Examensarbete

Exposure of Patterns in Parallel Memory Access

Björn Lundgren, Anders Ödlund

LiTH-ISY-EX--07/4005--SE
Exposure of Patterns in Parallel Memory Access

ISY / Datorteknik, Linköpings Universitet

Björn Lundgren, Anders Ödlund

LiTH-ISY-EX--07/4005--SE

Examensarbete: 20 p

Level: D

Supervisor: Dake Liu,
ISY / Datorteknik, Linköpings Universitet

Examiner: Dake Liu,
ISY / Datorteknik, Linköpings Universitet

Linköping: September 2007
Exposure of Patterns in Parallel Memory Access

Björn Lundgren, Anders Ödlund

The concept and advantages of a Parallel Memory Architecture (PMA) in computer systems have been known for long but it’s only in recent time it has become interesting to implement modular parallel memories even in handheld embedded systems. This thesis presents a method to analyse source code to expose possible parallel memory accesses. Memory access Patterns may be found, categorized and the corresponding code marked for optimization. As a result a PMA compatible with found pattern(s) and code optimization may be specified.

Nyckelord
ASIP, Parallel Memory, Memory Access Pattern, Static Code Analysis
Abstract

The concept and advantages of a Parallel Memory Architecture (PMA) in computer systems have been known for long but it’s only in recent time it has become interesting to implement modular parallel memories even in handheld embedded systems. The main uses of PMAs are with vector and macro block access in applications with high quantities and rates of data, e.g. streaming media, computer graphics and telecommunications.

To fully utilize a PMA it still need to be adapted to the proposed system and the minimal resulting complexity will depend on the application. This thesis presents a method to analyse source code to expose possible parallel memory accesses. It also discuss the use of the result form this analysis when finalizing the system.

Through an extension to the popular GCC compiler, the intermediate representation of compiled code can be made accessible to the developer and by analysing this representation of the code, the parts relating to memory accesses can be exposed. A tool for doing this, Memorizer, is presented. Memory access Patterns may be found, categorized and the corresponding code marked for optimization. As a result a PMA compatible with found pattern(s) and code optimization may be specified.

The thesis include case studies on method use in a streaming DSP system.

Keywords: ASIP, Parallel Memory, Memory Access Pattern, Static Code Analysis
Acknowledgements

We would like to thank Prof. Dake Liu, for support and help, our opponents Kristofer Bergh and Oskar Karlsson for their review of the thesis and Björn Skoglund for his initial work on Relief and his ideas on writing profiling tools. We would also like to thank Anders Lundgren for his proof-reading of and linguistic comments on our final publication. Finally, thanks to family and friends for emotional and motivational support during our thesis work.
Nomenclature

Most of the reoccurring abbreviations and symbols are described here.

Abbreviations

1D One-Dimensional
2D Two-Dimensional
3D Three-Dimensional
ADD Application-Driven Design
ASIP Application Specific Instruction set Processor
B Bytes
BBP Base Band Processing
DMA Direct Memory Access
DP Data Path
DSP Digital Signal Processing
GCC Gnu Compiler Collection, A popular free compiler.
GEM GCC Extension Modules, Framework for plugins to GCC.
GPU Graphical Processing Unit
HdS Hardware dependent Software
ILP Instruction Level Parallel
IR Intermediate Representation
LUT Look Up Table
MaCT Memory Access Code Template
MaE Memory Access Exposition
MaP Memory Access Pattern
P3RMA Predictable Programmable Parallel memory architecture for Random Memory Access
PMA Parallel Memory Architecture
PVM Parallel Vector Memories
RF Register File
SIMD Single Instruction, Multiple Data
STDERR Standard Error, where error and debug output from a program is normally written.
STDOUT Standard Output, where output from a program normally is written.
STL Standard Template Library
ULRF Ultra Large Register File
VLIW Very Long Instruction Word
Writing Conventions

filename  Names of files.
function  Names of functions source files.
ClassName  Names of classes in source files.
Keyword  Keywords, like the ones in configuration files.

Mathematical Notations

$N$  Number of memory modules in a specific PMA
$r$  sample address
$R$  Scanning field, set of sample addresses, $r \in R$
$S(r)$  Module assignment function, $S : R \mapsto \{0,1,\ldots,N-1\}$
$a(r)$  Address function, $a : R \mapsto \{0,1,\ldots,a_{max}\}$
$F$  Access format
$M$  Number of sample accesses in a specific access format, $M \in \{1,2,\ldots,N\}$
$F(r)$  Access format placed at $r$
$\pi$  output permutation function
$\pi^{-1}$  input permutation function
$\mathbb{Q}$  The set of rational numbers
$\mathbb{Z}$  The set of integers
$\mathbb{N}$  The set of positive integers
$L$  Row length, raster width
$n$  Number of (1D) MaP elements, $n \in \mathbb{N}$
$P$  MaP, general
$a$  MaP element, absolute
$P(r)$  MaP, relative
$e$  MaP element, relative
$\dot{P}(r)$  MaP, differential
$\dot{e}$  MaP element, differential
$P_s$  Constant stride MaP
$s$  Stride, $s \in \mathbb{Z}$
$P^1$  1D MaP
$P^2$  2D MaP
$P^D$  MaP of dimension $D$
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>ix</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>xi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Project Description</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Limitations &amp; Scope</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Method Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Workflow and Work Distribution</td>
<td>3</td>
</tr>
<tr>
<td>1.6 Thesis Outline</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Design for Parallel Data Access</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 P3RMA</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Parallel Vector Memories</td>
<td>6</td>
</tr>
<tr>
<td>2.3 System Design with P3RMA</td>
<td>7</td>
</tr>
<tr>
<td><strong>I Theory</strong></td>
<td>9</td>
</tr>
<tr>
<td><strong>3 Parallel Memory Architecture</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 General PMA Concepts and Model</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 Data Representation</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Assignment Functions</td>
<td>12</td>
</tr>
<tr>
<td>3.2.3 Access Format</td>
<td>13</td>
</tr>
<tr>
<td>3.2.4 Permutation</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Conflict Free Access</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Raster Memory Representation</td>
<td>16</td>
</tr>
<tr>
<td><strong>4 Static Code Analysis</strong></td>
<td>17</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Compiler Basics</td>
<td>17</td>
</tr>
<tr>
<td>4.2.1 Structure of a Compiler</td>
<td>18</td>
</tr>
<tr>
<td>4.2.2 Intermediate Representation</td>
<td>18</td>
</tr>
<tr>
<td>4.2.3 Basic Blocks</td>
<td>21</td>
</tr>
<tr>
<td>4.2.4 Loop Detection</td>
<td>21</td>
</tr>
<tr>
<td>4.3 Where to Analyse</td>
<td>22</td>
</tr>
</tbody>
</table>
## II Model and Method

### 5 Memory access Pattern Concepts

#### 5.1 Introduction

#### 5.2 Definition and Relations

- 5.2.1 Memory Access Pattern (MaP)
- 5.2.2 Memory Access Exposition (MaE)
- 5.2.3 Memory Access Code Template (MaCT)

#### 5.3 MaP Application

- 5.3.1 Patternable MaE and Exposable Code
- 5.3.2 MaCT Application

#### 5.4 Mathematical Description and Corresponding Templates

- 5.4.1 MaP as an Array
- 5.4.2 MaP as a Function

#### 5.5 Multidimensional MaP

- 5.5.1 Multidimensional MaP as a Matrix
- 5.5.2 Multidimensional MaP as a Function

#### 5.6 MaP Categories

- 5.6.1 Constant Stride Memory Access

### 6 Memorizer

#### 6.1 Introduction

#### 6.2 Design of Memorizer

- 6.2.1 GEM
- 6.2.2 Relief
- 6.2.3 From Relief to Memorizer
- 6.2.4 Requirement Specification
- 6.2.5 Input
- 6.2.6 Function
- 6.2.7 Output
- 6.2.8 Limitations of Memorizer

#### 6.3 Implementation

- 6.3.1 Patching GCC
- 6.3.2 Classes
- 6.3.3 Debugging
- 6.3.4 GEM Interface

#### 6.4 User Guide

- 6.4.1 Installing Memorizer
- 6.4.2 Compiling with Memorizer
- 6.4.3 Configuring Memorizer
- 6.4.4 Memorizer Workflow
III  Case studies and Result

7  Pipe Clean by Case Studies
7.1 Introduction
7.2 Proposed Memory System
7.3 P3RMA Analysis
  7.3.1 The Memorizer MaE
  7.3.2 MaP Categorizing
  7.3.3 PVM Access formats
7.4 How to read the MaE tables
7.5 DCT of Macroblock
  7.5.1 Memorizer Output and Construction of MaE
  7.5.2 Analysis of Output
7.6 Motion Compensation in x264
  7.6.1 Memorizer Output
  7.6.2 Construction of MaE

8  Result and Conclusion
8.1 Memory Access Pattern
8.2 Memorizer & Memory Access Exposition
8.3 Memory Access Exposition Analysis
8.4 Conclusion
8.5 Future Work
  8.5.1 Memory Access Pattern Concepts
  8.5.2 Memory Access Pattern Usage
  8.5.3 Memorizer

IV  Appendices

A  Memorizer XML Output

B  G++ Patch

C  jpeg_fdct_islow()

D  Memorizer Output from DCT Case Study

E  Memorizer Graph Output from DCT Case Study

F  motion_compensation_chroma()

G  Memorizer Output from x264 Case Study

List of Listings

List of Figures

Bibliography

Index
Chapter 1

Introduction

The concept and advantages of a Parallel Memory Architecture (PMA) in computer systems have been known for long but it’s only in recent time it has become interesting to implement modular parallel memories even in handheld embedded systems. The main uses of PMAs are with vector and macro block access in applications with high quantities and rates of data, e.g. streaming media, computer graphics and telecommunications.

1.1 Project Description

At the beginning of our work, the following text was written together with Prof. Dake Liu to be used as the project description under which our thesis would be written.

ASIP DSP sales in 2006 were USD 15 billion out of a total USD 208 billion semiconductor sale; however, academic research of ASIP is not a trivial task. The research can be too complicated and the scale of a project may be too large to be managed at a university department. To speed up and further qualify our research, finding the right methodology and establishing our supporting tool becomes the next step. At the same time, the industry requires qualified tools to design and use ASIP. Previous research can be found for ASIP tool chain, e.g. LISA, EASCAL, or Liberty, and there are already some profiling tools available, However neither of these supply fast yet accurate source code analysis for architecture and instruction set decision in ASIP design.

The goal of the project is to investigate methods and design a source code profiling tool to expose run time cost, memory costs, and vector addressing. The tool will be used to identify 10-90% localities, function coverage, identical subroutines, and similar subroutines. The tool will also be used to match vector addressing models for data pre-allocation of the main memory and permutation in scratch pad memories of ASIP.

The research conducts source code analysis by static profiling and cost annotation by dynamic profiling. All research activities are
Introduction

based on GNU. The Alpha version is working now. A Beta version can be expected and available in September 2007.

1.2 Objectives

Since August 2001 the research focus of the computer engineering division at Linköpings Universitet has been ASIP design. ASIPs give several benefits over general purpose processors, better performance, lower power consumption and lower silicon cost, the drawback is that designing an ASIP for a product will most probably increase the design time compared to if a general purpose microprocessor had been used. To counter this, techniques to cut the design time of ASIPs are developed.

As stated in the project description, the goal of the project is to investigate methods and design a source code profiling tool to expose run time cost, memory costs, and vector addressing. This thesis concentrates on the vector addressing part, and template based design for vector addressing.

We have chosen our own construct “Memory Access Pattern” (MaP) to avoid confusion due to multiple meanings and interpretations of words such as “template”. With the project goal in mind, and the new MaP construct we concluded that the question we want to answer in this thesis is.

- Is it possible to expose and identify Memory Access Patterns in source code?

The keywords in that question are expose, identify and Memory Access Patterns which leads us to the following three sub-questions.

- How do we define a Memory Access Pattern?
- How do we expose a Memory Access Pattern from Source Code?
- How can we identify a Memory Access Pattern?

This thesis will try and give answers to those three questions.

1.3 Limitations & Scope

The completion of work laid out in the project description would be more suited to a Ph.D. project, this final year project is therefore only a small part of the complete project. Listed here are limitations we made in order to make our part manageable.

- We are not going in depth into the physical implementation of our results, but are concentrating on a logical theoretical level.
- The static code analysis is done in the middle end of the compiler where no physical attributes is taken into account, e.g. a pointer is a logical representation of a memory address location that does not have a specific physical address connected to it.
• The parallel memory architectures we work with are also logical repre-
sentations, which prevents us from being locked to any specific physical
implementation. For example a PMA can have any number of memory
banks and are not confined to be just four or eight.

• We don’t give any general solution how to automatically go from source
code to DMA and Permutation table, instead the case studies we perform
shows how to use our tool and method to create these tables.

1.4 Method Overview

In this thesis a tool, Memorizer is used to expose memory accesses and address-
calculations in source code using a technique called Static Code Analysis,
from the output of Memorizer, a Memory access Exposition (MaE) is created.
From this MaE, if possible, a Memory access Pattern (MaP) is generated. This
MaP can be used as help when designing a Parallel Memory Architecture (PMA)
or to optimize the program for a predefined PMA.

1.5 Workflow and Work Distribution

The first part of the thesis work were to set up the objectives and foundations for
our later work, most of the material handled during this period were not included
in this report for reasons of limitation and relevance. After basic objectives were
set the work were divided into two parts; The construction of a tool to expose
memory access (Anders) and to find out how to identify and apply exposed data
(Björn). The identification/application part became dependent on the results
from the tool part and were therefore focused on setting requirements on the
later.

In parallel with the two main parts a case study model and workflow were
decided. With preliminary results from the two parts the research went into a
test phase with case studies, resulting in revising of both tool and formalization.
Revised model, case studies and results were then presented in this thesis report.

1.6 Thesis Outline

The introducing chapters of the thesis include background information and con-
text description.

Chapter 1 Introduction Thesis background, scope and overview.

Chapter 2 Design for Parallel Data Access This chapter introduce the back-
ground concepts and foundations of the thesis research.

Part I, Theory

In the theory part reference knowledge gained during the work is summarized.

Chapter 3 Parallel Memory Architecture This chapter is a summary on
the subject of parallel memory architecture with focus on a formal model
for describing them.
Chapter 4 Static Code Analysis  Description of Static Code Analysis in general and in combination with GCC in particular

Part II, Model and Method
This part describe model, concepts and tools used and how they were used.

Chapter 5 Memory access Pattern Concepts  This chapter will define and describe the concept of a Memory access Pattern (MaP), a formal representation of parallel memory accesses, invented as a part of the thesis research.

Chapter 6 Memorizer  Description of the tool Memorizer, a proof of concept that memory access and addressing calculations in fact can be exposed.

Part III, Case studies and Result
Concluding part, presenting the case studies and thesis results.

Chapter 7 Pipe Clean by Case Studies  Pipe Cleaning model and examples, from source code via exposed MaP to DMA- and Permutation-tables.

Chapter 8 Result and Conclusion  Thesis result presentation, summary of the thesis work related to set objectives and suggestions on future work in the area.

Part IV, Appendices
In this part, the appendices, our references and an index is included.
Chapter 2

Design for Parallel Data Access

This chapter introduce the background concepts and foundations of the thesis research.

2.1 Introduction

A common problem in computer engineering is the limited bandwidth between memory and processors, the so called von Neumann bottleneck. The memory system can’t supply data at the same rate as it’s processed, which results in a hungry data path when the processor has to stall waiting for data. This is especially the case in many SIMD, VLIW and vector processing systems where the data path bandwidth usually is much greater than the memory bandwidth. Figure 2.1 shows the relative costs of different memory accesses in different applications. For some applications it’s clear that improving the transfer from the main memory to the vector memory would greatly increase the total performance.

2.2 P3RMA

P3RMA stands for Programmable Parallel memory architecture for Predictable Random Memory Access. P3RMA is one of the main memory solutions to supply parallel data to computing engines including its hardware architecture and methodologies of embedded parallel programming. [7]

A P3RMA based system uses parallel scratch pad memories instead of relying on caches or ultra large register files. These on chip scratch pads are connected to the main memory via a wide data bus. Data shuffling between the main memory and the scratch pads is taken care of by a DMA unit and a special permutation unit.

The most important thing with PR3MA is a strong programmers tool chain and methodology to utilize the hardware.
2.2.1 Parallel Vector Memories

In [7] the scratch pad based Parallel Vector Memories (PVM) illustrated in figure 2.2 is presented to replace caches and ultra large register files (later sections will explain why those two alternatives aren’t good solutions to the von Neumann Bottleneck.

This PVM allows for parallel access to eight values in parallel, with the restriction being that it can only read one value from one Data Memory Block each clock cycle. With predictable addressing algorithms the Permutation Network can be configured to allow conflict free access to data. Programming or designing a PVM requires knowledge of the addressing algorithms and is a good candidate for design automation.
Discarded alternative 1: Cache

Caches are a good way to increase performance when a strong temporal locality exists in the data, this is often not the case in streaming DSP applications and the missrate in the cache will therefore be rather high. Cache misses will vary the access time which is not ideal for realtime processes. Another issue with caches is that they don’t support different access formats, like row- vs. column-access, whereas a well designed PVM can access a wide variety of access formats, see figure 2.3.

There are of course benefits of caches as well, they are transparent to the software developer and might give a performance increase without any special input from the developer.

![Figure 2.3: Advantage of using PVM, from 7](image)

Discarded alternative 2: Ultra Large Register File

A common solution to the streaming DSP memory bandwidth problem is to increase the size of the Register File (RF). Ultra Large Register Files (ULRF) can store many kilobytes of data and all registers are available as input to vector instructions. ULRF works well for data that shows strong spatial locality if there are no restrictions on silicon area or power consumption. For embedded systems or high volume products ULRF is often avoided, ULRF is also a bad choice for applications with very big addressing space, like parallel video signal processing. [7]

2.3 System Design with P3RMA

System Design using P3RMA methodology follows the basic steps outline in figure 2.4. Algorithms are described on behavioural level in C-code or other some other high level, such as MATLAB. The algorithms are rewritten to a Fixed-Point Behavioural Model (Bit Accurate Model), this software model should be equivalent to a behavioural hardware model.
The Fixed-Point Behavioural Model is further refined to Memory Accurate Model by partitioning different parts of the software to different parts of the hardware and allocating data to different parts of the memory system. When Memory partitioning is done the model is turned into a Transaction Accurate Model by adding transaction synchronization and taking transfer times between memories in account.

When the above steps are done the algorithm is translated to assembler code following the given processor and memory constraints.

The work presented in thesis is connects to the Memory Accurate Model, the Memorizer tool helps exposing memory costs and the MaP concept is a useful tool when choosing or designing a PVM system.
Part I

Theory
Chapter 3

Parallel Memory Architecture

This chapter is a summary on the subject of parallel memory architecture with focus on a formal model for describing them.

3.1 Introduction

A Parallel Memory Architecture (PMA) could be anything from a local on chip scratchpad memory to an external distributed memory system. The common element of PMA is that the system include several so called memory modules and that data are dispersed between these modules to enable parallel access.

