cross-checking between other nDRF regions with matching synchronisation resources.

7.3 Construction of xDRF regions

When constructing xDRF regions in a program, there are a couple of useful design decisions that are made for practical reasons. (I) Instead of trying to construct xDRF regions directly by extending DRF regions wherever possible, as suggested by the examples in the previous section, the delineation is performed in two principal passes. First, the available nDRF regions are analysed to determine if they are enclave or not. Then, xDRF regions can be formed by grouping consecutive DRF regions that are only separated by enclave nDRF regions. (II) nDRF regions that contain a barrier or signal-wait are always considered to be non-enclave nDRF regions. This decision is taken because their use is a strong hint from the programmer that a happens-before relationship is required to avoid a data race, or that that kind of behaviour is desired. This allows skipping a costly analysis for this type of nDRF regions.

The construction of xDRF regions is implemented in as a compiler pass in LLVM [31]. To improve the accuracy of the alias analysis involved in analysing nDRF regions, a state-of-the-art pointer analysis tool [39] is used. The overarching analysis procedure is outlined here, for more details previous work [28] or the more recent paper including
conflict isolation \cite{27} are recommended.

1. Identify nDRF regions (critical sections, barriers, signal-wait) and build a control-flow-graph (CFG) between them, this graph is denoted Sync-CFG.

2. Mark the first reachable nDRF region (of a depth-first-search (DFS)) in the Sync-CFG in each thread function as an entry nDRF region.

3. Determine what resource each nDRF region synchronise on (e.g. lock on L as in the examples in section 7.2). Correlate, across all thread functions, nDRF regions that use the same synchronisation resource and call these matching nDRF regions.

4. Mark nDRF regions with a barrier or signal-wait as non-enclave as per the design decision (II) mentioned earlier.

5. Walk the Sync-CFG in DFS order, starting from each entry nDRF region. For each nDRF region, do following:

   5.1. Build a pair of sets of preceding-xDRF-paths and following-xDRF-paths. These are built by recording the control-flow-paths visited when walking the Sync-CFG in reverse and forwards direction, respectively, starting with the current nDRF region, walking across enclave nDRF regions, and stopping at non-enclave or not yet processed nDRF regions.

   5.2. Identify conflicts
   
   5.2.1. between the preceding-xDRF-paths and following-xDRF-paths of matching nDRF regions,

   5.2.2. between the instructions within the matching nDRF regions and the preceding and following-xDRF-paths, and

   5.2.3. between the instructions within any enclave nDRF region crossed when building the xDRF paths of the matching nDRF regions and the preceding and following-xDRF-paths of the current nDRF region.

   5.3. If there are no conflicts, the nDRF region is enclave, otherwise non-enclave.

6. Form xDRF regions by walking the Sync-CFG in DFS order, starting from each entry nDRF region. For each nDRF region, do following:

   6.1. If the current nDRF region is enclave, extend the current xDRF region, otherwise break the current xDRF region at this point and start a new one.

   6.2. If the current nDRF region is already enclave in another xDRF region, merge that xDRF region with the current xDRF region.
Chapter 8

Contribution

xDRF with Conflict Isolation

A weakness of the established xDRF analysis (section 7.3) is that it is heavily dependent on quality alias analysis. At compile-time it is, however, hard to perform accurate analysis and often results are inconclusive. When this happens while analysing an nDRF region, the xDRF analysis pass must, for correctness, make the conservative assumption that there is a collision and the nDRF region must be marked non-enclave, limiting the extension of the xDRF region. The contribution of this work seeks to take an alternative approach when finding conflicts using a technique dubbed conflict isolation.

Under the xDRF-CI analysis, the recourse when finding a conflict (in step 5.3 of the established xDRF analysis) for an nDRF region is to still mark it as enclave, allowing xDRF extension across it. The potential conflict is dealt with by placing the parties of it in isolated data-race-free (iDRF) regions. For all intents and purposes, iDRF regions are equivalent to enclave nDRF regions, though they do not contain synchronised code. In fact, as an implementation simplification, iDRF regions are represented by reusing enclave nDRF regions with a flag indicating they are in fact iDRF regions.