A PMA got four main tasks:[10]

1. Provide any conflict free parallel access required by the application.
2. Keep track of to which module the data has been assigned.
3. Keep track of the data address inside the memory module.
4. Permutation of input and output data.

In short no. 2, 3 and 4 is essential to make no. 1 possible. Conflict free access imply that only one access per memory module is requested at one time, which enables low latency parallel access.[5]

3.2 General PMA Concepts and Model

This section is mainly based upon a PMA model presented in [5]. The purpose of this model is to formalize the handling of data accesses within a PMA system and present some concepts to use in later analysis.

In the figure 3.1 a general PMA is presented, it consists of \( N \) memory modules \((S_0, S_1, \ldots, S_{N-1})\), a address computation unit \( A \) and a input/output permutation unit \( \Pi \).
3.2.1 Data Representation

The model uses a semi-logical data representation where the memory modules store samples grouped into scanning fields.

A sample represents the data of one (nonparallel) access to one memory module. The model does not specify detailed properties, like range or complexity, of a sample and theoretically it can represent data of any size. Practically a sample usually represents a common access element like a byte, a pixel or a floating point number. In the model each sample has its own unique logical address $r$ and explicit value $v(r)$.

A scanning field represents a set of (dependent) samples. Practically scanning fields usually represent known data objects like an image macroblock, a table or a matrix. In the model a scanning field is presented as a set $R$ of sample addresses $r \in R$. In this model $R$ is usually a straight-line scanning field, that is, defined as an address range $[r_{\text{min}}, r_{\text{max}}]$, e.g. $r \in R = \{0, 1, \ldots, 15\}$ for a simple straight-line scanning field of 16 samples.

3.2.2 Assignment Functions

Assignment functions are functional representations of the sample storage allocation mechanisms of the PMA.

The module assignment function assigns each sample to a specific module. It’s usually an integer function returning the module number to place the sample $(r)$ in.

$$ S : R \mapsto \{0, 1, \ldots, N - 1\} $$

That is, if $S(r) = i$ then sample $r$ is placed in module $S_i$, $i \in \{0, 1, \ldots, N - 1\}$.
3.2 General PMA Concepts and Model

The address function assigns each sample its in-module sample address, usually using a range from 0 to $a_{\text{max}}$ ($a_{\text{max}} = \text{module size in samples} - 1$).

$$a : R \mapsto \{0, 1, \ldots, a_{\text{max}}\}$$

Example 3.1 (PMA assignment functions). Figure 3.2 show an example scanning field of a 4x4 monochrome picture ($r \in R = [0,15]$, $v(r) \in \{0,1\}$). This scanning field is stored in a PMA using four modules ($S(r) \in \{0, 1, 2, 3\}$) of four samples each ($a(r) \in \{0, 1, 2, 3\}$). The PMA uses assignment functions:

$$S(r) = \left\lfloor r + \frac{r}{4} \right\rfloor \mod 4$$

$$a(r) = \left\lfloor \frac{r}{4} \right\rfloor$$

Allocation result for the scanning field is presented in figure 3.3 and resulting module contents is presented in figure 3.4.

3.2.3 Access Format

The PMA access control signal (input $F$ in Figure 3.1), referred to as the access format, is identifying which samples to be accessed and their order of access.

$$F = \{r^0, r^1, \ldots, r^{M-1}\}, \quad M \in \{1, 2, \ldots, N\}$$
Definition 3.1 (Access format). The access format $F$ is a mathematical formalization of the logical values representing the control signals used for data access in a PMA.

By including a placement sample $r \in R$ it can be described as a set of offsets $e$, relative to the placement sample. This position independent access format representation $F$ with $F(r) (F_M : R \rightarrow R^M)$ is quite useful for later use.

$$F = (e^0, e^1, \ldots, e^{M-1})$$

$$F = F(r) = \{r + e^0, r + e^1, \ldots, r + e^{M-1}\}, \ M \in \{1,2,\ldots,N\}$$

3.2.4 Permutation

Given $F$ and $S(r)$ the corresponding output ($\pi$) and input ($\pi^{-1}$) permutations becomes:

$$\pi(F, S) = \begin{pmatrix} S(r^0) & S(r^1) & \cdots & S(r^{M-1}) & \cdots \\ 0 & 1 & \cdots & M-1 & \cdots & N-1 \end{pmatrix}$$

$$\pi^{-1}(F, S) = \begin{pmatrix} S(r^0) & 1 & \cdots & M-1 & \cdots & N-1 \\ 0 & S(r^1) & \cdots & S(r^{M-1}) & \cdots \end{pmatrix}$$

Example 3.2 (PMA access). Figure 3.5 shows an example of read access to a PMA with data and assignment functions from example 3.1:

1. The access targets the second column in the image (see figure 3.2):

   Column access format,
   $$F_c = (0,4,8,12)$$

   placed at $r = 1$

   $$F = F_c(1) = \{1,5,9,13\}$$

2. The address computation unit $A$ convert the access format to corresponding sample addresses by use of the assignment functions (from example 3.1):

   $$S(F) = \{1,2,3,0\}$$
   $$a(F) = \{0,1,2,3\}$$

3. Out of order sample values from memory modules.
4. The permutation unit Π output in order sample values (v(F)) by use of output permutation function:

\[ \pi(F) = \begin{pmatrix} 1 & 2 & 3 & 0 \\ 0 & 1 & 2 & 3 \end{pmatrix} \]

### 3.3 Conflict Free Access

In the above examples a so called linear module assignment function have been used, that is when:

\[ S(r) = \lfloor q \cdot r + p \rfloor \mod N \quad q, p \in \mathbb{Q} \]

There are a number of common module assignment functions, all with the purpose of providing conflict free access with respect to specific access format. [5]

**Definition 3.2** (Conflict free access). A module assignment function \( S(r) \) is said to provide conflict free access with respect to the access format \( F(r) \) if, for all \( r', r \in F(r) \) with \( r' \neq r \):

\[ S(r') \neq S(r) \]

**Example 3.3** (Conflict free access). With the module assignment function from example 3.1 conflict free access is provided for row access (placed at \( r = 8 \), third row)

\[ F_r = (0, 1, 2, 3) \Rightarrow S(F_r(8)) = (2, 3, 0, 1) \]

and column access (placed at \( r = 3 \), fourth column)

\[ F_c = (0, 4, 8, 12) \Rightarrow S(F_c(3)) = (3, 0, 1, 2) \]
but not for diagonal access (placed at $r = 0$, forward diagonal)

$$F_d = (0, 5, 10, 15) \Rightarrow S(F_d(0)) = (0, 2, 0, 2).$$

### 3.4 Raster Memory Representation

When applying the PMA model two-dimensional scanning fields, e.g. image or matrix data, it’s common to use a two-dimensional scanning field representation, a so called raster. In that case the samples $r$ get a dual parameter representation $r = (i, j)$, where $r == i + j \cdot L$, $i = r \mod L$ and $j = \lfloor \frac{r}{L} \rfloor$. The constant $L$ refers to the (row) width of the raster.

**Example 3.4 (Raster representation).** The data from example 3.1 can be represented in a raster $r = (i, j)$ of width $L = 4$ where $i = r \mod 4$ and $j = \lfloor \frac{r}{4} \rfloor$. This results in new assignment functions

$$S(r) = S(i, j) = i + j \mod 4$$

$$a(r) = a(i, j) = j$$

and new access formats for row access;

$$F_r = ((0, 0), (1, 0), (2, 0), (3, 0)) = (0, 1, 2, 3) (1, 0)$$

column access;

$$F_c = ((0, 0), (0, 1), (0, 2), (0, 3)) = (0, 1, 2, 3) (0, 1)$$

and diagonal access;

$$F_d = ((0, 0), (1, 1), (2, 2), (3, 3)) = (0, 1, 2, 3) (1, 1)$$
Chapter 4

Static Code Analysis

This chapter will introduce the reader to Static Code Analysis, the technique we use to expose parallel memory accesses from source code during compile time.

4.1 Introduction

Static Code Analysis is an analysis of computer software done without executing the program, in contrary to Dynamic Analysis, which is done during run time of the software.

Static Code Analysis is a good choice when developing for embedded systems since the analysis is performed on the textual source code, or some binary representation of it, which means that the analysis can be done on the development platform and not on the target platform. The analysis is also not depending on input data, execution of a program can differ from time to time depending on different input data.

Doing a static analysis of the code is also good when you want to expose patterns in the source code, and be able to link the output of the analysis to the source code.

The strengths of Static Code Analysis are also it drawbacks, for example if data dependency has to be taken into account Dynamic Analysis has to be done instead.

4.2 Compiler Basics

A compiler is a tool that translates a program written in a programming language such as C or Pascal into object code that a computer can understand and execute.

![Figure 4.1: A Compiler](image)

In the process of translating the code, the compiler also check for errors,
telling the developer if a mistake has been made, optimizes code and take care of nuisances that developers might not want to care about like memory handling.

4.2.1 Structure of a Compiler

Traditionally [1] a compiler is divided into two major parts, front end and back end. The front end is responsible for analysing the source code and translate it into an Intermediate Representation (IR) that the back end in turn translates into object code for the target platform. This division ideally puts all source language details in the front end, and all details regarding the target platform are handled by the back end.

![Figure 4.2: Front End and Back End](image)

Before sending the Intermediate Representation to the back end, the compiler can perform various hardware independent optimizations, or transform the IR in other ways. This leads to a three-tiered structure of a compiler, with a front end, a back end and a middle end.

![Figure 4.3: A three tiered compiler](image)

4.2.2 Intermediate Representation

Intermediate Representation (IR) is used by compilers as the common language between the Front End and the Back End. By using a well defined IR, different front ends can be combined with different back ends to create a plethora of compilers for different source languages to different target platforms. As will be shown later, the IR is also very useful when perform static code analysis.

In this section two different IRs will be presented, one theoretical and one real implementation that is used in the GCC compiler.

Three-Address Code

Implementations of Three-address Code is a common Intermediate Representation used by compilers during the optimization phases of compiling. In Three-Address Code each statement consists of at most one result operand and two source operands. For example the statement $a = b + c * d$ would be translated into:

1 The temporary variable $T_2$ is strictly not necessary but added for the sake of the example.
\[ T_1 = c \times d \]
\[ T_2 = b + T_1 \]
\[ a = T_2 \]

It is interesting to note here that in each basic block, it’s easy to find dependencies between variables, since all statements will be executed in order. It’s therefore obvious that in the the example above the variable \( a \) is depending explicitly on \( b, c \) and \( d \) through the temporary variables \( T_1 \) and \( T_2 \).

Dependencies can also be implicit, consider the following Three-Address Code.

\[ a = b \times 2 \]
\[ b = c \]

If this code would stand by itself, \( a \) would not be dependant on \( c \), but if the code is part of a loop \( a \) might depend on \( c \) implicitly.

**GIMPLE**

GIMPLE and its subset GIMPLE is the Intermediate Representation used in GCC. It’s an Abstract Syntax Tree (AST) representation of Three-Address Code.

Every node in the AST is a tree of it’s own, so in this context the words tree and node will be used interchangeable.

Each tree has a code which identify what it represents. Tree codes are grouped in classes, a listing of some of the codes and classes can be found in Table 4.1. This is only a selection of codes to give the reader a feeling for what kind of data each node represents. For full documentation of all available tree codes, look in the file `tree.def` in the GCC source [4].

<table>
<thead>
<tr>
<th>Tree Code Class</th>
<th>Tree Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcc_declaration</td>
<td>VAR_DECL</td>
<td>Declaration of a variable.</td>
</tr>
<tr>
<td></td>
<td>PARM_DECL</td>
<td>Declaration of a function parameter.</td>
</tr>
<tr>
<td>tcc_reference</td>
<td>INDIRECT_REF</td>
<td>Pointer references. (Unary * operator)</td>
</tr>
<tr>
<td></td>
<td>ARRAY_REF</td>
<td>Reference to array elements.</td>
</tr>
<tr>
<td>tcc_expression</td>
<td>MODIFY_EXPR</td>
<td>Assignment expression.</td>
</tr>
<tr>
<td></td>
<td>CALL_EXPR</td>
<td>Function call.</td>
</tr>
<tr>
<td>tcc_binary</td>
<td>PLUS_EXPR</td>
<td>Addition of two values.</td>
</tr>
<tr>
<td></td>
<td>MAX_EXPR</td>
<td>Maximum of two values.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tree Code Class</th>
<th>Tree Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcc_declaration</td>
<td>VAR_DECL</td>
<td>Declaration of a variable.</td>
</tr>
<tr>
<td></td>
<td>PARM_DECL</td>
<td>Declaration of a function parameter.</td>
</tr>
<tr>
<td>tcc_reference</td>
<td>INDIRECT_REF</td>
<td>Pointer references. (Unary * operator)</td>
</tr>
<tr>
<td></td>
<td>ARRAY_REF</td>
<td>Reference to array elements.</td>
</tr>
<tr>
<td>tcc_expression</td>
<td>MODIFY_EXPR</td>
<td>Assignment expression.</td>
</tr>
<tr>
<td></td>
<td>CALL_EXPR</td>
<td>Function call.</td>
</tr>
<tr>
<td>tcc_binary</td>
<td>PLUS_EXPR</td>
<td>Addition of two values.</td>
</tr>
<tr>
<td></td>
<td>MAX_EXPR</td>
<td>Maximum of two values.</td>
</tr>
</tbody>
</table>

Table 4.1: Some of the different trees in GCC

Each tree has a numbers of operands, which also are trees. The number of operands are known depending on tree code, for example a PLUS_EXPR always has 2 operands.
In order to build lists of trees, trees may be chained together in a linked list, the list ends when a tree is chained to NULL_TREE.

To give an example of how the tree structure is used, consider the code in Listing 4.1

int a, b, c;
a = b + c;

Listing 4.1: C code of a=b+c

GCC provides the function debug_tree that dumps a human readable representation of a tree to STDERR, as seen in Listing 4.2 (The listing is edited for readability). The tree in Listing 4.2 starts with a modify_expr node, operand 0 is the var_decl tree representing the variable $a$ where the result will be saved, operand 1 is a PLUS_EXPR tree with var_decl nodes for $b$ and $c$ as it’s two operands.

The observant reader might wonder why the different var_decl nodes contain such a varying amount of information. This is a result of how the function debug_tree is implemented, it strives not to go too deep into the tree when printing it, and therefore the difference is purely cosmetic.
Of interest is also the chaining between the var_decl nodes that can be seen on lines 15-19; all variables declared in the same scope is chained together like this to form a list.

### 4.2.3 Basic Blocks

In the Dragon book [1] a Basic Block is defined as

**Definition 4.1 (Basic Block).** A maximal sequence of instructions with the following properties

1. The flow control can only enter the basic block through the first instruction in the block.
2. Except for at the last instruction of the block, control will not leave the block due to halting or branching.

Following this definition a basic block is a very powerful entity when analysing the Intermediate Representation of the source code. All instructions in the basic block will be executed in the exact order they are given in the block [2], thus you can find out relations between variables and expressions in a reliable way.

### 4.2.4 Loop Detection

A Basic Block needs to know from which other blocks the control flow can enter, and to what blocks it can exit. This information forms a Control Flow Graph and can be used to find loops in the code, to do this we introduce the concept of Dominators.

**Definition 4.2 (Dominator).** A node d dominates a node n if every path in the control flow graph from the entry node to node n goes through d. d is then called a dominator of n, this also means that every node is a dominator of itself.

From this definition it can be deducted that if a node has an outgoing edge leading to one of it’s dominators, that edge is a so called back edge, and it’s the edge that closes the loop.

To exemplify, look at Figure 4.4. Node 1 is the entry node which means it dominates every other node in the graph, including itself. Node 2 is dominated by 1 and 2. Node 3 is not dominated by 2 since the flow can go directly from Node 1 to Node 3, thus the dominators for 3 is 1 and 3. Node 4 is dominated by 1, 3 and 4. Since 4 has an edge leading to 3, one of it’s dominators, that edge is forming a loop including the nodes 3 and 4. Node 5 and 6 are both dominated by 1, 3 and 4, and 5 respective 6. Node 7 is dominated by 1, 3, 4 and 7, but neither of 5 or 6. Node 7 has an edge leading back to 1, one of it’s dominator, and a loop is formed.

One problem can arise when using this way to find loops in the code. If a node has a back edge to a node that is not one of it’s dominators, as in Figure 4.5, the graph is said to be nonreducible and the mentioned way to find loops doesn’t work.

Fortunately, this kind of loops seldom occur, since normal control flow structures like if-then-else and do-while does not result in nonreducible code [1].

---

[1]: Unless an interrupt occurs
4.3 Where to Analyse

Static code Analysis can be done on different representations of the program. The source code can be parsed and analysed in its original form, the resulting object code available after compilation can be analysed, or the analysis can be done on some intermediate representation used by the compiler during compilation.

To compare these three different options the C code in Listing 4.3 will be used. It’s a simple arithmetic expression $b + c \times d$ where the result is stored in the variable $a$ referenced by the pointer $p$.

4.3.1 Source Code

```c
int main(void)
{
    int a, b, c, d;
    int *p;
    p = &a;
    *p = b + c * d
}
```

Listing 4.3: C code of $b + c \times d$

In order to analyse the code in Listing 4.3 a parser for the C code needs to be written. This is a good option if you don’t want to be dependent on a specific compiler but requires a lot of work. The example above looks rather easy to parse, but usage of for example preprocessor macros can make the parsing a lot harder.

4.3.2 Intermediate Representation

Listing 4.4 is the output of the function `debug_bb()` in GCC, this function prints out the content of a single basic block. It doesn’t look very different from the C source code, but the calculation is split up into Three Address Code as
4.3 Where to Analyse

\[
\begin{align*}
\text{;; basic block 0, loop depth 0, count 0} \\
\text{;; prev block -1, next block -2} \\
\text{;; pred: ENTRY (fallthru)} \\
\text{;; succ: EXIT} \\
<\text{bb 0}>: \\
p &= \&a; \\
D.1284 &= c \ast d; \\
D.1285 &= D.1284 + b; \\
*p &= D.1285; \\
\text{return};
\end{align*}
\]

Listing 4.4: Intermediate Representation of \(b + c \ast d\)

described in section 4.2.2. The listing is only a textual representation of the tree structure used in the compiler, a more detailed print out of a tree showing the expression \(a = b + c\) can be found in Listing 4.2.

The main benefits of using IR instead of the original source code is that it’s more structured; every statement has at most 3 operands, and it’s already parsed and available as tree structure. The drawback is that every compiler has its own IR making your analyser dependent of that specific compiler.

4.3.3 Object Code

\[
\begin{align*}
1c: 8d 45 \text{ ec} &\quad \text{lea} 0 \text{fffffff}e(\%ebp),\%eax \\
1f: 89 45 \text{ fc} &\quad \text{mov} \%eax,0 \text{fffffff}f(\%ebp) \\
22: 8b 45 \text{ f4} &\quad \text{mov} 0 \text{fffffff}4(\%ebp),\%eax \\
25: 0f af \text{ f8} &\quad \text{imul} 0 \text{fffffff}8(\%ebp),\%eax \\
29: 8b \text{ c2} &\quad \text{mov} \%eax,\%edx \\
2b: 03 55 \text{ f0} &\quad \text{add} 0 \text{fffffff}0(\%ebp),\%edx \\
2e: 8b 45 \text{ fc} &\quad \text{mov} 0 \text{fffffff}c(\%ebp),\%eax \\
31: 89 \text{ 10} &\quad \text{mov} \%edx,(\%eax) \\
33: \text{ c9} &\quad \text{leave} \\
34: \text{ c3} &\quad \text{ret}
\end{align*}
\]

Listing 4.5: Object Code version of \(b + c \ast d\)

In listing 4.5 the i386 object code created from the C code in listing 4.3 and its disassembly are shown. This object code is easily parsed by a machine, its structure is very strict and memory accesses can be detected. The drawbacks are that the structure from the original source code is completely lost and it’s hard to match the information to the different variables, making object code a bad choice for exposing Memory Access Patterns.

4.3.4 Conclusion

By analysing the Intermediate Representation, you get structured code that still have enough ties back to the original source code to give the developer usable information. In the case of GCC and GEM, the IR is already parsed and available for use, so more time can be spent on analysing the code instead of parsing it.
Part II

Model and Method
Chapter 5

Memory access Pattern Concepts

This chapter will define and describe the concept of a Memory access Pattern (MaP), a formal representation of parallel memory accesses, invented as a part of the thesis research.

5.1 Introduction

Already in the early work of the thesis it was found that to make further analysis in the area of parallel memory access some kind of formal model was needed. The concept, named Memory access Pattern (MaP), was invented and later defined based upon the PMA model in [5] (summarized in 3.2). The purpose of a MaP is to provide a formal representation of parallel memory accesses exposed from source code.

5.2 Definition and Relations

The notion of memory access patterns have been used in many different ways and in this thesis it’s presented as a central object for use in memory access software and hardware design.

5.2.1 Memory Access Pattern (MaP)

A MaP is a number of ordered accesses to a memory. Each individual access can be represented either by:

1. Absolute address
2. Relative offset to a reference point address
3. Relative offset to previous access

These choices of representation correspond to common variants of MaP code implementations. A MaP can contain read-only-, write-only- or read-write-mixed-accesses, although a mixed MaP usually can be divided into one read-MaP and one write-MaP.
5.2.2 Memory Access Exposition (MaE)

A MaE is the result of an analysis of software containing memory access code. The MaE presents all memory accesses within the code and information about memory access control-flow (e.g. loop constructs). The structure and appearance of a MaE depend on the code analysis method but the result should in some way be comparable to a MaP. In this thesis a tool for code analysis and visualization of resulting MaE is proposed (see Chapter 6).

Definition 5.1 (Patternable). A MaE is called patternable if the memory accesses it presents can be identified as a specific MaP.

The purpose of the MaE is to be an in-between between software and MaP representation. This way any software can result in a MaE but any MaE does not have to be convertible to a MaP.

Definition 5.2 (Exposable). A source code is called exposable if it’s resulting MaE is patternable.

5.2.3 Memory Access Code Template (MaCT)

A MaCT is a code implementation of a specific MaP. To implement a MaP it has to be divided into one or more hardware supported access formats. This makes a MaCT somewhat hardware dependent since it imply a specific memory system.

Theorem 5.1. The MaE resulting from analysis of a MaCT will present memory accesses that can be identified as a specific MaP.

Proof. This is rather an extension of the MaE definition than a true theorem, stating how a MaE is supposed to work, related to the MaCT definition.

Theorem 5.2. All MaCT are exposable.

Proof. Follows from theorem 5.1 and definitions.

And from that follows:

Theorem 5.3. Any code not exposable can not be a MaCT.

5.3 MaP Application

A MaP is to be regarded as a memory access representation that is both hardware and software design independent. This makes the concept quite useful in further design specification. Here are the main intended fields of application:

PMA software implementation - In a case when a given (hardwired) PMA exists, the MaP can be used to find a optimal software memory access implementation. In this case one can also list a set of coding templates (MaCTs) supported by the PMA, each corresponding to a specific MaP.

PMA configuration - When using a configurable PMA, the MaP can be used to find an optimal configuration. After the configuration is set the MaP can also be used to optimize software implementation.
PMA design - One MaP or several, resulting for analysis of specific applications, may be used to design a PMA for the application specific system.

Applied with the PMA model presented in [5] (summarized in 3.2) this imply the ability either to divide or form any MaP into a set of access formats that are, can or will be supported by the intended PMA. This thesis focus on a MaP as a intermediate between software and access format and the MaP representation is based upon the access format representation.

5.3.1 Patternable MaE and Exposable Code

In this thesis, the MaP concept is foremost intended to tell what memory access code can be exposed and in turn be represented by a MaP. The MaP concept is to be viewed as requirements on the properties of exposed memory accesses needed in further analysis.

5.3.2 MaCT Application

The purpose of the MaCT is to give a direct connection between a specific MaP and a its code implementation. To use a MaCT in source code implementation is to implement with a specific MaP in mind. An intended use of MaCTs is to exchange generic (non-exposable) memory access code with (exposable) MaCTs and thereby make further analysis simpler.

5.4 Mathematical Description and Corresponding Templates

To describe an abstract object such as a MaP a mathematical model is well in place. In this section different matematical MaP representations are presented. It also include examples on corresponding MaCT implementations, all intended for a system using four integer memory accesses in parallel.

5.4.1 MaP as an Array

In access formats, \( F \) or \( F(r) \), are represented by elements set in arrays, this model can also apply to MaPs. In MaCT coding terms this means that an array is first constructed then used in memory access.

Absolute array MaP Generally a MaP can be described by the set \( P \) of memory addresses \( a_i \), pointing out the samples to be accessed.

\[
P = (a_0, a_1, \ldots, a_{n-1})
\]

```c
void read_4_absolute (const (int *) A0, A1, A2, A3, (register int *) RF)
{
    int * P[4] = {A0, A1, A2, A3};
    RF[0] = *(P[0]);
    RF[1] = *(P[1]);
    RF[2] = *(P[2]);
```
Relative array MaP

An alternativ form $P(r)$, closly related to the access format conterpart $F(r)$, structured as relative offsets to a reference point address, e.g. $a_1$.

$$P(r) = (e_0, e_1, \cdots, e_{n-1})$$

$$e_i = a_i - r$$

$$(r = a_0 \Rightarrow e_0 = 0)$$

```c
void read_4_relative(const int *r, const int E0, E1, E2, E3, (register int) *RF)
{
    int P_r[4] = {E0, E1, E2, E3};
    RF[0] = *(r + P_r[0]);
    RF[1] = *(r + P_r[1]);
    RF[2] = *(r + P_r[2]);
    RF[3] = *(r + P_r[3]);
}
```

Listing 5.2: C MaCT: Relative array MaP, 4 reads

Differential array MaP

A second alternative $\dot{P}(r)$, structured as relative offset to previous access,

$$\dot{P}(r) = (\dot{e}_0, \dot{e}_1, \cdots, \dot{e}_{n-1})$$

$$\dot{e}_i = a_i - a_{i-1}$$

either with $\dot{e}_0 = 0$ relative to $r = a_0$ or in some cases one prefer $\dot{e}_0 = \dot{e}_1$ relative to $r = a_0 - \dot{e}_1$.

```c
void read_4_diff(const int *r, const int dE0, dE1, dE2, dE3, (register int) *RF)
{
    int * r_temp = r;
    int dP_r[4] = {dE0, dE1, dE2, dE3};
    RF[0] = *(r_temp += dP_r[0]);
    RF[1] = *(r_temp += dP_r[1]);
    RF[2] = *(r_temp += dP_r[2]);
    RF[3] = *(r_temp += dP_r[3]);
}
```

Listing 5.3: C MaCT: Differential array MaP, 4 reads

Example 5.1 ($P(r)$).

Arbitrary stride access of size $n$: In memory access with arbitrary stride the offset increase by a constant $s$ for each access (first access address $r$):

$$P_s = (r, r + s, r + 2 \cdot s, \cdots, r + (n - 1) \cdot s)$$
In this chapter the \( P(r) \) MaP representation will be the one commonly used.

\[
P_s(r) = (0,s,2\cdot s,\ldots,(n-1)\cdot s)
\]
\[
\dot{P}_s(r-s) = (s,s,\ldots,s)
\]

5.4.2 MaP as a Function

Although any MaP can be represented as an array there are many interesting cases where functional representation, with \( a_i \), \( e_i \) or \( \dot{e}_i \) replaced by a index function \( f(i) \), is possible and most useful, especially for coding purposes.

**Absolute functional MaP** A MaP in the form

\[
P = f(i) \quad i \in \{0,\ldots,n-1\}
\]

is only relevant if \( f(i) \) returns values from a table \( f(i) = a_i \).

**Relative functional MaP** In many cases the most simple way to represent a MaP. The relative functional MaP can be implemented by use of iteration and is therefore common in expositions of high level code.

\[
P(r) = r + f(i) \quad i \in \{0,\ldots,n-1\}
\]

**Listing 5.4: C MaCT Example: Arbitrary stride with array MaP, 4 reads**

```c
void arbitrary_stride_read_4(const int * r, const int s, (register int) * RF) {
    // int * P[4] = {r, r+s, r+2*s, r+3*s}
    // int P\_[r][4] = {0, s, 2\_* s, 3\_* s};
    // int dP_r[4] = {s, s, s, s}
    int dP_r = s; // the constant diff. MaP
    int * r_temp = r - dP_r;
    RF[0] = *(r_temp += dP_r);
    RF[1] = *(r_temp += dP_r);
    RF[2] = *(r_temp += dP_r);
    RF[3] = *(r_temp += dP_r);
}
```

**Listing 5.5: C MaCT: functional relative MaP, 4 reads**

```c
void read_4_relative(const int * r, (const int) * F, (register int) * RF) {
    int i;
    for (i = 0; i < 4; i++)
        RF[i] = *(r + F[i]);
}
```
Differential functional MaP Because of it’s recursive nature $\dot{P}(r)$ doesn’t have a useful functional representation:

$$\dot{P}(r) = r + \sum_{j=0}^{i} \dot{e}_j \quad i \in \{0, \ldots, n - 1\}$$

Although when viewing it as the differential of the relative representation,

$$df(i) = \dot{e}_i di$$

$$f(i) = f(i - 1) + \dot{e}_i$$

the relation to commonly used incremental addressing is more clear and propose an alternative representation:

$$\dot{P}(r) = f(i) \quad i \in \{0, \ldots, n - 1\}$$

$$f(i) = \begin{cases} 
    r + \dot{e}_0 & i = 0 \\
    f(i - 1) + \dot{e}_i & i \in \{1, \ldots, n - 1\}
\end{cases}$$

This representation is especially interesting when $\dot{e}_i$ is constant $\forall i$.

```c
void read_4_diff(const int * r, const int * dF, (register int) * RF)
{
    int * r_temp = r;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp += dF[i]);
    }
}
```

Listing 5.6: C MaCT: functional differential MaP, 4 reads

**Example 5.2** (Relative functional MaP). Arbitrary stride access of size $n$, represented with a relative functional MaP:

$$P_s(r) = r + f(i) = r + i \cdot s \quad i \in \{0, \ldots, n - 1\}$$

```c
void read_4_relative(const int * r, const int s, (register int) * RF)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r + i*s);
    }
}
```

Listing 5.7: C MaCT Example: Arbitrary stride with relative functional MaP, 4 reads
Example 5.3 (Differential functional MaP). Arbitrary stride access of size \( n \), represented with a differential functional MaP:

\[
\dot{P}_s(r) = f(i) \quad i \in \{0, \ldots, n-1\}
\]

\[
f(i) = \begin{cases} 
  r & i = 0 \\
  f(i-1) + s & i \in \{1, \ldots, n-1\}
\end{cases}
\]

```c
void read_4_diff(const (int *) r, const int s, (register int)∗ RF)
{
    int ∗ r_temp = r - s;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp += s);
    }
}
```

Listing 5.8: C MaCT Example: Arbitrary stride with differential functional MaP, 4 reads

5.5 Multidimensional MaP

A common feature of MaEs examined are that they relate to sets of MaPs rather than a single one. The simple solution would be to split these MaEs into single MaP parts, but for many cases this is not preferred as some parts are clearly related (e.g. identified as different iterations of the same code and/or using a common scanning field). To make it possible to relate this kind of data to a single MaP the definition is extended with dimensions.

If the MaP representations presented in 5.4 are considered to be one-dimensional (1D):

**Definition 5.3** (1D MaP, \( P^1 \)). The 1D MaP is a ordered set of memory accesses.

A two-dimensional (2D) definition is possible:

**Definition 5.4** (2D MaP, \( P^2 \)). The 2D MaP is a ordered set of 1D MaPs with the same size.

Which iteratively give the multidimensional definition:

**Definition 5.5** (Multidimensional MaP, \( P^D \)). A MaP of dimension \( D > 1 \) is a ordered set of MaPs of dimension \( D - 1 \) with the same size.

The definitions include requirements on MaP size which also need to be defined:

**Definition 5.6** (MaP size). The size of a 1D MaP is equivalent to the number of elements (memory accesses) it include:

\[
P^1 = (a_0, a_1, \ldots, a_{n-1}) \Rightarrow \text{size}(P^1) = n
\]
The size of a multidimensional MaP is written as size of the MaPs in the set:

\[ P^D = \{ P^{D-1}_0, P^{D-1}_1, \ldots, P^{D-1}_{n-1} \}, \]

\[ \text{size} (P^{D-1}_0) = \text{size} (P^{D-1}_1) = \ldots = \text{size} (P^{D-1}_{n-1}) = \text{size} (P^{D-1}) \]

\[ \Rightarrow \text{size} (P^D) = \text{size} (P^{D-1}) \times n \]

**Example 5.4** (General \( P^2 \)). \( m \) 1D MaPs,

\[ P^i_1, \quad i \in \{0, \ldots, m-1\} \]
of common size \( \text{size} (P^i_1) = n \), could be set into a 2D MaP,

\[ P^2 = \{ P^i_0, \ldots, P^{m-1}_1 \} \]
of size \( \text{size} (P^2) = n \times m. \)

**Example 5.5** (Absolute \( P^3 \), specific size). \( 9 \) 1D MaPs,

\[ P^{i,j}_1 = (a_{ij0}, a_{ij1}, a_{ij2}, a_{ij3}) \quad i \in \{0, 1, 2\}, \quad j \in \{0, 1, 2\} \]
of common size \( \text{size} (P^{i,j}_1) = 4 \), could be set into 3 2D MaPs,

\[ P^2_j = \{ P^{i,j}_0, P^{i,j}_1, P^{i,j}_2 \} \]
of common size \( \text{size} (P^2_j) = 4 \times 3 \), that in turn could be set into a 3D MaP,

\[ P^3 = \{ P^2_0, P^2_1, P^2_2 \} \]
of size \( \text{size} (P^3) = 4 \times 3 \times 3. \)

### 5.5.1 Multidimensional MaP as an Matrix

The multidimensional equivalent of an array is a matrix and although the matrix isn’t truly a set of arrays (cp. Definition 5.4) it can represent one (and in common logical representations, like C-code, a matrix is a set of arrays). In Listing 5.9 a general 2D relative MaP is implemented using the C, semi-matrix, array of arrays type.

```c
void read_4x4_relative(const int * r , ( const int ) * * P2_r , ( register int ) * RF)
{
    int i ;
    for( i = 0 ; i < 4 ; i++ )
    {
        RF[4*i] = *( r + P2_r[0][i] ) ;
        RF[1 + 4*i] = *( r + P2_r[1][i] ) ;
        RF[2 + 4*i] = *( r + P2_r[2][i] ) ;
        RF[3 + 4*i] = *( r + P2_r[3][i] ) ;
    }
}
```

Listing 5.9: C MaCT: Relative 2D matrix MaP, 4x4 reads
Example 5.6 (2D arbitrary stride MaP as matrix). A common set (2D MaP) of MaPs,

\[ P_s^2 = \{ P_s^{1_0}, P_s^{1_1}, \ldots, P_s^{1_{m-1}} \} \]

\[ P_s^{1_i} = P_s(r + i \cdot t), \quad i = 0, 1, \ldots, m - 1 \]

\[ P_s^1(r) = (0, s, 2 \cdot s, \ldots, (n - 1) \cdot s) \]

\[ P_s^{1_i}(r) = (i \cdot t, s + i \cdot t, 2 \cdot s + i \cdot t, \ldots, (n - 1) \cdot s + i \cdot t), \quad i = 0, 1, \ldots, m - 1 \]

where \( s, t \) are arbitrary constants, can be represented by the matrix

\[ P_s^2 = \{ P_s^{1_0}^T, P_s^{1_1}^T, \ldots, P_s^{1_{m-1}}^T \} = \]

\[
\begin{pmatrix}
0 & t & \cdots & (m - 1) \cdot t \\
s & s + t & \cdots & s + (m - 1) \cdot t \\
\vdots & \vdots & \ddots & \vdots \\
(n - 1) \cdot s & (n - 1) \cdot s + t & \cdots & (n - 1) \cdot s + (m - 1) \cdot t
\end{pmatrix}
\]

\[ \text{size}(P_s^2) = n \times m \]

This is called a (2D) \( n \times m \) arbitrary stride matrix MaP.