Figure 8.1 illustrates the difference between how the xDRF and xDRF-CI analyses handle a conflict. Based on example (d) in figure 7.1, the example to the left shows how the regular xDRF analysis handles a conflict between $x$ and $x'$, which as a result of an inconclusive alias analysis may or may not refer to the same memory location. Since there could be a conflict between the two accesses, the xDRF analysis is forced to make a conservative decision and mark the nDRF regions non-enclave which means that DRF extension is not possible. The xDRF-CI analysis, to the right, tackles the same situation by enclosing the offending operations in iDRF regions and marks the nDRF regions enclave. DRF extension is now possible across the nDRF regions as well as across the newly inserted iDRF regions.

The way the xDRF-CI analysis preserve correctness under this altered procedure is by placing the conflicts outside the xDRF regions (by enclosing them in iDRF regions). This means that the conflicts no longer enjoy the sequential-like semantics which would allow them to move across the nDRF regions. Instead, they are fixed in position and
Chapter 8. Contribution: xDRF with Conflict Isolation

Figure 8.1: Difference between xDRF and xDRF-CI.

To the left is a similar situation to example (d) from figure 7.1. Here there is an inconclusive alias between \( x \) and \( x' \), indicating a potential conflict (marked in red) that prevents DRF extension under the xDRF analysis. In the xDRF-CI analysis, to the right, the potentially conflicting operations are encapsulated in iDRF regions (red boxes), allowing formation of xDRF regions (shaded in colour) that extends over both nDRF and iDRF regions.

subject to the general happens-before relationship imposed by the nDRF regions.

A concern with the xDRF-CI analysis is that it applies the isolation technique to all conflicts that it comes across. Since this allows the xDRF regions to become larger than before, the risk of encountering more conflicts increases as well, whether they are true or just a result of incomplete alias analysis. This can lead to quite large xDRF regions that are peppered with iDRF regions. This would jeopardise the sequential-like semantics for a lot of accesses, which was what was sought after to begin with. Ultimately, whether ‘clean’ or large xDRF regions are preferred may depend on application.
Chapter 9

Evaluation

The evaluation of the xDRF-CI analysis is performed on applications from the SPLASH-3 [38] (a modernised, data-race-free version of the SPLASH-2 suite [41]) and PARSEC-2.1 [22] benchmark suites. Both suites contain applications for testing multi-threaded systems with shared address space memory. The applications are executed with the standard inputs for SPLASH-2/3 and the ‘simsmall’ inputs for PARSEC-2.1. The evaluation considers both static results from compilation with the xDRF and xDRF-CI analyses—non-enclave nDRF regions left in code (section 9.1) and compilation time (section 9.3)—as well as runtime statistics gathered with a simulator developed in the xDRF project behind the xDRF-CI paper [27]—non-enclave nDRF regions encountered during execution (section 9.2) and performance of the state-of-the-art, dual-mode cache coherency protocol SPEL++ [35] hinted at in the introduction (section 9.4). The simulation tool also checks whether any data races occur in practice (in case of problems with the implementation of the analyses) using a consistency checker tool similar to FAST&FURIOUS [36], extended to support xDRF regions. A hundred runs of each application and xDRF analysis variation revealed no consistency violation.

Four different DRF oriented delineations are considered in the evaluation. The main feature is the xDRF-CI analysis (labelled xDRF-CI-Compiler) proposed in this work (and presented in [27]). This is contrasted against the predecessor xDRF analysis (labelled xDRF-Compiler without conflict isolation presented in previous work [28]¹, a manual delineation based on code inspection [37] (labelled xDRF-Manual), and the baseline DRF version (labelled DRF) based on the xDRF-Compiler version where all nDRF regions are considered non-enclave.

¹Some improvements have been made to the xDRF analysis by the xDRF team for the updated version of the paper [27]. This leads to a few more enclave nDRF regions in the Dedup, Ocean, Ocean-nc, Radiosity, and Water-Sp benchmarks.
Chapter 9. Evaluation

Figure 9.1: Non-enclave nDRF regions in code.