5.5.2 Multidimensional MaP as a Function

Most uses of 2D MaPs are when accessing 2D structured data and in these cases one usually like to use a double index function, \( f(i, j) \), for address (offset) calculation, which result in a relative MaP:

\[ P^2(r) = r + f(i, j) \]

```c
void write_4x4_relative((register int) * RF, const (int *) r, ((const int *) *) F)
{
    int i, j;
    for (j = 0; j < 4; j++)
    {
        for (i = 0; i < 4; i++)
        {
            *(r + F[i][j]) = RF[i + 4*j];
        }
    }
}
```

Listing 5.10: C MaCT: Multidimensional functional relative MaP, 4x4 writes

Double index is one common version of 2D functional relative MaP another is single index, double parameter \((i + N \cdot j)\), also used in Listing 5.10. The differential 2D functional MaP may look a little different:

\[ \dot{P}^2(r) = f(i, j) \quad i \in \{0, \ldots, n-1\}, \quad j \in \{0, \ldots, m-1\} \]

\[ f(i, j) = \begin{cases} 
    r + \dot{e}_{00} & i, j = 0 \\
    f(0, j - 1) + \dot{e}_{0j} & i = 0, \quad j \in \{1, \ldots, m-1\} \\
    f(i - 1, j) + \dot{e}_{ij} & i \in \{1, \ldots, n-1\}, \quad \forall j
\end{cases} \]
Example 5.7 (2D stride MaP $P_s^2(r, s_0, s_1)$). 2D arbitrary stride differential MaP of size $n \times m$:

$$P_s^2(r, s_0, s_1) = f(i, j) \quad i \in \{0, \ldots, n - 1\}, \quad j \in \{0, \ldots, m - 1\}$$

$$f(i, j) = \begin{cases} r & i, j = 0 \\ f(0, j - 1) + s_1 & i = 0, \quad j \in \{1, \ldots, m - 1\} \\ f(i - 1, j) + s_0 & i \in \{1, \ldots, n - 1\}, \quad \forall j \end{cases}$$

```c
void write_4x4_relative((register int) * RF, const (int *) r,
((const int) *) dE)
{
    int i, j;
    int * r_temp = r;
    int * r_temp_temp = r_temp;
    for (j = 0; j < 4; j++)
    {
        // i = 0;
        *(r_temp += dE[0][j]) = RF[4*j];
        r_temp_temp = r_temp;
        for (i = 1; i < 4; i++)
        {
            *(r_temp_temp += dE[i][j]) = RF[i + 4*j];
        }
    }
}
```

Listing 5.11: C MaCT: Multidimensional functional differential MaP, 4x4 writes

```c
void write_4x4_diff((register int) * RF, const int r, const int s_0, s_1)
{
    int i, j;
    int * r_temp = r - s_1;
    int * r_temp_temp = r_temp;
    for (j = 0; j < 4; j++)
    {
        // i = 0;
        *(r_temp += s_1) = RF[4*j];
        r_temp_temp = r_temp;
        for (i = 1; i < 4; i++)
        {
            *(r_temp_temp += s_0) = RF[i + 4*j];
        }
    }
}
```

Listing 5.12: C MaCT: 2D arbitrary stride with differential functional MaP, 4x4 writes
5.6 MaP Categories

The wide scope of the MaP concept makes it possible to represent the same memory access behavior in several different ways. This inconsistency isn’t entirely undesirable since it also broadens the sum of exposeable code (see Definition 5.2), although it makes it impossible to define a specific MaP to represent a specific memory access behavior. Instead MaPs may be categorized according to behavior.

5.6.1 Constant Stride Memory Access

This group of MaPs, categorized as arbitrary stride in an arbitrary number of dimensions, cover the greater part of all parallel data memory accesses.

The typical 1D constant stride MaP use differential functional representation (see 5.4.2 Example 5.2 with Listing 5.7). Adding a dimension is then the same as reuse of the same MaP elements but with a stride in the reference address (see Example 5.6). In the same way any number of dimensions may be added. Using unlimited number of dimensions any memory access behavior can be categorized as stride access but here the focus will be on stride MaPs of the first and second dimension.

1D stride MaP \( P_s^1(r, s) \)

\[
P_s^1(r, s) = f(i) \quad i \in \{0, \ldots, n - 1\}
\]

\[
f(i) = \begin{cases} 
  r & i = 0 \\
  f(i - 1) + s & i \in \{1, \ldots, n - 1\}
\end{cases}
\]

**Burst access**, \( s = 1 \) a.k.a. straight, vector or row (2D data) access. Simply to access adjacent data samples, definitely the most used and supported access format.

\[ f(i) = f(i - 1) + 1 \]

```c
void read_4_burst (const (int *) r, (register int) * RF)
{
    int * r_temp = r;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp++);
    }
}
```

Listing 5.13: C MaCT Example: Burst access, 4 reads

**Radix access**, \( s = n^l \), radix-\( n \) access of level \( l \), \( n \) usually a power of 2, usually in a MaP of size \( n \).

\[ f(i) = f(i - 1) + n^l, \quad n \geq 2, \quad l \geq 0 \]
void read_4_radix4_2(const int * r, (register int) * RF)
{
    // s = 4^2 = 16
    int * r_temp = r - 16;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp += 16);
    }
}

Listing 5.14: C MaCT Example: Radix-4 access level 2, 4 reads

Column access, \( s = N \), of 2D data, structured in rows of \( N \) samples.
\[
f(i) = f(i - 1) + N, \quad N \geq 2
\]

void read_4_col_in12(const int * r, (register int) * RF)
{
    // s = N = 12
    int * r_temp = r - 12;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp += 12);
    }
}

Listing 5.15: C MaCT Example: Column access 12 sample row, 4 reads

Diagonal access, \( s = \pm(N \pm 1) \) (4 cases), of 2D data, structured in rows of \( N \) samples.
\[
f(i) = f(i - 1) \pm (N \pm 1), \quad N \geq 2
\]

void read_4_diag_in12(const int * r, (register int) * RF)
{
    // s = N+1 = 13
    int * r_temp = r - 13;
    int i;
    for (i = 0; i < 4; i++)
    {
        RF[i] = *(r_temp += 13);
    }
}

Listing 5.16: C MaCT Example: Diagonal access 12 sample row, 4 reads
5.6 MaP Categories

2D stride MaP $P_s^2(r, s_0, s_1)$

$$P_s^2(r, s_0, s_1) = f(i, j) \quad i \in \{0, \ldots, n-1\}, \quad j \in \{0, \ldots, m-1\}$$

$$f(i, j) = \begin{cases} 
    r & i, j = 0 \\
    f(0, j - 1) + s_1 & i = 0, \quad j \in \{1, \ldots, m-1\} \\
    f(i - 1, j) + s_0 & i \in \{1, \ldots, n-1\}, \quad \forall j 
\end{cases}$$

**Block burst**, $s_0 = 1$, more commonly known as square/rectangle access with data structured in rows of $N$ samples and $s_1 = N$.

$$f(i, j) = \begin{cases} 
    r & \\
    f(0, j - 1) + N & \\
    f(i - 1, j) + 1 & 
\end{cases}$$

```c
void write_4x4_sq_in12((register int) * RF, const (int *) r)
{
    //N = 12
    int i, j;
    int * r_temp = r - 12;
    int * r_temp_temp = r_temp;
    for (j = 0; j < 4; j++)
    {
        //i = 0;
        *(r_temp += 12) = RF[4*j];
        r_temp_temp = r_temp;
        for (i = 1; i < 4; i++)
        {
            *(r_temp_temp += 1) = RF[i + 4*j];
        }
    }
}
```

Listing 5.17: C MaCT Example: Block burst (square access) 12 sample row, 4x4 writes

**Crumbled square**, $s_0 = s, s_1 = N \cdot s$, standard 2D access of 2D data, structured in rows of $N$ samples, of size $n \times n$ (cp. crumbled rectangle if $s_0 \neq s$ or non-square size). Commonly supported access format.

$$f(i, j) = \begin{cases} 
    r & \\
    f(0, j - 1) + N \cdot s & \\
    f(i - 1, j) + s & 
\end{cases}$$

```c
void write_4x4_crsq_2in12((register int) * RF, const (int *) r)
{
    //s = 2, s*N = 24
    int i, j;
    int * r_temp = r - 24;
    int * r_temp_temp = r_temp;
```
for (j = 0; j < 4; j++)
{
    // i = 0;
    *(r_temp += 24) = RF[4*j];
    r_temp_temp = r_temp;
    for (i = 1; i < 4; i++)
    {
        *(r_temp_temp += 2) = RF[i + 4*j];
    }
}

Listing 5.18: C MaCT Example: Crumbled square stride 2 access 12 sample row, 4x4 writes
Chapter 6

Memorizer

This chapter will go into detail about the application Memorizer, which was built to expose memory access and addressing calculations in C code.

6.1 Introduction

Memorizer is a tool built as an extension to the popular GCC\textsuperscript{1} compiler using the GEM 1.7 plugin framework. GEM is a patch for GCC that includes a set of function pointers, which are implemented in the extension module. During compilation of a file, the GEM enabled GCC checks whether a specific function pointer is implemented and if so calls the function.

The idea behind Memorizer stems from the tool Relief developed by Björn Skoglund as his master thesis work\textsuperscript{8}. Although Memorizer is not a derivative of Relief, its design is heavily influenced by it.

6.1.1 Goal

The goal of Memorizer is to expose memory accesses, that can be matched against a memory access pattern and facilitate automatic creation of DMA linking tables and permutation tables for parallel memory access.

Memorizer neither find the most frequently used memory accesses, nor the most costly, but can be used for showing how memory access in a specific function is done. A tool like Relief can be used to show which functions that should be targeted by Memorizer.

The usefulness of Memorizer may be debated, but it should be seen as a proof of concept, since it shows that addressing code and memory accesses can be exposed even if they are intertwined with other code.

6.2 Design of Memorizer

This section is about the design of Memorizer, underlying technologies and the work it is based on.

\textsuperscript{1}GNU Compiler Collection
6.2.1 GEM

GEM is a framework for writing compiler extensions as dynamically loaded modules [9]. GEM is implemented as a patch for GCC, that defines a number of hooks in the GCC source code. A hook is a function pointer that is initially set to NULL. When a module is loaded that want to use the specific hook, it sets the function pointer to one of the functions included in the module. A GEM Module is written as a stand-alone program, which is compiled into a dynamically linkable object file. This file can be loaded with a command-line argument to GCC. When GCC loads the module and calls its initialization function, the module registers the hooks it’s interested in and will then get callbacks during the compilation.

6.2.2 Relief

Relief is a profiling tool built by Björn Skoglund [8] as his Master Thesis work at ISY / Computer Engineering. It’s a combinational static-dynamic profiler. During compile time it instruments the code with functions for time-taking and collects information about each basic block, such as its place in the control flow graph and what kinds of and how many operations that are executed in the block. During execution of the profiled program, statistics are generated to record what blocks that are executed most frequently. The output from the static and dynamic analysis is then combined and explored using a special post analysis program included in the Relief suite.

6.2.3 From Relief to Memorizer

Using Relief as a template, Memorizer is built using the same GEM callback functions to get notified when a complete function is available as GIMPLE code. Memorizer shares no code with Relief, but the design of Memorizer is heavily influenced by it.

The goals of Relief and Memorizer are also different. Relief is meant to point out in what parts of a program most of the execution time is spent, whereas Memorizer exposes the memory accesses and addressing calculations.

The best use of Memorizer is together with a tool like Relief, that can show the developer what parts of the program is interesting to optimize, or where there are many memory accesses. Once this information is found, Memorizer can be configured to only examine the functions pointed out by Relief.

6.2.4 Requirement Specification

The tool will also be used to match vector addressing models for data pre-allocation of the main memory and permutation in scratch pad memories of ASIP.

The research conducts source code analysis by static profiling and cost annotation by dynamic profiling. All research activities are based on GNU.

This part of the Project Description (see 1.1) describes a tool based on available GNU tools, i.e., GCC. Memorizer is meant to be the first part of that tool, extracting the vector addressing code from other program logic. The
int foo ()
{
    char ∗MEM;
    register int i;
    for (i = 0; i < 8; i++)
    {
        *(MEM++) = 1;
    }
}

Listing 6.1: Example of Memorizer input

extracted code can then be matched manually or in the future by an automatic tool.

6.2.5 Input

In this section, the code in Listing 6.1 will be used as an example.

Since Memorizer plugs into GCC and access the code at an intermediate representation level, the raw C code is not the real input to Memorizer but instead Memorizer takes the GIMPLE representation of the code. In figure 6.1 a graphical representation of the GIMPLE tree can be found. This is just an outline of the tree structure to facilitate easy visualization, for a more in depth view of the GIMPLE tree, a textual printout is given in listing 4.2.

The dashed lines in the figures illustrates the separation in basic blocks. Lines between the boxes describe relationship between the nodes; nodes to the right in the diagram are operands of nodes to the left.

6.2.6 Function

The process of exposing memory accesses in Memorizer can be broken down into a few steps.

Collecting Information  Parsing through the code and built up a tree structure of the interesting information.

Finding Memory Accesses  Identifying the nodes in the tree that are memory accesses.

Finding Addressing Calculations  Recursively iterate through the tree to find all calculations leading up to the address of a memory access.

Finding Loops  Detecting loops in the code and the conditions governing the execution of them.

This section describes the steps in detail.

Collecting Information

For each source file GCC encounters during the compilation of a project, the first call to Memorizer is done via the function gem_init. This function reads in the configuration file and set up the callback hooks.
Figure 6.1: Visualization of the GIMPLE tree
When GCC has parsed a complete function, and transformed the source into GIMPLE, a call is made to the GEM function gem_build_tree_cfg. Each function is divided into one or more basic blocks, which are iterated. For each block the statements in the block are identified and Memorizer builds a tree structure of its own with dependency information between expressions and operators.

The tree structure built by Memorizer is a graph describing the data dependencies and is called the **DDG Nodes** (**DDG** is short for Data Dependency Graph). Every node in the DDG Nodes have a corresponding node in the GIMPLE Tree and the tree codes of the GIMPLE nodes are saved in the DDG Node to identify the type of node. A graphical representation can be found in figure 6.2.

The big difference between the GIMPLE tree and the DDG Nodes is that the modify_expr nodes are removed and replaced with dependencies between two nodes. Compare the nodes numbered 1-4 in figure 6.1 and figure 6.2.

Another difference is that the explicit control flow of the program is not reflected in the DDG Nodes, the goto_expr and label_expr nodes are discarded. This has to do with one of the limitations in Memorizer: control flow is not analyzed beyond the extraction of loop conditions. Information about the control flow is still recorded in Memorizer on basic block level though.

Once GCC is done with one source file the function gem_destroy is called and the post processing starts. This is the real work of Memorizer and it is done in a few steps, outlined below.

**Finding Memory Accesses** Once again each basic block of each function is traversed, this time using the DDG Nodes. Nodes that are of the types ARRAY_REF and INDIRECT_REF are marked as being memory accesses and are added to a list of memory access nodes. In the example used, the filled node in figure 6.2 is marked to be a memory access. In Memorizer the assumption is made that all memory accesses are either done by indirect reference from a pointer or by accessing an element of an array (INDIRECT_REF and ARRAY_REF respectively). As long as this holds all memory accesses apart from those done...
in blocks of inline assembler will be found and exposed.

This marking is done recursively and all explicit dependencies are found in
the first iteration of the blocks, but the detection needs to be smarter than this.
The inner loop in Listing 6.1 will be translated into the following Three-address
code

\[
\begin{align*}
*MEM &= 1 \\
MEM &= MEM + 1 \\
i &= i + 1
\end{align*}
\]

Since the increment of the \(MEM\) pointer is done after the write, just mark-
ing explicit dependencies would miss that the statement \(MEM = MEM + 1\) is
actually an addressing calculation.

**Finding Addressing Calculations** Once all memory accesses and their ex-
PLICIT dependencies are found more iterations has to be done to find implicit
dependencies. During the first iteration, a list with the names of all variables
of interest for the addressing was created, let’s call this list \(L\).

A new iteration of the DDG Nodes are done. This time the variables pointed
out by \(L\) and their explicit dependencies are marked as being of interest. If any
new variables of interest are found they will be added to \(L\), and the iteration
will repeat until \(L\) remains constant during one iteration.

To exemplify, imagine that the inner loop of a program looks like

\[
\begin{align*}
MEM &= MEM + a \\
*MEM &= 1 \\
a &= a + b \\
b &= 2\ast b
\end{align*}
\]

From the very first iteration done in order to find memory accesses the list
of interesting nodes would be

\[L = \{MEM, a\}\]

In a second iteration the expression \(a = a + b\) is found and \(b\) is added to the list.

\[L = \{MEM, a, b\}\]

A third iteration is done but this time no new nodes will be found and \(L\) remains
constant which indicates that all nodes of interest have been found.

**Finding Loops** Knowing the memory accesses and addressing calculations
in each iteration of the loop is of course interesting, but to get anything really
useful out of the exposition, information about how many times something is
done is vital. It’s nice to know that memory is accessed one sample at a time
from adjacent addresses, but until we know if it is 16 or 1024 samples that are read in total the information is not really usable.

Loops are found by examining the Control Flow Graph of Basic Blocks, using the theory explained in section 4.2.4. In figure 6.3 the control flow graph of the example code is shown. Each box in the graph represents one basic block, the numbers next to the boxes are the node numbers, and the node’s dominators. In the example, the edge leading from block 1 to block 2 will be found to form a loop. The last statement in each node that is the head of a loop is a COND_EXPR and from this expression, the condition governing the loop is extracted.

Memorizer searches through all blocks that earlier have been found to contain memory accesses or addressing calculations and uses the same repeating iterations method as used for the addressing calculations to find all variables and expressions that are of interest for the loop control calculations.

### 6.2.7 Output

The output from memorizer is our representation of an MaE. Separate outputs are given for each function analysed, see section 6.4.3 for details. The C code in listing 6.1 is used as input to give examples of the four different output alternatives.

**Full Dependency Graph** A tree representation of all operations in the function, memory addressing operations are highlighted, see Figure 6.4. The graph is divided into multiple subgraphs, one showing the control flow of the basic blocks in the function and one subgraphs per basic block showing...
the operations in that block. Nodes that are identified as being a memory access are filled with a red color (Dark grey) and nodes that are found as being part of the addressing calculations are yellow (Lighter grey). A double bordered node indicates that it is a return statement. In Appendix E two more detailed examples can be found.

**Addressing Dependency Graph** A tree representation of all expressions relevant for memory access and addressing calculations, see Figure 6.5. For each basic block up to three subgraphs are available, *Addressing Calculations, Memory Writes* and *Memory Reads*. The same colors as in the Full Dependency Graph are used.

**Memory Access Tables** Textual representation of the Memory Accesses, Addressing Calculations and Loop Control, see listing 6.2. For each basic block, the name and the loop depth is given. If the block is part of a loop, the conditions governing the loop is given as well.

The main part of the information here is given in a table form, with four columns.

- **Index** This is just a running number given to the nodes when memorizer searches through the basic block, to keep the order of nodes intact.
- **Size** Sample size in bytes of the data being read or written, for character data, the size is typically 1. For integer data, assuming 32 bit integers are used, it’s 4.
- **Type** This column tells what kind of expression the row represents, it can be one of either WRITE, READ, ADDR or LOOP for memory writes, memory reads, addressing calculations or loop control respectively.
- **Expression** The expression in question, given in C-like syntax.

**Memory Access XML** XML Representation of the exact same information as the Memory Access Tables. This representation could be used by other applications for automatic matching of Memory Access Patterns or other analysis. See listing A.1 in Appendix A for an example of this XML file.

6.2.8 Limitations of Memorizer

Memorizer can analyse any C code, though some restriction applies for getting useful results.
Control Flow  The more control logic there is intermixed with the addressing calculations, the harder it will be to get any useful result. Memorizer is best at exposing addressing in programs that consists of large basic blocks.

Well behaved code  Pointers to pointers to pointers... you get the point; having to intricate structures of pointers will make it harder to understand the exposed information.

Functions Memorizer only works on function level and can’t find relations in between functions. Therefore, if it is possible, addressing code should be kept in the same function as its surrounding loop.

Inline Assembly  Code which includes inline assembly will not be analysed, since no GIMPLE tree is created for blocks of assembler code.

Pointers and Arrays Memorizer assumes that all memory accesses are done either by referencing a pointer or by accessing an element of an array, if a memory access is done in some other obscure way it will not be found.

6.3 Implementation

Björn Skoglund [8] successfully showed that using GCC and GEM to write profiling and analysis tools is a good choice when he developed Relief. The choice of platform for developing Memorizer was therefore given at the start of the project.
This section will describe how Memorizer is implemented and what patches apart from GEM was needed to be applied to GCC in order to get it to work.

### 6.3.1 Patching GCC

```c
typedef int type;
struct s {
    type type;
};
```

**Listing 6.3:** Code valid in C but not in C++

In order to be able to use STL we decided to use C++ for implementing Memorizer. This meant that patching of GCC was needed due to a difference in `typedef` between C and C++. In C members of `struct`s can have the same name as a defined type, which is not allowed in C++. This means that the code in listing 6.3 would pass through a C-compiler, but not a C++ one. This feature is exploited in the GCC source. The problem is fixed by replacing the user defined type with the type it represents. The complete patch to do this is included in Appendix B.

### 6.3.2 Classes

Since Memorizer is written in C++, separating the code into different classes was the natural way to go. There are three main classes representing the different data entities that are used in Memorizer: `Function`, `BasicBlock` and `DDGNode`. These three classes are all implemented in the files `ddg.hh` and `ddg.cc`. The class `Memorizer` is used for handling the configuration file.

Two special classes, `rptr` and `robj` were written for easier memory management. The class `Tree` is a C++ wrapper around the GCC `tree` structure.

The different classes in the program will be briefly described below, detailed documentation is available in Doxygen format from the source code.

**Function** An instance of this class is created whenever a new function is found in the source file being analyzed. Each instance has a map of `BasicBlocks` keyed by name. The blocks are accessed by name instead of just storing them in a sequence since one block might refer to a later block not yet encountered. In order to get references right, the name is stored in the current block instead of a pointer to a block that doesn’t yet exist.

When Memorizer is done parsing a source file, it iterates over all functions and performs the analysis. Loops are detected and marked up, and the same thing is done for memory accesses and addressing calculations.

**BasicBlock** For each basic block encountered an instance of this class is created. The `BasicBlock` class contains an ordered list of the instructions in the block represented by `DDGNodes`
DDGNode  Base class for all nodes. Nodes are created using the static function DDGNode::CreateDDG() this factory function takes a GIMPLE Tree as argument and creates the right type of node from it. Each DDGNode subclass has functions for printing the node and for special handling of dependencies.

Memorizer The Memorizer class is a class with only static data members. Functions in this class are called during parsing of the memorizer.conf file. The parsing of this file is implemented using flexx++. The class is defined in the files memorizer.cc and memorizer.hh and the rules for the parser generator are written in the file conf lexer.1l.

rptr & robj rptr is a template class for memory management, inspired by the Objectiv-C Runtime system it uses retain and release functions. An rptr object is a wrapper around any other object that inherits from the robj class.

Whenever an object wants to keep a reference to a pointer, it calls the retain() function in the rptr object wrapping the pointer, and when it doesn’t need the reference any longer it calls the release(). These functions simply increase and decrease a counter variable defined in robj. When an rptr goes out of scope it checks the retain count of the wrapped object, and if it is zero it deletes it. This ensures that if multiple objects keeps a reference to the same pointer, it will not be deallocated until all of them have released it.

The −> operator is overloaded in rptr and enables transparent access to the wrapped pointer.

rptr and robj are implemented in the rptr.hh file.

6.3.3 Debugging

Debugging GCC is not as easy as for a normal application, one of the reasons is that you need to run the debugger on the cc1 executable instead of gcc, which is the executable normally used during compilation. To facilitate easy debug printouts the class Trace is implemented. It is used by putting the macro TRACE as the first line in a function, and will print out the function name to STDOUT when the function is entered and when execution leaves that function. In order to enable the trace functionality the file trace.hh has to be edited and the #if 0 statement at the top of the file should be changed to #if 1.

6.3.4 GEM Interface

In Memorizer, the following GEM hooks and functions are used.

gem_init This is the first function called for each source file being compiled. Here the function pointers are set up and the configuration file is parsed.

gem_start_function This hook is called when a new function is found in the source file. This instructs Memorizer to create a new Function object.

gem_build_tree_cfg This hook is called when the control flow graph of the basic blocks in each function is completed. This function was added by Björn Skoglund during his work on Relief. The function has a two levels deep nested loop; the first loop iterates over all the basic blocks in the
function, and the inner loop iterates over all statements in the function and create DDG Nodes based on this information.

```
gem_destroy
```
Called when GCC is done with one source file, in this function the post processing of the DDG Nodes is executed.

### 6.4 User Guide

This section goes through how to install Memorizer and how to enable analysis of source code during compilation. It also contains a section describing the configuration file that Memorizer uses. The last part of this section goes through the workflow of using memorizer as a tool in your design chain.

#### 6.4.1 Installing Memorizer

Memorizer requires gcc-core 4.1.0, gcc-g++ 4.1.0, and GEM 1.7 the source of all these need to be available to install. Here are step by step instructions how to install Memorizer.

**Prerequisites**

Memorizer is developed and tested on SUSE Linux 10.0, but should work on any modern Linux installation. In order to compile your own GCC and later on Memorizer, a working installation of GCC and Flex is needed.

**Download GEM**

GEM can be downloaded from [http://www.ecsl.cs.sunysb.edu/gem/](http://www.ecsl.cs.sunysb.edu/gem/). Memorizer is developed and tested against version 1.7 of the framework, but later versions might work if available.

Unpack the compressed tar archive where you want the tool to be located, the rest of this guide assumes you unpack it in your home directory.

```
[~]> tar xzf gem-1.7.tar.gz
```

GEM is by default configured to compile against GCC 3.4.1, but Memorizer is built for GCC 4.1.0 so this setting must be changed in the file `Makefile` in the `gem-1.7` folder just created. Change the first lines to look like this

```
# GCC_RELEASE = 3.4.1
GCC_RELEASE = 4.1.0
```

**Download GCC**

The build system for GEM includes functionality to download GCC on its own, but since Memorizer makes use of G++ as well, it’s recommended to download GCC manually. A list of places to download GCC and G++ can be found at [http://www.gnu.org/order/ftp.html](http://www.gnu.org/order/ftp.html). The interesting files will most probably be found in the folder `/pub/gnu/gcc/releases/gcc-4.1.0`.  

---

2GCC 4.0.2 and Flex 2.5.4 have been used during development
Once the folder is located, download the files gcc-core-4.1.0.tar.gz and gcc-g++-4.1.0.tar.gz to the gem-1.7 folder created earlier.

```
[~/gem-1.7] > tar xzf gcc-core-4.1.0.tar.gz
[~/gem-1.7] > tar xzf gcc-g++-4.1.0.tar.gz
```

Patching GCC

A Patch is made to enable the use of C++ in GEM modules see appendix B. This patch is available from the Subversion server at ISY. The following commands downloads the entire Memorizer repository, including the patch file.

```
[~/gem-1.7] > svn co https://svn.isy.liu.se/daxjobb/relief
[~/gem-1.7] > cd gcc-4.1.0
[~/gem-1.7/gcc-4.1.0/] > patch -p0 < ../relief/patch_for_g++.patch
```

Compiling GCC

Once downloading and patching is done it’s time to compile. Change your working directory to the gem-1.7 folder and invoke make to start the compilation.

```
[~/gem-1.7] > make
```

You should now have a functional GCC+GEM installation and it’s time to compile Memorizer itself.

Compiling Memorizer

When the GCC compilation is finished the time has come to Memorizer. The source code should already have been downloaded from the Subversion server, if GEM was installed directly in your home directory nothing has to be edited. If you placed it somewhere else, the path to your GCC installation has to be set to it’s correct value in the file relief/code/common/Makefile.conf. Make sure the correct path is listed on the first line of that file:

```
GCC_BASE=${HOME}/gem-1.7/gcc-4.1.0
```

When this is done it’s just to compile Memorizer by running make in the relief/code/memorizer directory.

```
[~/relief/code/memorizer] > make
```

6.4.2 Compiling with Memorizer

Since Memorizer is run during compile time of your software, you need to tell GCC to load the newly compiled GEM module, this is done by giving the -fextension-module option to the compiler.

It’s very probable that you already have another copy of GCC installed, and therefore you must also make sure to use the GEM enabled version like in the following example.
# This is a sample memorizer.conf file

[analyze]

# We are not interested in analyzing the main function

[noanalyze]
main

# We only want the textual memory access templates and the full graph, the three is not optimized since we are interested in seeing all temporary variables.

[options]
  #optimise_tree
  print_memory_access_tables
  #print_memory_access_xml
  no_export_full_graph
  #no_export_addressing_graph
  #print_basic_blocks
  #expand_variables_in_expressions

Listing 6.4: An example of memorizer.conf

[~]\> set CC = ~/gem-1.7/gcc-4.1.0/bin/bin/gcc
[~]\> set EXT = -fextension-module=${HOME}/relief/code/memorizer/bin/mm_gem
[~]\> ${CC} ${EXT} source.c

6.4.3 Configuring Memorizer

Memorizer is configured by the file memorizer.conf. This file must be placed in the working directory of your compiler, most often that is the same directory as the source files resides in. The configuration files is divided into three sections, the first two are used to filter what functions will be analysed, the third section modifies the output. Comments starts with the character # and can be placed anywhere in the file, the comment runs to the end of the line. A sample configuration file is listed in Listing 6.4, the following sections will describe in detail the options in this file.

Filtering Input

The sections [analyze] and [noanalyze] in the configuration file decides what will and what will not be included in the static analysis.

If any function is listed under [analyze], only those functions that are listed there will be analysed. If not functions are given under [analyze], all functions but those listed under [noanalyze] will be analysed.
Customizing Output

The [options] section contains configuration directives to control the output of memorizer. If an option is not set in the configuration file it’s default value will be used. That’s why there are options that starts with no, since their default value is to enable the feature.

**optimize_tree** If this option is set, some nodes will be removed from the tree to make the output tidier, for example will notes representing type casts and temporary variables be removed.

**print_memory_access_tables** If this option is set, tables describing memory accesses, addressing calculations and loop control expressions will be printed to a file named `<source file name>_<function name>_addr.txt`.

**print_memory_access_xml** If this option is set, and XML document describing memory accesses, addressing calculations and loop control expressions will be printed to a named `<source file name>_<function name>_addr.xml`.

**no_export_full_graph** If this option is not set, a graphviz compatible file will be created including all basic blocks and all nodes in the computational tree. Nodes of interest for the addressing calculations will be highlighted. This file will be named `<source file name>_<function name>.dot`.

**no_export_addressing_graph** If this option is not set, a graphviz compatible file will be created including all nodes of interested for the addressing calculations. The file name will be `<source file name>_<function name>.addr.dot`.

**print_basic_blocks** If this option is set, a debug print out of each basic block will be written to `STDOUT` during compilation.

**expand_variables_in_expressions** If this options is set the textual output of variables will if possible be expanded into the expression leading up to them. See figure 6.6 for an example.

\[
\begin{align*}
  a &= 5; \\
  b &= a + 3; \\
  A[b];
\end{align*}
\]

\[\Rightarrow A[5 + 3];\]

Figure 6.6: Example of expand_variables_in_expressions

6.4.4 Memorizer Workflow

By default, Memorizer analyses and produces output for all functions in the source code being compiled. This makes the compile time longer and might produce vast amounts of output in larger projects. To prevent this, the first step when using Memorizer is to identify which part of the code that is of interest to analyse. This can for example be done with a tool such as Relief [8].

Once the interesting functions have been identified and Memorizer have been configured to analyse only those functions, the source is being compiled with the GEM enabled compiler using the Memorizer extension module.
So far, the workflow is the same whether the source code is written using MaCT methods or if it is generic code.

In the case of generic code the output from memorizer is looked at by the developer, with the memorizer output showing the addressing part of the code and access to the source code, the code can be rewritten to match an MaCT, once this is done the code is compiled again. This takes us to the same step that we would have been at if the code had been template based from start. The shaded boxes in figure 6.7 represent these steps. If the code was template based from start, these steps are obviously skipped.

Once the Memorizer output from the template based code is available, matching against the template will be done. Currently this step has to be done manually but the idea is to make this matching automatic in the future. When the matching is done, the output from Memorizer, together with the knowledge about what templates that are used, DMA Linking Tables and Permutation Tables can be automatically generated. These two steps are illustrated by the boxes in figure 6.7 with dashed borders.
Part III

Case studies and Result
Chapter 7

Pipe Clean by Case Studies

An example of a pipe clean, from source code, via exposed and identified MaP, to DMA- and permutation tables.

7.1 Introduction

This chapter describes case studies on P3RMA in a proposed PVM system, by use of Memorizer in combination with the method and model described in the thesis on two cases of generic code. The first code case examines a DCT implementation and the second examines a motion compensation implementation.

The general workflow of the case studies are to examine the output from Memorizer, find what kind of memory accesses are being done and construct DMA linking table and Permutation table for the analysed code. These tables could be used for implementations of the different algorithms on the proposed PVM system.

7.2 Proposed Memory System

For the case studies in this thesis the PVM system from [7], presented in figure 7.1, is the target. The RISC DSP core works as task manager, mastering a number of vector engines using PVM (dashed line in figure). A vector engine
can be any type of ILP processing element, e.g. a SIMD DP or a slave SIMD processor. To supply data for the program running in a vector engine, two tables shall be prepered, a *DMA linking table* and a *permutation table*.

The **DMA linking table** specifies uploading and downloading of scanning fields from main memory into PVM. It consist of a chain of DMA operations, identified by start address and data block length.

The **permutation table** is the PVM equivalent of PMA assignment functions, specifying allocation of stored samples within the PVM, stated as row-wise permutations. This imply that when passed thru the PVM permutation network a scanning field is divided into rows of 8 or less (number of memory modules in the PVM = 8) 16 bit samples (16 bit = PVM sample size) and permuted according to the permutation table before stored. E.g. in the common case of a straight-line scanning field, the difference between a permutation table and $S/a$ assignment functions is that the PVM implies that $a(r) = \left\lfloor \frac{r}{8} \right\rfloor$ and therefore can only be used to implement only a part of all module assignment functions.(cp. 3.2)

### 7.3 P3RMA Analysis

The object of this thesis to expose and identify parallel memory accesses and to make use of them, the method for this is called P3RMA analysis. The case studies try to apply this with the system presented in Chapter 6. Chapter 6 describe the tool Memorizer that can be used to expose memory access by source code analysis and in chapter 5 the concept of a MaP is defined. The combination of these two create a foundation for the P3RMA analysis workflow (Figure 7.2):

1. Find opportunities for parallelization by use of code profiling (e.g. Relief, see 6.2.2).
2. Expose memory accesses in code by use of Memorizer and create a MaE.
3. Extract scanning field information from the MaE
4. Generate the DMA linking table from scanning field information.
5. Categorize the MaE to create a MaP. If no MaP category match, try to define a new one or use array MaP. For this step to succeed it’s required for the MaE to be *patternable* (see 5.2.2 and 5.3.1).
6. If needed, the MaP is to be reordered to match the PVM sample size (16 bits).
7. Divide the MaP into PVM access formats (groups of 1-8 samples to be accessed in parallel).
8. Generate a permutation table for the PVM that supplies conflict free access for all access formats. If a conflict free solution can’t be found, try to redo the previous step.
9. Implement PVM configuration (DMA linking table and premutation table) as HdS.
7.3 P3RMA Analysis

Source Code

1. Profiler (Real): Identify opportunities of parallelization

Exposer (Memorizer): Expose Memory Accesses

2. Scanning field information

3. MaE

4. Address generator: Generate DMA linking table based on scanning field information

5. MaE Identification (manual): Categorize the MaE to create a MaP

6. Divide into access formats

7. Assignment function generator: Generate a permutation table for the PVM that supplies conflict free access for all access formats needed

8. HdS Code including DMA linking table and permutation table

Figure 7.2: Workflow in P3RMA analysis, based on [7]
The case studies start with selected code at step 2 and does not include the final implementation in step 9.

### 7.3.1 The Memorizer MaE

In step 2 of the P3RMA analysis, the output from Memorizer is to be converted into a MaE (see 5.2.2 and 6.2.7). This is done through a number of substeps:

1. Find blocks of consecutive read or write accesses using the same base address.
2. Put exposed access offsets into tables, keep base address and access type as identifiers. Also include iteration variables.
3. Divide exposed offsets by sample size.
4. Sort tables on offset values.
5. Merge doublets into single offset elements (but keep track of destinations)

The resulting MaE can be divided into two parts, **scanning field information** and **access pattern information**.

**The scanning field information** include access type and range data, used to create the DMA linking table.

**The access pattern information** focus on the sample offset data to create a MaP.

### 7.3.2 MaP Categorizing

In step 5 of the P3RMA analysis, the MaE access pattern information is to be converted into a MaP and categorized. For this step to be possible the MaE needs to hold enough information to create a MaP, that is the MaE needs to be patternable (see 5.2.2 and 5.3.1).

This thesis does not include a general MaP categorization method, but since the cases studied only include constant stride MaPs a specific method can be used:

**Start:** Convert (if needed) the exposed MaP into a differential MaP.

**Case 1:** Constant differential MaP elements indicates 1D constant stride MaP.

**Case 2:** Groups of, or iterations with, constant differential MaP elements indicates 2D constant stride MaP.

**Case 3:** Iterations with groups of constant differential MaP elements indicates set of 2D constant stride MaPs (a 3D constant stride MaP).
7.3.3 PVM Access Formats

In steps 7 and 8 of the P3RMA analysis, the outcome depends on what access formats the PVM supports. As the PVM is a configurable PMA the set of supported access formats is close to unlimited depending on the configuration used.

The extensive MaP analysis to find the optimal PVM access formats is not included in this thesis so the case studies focus on finding a simple but still useful PVM solution.

7.4 How to read the MaE tables

The Memory Access Expositions done in the case studies are presented in table form, below is a description of the contents in those tables.

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Tells what kind of memory accesses are being done, Read or Write.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Address</td>
<td>The base address of the memory exposition, often the name of a variable, a $-sign in front of a variable name indicates if it’s an address or a value.</td>
</tr>
<tr>
<td>Initial Value</td>
<td>The initial value of the Base Address.</td>
</tr>
<tr>
<td>Iteration Expr</td>
<td>Expressions done each iteration of a loop that are interest to the addressing calculations. If there are nested loops there might be multiple Iteration Expr lines, if so they are numbered so inner loops have higher number than outer loops.</td>
</tr>
<tr>
<td>Iteration Init</td>
<td>Initializations of variables before an iteration starts, the same numbers applies to Iteration Init as for Iteration Expr</td>
</tr>
<tr>
<td>Iterations</td>
<td>The number of iterations in a loop, the same numbering applies to Iterations as for Iteration Expr.</td>
</tr>
<tr>
<td>Sample Size</td>
<td>The size in bytes of each sample.</td>
</tr>
<tr>
<td># Elements</td>
<td>The number of elements (samples) read in one iteration of the innermost loop.</td>
</tr>
<tr>
<td>Offset n</td>
<td>The offset to every element, counted from Base Address.</td>
</tr>
</tbody>
</table>

7.5 DCT of Macroblock

This example will show the usage of Memorizer and Memory Access Patterns to expose the memory accesses and help creating DMA linking tables and Permutation tables. For the example we are using the libjpeg software package. This library is used in the course TSEA20 Computer hardware – a computer system on a chip given on the Computer Engineering Division at Linköpings Universitet and thus is familiar territory.