9.1 Static efficiency

Figure 9.1 shows for each of the delineation variants how many non-enclave nDRF regions that can be found in the compiled code. Since all nDRF regions are considered non-enclave in DRF, this category show the total number of nDRF regions in each benchmark, with an average (geometric mean) of 19.8. The xDRF-Compiler does a decent job of making nDRF regions enclave, leaving on average 14.9 non-enclave nDRF regions or 75%, however, the xDRF-Manual delineation is often slightly better with, on average, 12.1 non-enclave nDRF regions left. Since the manual approach takes additional information into account, it can be presumed to perfectly distinguish when a happens-before relationship is needed and only mark such nDRF regions non-enclave. The gap represents the cases where the xDRF-Compiler is forced to take a conservative decision due to incomplete alias analysis. The xDRF-Manual, on the other hand, closes this gap and in some cases even exceeds it. This is natural due to its behaviour as it always marks nDRF regions based on critical sections enclave. This must match the xDRF-Manual approach and in the cases it exceeds, this gap represents the CS based nDRF regions that do impose a happens-before relationship but is turned enclave anyway with isolated conflicts. The xDRF-CI-Compiler only leaves an average of 11.7 non-enclave nDRF regions which is 21% fewer compared to the xDRF-Compiler. The peak improvement the xDRF-CI-Compiler has over the xDRF-Compiler is in Radiosity where the xDRF-CI-Compiler leaves 6 non-enclave nDRF regions, 82% fewer than the xDRF-Compiler’s 33.

In seven of the 17 benchmarks, Dedup, FFT, Fluidanimate, LU, LU-nc, Radix, and Streamcluster, the xDRF-Compiler is enough to achieve a perfect analysis result; neither xDRF-Manual nor the xDRF-CI-Compiler provides any further improvements which means that all CS based nDRF regions are correctly identified as not causing a happens-before relationship without additional help from conflict isolation. In Streamcluster there actually are no CS based nDRF regions which means that no nDRF regions can be made enclave since it is only the CS based regions that are considered.
In the remaining benchmarks, *Barnes*, *Cholesky*, *FMM*, *Ocean*, *Ocean-nc*, *Radiosity*, *Raytrace*, *Volrend*, *Water-Nsq*, and *Water-Sp*, the *xDRF-Compiler* misses some extension opportunities as captured by the *xDRF-Manual* delineation. In all these benchmarks, apart from *FMM* and *Radiosity*, the addition of conflict isolation allows the *xDRF-CI-Compiler* to bypass the conflicts and match *xDRF-Manual*. In *FMM* and *Radiosity* there are a few additional CS based nDRF regions identified by *xDRF-Manual* to have happens-before semantics which the *xDRF-CI-Compiler* marks as enclave with the help of conflict isolation.

### 9.2 Dynamic efficiency

While static results say something about the power of the analysis, it does not give guarantees about its practical impact. Figure 9.2 shows the number of non-enclave nDRF regions encountered in the parallelised parts of the benchmarks while running them, normalised to the results of the plain *DRF* delineation (where all nDRF regions are non-enclave).

Looking at specific benchmarks, there are a few patterns that stand out. One of these are benchmarks exhibiting the same characteristics as *DRF* in all approaches of DRF extension. For *Streamcluster* this is natural since all synchronisation is handled by barriers and signal-waits, which is also reflected in the static results. The other benchmarks that does not see any improvement over *DRF*, *FFT*, *LU*, *LU-nc*, and *Radix*, are all from the same static category where the *xDRF-Compiler* reaches optimal DRF extension. These also have in common that they only have a small number of CS based nDRF regions to be made enclave, however, these are not executed in the parallelised parts of the programs where non-enclave barrier/signal-wait type synchronisations dominate instead, which explains why these benchmarks behave the same compared to *DRF*.

Other benchmarks beat *DRF* but still without differences between extension tactics.
These are Dedup, Fluidanimate, Ocean, Ocean-nc, Volrend, Water-Nsq, and Water-Sp. The first two are also of the category where conflict isolation does not enable any additional enclave nDRF regions over the xDRF-Compiler, yet the enclave nDRF regions that are identified are visited regularly in the parallelised part of the programs. The rest of the mentioned benchmarks do have some extra extended regions under the xDRF-CI-Compiler, but the relevant enclave nDRF regions are visited too rarely or not at all in the parallelised part of the programs to make a big practical impact.

Finally, there are the benchmarks that do see some benefit from conflict isolation, both in terms of improved DRF extension and its relevance in hot parts of the applications, Barnes, Cholesky, FMM, Radiosity, and Raytrace.