The code we are using for this example is the function jpeg_fdct_islow(), this function can be seen in Appendix C.

\textsuperscript{1}http://www.da.isy.liu.se/courses/tsea02/
7.5.1 Memorizer Output and Construction of MaE

The output of memorizer is listed in Listing D.1 and D.2, found in Appendix D. By applying the steps given in the section 7.3.1, we get a table for first 16 read accesses in Listing D.1, see figure 7.3. Following the same steps for the first set of write accesses, we get the table shown in figure 7.4. The same steps are done for the output in listing D.2, steps for the read accesses are shown in figure 7.5.

7.5.2 Analysis of Output

From the MaE, it’s possible to extract DMA task linking tables and permutation tables for the PMA.

DMA Table

The data given in the tables in figure 7.3 is enough to create a DMA task linking table for the first block of memory reads. This DMA table has information about source and destination memories, the data of the transactions is split up in different blocks each with a source address, a destination address and a length.

The condition governing the loop is $\text{ctr} > 0$, the variable $\text{ctr}$ is initialized to 7 and in each iteration the expression $\text{ctr} = \text{ctr} - 1$ is executed, which means the number of iterations in the MaE is 8, and since all the data in each iteration resides at adjacent addresses the DMA table will consist of 8 blocks. Each block of data consists of 8 values, all 4 bytes in size, resulting in a block length of 32 bytes. The source address of each block is given by the variable $\text{dataptr}$, and in each iteration of the loop this variable is increased by 32.

The steps above will result in the DMA linking table shown in figure 7.6, but by examining the MaE a bit further it can be optimized quite a bit. In this particular example the length of each block of data, and the increase to the variable $\text{dataptr}$ are both 32, this means that not only are the samples in each block adjacent, but the 8 data blocks are also adjacent in memory. So we can reduce the DMA transaction to only one block that fetches all the data at once as seen in figure 7.7.

The memory accesses done by the second set of reads, shown in the MaE in figure 7.5, are not as easy to use as a base for constructing a DMA task linking table since the accesses are not on adjacent addresses in the main memory.

By comparing the scanning field of the first and the second set of read accesses, it can be seen that they work on the same set of data, and thus is no extra DMA needed to get the second block, the MaE for the second set will therefore only be used when creating the permutation tables.

Permutation Table

The DCT functions use one scanning field (int[256] $\text{dataptr}$). Using the proposed PMA (PVM, see 2.2.1), this results in two used access formats.

Row access, one sample stride, from the first exposition.

Column access, eight sample stride, from second exposition.
### Step 1 & 2
First Block of Consecutive Reads

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$ dataptr</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$ data</td>
</tr>
<tr>
<td>Iteration Expr</td>
<td>$ dataptr + = 32</td>
</tr>
<tr>
<td>Iterations</td>
<td>8</td>
</tr>
<tr>
<td>Sample Size</td>
<td>4</td>
</tr>
<tr>
<td># Elements</td>
<td>16</td>
</tr>
<tr>
<td>Offset 1</td>
<td>0</td>
</tr>
<tr>
<td>Offset 2</td>
<td>28</td>
</tr>
<tr>
<td>Offset 3</td>
<td>0</td>
</tr>
<tr>
<td>Offset 4</td>
<td>28</td>
</tr>
<tr>
<td>Offset 5</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Offset 15</td>
<td>12</td>
</tr>
<tr>
<td>Offset 16</td>
<td>16</td>
</tr>
</tbody>
</table>

### Step 3
Offsets Divided by Sample Size

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$ dataptr</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$ data</td>
</tr>
<tr>
<td>Iteration Expr</td>
<td>$ dataptr + = 32</td>
</tr>
<tr>
<td>Iterations</td>
<td>8</td>
</tr>
<tr>
<td>Sample Size</td>
<td>4</td>
</tr>
<tr>
<td># Elements</td>
<td>16</td>
</tr>
<tr>
<td>Offset 1</td>
<td>0</td>
</tr>
<tr>
<td>Offset 2</td>
<td>7</td>
</tr>
<tr>
<td>Offset 3</td>
<td>0</td>
</tr>
<tr>
<td>Offset 4</td>
<td>7</td>
</tr>
<tr>
<td>Offset 5</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Offset 15</td>
<td>3</td>
</tr>
<tr>
<td>Offset 16</td>
<td>4</td>
</tr>
</tbody>
</table>

### Step 4
Offset Values Sorted

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$ dataptr</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$ data</td>
</tr>
<tr>
<td>Iteration Expr</td>
<td>$ dataptr + = 32</td>
</tr>
<tr>
<td>Iterations</td>
<td>8</td>
</tr>
<tr>
<td>Sample Size</td>
<td>4</td>
</tr>
<tr>
<td># Elements</td>
<td>16</td>
</tr>
<tr>
<td>Offset 1</td>
<td>0</td>
</tr>
<tr>
<td>Offset 3</td>
<td>0</td>
</tr>
<tr>
<td>Offset 5</td>
<td>1</td>
</tr>
<tr>
<td>Offset 7</td>
<td>1</td>
</tr>
<tr>
<td>Offset 9</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Offset 2</td>
<td>7</td>
</tr>
<tr>
<td>Offset 4</td>
<td>7</td>
</tr>
</tbody>
</table>

### Step 5
Duplicates Merged

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$ dataptr</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$ data</td>
</tr>
<tr>
<td>Iteration Expr</td>
<td>$ dataptr + = 32</td>
</tr>
<tr>
<td>Iterations</td>
<td>8</td>
</tr>
<tr>
<td>Sample Size</td>
<td>4</td>
</tr>
<tr>
<td># Elements</td>
<td>8</td>
</tr>
<tr>
<td>Offset 1,3</td>
<td>0</td>
</tr>
<tr>
<td>Offset 5,7</td>
<td>1</td>
</tr>
<tr>
<td>Offset 9,11</td>
<td>2</td>
</tr>
<tr>
<td>Offset 13,15</td>
<td>3</td>
</tr>
<tr>
<td>Offset 14,16</td>
<td>4</td>
</tr>
<tr>
<td>Offset 10,12</td>
<td>5</td>
</tr>
<tr>
<td>Offset 6,8</td>
<td>6</td>
</tr>
<tr>
<td>Offset 2,4</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 7.3: From exposed memory accesses to MaE for the first block of reads
<table>
<thead>
<tr>
<th>MaE of the first write accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Content</strong></td>
</tr>
<tr>
<td>Access Type</td>
</tr>
<tr>
<td>Base Address</td>
</tr>
<tr>
<td>Initial Value</td>
</tr>
<tr>
<td>Iteration Expr</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Sample Size</td>
</tr>
<tr>
<td># Elements</td>
</tr>
<tr>
<td>Offset 1</td>
</tr>
<tr>
<td>Offset 8</td>
</tr>
<tr>
<td>Offset 3</td>
</tr>
<tr>
<td>Offset 7</td>
</tr>
<tr>
<td>Offset 2</td>
</tr>
<tr>
<td>Offset 6</td>
</tr>
<tr>
<td>Offset 4</td>
</tr>
<tr>
<td>Offset 5</td>
</tr>
</tbody>
</table>

Figure 7.4: MaE for the first block of writes

There are a lot of simple module assignment functions that support both access formats so let’s take a linear module assignment function:

$$S(r) = \left\lfloor \frac{r + \ell}{8} \right\rfloor \mod 8$$

And because

$$a(r) = \left\lfloor \frac{r}{8} \right\rfloor$$

it can be implemented as a permutation table.

### 7.6 Motion Compensation in x264

x264 is an open source H.264/AVC video encoding library [2]. This case study will investigate the motion compensation on chroma values in this software package, the function in question is `motion Compensation Chroma()` the source code of this function can be found in listing F.1 in appendix F.

#### 7.6.1 Memorizer Output

The Memorizer output from can be found in figure G.1 in appendix G. One problem we face here is that due to the lack of inter-function dependencies we don’t know any numerical values of the parameters, however since the code is part of the motion compensation in a H.264 we can make qualified assumptions on the numerical values.

In H.264 the most common block size for chroma motion compensation is 8x8 [11] so `i_width` and `i_height` will both be assumed to be 8. To get an idea of the other values the x264 encoder was executed on a test file and the values recorded, that gave us the two strides, `i_src_stride = 120` and `i_dst_stride = 8`. `mvx` and `mvy` took values in the range $-31$ to 1, but since they only affect the
Step 1 & 2: Second Block of Consecutive Reads

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
<th>Access Type</th>
<th>Base Address</th>
<th>Initial Value</th>
<th>Iteration Expr</th>
<th>Iterations</th>
<th>Sample Size</th>
<th># Elements</th>
<th>Offset 1</th>
<th>Offset 2</th>
<th>Offset 3</th>
<th>Offset 4</th>
<th>Offset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
<td>Base Address</td>
<td>$dataptr$</td>
<td>$data$</td>
<td>$dataptr+=4$</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>224</td>
<td>0</td>
<td>224</td>
<td>32</td>
</tr>
<tr>
<td>Base Address</td>
<td></td>
<td>$dataptr$</td>
<td></td>
<td>$data$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 3: Offsets Divided by Sample Size

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
<th>Access Type</th>
<th>Base Address</th>
<th>Initial Value</th>
<th>Iteration Expr</th>
<th>Iterations</th>
<th>Sample Size</th>
<th># Elements</th>
<th>Offset 1</th>
<th>Offset 2</th>
<th>Offset 3</th>
<th>Offset 4</th>
<th>Offset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
<td>Base Address</td>
<td>$dataptr$</td>
<td>$data$</td>
<td>$dataptr+=4$</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>Base Address</td>
<td></td>
<td>$dataptr$</td>
<td></td>
<td>$data$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 4: Offset Values Sorted

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
<th>Access Type</th>
<th>Base Address</th>
<th>Initial Value</th>
<th>Iteration Expr</th>
<th>Iterations</th>
<th>Sample Size</th>
<th># Elements</th>
<th>Offset 1</th>
<th>Offset 2</th>
<th>Offset 3</th>
<th>Offset 4</th>
<th>Offset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
<td>Base Address</td>
<td>$dataptr$</td>
<td>$data$</td>
<td>$dataptr+=4$</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>Base Address</td>
<td></td>
<td>$dataptr$</td>
<td></td>
<td>$data$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 5: Duplicates Merged

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
<th>Access Type</th>
<th>Base Address</th>
<th>Initial Value</th>
<th>Iteration Expr</th>
<th>Iterations</th>
<th>Sample Size</th>
<th># Elements</th>
<th>Offset 1</th>
<th>Offset 2</th>
<th>Offset 3</th>
<th>Offset 4</th>
<th>Offset 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
<td>Base Address</td>
<td>$dataptr$</td>
<td>$data$</td>
<td>$dataptr+=4$</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>Base Address</td>
<td></td>
<td>$dataptr$</td>
<td></td>
<td>$data$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offset 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.5: From exposed memory accesses to MaE for the second block of reads
<table>
<thead>
<tr>
<th>Word</th>
<th>Content</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Task Name</td>
<td>Load a Macro Block</td>
</tr>
<tr>
<td>2</td>
<td>Mode and Priority</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>Source Device</td>
<td>Main off-chip Memory</td>
</tr>
<tr>
<td>4</td>
<td>Destination Device</td>
<td>Parallel Scratch Pad Memory</td>
</tr>
<tr>
<td>5</td>
<td>Number of blocks</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Block 1 Source Address</td>
<td>$\text{dataptr}$</td>
</tr>
<tr>
<td>7</td>
<td>Block 1 Destination Address</td>
<td>Scratch Pad Base Address + 0</td>
</tr>
<tr>
<td>8</td>
<td>Block 1 Length</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>Block 2 Source Address</td>
<td>$\text{dataptr} + 32$</td>
</tr>
<tr>
<td>10</td>
<td>Block 2 Destination Address</td>
<td>Scratch Pad Base Address + 32</td>
</tr>
<tr>
<td>11</td>
<td>Block 2 Length</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>Block 3 Source Address</td>
<td>$\text{dataptr} + 2 \times 32$</td>
</tr>
<tr>
<td>13</td>
<td>Block 3 Destination Address</td>
<td>Scratch Pad Base Address + 2 \times 32</td>
</tr>
<tr>
<td>14</td>
<td>Block 3 Length</td>
<td>32</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Block 8 Source Address</td>
<td>$\text{dataptr} + 8 \times 32$</td>
</tr>
<tr>
<td>28</td>
<td>Block 8 Destination Address</td>
<td>Scratch Pad Base Address + 8 \times 32</td>
</tr>
<tr>
<td>29</td>
<td>Block 8 Length</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 7.6: First draft of a DMA Table for reading the data

<table>
<thead>
<tr>
<th>Word</th>
<th>Content</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Task Name</td>
<td>Load a Macro Block</td>
</tr>
<tr>
<td>2</td>
<td>Mode and Priority</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>Source Device</td>
<td>Main off-chip Memory</td>
</tr>
<tr>
<td>4</td>
<td>Destination Device</td>
<td>Parallel Scratch Pad Memory</td>
</tr>
<tr>
<td>5</td>
<td>Number of blocks</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Block 1 Source Address</td>
<td>$\text{dataptr}$</td>
</tr>
<tr>
<td>7</td>
<td>Block 1 Destination Address</td>
<td>Scratch Pad Base Address + 0</td>
</tr>
<tr>
<td>8</td>
<td>Block 1 Length</td>
<td>256</td>
</tr>
</tbody>
</table>

Figure 7.7: DMA Table for reading the data
initial value of the source address, they will both be assumed to be 0 in the case study.

Each iteration of the loop one sample is written, it is a pure sequential index based write, done one sample at time, so without unrolling the loop no parallelization can be done.

The reading is a bit more interesting. Four samples are read each iteration, adjacent in pairs and with a set distance between them, depending on the i_src_stride variable. A visual example of how the memory reads could look is given in figure 7.8. In this example \( i_{\text{height}} = 8, i_{\text{width}} = 8, i_{\text{src_stride}} = 32 \) The first iteration of the loop, the four samples marked with A are read, the second iteration the samples marked with B are read and up till the 16th iteration where the blocks marked with P are read. The iterations after 16 are not shown in the figure since they happen outside of the 8x8 square.

\[
\begin{array}{ccccccccc}
A & AB & BC & CD & DE & EF & FG & GH \\
H & & & & & & & \\
\hline
\hline
A & AB & BC & CD & DE & EF & FG & GH \\
I & & & & & & & \\
\hline
A & AB & BC & CD & DE & EF & FG & GH \\
P & & & & & & & \\
\end{array}
\]

Figure 7.8: x264 reading pattern

### 7.6.2 Construction of MaE

Following the steps outlined in section 7.3.1 for the memory reads, the MaE listed in figure 7.9 is constructed. When constructing this MaE, the fact that srcp can be described as \( src + i_{\text{src_stride}} \) was used. The reason this is not explicitly shown in the memorizer output is that variable dependencies between basic blocks is not exposed the same way as dependencies inside the blocks.

**DMA Table**

The data in figure 7.10 is used to find the scanning field for the accesses, which in turn will be used to create the DMA task linking table.

First we create the scanning field for the inner loop, it reads four values each of the eight iterations. The base address of the pointers is not yet taken into account.

\[
R_{\text{inner}} = \{ x, x + 1, x + 120, x + 121 : x \in \mathbb{N}, 0 \leq x < 8 \}
\]
<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$src$</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$src + (mv_{y} &gt;&gt; 3) \cdot i_{src _stride} + (mv_{x} &gt;&gt; 3)$</td>
</tr>
<tr>
<td>Iteration 1 Expr</td>
<td>$src + = i_{src _stride}$</td>
</tr>
<tr>
<td>Iteration 1 Init</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>Iteration 2 Expr</td>
<td>$x += 1$</td>
</tr>
<tr>
<td>Iterations 1</td>
<td>$i_{height}$</td>
</tr>
<tr>
<td>Iterations 2</td>
<td>$i_{width}$</td>
</tr>
<tr>
<td>Sample Size</td>
<td>1</td>
</tr>
<tr>
<td># Elements</td>
<td>4</td>
</tr>
<tr>
<td>Offset 1</td>
<td>$x$</td>
</tr>
<tr>
<td>Offset 2</td>
<td>$x + 1$</td>
</tr>
<tr>
<td>Offset 3</td>
<td>$i_{src _stride} + x$</td>
</tr>
<tr>
<td>Offset 4</td>
<td>$i_{src _stride} + x + 1$</td>
</tr>
</tbody>
</table>

Figure 7.9: MaE of the memory reads in the chroma motion compensation without numerical values

<table>
<thead>
<tr>
<th>Content</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
<td>Read</td>
</tr>
<tr>
<td>Base Address</td>
<td>$src$</td>
</tr>
<tr>
<td>Initial Value</td>
<td>$src + = 120$</td>
</tr>
<tr>
<td>Iteration 1 Expr</td>
<td>$src + = 120$</td>
</tr>
<tr>
<td>Iteration 2 Init</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>Iteration 2 Expr</td>
<td>$x += 1$</td>
</tr>
<tr>
<td>Iterations 1</td>
<td>8</td>
</tr>
<tr>
<td>Iterations 2</td>
<td>8</td>
</tr>
<tr>
<td>Sample Size</td>
<td>1</td>
</tr>
<tr>
<td># Elements</td>
<td>4</td>
</tr>
<tr>
<td>Offset 1</td>
<td>$x$</td>
</tr>
<tr>
<td>Offset 2</td>
<td>$x + 1$</td>
</tr>
<tr>
<td>Offset 3</td>
<td>$x + 120$</td>
</tr>
<tr>
<td>Offset 4</td>
<td>$x + 121$</td>
</tr>
</tbody>
</table>

Figure 7.10: MaE of the memory reads in the chroma motion compensation with numerical values
7.6 Motion Compensation in x264

<table>
<thead>
<tr>
<th>MaE of the memory writes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Content</strong></td>
</tr>
<tr>
<td>Access Type</td>
</tr>
<tr>
<td>Base Address</td>
</tr>
<tr>
<td>Initial Value</td>
</tr>
<tr>
<td>Iteration 1 Expr</td>
</tr>
<tr>
<td>Iteration 2 Init</td>
</tr>
<tr>
<td>Iteration 2 Expr</td>
</tr>
<tr>
<td>Iterations 1</td>
</tr>
<tr>
<td>Iterations 2</td>
</tr>
<tr>
<td>Sample Size</td>
</tr>
<tr>
<td># Elements</td>
</tr>
<tr>
<td>Offset 1</td>
</tr>
</tbody>
</table>

Figure 7.11: MaE of the memory writes in the chroma motion compensation without numerical values

Since we are not interested in duplicate values when constructing the DMA table this can be simplified as

\[ R_{\text{inner}} = \{x, x + 120 : x \in \mathbb{N}, 0 \leq x \leq 8\} \]

By adding the information from the outer loop we end up with the following scanning field for the whole set of accesses

\[ R = \{120 \cdot y + x : x, y \in \mathbb{N}, 0 \leq y < 8, x \in R_{\text{inner}}\} \]

By expanding the variables in the sets above we get

\[ R = \{0, 1, 2, 3, 4, 5, 6, 7, 8, \\
120, 121, 122, 123, 124, 125, 126, 127, 128, \\
120, 121, 122, 123, 124, 125, 126, 127, 128, \\
240, 241, 242, 243, 244, 245, 246, 247, 248, \\
240, 241, 242, 243, 244, 245, 246, 247, 248, \\
\vdots \\
840, 841, 842, 843, 844, 845, 846, 847, 848 \\
960, 961, 962, 963, 964, 965, 966, 967, 968\} \]

The duplicate values are still of no interest and the expression for \( R \) can be written as

\[ R = \{120 \cdot y + x : x, y \in \mathbb{N}, 0 \leq x \leq 8, 0 \leq y \leq 8\} \]

From this scanning field it’s easy to construct the DMA table in figure 7.12

**Permutations**

The memory accesses matches the Block burst MaP category described in 5.6 with a row size of 120. \((P_{s}^{r}(r, 1, 120))\)

\[
f(i, j) = \begin{cases} 
    r & i, j = 0 \\
    f(0, j - 1) + 120 & i = 0, \quad j = 0 \\
    f(i - 1, j) + 1 & i = 1, \quad \forall j 
\end{cases}
\]
### Table 7.12: DMA Table to fetch the data needed for the motion compensation

<table>
<thead>
<tr>
<th>Word</th>
<th>Content</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Task Name</td>
<td>Load Data for Motion Compensation</td>
</tr>
<tr>
<td>2</td>
<td>Mode and Priority</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>Source Device</td>
<td>Main off-chip Memory</td>
</tr>
<tr>
<td>4</td>
<td>Destination Device</td>
<td>Parallel Scratch Pad Memory</td>
</tr>
<tr>
<td>5</td>
<td>Number of blocks</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Block 1 Source Address</td>
<td>$src</td>
</tr>
<tr>
<td>7</td>
<td>Block 1 Destination Address</td>
<td>Scratch Pad Base Address + 0</td>
</tr>
<tr>
<td>8</td>
<td>Block 1 Length</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Block 2 Source Address</td>
<td>$src + 1 * 120</td>
</tr>
<tr>
<td>10</td>
<td>Block 2 Destination Address</td>
<td>Scratch Pad Base Address + 1 * 8</td>
</tr>
<tr>
<td>11</td>
<td>Block 2 Length</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Block 8 Source Address</td>
<td>$src + 8 * 120</td>
</tr>
<tr>
<td>28</td>
<td>Block 8 Destination Address</td>
<td>Scratch Pad Base Address + 8 * 8</td>
</tr>
<tr>
<td>29</td>
<td>Block 8 Length</td>
<td>8</td>
</tr>
</tbody>
</table>

With this knowledge the module assignment function $S(r)$ for an eight-way PVM like the one described in 2.2.1 can be created. The expression $2 \cdot r$ stems from the block being 2 pixels wide.

$$S(r) = \left( r + \left\lfloor \frac{2 \cdot r}{120} \right\rfloor \right) \mod 8$$

The address function $a(r)$ needs a way to compensate for the jump in addresses. This is done by downscaling by a factor $10/120$ to decrease the number of unused cells in each memory bank.

$$a(r) = \left\lfloor \frac{10 \cdot r}{120} + r \mod 120}{8} \right\rfloor$$

These two functions will lead to the partitioning of the data shown in figure 7.13. This figure shows how the data from one DMA transaction will be partitioned in the PVM. The number in the upper left corner of each square is the memory bank the sample will be placed in, and the number in the lower right is the address in that memory bank. The shaded squares in the figure shows the accesses done for $y = 0, x = 0$ and $y = 2, x = 3$.

The observant reader will notice that some addresses will not be used in all memory banks. This is due to the difference between the number of values in each row of data (9) and the factor used to scale down the addresses (10).
### Figure 7.13: Partitioning of the data in the PVM
Chapter 8

Result and Conclusion

Thesis result presentation, conclusion of the thesis work related to set objectives and suggestions on future work in the area.

8.1 Memory Access Pattern

The concept of a MaP, corresponding definitions and representation descriptions are major results of the research presented in this thesis. A MaP is identified and defined as an abstract object for representing memory access and presented with different mathematical representations, all related to common types of memory accesses exposed in code. Although this close relation to target and application specific code the MaP itself is a system independent representation.

Using the presented MaP representations any memory access can be represented by a MaP, although there are some MaP results that are found more common and interesting than others. These results, which are categories of MaPs rather than specific ones, are defined and exemplified in the thesis. By use of these categories most MaPs resulting from code analysis can be categorized.

As a result of the thesis research, different applications of MaP results are suggested, depending on context:

**PMA software implementation** - In a case when a given (hardwired) PMA exists the MaP can be used to find an optimal software memory access implementation. In this case one can also list a set of coding templates supported by the PMA, each corresponding to a specific MaP.

**PMA configuration** - When using a configurable PMA the MaP can be used to find an optimal configuration. After the configuration is set the MaP can also be used to optimize software implementation.

**PMA design** - One MaP or several, resulting for analysis of specific applications, may be used to design a PMA for the application specific system.

The thesis include case studies in the case of a system with a configurable PMA but does not prove any of the suggested applications.
8.2 Memorizer & Memory Access Exposition

In order to investigate if and how MaPs could automatically be exposed from source code we built the tool Memorizer. This tool finds pointer and array accesses and the calculations leading up to what address or index is used for the memory access. The output from Memorizer can be used to create a Memory Access Exposition (MaE).

The Memorizer output includes all the information needed to get the scanning field for a set of memory accesses and create a DMA task linking table, the output can also be used to list the needed access formats for the memory access and be used when creating Permutation Tables for a parallel scratch pad memory. Finally the resulting MaE can also be used to identify MaPs or to construct MaCTs.

Memorizer is not a complete tool, but should be seen as a proof of concept that automatic exposure of memory accesses can be done. Section 8.5.3 lists a number of improvements that can be done before Memorizer becomes a usable tool.

8.3 Memory Access Exposition Analysis

In the cases investigated in chapter 7 it’s shown that by analysing the memorizer output it’s possible to identify MaP categories (see 5.6). The resulting category gives the access formats in use, and when the access formats is known it’s possible to create a permutation table.

From the Memorizer output scanning fields for the set of memory accesses done in the function can be detected. This information is directly usable to create a DMA task linking table as described in 7.2.

8.4 Conclusion

In the beginning of the work three questions were asked, we will look back to those questions and see how they are answered. This section does not give complete answers to the questions since that are being done in detail in earlier chapters.

How do we define a Memory Access Pattern? An abstract object for representing memory access, presented by a mathematical representation, related to common types of memory accesses exposed in code.

How do we expose a Memory Access Pattern from Source Code? The tool Memorizer exposes memory access and presents the user with different representations of the interesting operations and expressions. The output from Memorizer is not a Memory Access Pattern, but can be used to identify or construct one.

How can we identify a Memory Access Pattern? The output from Memorizer is enough information to identify and categorize Memory Access Patterns, this is shown for specific cases in our case studies.
8.5 Future Work

As stated in the introduction, this thesis is a small part of a larger project, in order to leave the work over to someone else we here present our thoughts on where to continue the work.

8.5.1 Memory Access Pattern Concepts

The MaP concepts defined and described in this thesis can be seen as a first step on the way to a hardware independent P3RMA formalization. So far the concept has focused on the definition of *exposable code* and *patternable expositions*, to supply requirements on memory access exposition (applied in Memorizer), a suggestion of future focus would be on analyze a MaP and enable parallel implementation. If this means an extension, reduction or total redefinition of the MaP concept is still to be seen.

8.5.2 Memory Access Pattern Usage

On the question of MaP usefulness this thesis does only supply suggestions, all concerning PMA design and implementation. What is still to be done is to prove that these uses are possible. There are also possible to find other types of MaP analysis and intended use.

8.5.3 Memorizer

As stated earlier, Memorizer is a proof of concept, showing that Memory Accesses can be exposed from source code. The output from Memorizer should be possible to use for automatic matching against a MaP if a program for that is written. This could either be done by using the XML output, or Memorizer could be altered to export in some other format, maybe with more information.

There are also areas where Memorizer as a tool could be improved. Currently information about conditional execution is being discarded, Memorizer keeps track of different paths of execution, but not the conditions deciding which part is taken.

The exposure of variable dependencies between Basic Block could be improved, as well as relations between functions. It should be possible to map the input variables of a function to the parameters of said function if the call to the function is defined in the same file as the function.

To give an example of the conditional execution and the inter basic block dependencies, consider the code in listing 8.1. For the human reader, it’s easy to see that expression for \( j \) is either \( a + 1 \) or \( b + 1 \) depending on wether \( a \) is bigger than \( b \) or not, but this information is lost in Memorizer.

Memorizer is initialized and destroyed once for each source file encountered during compilation. If dependencies between functions defined in different source files are to be found, the internal DDG Nodes for each source file must be saved to disk and once all source files are compiled and analyzed, a second stage of static analysis can be done, examining all the data, of all functions in all source files at once.

Another way to get dependencies between functions would be to add instrumentation calls during compilation that tracks what variables are used as...
int foo(int a, int b)
{
    int i, j;
    if (a > b)
        i = a;
    else
        i = b;
    j = i + 1;
}

Listing 8.1: Dependencies that Memorizer fails to expose

parameters to functions. Once the program is compiled it can be executed and store this information to a file. Then the static output from Memorizer and the dynamic output from the instrumentation library could be combined to create a full dependency graph even between functions.
Part IV

Appendices
Appendix A

Memorizer XML Output

This is the XML output from Memorizer for the example code given in listing 6.1

```xml
<function>
  <name>foo</name>
  <basic_block>
    <name>foo_0000</name>
    <loop_depth>0</loop_depth>
    <nodes>
      <node id="0">
        <index>1</index>
        <ref_size>0</ref_size>
        <type>LOOP</type>
        <expr>i = 0</expr>
      </node>
    </nodes>
  </basic_block>
  <basic_block>
    <name>foo_0001</name>
    <loop_depth>1</loop_depth>
    <nodes>
      <node id="0">
        <index>4</index>
        <ref_size>1</ref_size>
        <type>WRITE</type>
        <expr>*(MEM)</expr>
      </node>
      <node id="1">
        <index>8</index>
        <ref_size>1</ref_size>
        <type>ADDR</type>
        <expr>MEM = MEM + 1</expr>
      </node>
      <node id="2">
        <index>12</index>
        <ref_size>0</ref_size>
        <type>LOOP</type>
        <expr>i = i + 1</expr>
      </node>
    </nodes>
  </basic_block>
</function>
```
Listing A.1: Memory Access XML output from Memorizer
Appendix B

G++ Patch

In order to be able to use C++ code in the GEM module some changes were needed to work around the difference in typedef in C and C++, the patch is included here for completeness.

```
diff -u ../tmp/gcc-4.1.0/gcc/cfgloop.h gcc/cfgloop.h
--- ../tmp/gcc-4.1.0/gcc/cfgloop.h 2005-08-24 09:56:56.000000000 +0200
+++ gcc/cfgloop.h 2007-02-21 15:39:02.000000000 +0100
@@ -417,7 +417,7 @@
 static inline struct niter_desc *
     simple_loop_desc ( struct loop *loop )
 {
-    return loop->aux;
+    return ( struct niter_desc *)loop->aux;
 }

 /* The properties of the target. */
```

```
Common subdirectories: ../tmp/gcc-4.1.0/gcc/config and gcc/
config
Only in gcc: cp
Common subdirectories: ../tmp/gcc-4.1.0/gcc/doc and gcc/doc
Common subdirectories: ../tmp/gcc-4.1.0/gcc/ginclude and gcc/
ginlude
Only in gcc: include
Common subdirectories: ../tmp/gcc-4.1.0/gcc/objcp and gcc/
objcp
Common subdirectories: ../tmp/gcc-4.1.0/gcc/po and gcc/po
diff -u ../tmp/gcc-4.1.0/gcc/tree-flow.h gcc/tree-flow.h
@@ -361,7 +361,7 @@
 struct edge_prediction GTY( ( chain_next ( "%h.next" ) ) )
 {
     struct edge_prediction *next;
-    edge edge;
+    enum br_pred predictor predictor;
```
int probability;
};
diff -u ../tmp/gcc-4.1.0/gcc/tree-flow-inline.h gcc/tree-flow-inline.h
--- ../tmp/gcc-4.1.0/gcc/tree-flow-inline.h 2006-02-09
  15:00:59.000000000 +0100
+++ gcc/tree-flow-inline.h 2007-02-21 15:44:42.000000000 +0100
@@ -78,7 +78,7 @@
    first_referenced_var (referenced_var_iterator *iter)
    {
        struct int_tree_map *itm;
-       itm = first_htab_element (&iter->hti, referenced_vars);
+       itm = (int_tree_map*) first_htab_element (&iter->hti, referenced_vars);
        if (!itm)
            return NULL;
        return itm->to;
@@ -100,7 +100,7 @@
    next_referenced_var (referenced_var_iterator *iter)
    {
        struct int_tree_map *itm;
-       itm = next_htab_element (&iter->hti);
+       itm = (int_tree_map*)next_htab_element (&iter->hti);
        if (!itm)
            return NULL;
        return itm->to;
    }

Common subdirectories: ../tmp/gcc-4.1.0/gcc/treelang and gcc/treelang
diff -u ../tmp/gcc-4.1.0/gcc/varray.h gcc/varray.h
--- ../tmp/gcc-4.1.0/gcc/varray.h 2005-06-25
  04:02:01.000000000 +0200
+++ gcc/varray.h 2007-02-21 15:31:46.000000000 +0200
@@ -90,11 +90,11 @@
    struct bitmap_head_def *gty ( (length ("%0.num_elements")),
        tag ("VARRAY_DATA_BITMAP")) bitmap[1];
@@ -106,7 +106,7 @@
    struct edge_def *gty ( (length ("%0.num_elements")),
        tag ("VARRAY_DATA_EDGE")) e[1];
    struct tree_def *gty ( (length ("%0.num_elements")),
        tag ("VARRAY_DATA_TREE")) tree[1];
skip ("")
+
+ union tree_node *
+ *GTY ((length ("%0.num_elements"),
+ skip (""),
+ tag ("VARRAY_DATA_TREE_PTR"))) tp
+ [1];
}

varray_data;

Only in gcc: varray.h.gch

Listing B.1: Patch to allow using GEM and G++ together
Appendix C

jpeg_fdct_islow()

This is the code of the function jpeg_fdct_islow as found in libjpeg, used in the course TSEA04¹ at Linköpings Tekniska Högskola.

/*
 * Perform the forward DCT on one block of samples.
 */

void jpeg_fdct_islow(DCTELEM * data)
{
    INT32 tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    INT32 tmp10, tmp11, tmp12, tmp13;
    INT32 z1, z2, z3, z4, z5;
    DCTELEM * dataptr;
    int ctr;
    SHIFT_TEMPS
    unsigned int startcycle;

    int i;
    short val;

    startcycle = gettimer();

    /*
     * Pass 1: process rows. Note results are scaled up by sqrt (8)
     * compared to a true DCT;
     */

    dataptr = data;
    for (ctr = DCTSIZE - 1; ctr >= 0; ctr--)
    {
        tmp0 = dataptr[0] + dataptr[7];
        tmp7 = dataptr[0] - dataptr[7];
        tmp1 = dataptr[1] + dataptr[6];
        tmp6 = dataptr[1] - dataptr[6];
        tmp2 = dataptr[2] + dataptr[5];
        tmp5 = dataptr[2] - dataptr[5];
        tmp3 = dataptr[3] + dataptr[4];

¹Computer hardware - a computer system on a chip
tmp4 = dataptr[3] - dataptr[4];

/*
 * Even part per LLEM figure 1 — note that published figure is faulty;
 * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
 */
tmp10 = tmp0 + tmp3;
tmp13 = tmp0 - tmp3;
tmp11 = tmp1 + tmp2;
tmp12 = tmp1 - tmp2;
 dataptr[0] = (DCTELEM) (tmp10 + tmp11);
dataptr[4] = (DCTELEM) (tmp10 - tmp11);

dataptr[2] = (DCTELEM) DESCALE(MULTIPLY(tmp12, FIX_C6) +
             MULTIPLY(tmp13, FIX_S6), CONST_BITS);
dataptr[6] = (DCTELEM) DESCALE(-MULTIPLY(tmp12, FIX_S6) +
             MULTIPLY(tmp13, FIX_C6), CONST_BITS);

/*
 * Odd part per figure 8 — note paper omits factor of sqrt(2). cK
 * represents cos(K*pi/16). i0...i3 in the paper are tmp4..tmp7 here.
 */
z1 = tmp4 + tmp7;
z2 = tmp5 + tmp6;
z3 = tmp4 + tmp6;
z4 = tmp5 + tmp7;
z5 = MULTIPLY(z3 + z4, FIX_1_175875602);
tmp4 = MULTIPLY(tmp4, FIX_0_298631336);
tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
z1 = MULTIPLY(z1, -FIX_0_899976223);
z2 = MULTIPLY(z2, -FIX_2_562915447);
z3 = MULTIPLY(z3, -FIX_1_961570560);
z4 = MULTIPLY(z4, -FIX_0_390180644);

z3 += z5;
z4 += z5;

dataptr[7] = (DCTELEM) DESCALE(tmp4 + z1 + z3, CONST_BITS);
dataptr[5] = (DCTELEM) DESCALE(tmp5 + z2 + z4, CONST_BITS);
dataptr[3] = (DCTELEM) DESCALE(tmp6 + z2 + z3, CONST_BITS);
dataptr[1] = (DCTELEM) DESCALE(tmp7 + z1 + z4, CONST_BITS);


dataptr += DCTSIZE; /* advance pointer to next row */

/*
* Pass 2: process columns. We remove the PASS1_BITS scaling,
* but leave the
* results scaled up by an overall factor of 8.
*/
dataptr = data;
for (ctr = DCTSIZE - 1; ctr >= 0; ctr--)
{
    tmp0 = dataptr[DCTSIZE * 0] + dataptr[DCTSIZE * 7];
    tmp7 = dataptr[DCTSIZE * 0] - dataptr[DCTSIZE * 7];
    tmp1 = dataptr[DCTSIZE * 1] + dataptr[DCTSIZE * 6];
    tmp6 = dataptr[DCTSIZE * 1] - dataptr[DCTSIZE * 6];
    tmp2 = dataptr[DCTSIZE * 2] + dataptr[DCTSIZE * 5];
    tmp5 = dataptr[DCTSIZE * 2] - dataptr[DCTSIZE * 5];
    tmp3 = dataptr[DCTSIZE * 3] + dataptr[DCTSIZE * 4];
    tmp4 = dataptr[DCTSIZE * 3] - dataptr[DCTSIZE * 4];

    tmp10 = tmp0 + tmp3;
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE * 0] = (DCTELEM) tmp10 + tmp11;
    dataptr[DCTSIZE * 4] = (DCTELEM) tmp10 - tmp11;

    dataptr[DCTSIZE * 2] = (DCTELEM) DESCALE(MULTIPLY(tmp12,
        FIX_C6) + MULTIPLY(tmp13, FIX_S6), CONST_BITS);
    dataptr[DCTSIZE * 6] = (DCTELEM) DESCALE(-MULTIPLY(tmp12,
        FIX_S6) + MULTIPLY(tmp13, FIX_C6), CONST_BITS);

    z1 = tmp4 + tmp7;
    z2 = tmp5 + tmp6;
    z3 = tmp4 + tmp6;
    z4 = tmp5 + tmp7;
    z5 = MULTIPLY(z3 + z4, FIX_1.175875602);
    tmp4 = MULTIPLY(tmp4, FIX_0.298631336);

}
tmp5 = MULTIPLY(tmp5, FIX_2_053119869);
tmp6 = MULTIPLY(tmp6, FIX_3_072711026);
tmp7 = MULTIPLY(tmp7, FIX_1_501321110);
z1 = MULTIPLY(z1, -FIX_0_899976223);
z2 = MULTIPLY(z2, -FIX_2_562915447);
z3 = MULTIPLY(z3, -FIX_1_961570560);
z4 = MULTIPLY(z4, -FIX_0_390180644);

z3 += z5;
z4 += z5;

dataptr[DCTSIZE*7] = (DCTELEM) DESCALE(tmp4 + z1 + z3, CONST_BITS);
dataptr[DCTSIZE*5] = (DCTELEM) DESCALE(tmp5 + z2 + z4, CONST_BITS);
dataptr[DCTSIZE*3] = (DCTELEM) DESCALE(tmp6 + z2 + z3, CONST_BITS);
dataptr[DCTSIZE*1] = (DCTELEM) DESCALE(tmp7 + z1 + z4, CONST_BITS);

dataptr++; /* advance pointer to next column */

Listing C.1: Original jpeg_fdct_islow()
## Appendix D

### Memorizer Output from DCT Case Study

The output from memorizer is a bit too large to fit nicely into the text, therefore it’s included in this appendix.