9.3 Compilation performance

Beside the impact in the compiled benchmarks, the speed of the compilation procedure itself has importance. Long compilation times are usually only acceptable if the undertaking is rare. Table 9.1 displays the compilation times of the considered benchmarks on a machine running Ubuntu 14.04 with an Intel Core i5-4250U CPU (1.3GHz, 32KiB+32KiB L1I/D, 256KiB L2, 3072KiB L3) and 8GiB of DDR3L-1600 RAM.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base (s)</th>
<th>xDRF (s) (norm.)</th>
<th>xDRF-CI (s) (norm.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>1.93</td>
<td>8.38 4.35</td>
<td>8.35 4.33</td>
</tr>
<tr>
<td>Cholesky</td>
<td>10.23</td>
<td>2201.56 215.25</td>
<td>2046.02 200.04</td>
</tr>
<tr>
<td>Dedup</td>
<td>1.38</td>
<td>2.02 1.46</td>
<td>2.03 1.47</td>
</tr>
<tr>
<td>FFT</td>
<td>1.12</td>
<td>1.54 1.38</td>
<td>1.54 1.37</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>1.90</td>
<td>2.44 1.28</td>
<td>2.44 1.28</td>
</tr>
<tr>
<td>FMM</td>
<td>3.65</td>
<td>168.15 46.06</td>
<td>194.93 53.39</td>
</tr>
<tr>
<td>LU</td>
<td>1.39</td>
<td>1.72 1.23</td>
<td>1.72 1.24</td>
</tr>
<tr>
<td>LU-nc</td>
<td>1.66</td>
<td>2.08 1.25</td>
<td>2.08 1.25</td>
</tr>
<tr>
<td>Ocean</td>
<td>10.60</td>
<td>14.67 1.38</td>
<td>14.67 1.38</td>
</tr>
<tr>
<td>Ocean-nc</td>
<td>7.38</td>
<td>9.33 1.27</td>
<td>9.33 1.27</td>
</tr>
<tr>
<td>Radiosity</td>
<td>4.25</td>
<td>4636.70 1089.96</td>
<td>4651.39 1093.41</td>
</tr>
<tr>
<td>Radix</td>
<td>0.80</td>
<td>1.03 1.29</td>
<td>1.03 1.29</td>
</tr>
<tr>
<td>Raytrace</td>
<td>5.53</td>
<td>24.50 4.43</td>
<td>94.23 17.04</td>
</tr>
<tr>
<td>Volrend</td>
<td>3.51</td>
<td>29.56 8.41</td>
<td>24.41 6.95</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>1.55</td>
<td>1.90 1.22</td>
<td>1.90 1.23</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>2.29</td>
<td>7.78 3.39</td>
<td>7.78 3.39</td>
</tr>
<tr>
<td>Water-Sp</td>
<td>2.55</td>
<td>5.70 2.23</td>
<td>5.88 2.31</td>
</tr>
</tbody>
</table>

G.Mean 2.71 12.30 4.54 13.25 4.89
Without xDRF delineation, all of the benchmarks have reasonable compilation times, only requiring a few to a dozen seconds to complete. Adding the xDRF-Compiler, the compilation times grows by a factor of 4.54 on average. It is, however, important to recognize that the overhead distribution is uneven across the benchmarks. In fact, the spikes in compilation overhead correlate to a degree with the amount of nDRF regions turned enclave (figure 9.1), which is explained by the increasing amount of cross-checking required for larger xDRF regions. For a few benchmarks the increased complexity results in extreme overhead: Cholesky (215x), FMM (46x), and Radiosity (1090x).

The introduction of the conflict isolation component in the xDRF-CI-Compiler sometimes allows xDRF regions to extend further than without it. When conflict isolation allows otherwise unavailable extensions there is theoretically an increased load on the xDRF analysis as it exposes a larger amount of code to be cross-checked in order to extend the xDRF region over the next nDRF region. The higher complexity manifests as an increase in compilation time over the standard xDRF-Compiler by 7.8% 4.89 times longer than compiling without xDRF extension. For the benchmarks where the standard xDRF-Compiler already achieves the perfect results, conflict isolation does not make any difference and the compilation takes the same time for both the xDRF-Compiler and the xDRF-CI-Compiler. Some of the remaining benchmarks, Barnes, Ocean, Ocean-nc, Water-Nsq, and Water-Sp, are barely affected in compilation time either. This can be attributed to the relatively small number of additional nDRF regions conflict isolation helps make enclave.

The rest of the benchmarks are a bit more interesting when comparing the xDRF-CI-Compiler compilation time against that of the xDRF-Compiler, Cholesky (−7.1%), FMM (+16%), Radiosity (+0.32%), Raytrace (+280%), and Volrend (−17%). Both Cholesky and Volrend counter-intuitively finish compilation faster under the xDRF-CI-Compiler while simultaneously making a few extra nDRF regions enclave. FMM and Raytrace also make a few extra nDRF regions enclave but as expected take longer to complete compilation under the xDRF-CI-Compiler, but in Raytrace the change is abnormally large with an almost 4 times as long compilation time. Another counter-intuitive result is Radiosity that despite having the best conversion rate for additional enclave nDRF regions with the xDRF-CI-Compiler over the xDRF-Compiler, has almost no relative difference in compilation time between the two. This can however be explained by the standard xDRF-Compiler already converting a large number of nDRF regions and performing a large quantity of cross-checking, hence the long compilation time. When adding conflict isolation even more nDRF regions are converted while most cross-checking is already done, causing only a small difference in compilation time.

Considering the counter-intuitive results it is fair to say that the number of extra converted nDRF regions is not a good predictor of how much longer a benchmark takes to compile under the xDRF-CI-Compiler compared to the xDRF-Compiler. Instead, the structure of the specific application is more likely to affect the compilation time.
9.4 Application: Dual-mode cache coherence protocol

The primary use of the xDRF delineation is as a guide for a dual-mode cache coherence protocol, SPEL++, described by Ros and Jimborean [35]. SPEL++ expands upon a standard directory-based cache coherency protocol by disabling coherence traffic inside DRF regions and performs necessary coherence transactions in bulk at the end of the regions such that overall coherence network traffic can be reduced and in turn save both execution time and energy.

With longer xDRF regions, in some cases made even longer with conflict isolation, coherence traffic can be consolidated over larger pieces of code, giving improved benefits. With xDRF regions, bulk transactions, or cache flushes, are performed at the end of the regions, i.e. when encountering a non-enclave nDRF region. As DRF semantics are not valid in any nDRF or iDRF regions, the coherency protocol momentarily switches to regular directory mode in enclave nDRF regions and iDRF regions that xDRF regions extend across.

In Jimborean et al.’s paper [27], the xDRF-CI and earlier delineations are evaluated with SPEL++ using the GEMS simulator [33], along with GARNET [21] modelling the interconnect network. A Pin tool [32] is used to provide information to the simulator such as executed instructions, memory references, and occurrences of symbols marking regions in the code. The simulator models a large-scale system with 64 in-order cores, private L1 caches, and a shared L2 cache.

Some of the results from the evaluation are repeated here to give insight into the benefits of conflict isolation in a practical application. More detailed simulation methodology, results, and analysis are available in Jimborean et al.’s paper [27].

As seen in figure 9.3, the SPEL++ protocol often outperforms the standard Directory protocol when it comes to network traffic, even under the relatively trivial DRF.
delineation. In most of the cases where the presumed optimal \textit{xDRF-Manual} delineation indicate remaining potential, the automatic \textit{xDRF-Compiler} delineation is able to match the expectation. The notable exception to this is \textit{Barnes} where the \textit{xDRF-Compiler} delineation does not improve upon \textit{DRF} under \textit{SPEL++}. Adding conflict isolation with the \textit{xDRF-CI-Compiler} delineation, the results for \textit{Barnes} improves to match the \textit{xDRF-Manual} approach turning the traffic overhead, compared to \textit{Directory}, into an improvement. Another beneficiary of conflict isolation is \textit{Radiosity} where the \textit{xDRF-CI-Compiler} delineation beats \textit{xDRF-Manual}, turning a small overhead into a small improvement. In \textit{Cholesky} and \textit{Raytrace} the \textit{xDRF-CI-Compiler} delineation also improves upon \textit{xDRF-Compiler}, but only to a very small degree.

Comparing the network traffic results to the occurrences of non-enclave \textit{nDRF} regions, particularly at runtime, discussed in previous sections, it is the same benchmarks that benefit from conflict isolation. The level of impact is, however, different for each benchmark and the scaling between delineations does not correspond well either (particularly obvious in \textit{Radiosity}).

On average, the \textit{xDRF-CI-Compiler} delineation reduces coherence network traffic by 32\% compared to \textit{Directory} and, mainly tanks to \textit{Barnes} and \textit{Radiosity}, by 3.1\% compared to the \textit{xDRF-Compiler}. The reduction of network traffic also lead to approximately proportional savings in energy consumption for the coherency interconnect.