<table>
<thead>
<tr>
<th>Basic Block: jpeg_fdct_islow_0000</th>
<th>Loop Depth: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>size</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Block: jpeg_fdct_islow_0001</th>
<th>Loop Depth: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>size</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td>47</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>4</td>
</tr>
<tr>
<td>61</td>
<td>4</td>
</tr>
<tr>
<td>67</td>
<td>4</td>
</tr>
<tr>
<td>75</td>
<td>4</td>
</tr>
<tr>
<td>81</td>
<td>4</td>
</tr>
<tr>
<td>89</td>
<td>4</td>
</tr>
<tr>
<td>95</td>
<td>4</td>
</tr>
<tr>
<td>103</td>
<td>4</td>
</tr>
<tr>
<td>109</td>
<td>4</td>
</tr>
</tbody>
</table>
Memorizer Output from DCT Case Study

<table>
<thead>
<tr>
<th>Index</th>
<th>Size</th>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr)</td>
</tr>
<tr>
<td>131</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 16)</td>
</tr>
<tr>
<td>147</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 8)</td>
</tr>
<tr>
<td>163</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 24)</td>
</tr>
<tr>
<td>216</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 28)</td>
</tr>
<tr>
<td>228</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 20)</td>
</tr>
<tr>
<td>240</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 12)</td>
</tr>
<tr>
<td>252</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 4)</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>ADDR</td>
<td>dataptr = dataptr + 32</td>
</tr>
<tr>
<td>260</td>
<td></td>
<td>LOOP</td>
<td>ctr = ctr - 1</td>
</tr>
</tbody>
</table>

Listing D.1: Output of original jpeg_fdct_islow part 1

| Basic Block: jpeg_fdct_islow_0003 |
| Loop Depth: 0 |

<table>
<thead>
<tr>
<th>Index</th>
<th>Size</th>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>266</td>
<td>4</td>
<td>ADDR</td>
<td>dataptr = data</td>
</tr>
<tr>
<td>268</td>
<td></td>
<td>LOOP</td>
<td>ctr = 7</td>
</tr>
</tbody>
</table>

| Basic Block: jpeg_fdct_islow_0004 |
| Loop Depth: 1 |

<table>
<thead>
<tr>
<th>Index</th>
<th>Size</th>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>270</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr)</td>
</tr>
<tr>
<td>276</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 224)</td>
</tr>
<tr>
<td>281</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr)</td>
</tr>
<tr>
<td>287</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 224)</td>
</tr>
<tr>
<td>295</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 32)</td>
</tr>
<tr>
<td>301</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 192)</td>
</tr>
<tr>
<td>309</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 32)</td>
</tr>
<tr>
<td>315</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 192)</td>
</tr>
<tr>
<td>323</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 64)</td>
</tr>
<tr>
<td>329</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 160)</td>
</tr>
<tr>
<td>337</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 64)</td>
</tr>
<tr>
<td>343</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 160)</td>
</tr>
<tr>
<td>351</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 96)</td>
</tr>
<tr>
<td>357</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 128)</td>
</tr>
<tr>
<td>365</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 96)</td>
</tr>
<tr>
<td>371</td>
<td>4</td>
<td>READ</td>
<td>*(dataptr + 128)</td>
</tr>
<tr>
<td>386</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr)</td>
</tr>
<tr>
<td>393</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 128)</td>
</tr>
<tr>
<td>409</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 64)</td>
</tr>
<tr>
<td>425</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 192)</td>
</tr>
</tbody>
</table>

Conditions:

ctr >= 0
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 224)</td>
</tr>
<tr>
<td>490</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 160)</td>
</tr>
<tr>
<td>502</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 96)</td>
</tr>
<tr>
<td>514</td>
<td>4</td>
<td>WRITE</td>
<td>*(dataptr + 32)</td>
</tr>
<tr>
<td>518</td>
<td>4</td>
<td>ADDR</td>
<td>dataptr = dataptr + 4</td>
</tr>
<tr>
<td>522</td>
<td></td>
<td>LOOP</td>
<td>ctr = ctr - 1</td>
</tr>
</tbody>
</table>

Conditions:

- ctr >= 0

Listing D.2: Output of original jpeg_fdct_islow part 2
Appendix E

Memorizer Graph Output from DCT Case Study

In this Appendix the parts of the Full Dependency Graph output from Memorizer is included.

The size of the figures might make the text in the nodes a bit hard to read, but the important part is to give an example of Memorizer output for a larger function than function used in Chapter 6. Dark grey nodes are nodes identified as memory accesses and the lighter grey nodes are nodes identified as addressing calculations.
Figure E.1: Part of the Full Dependency Graph Output from the jpeg_fdct_islow() function in the DCT case study.
Figure E.2: Part of the Full Dependency Graph Output from the jpeg_fdct_islow() function in the DCT case study.
Appendix F

**motion Compensation_chroma()**

This is the source of the `motion_compensation_chroma()` function used in the x264 case study.

```c
static void motion_compensation_chroma( uint8_t *src,
    int i_src_stride, uint8_t *dst, int i_dst_stride,
    int mvx, int mvy, int i_width, int i_height )
{
    uint8_t *srcp;
    int x, y;

    const int d8x = mvx & 0x07;
    const int d8y = mvy & 0x07;

    const int cA = (8 - d8x) * (8 - d8y);
    const int cB = d8x * (8 - d8y);
    const int cC = (8 - d8x) * d8y;
    const int cD = d8x * d8y;

    src += (mvy >> 3) * i_src_stride + (mvx >> 3);
    srcp = &src[i_src_stride];

    for( y = 0; y < i_height; y++ )
    {
        for( x = 0; x < i_width; x++ )
        {
            dst[x] = ( cA * src[x] + cB * src[x+1] +
                       cC * srcp[x] + cD * srcp[x+1] + 32 ) >> 6;
        }
        dst += i_dst_stride;

        src = srcp;
        srcp += i_src_stride;
    }
}

Listing F.1: The motion_compensation_chroma() function in x264
motion_compensation_chroma()
Appendix G

Memorizer Output from x264 Case Study

The output from memorizer is a bit too large to fit nicely into the text, therefore it’s included in this appendix.

+----------------------------------------------------------+
| Basic Block : motion_compensation_chroma_0000          |
| Loop Depth:  0                                        |
|                                                        |
| index  size  type  expression                          |
+-------+------+-------+-----------------------------------+
| 3180 | 1    | ADDR  | src = (mvy >> 3) * i_src_stride + |
| 3187 | 1    | ADDR  | srcp = src + i_src_stride         |
| 3189 | | LOOP  | y = 0                               |
+-------+------+-------+-----------------------------------+

+----------------------------------------------------------+
| Basic Block : motion_compensation_chroma_0001          |
| Loop Depth:  1                                        |
|                                                        |
| index  size  type  expression                          |
+-------+------+-------+-----------------------------------+
| 3191 | | ADDR  | x = 0                               |
+----------------------------------------------------------+

| Conditions:                                             |
| +-------------------------------------------------------|
| y < i_height                                           |
+-------------------------------------------------------+

+----------------------------------------------------------+
| Basic Block : motion_compensation_chroma_0002          |
| Loop Depth:  2                                        |
|                                                        |
| index  size  type  expression                          |
+-------+------+-------+-----------------------------------+
| 3205 | 1    | READ  | *(src + x)                        |
+----------------------------------------------------------+
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3224</td>
<td>1</td>
<td>READ</td>
<td>*(src + x + 1)</td>
</tr>
<tr>
<td>3242</td>
<td>1</td>
<td>READ</td>
<td>*(srcp + x + 1)</td>
</tr>
<tr>
<td>3257</td>
<td>1</td>
<td>READ</td>
<td>*(srcp + x)</td>
</tr>
<tr>
<td>3276</td>
<td>1</td>
<td>WRITE</td>
<td>*(dst + x)</td>
</tr>
<tr>
<td>3280</td>
<td></td>
<td>ADDR</td>
<td>x = x + 1</td>
</tr>
</tbody>
</table>

Conditions:
- x < i_width

Listing G.1: Memorizer output of the function motion_compensation_chroma()
List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Tree representation of $a=b+c$</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>C code of $a=b+c$</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>C code of $b+c*d$</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Intermediate Representation of $b+c*d$</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>Object Code version of $b+c*d$</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>C MaCT: Absolute array MaP, 4 reads</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>C MaCT: Relative array MaP, 4 reads</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>C MaCT: Differential array MaP, 4 reads</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>C MaCT Example: Arbitrary stride with array MaP, 4 reads</td>
<td>31</td>
</tr>
<tr>
<td>5.5</td>
<td>C MaCT: functional relative MaP, 4 reads</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td>C MaCT: functional differential MaP, 4 reads</td>
<td>32</td>
</tr>
<tr>
<td>5.7</td>
<td>C MaCT Example: Arbitrary stride with relative functional MaP, 4 reads</td>
<td>32</td>
</tr>
<tr>
<td>5.8</td>
<td>C MaCT Example: Arbitrary stride with differential functional MaP, 4 reads</td>
<td>33</td>
</tr>
<tr>
<td>5.9</td>
<td>C MaCT: Relative 2D matrix MaP, 4x4 reads</td>
<td>34</td>
</tr>
<tr>
<td>5.10</td>
<td>C MaCT: Multidimensional functional relative MaP, 4x4 writes</td>
<td>35</td>
</tr>
<tr>
<td>5.11</td>
<td>C MaCT: Multidimensional functional differential MaP, 4x4 writes</td>
<td>36</td>
</tr>
<tr>
<td>5.12</td>
<td>C MaCT: 2D arbitrary stride with differential functional MaP, 4x4 writes</td>
<td>36</td>
</tr>
<tr>
<td>5.13</td>
<td>C MaCT Example: Burst access, 4 reads</td>
<td>37</td>
</tr>
<tr>
<td>5.14</td>
<td>C MaCT Example: Radix-4 access level 2, 4 reads</td>
<td>38</td>
</tr>
<tr>
<td>5.15</td>
<td>C MaCT Example: Column access 12 sample row, 4 reads</td>
<td>38</td>
</tr>
<tr>
<td>5.16</td>
<td>C MaCT Example: Diagonol access 12 sample row, 4 reads</td>
<td>38</td>
</tr>
<tr>
<td>5.17</td>
<td>C MaCT Example: Block burst (square access) 12 sample row, 4x4 writes</td>
<td>39</td>
</tr>
<tr>
<td>5.18</td>
<td>C MaCT Example: Crumbled square stride 2 access 12 sample row, 4x4 writes</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of Memorizer input</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>Memory Access Table output from Memorizer</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>Code valid in C but not in C++</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>An example of <code>memorizer.conf</code></td>
<td>54</td>
</tr>
<tr>
<td>8.1</td>
<td>Dependencies that Memorizer fails to expose</td>
<td>78</td>
</tr>
<tr>
<td>A.1</td>
<td>Memory Access XML output from Memorizer</td>
<td>81</td>
</tr>
<tr>
<td>B.1</td>
<td>Patch to allow using GEM and G++ together</td>
<td>83</td>
</tr>
<tr>
<td>C.1</td>
<td>Original <code>jpeg_fdct_islow()</code></td>
<td>87</td>
</tr>
<tr>
<td>D.1</td>
<td>Output of original <code>jpeg_fdct_islow</code> part 1</td>
<td>91</td>
</tr>
<tr>
<td>D.2</td>
<td>Output of original <code>jpeg_fdct_islow</code> part 2</td>
<td>92</td>
</tr>
<tr>
<td>F.1</td>
<td>The <code>motion_compensation_chroma()</code> function in x264</td>
<td>99</td>
</tr>
</tbody>
</table>
G.1 Memorizer output of the function motion_compensation_chroma() 101
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relative costs of memory access, from [7]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>A 128b wide 8 way PVM, RF and DP, based on [7]</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Advantage of using PVM, from [7]</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>P3RMA Design Flow</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>General PMA concept, based on [5]</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Scanning field example, $v(r)/r$, 4x4 monochrome picture</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>S/a storage example, monochrome 4x4 image</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>PMA module content example, monochrome 4x4 image</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>PMA read access example, monochrome 4x4 image</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>A Compiler</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Front End and Back End</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>A three tiered compiler</td>
<td>18</td>
</tr>
<tr>
<td>4.4</td>
<td>Dominator Example</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of a nonreducible flow graph</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Visualization of the GIMPLE tree</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>Visualization of the DFG-Nodes</td>
<td>45</td>
</tr>
<tr>
<td>6.3</td>
<td>Visualization of the Control Flow Graph</td>
<td>47</td>
</tr>
<tr>
<td>6.4</td>
<td>Full Dependency Graph output from Memorizer</td>
<td>47</td>
</tr>
<tr>
<td>6.5</td>
<td>Addressing Dependency Graph output from Memorizer</td>
<td>48</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of expand_variables_in_expressions</td>
<td>55</td>
</tr>
<tr>
<td>6.7</td>
<td>Memorizer workflow</td>
<td>56</td>
</tr>
<tr>
<td>7.1</td>
<td>Proposed PVM system, from [7]</td>
<td>59</td>
</tr>
<tr>
<td>7.2</td>
<td>Workflow in P3RMA analysis, based on [7]</td>
<td>61</td>
</tr>
<tr>
<td>7.3</td>
<td>From exposed memory accesses to MaE for the first block of reads</td>
<td>65</td>
</tr>
<tr>
<td>7.4</td>
<td>MaE for the first block of writes</td>
<td>66</td>
</tr>
<tr>
<td>7.5</td>
<td>From exposed memory accesses to MaE for the second block of reads</td>
<td>67</td>
</tr>
<tr>
<td>7.6</td>
<td>First draft of a DMA Table for reading the data</td>
<td>68</td>
</tr>
<tr>
<td>7.7</td>
<td>DMA Table for reading the data</td>
<td>68</td>
</tr>
<tr>
<td>7.8</td>
<td>$x264$ reading pattern</td>
<td>69</td>
</tr>
<tr>
<td>7.9</td>
<td>MaE of the memory reads in the chroma motion compensation</td>
<td>70</td>
</tr>
<tr>
<td>7.10</td>
<td>MaE of the memory reads in the chroma motion compensation</td>
<td>70</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.11</td>
<td>MaE of the memory writes in the chroma motion compensation without numerical values</td>
<td>71</td>
</tr>
<tr>
<td>7.12</td>
<td>DMA Table to fetch the data needed for the motion compensation</td>
<td>72</td>
</tr>
<tr>
<td>7.13</td>
<td>Partitioning of the data in the PVM</td>
<td>73</td>
</tr>
<tr>
<td>E.1</td>
<td>Part of the Full Dependency Graph Output from the jpeg_fdct_islow() function in the DCT case study</td>
<td>96</td>
</tr>
<tr>
<td>E.2</td>
<td>Part of the Full Dependency Graph Output from the jpeg_fdct_islow() function in the DCT case study</td>
<td>97</td>
</tr>
</tbody>
</table>
Bibliography


Index

Abbreviations, xi
Abstract, viii
Abstract Syntax Tree, 19
Access format, 13
Acknowledgements, ix
Address function, 12
Addressing Dependency Graph, 48
Array MaP, 29
back end, 18
Basic Block, 21
BasicBlock, 50
Cache, 7
compiler, 17
Conflict free access, 15
Control Flow Graph, 21
Data Dependency Graph, 45
DDG Nodes, 45
DDGNode, 51
DMA linking table, 60
Dominator, 21
Dynamic Analysis, 17
Exposable (def.), 28
front end, 18
Full Dependency Graph, 47
Function, 50
Functional MaP, 31
G++ Patch, 83
GEM, 42
General PMA model, 11
GENERIC, 19
GIMPLE, 19
hook, 42
Intermediate Representation, 18
jpeg_fdct_islow, 87
Linear module assignment function, 15
MaP dimension, 53
MaP Representations, 27
Absolute array MaP, 29
Absolute functional MaP, 31
Differential array MaP, 30
Differential functional MaP, 32
Multidimensional functional MaP, 35
Multidimensional matrix MaP, 34
Relative array MaP, 30
Relative functional MaP, 31
MaP size (def.), 33
Memorizer, 11 61
memorizer.conf, 54
[analyze], 54
[no_analyze], 54
[options], 55
expand_variables_in_expressions, 55
no_export_addressing_graph, 55
no_export_full_graph, 55
optimize_tree, 55
print_basic_blocks, 55
print_memory_access_tables, 55
print_memory_access_xml, 55
Memory access Code Template, MaCT, 28
Memory access Exposition, MaE, 28
Memory access Pattern, MaP, 27
Memory Access Tables, 48
Memory Access XML, 48
middle end, 18
Module assignment function, 12
Multidimensional MaP (def.), 33
Nomenclature, xi

109
P3RMA, 5
Parallel Memory Architecture, PMA, 11
Parallel Vector Memories, PVM, 6, 59
Patternable (def.), 28
Permutation, 14
Permutation table, 60
Raster memory, 16
Relief, 41, 42
robj, 51
rptr, 51
Sample, 12
Scanning field, 12
Static Code Analysis, 17
Straight-line scanning field, 12
Three-address Code, 18
Trace, 51
Ultra Large Register File, ULRF, 7
Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances. The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skyld mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

© 2007, Björn Lundgren, Anders Ödlund