Additionally, the coherence traffic reduction improves cache performance which translates into improved execution time. Figure 9.4 shows that change in execution time correlate to the reduction in coherence traffic but the impact is less pronounced since cache behaviour is not the only factor contributing to execution performance. On average, \textit{SPEL++} with the \textit{xDRF-CI-Compiler} delineation improves execution time by 8.5\% compared to \textit{Directory} and 0.63\% compared to \textit{xDRF-Compiler}.  

Figure 9.4: \textbf{Execution time, normalised to Directory.} \textit{Fluidanimate}, \textit{DRF}: 1.86.
Chapter 9. Evaluation

For the specific application of the dual-mode cache coherence protocol, the addition of conflict isolation improves upon the result of the standard xDRF analysis where it does not meet the expectation (Barnes). On occasion conflict isolation can even help to exceed the expectation (Radiosity). While improvements are rare, degradation in runtime performance following conflict isolation is never observed. This means that the addition is a useful incremental improvement worth using for the cases where it helps, without concern for negative effects in other cases. The only caveat to this is the significantly longer compilation time for Raytrace.
Chapter 10

Related work

Before being applied on general parallel code [28], xDRF classification was performed using synchronisation directives in OpenMP applications [34]. The benefit of doing this is that the OpenMP framework provides recognisable synchronisation patterns with information of what data is shared, making it easier to reason about what xDRF delineations are possible. The downside, which sparked the move to general code, is the limited applicability to programs that have been synchronised using OpenMP.

The idea of exploiting sequential semantics beyond synchronisation has been considered before. Joisha et al. [29] approach the problem by creating a procedural concurrency graph that indicates when objects in different procedures interfere with each other. This can be used to determine if there are any opportunities to use classical optimisation techniques.

Along similar lines, Effinger-Dean et al. [23, 24] create a data-centric classification called interference free regions (IFRs). An IFR is defined as the region of code around the use of a variable between the nearest acquire (or lock) before it and release (or unlock) after it. Assuming a data-race-free program, IFRs provide clues useful for a number of analyses. IFRs can also be used in dynamic data-race detection by looking for overlapping execution of matching regions. The xDRF analysis algorithm is, in contrast, centred on synchronisations. However, it statically performs a task similar to the IFR-based data-race detection. Making a synchronisation, or nDRF region, enclave would only be possible if there are no IFRs of a variable approaching the synchronisation, from before and after it in different threads, that would overlap if they were allowed to extend past the synchronisation.
Chapter 11

Conclusion

This project contributes a modification to the xDRF analysis developed in previous work to enlarge continuous reach of DRF semantics in parallel programs. The original analysis does this by creating xDRF regions that extend between DRF regions across synchronisations, called nDRF regions, where possible. The modification combats limitations in the original analysis arising from false positives when detecting memory conflicts. Instead of asserting an xDRF boundary when conflicts are detected, the conflicts are isolated in regions where DRF semantics are momentarily suspended. The outcome is longer xDRF regions at the cost of weakened semantics for potentially conflicting memory accesses between encompassed DRF regions.

The variants of the xDRF analysis are tested on benchmarks from the SPLASH-3 and PARSEC-2.1 suites. On average, the modified analysis reduces the number of xDRF boundaries by 41% or 88% depending on if counting statically in code or dynamically at runtime, respectively, an improvement over the original analysis by 21% or 51%. However, the improvements are mostly concentrated into a few select benchmarks. Used as a guide for a dual-mode cache coherence protocol, SPEL++, which reduces coherence traffic on the interconnect network and saves energy, the modified analysis improves results where the original did poorly.

The intent of the xDRF analysis is to discern what synchronisations enforces a happens-before behaviour and place xDRF boundaries there. The new conflict isolation breaks this idea as it allows extension in some happens-before situations as well, which for the SPEL++ application has turned out to be a positive where it makes a difference. It would be interesting to see if this aspect of conflict isolation can have further benefits if applied to barriers and signal-waits that are automatically assumed to be xDRF boundaries. Both for the present and any escalated form of conflict isolation, it is important to consider and investigate any detrimental effects isolation regions may have in any particular application.

Adding conflict isolation is a sure way to enlarge xDRF regions where false conflicts prevent extension. The price is the presence of isolation regions which could cause issues for some applications. However, for the primary application, SPEL++, conflict isolation provides a net improvement in the cases where the original xDRF analysis falls short of the expectation.
References


References


