Code profiling as a design tool for application specific instruction sets

Master’s thesis
performed in Computer Engineering
by
Björn Skoglund

Reg nr: LiTH-ISY-EX -- 07/3987 -- SE

February 27, 2007
Code profiling as a design tool for application specific instruction sets

Master’s thesis

performed in Computer Engineering,
Dept. of Electrical Engineering
at Linköpings universitet

by Björn Skoglund

Reg nr: LiTH-ISY-EX -- 07/3987 -- SE

Supervisor: Professor Dake Liu
Department of electrical engineering, Computer architecture

Examiner: Professor Dake Liu
Linköpings Universitet

Linköping, February 27, 2007
Sammanfattning
As the embedded devices has become more and more generalized and as their
product cycles keeps shrinking the field has opened up for the Application
Specific Instruction set Processor. A mix between the classic generalized
microcontroller and the specialized ASIC the ASIP keeps a set of general
processing instructions for executing embedded software but combines that
with a set of heavily specialized instructions for speeding up the data intense
application core algorithms. One important aspect of the ASIP design flow
research is cutting design time and cost. One way of that is automation of
the instruction set design. In order to do so a process is needed where the
algorithm to be ASIPed is analyzed and critical operations are found and
exposed so that they can be implemented in special hardware. This process
is called profiling. This thesis describes an implementation of a fine grained
source code profiler for use in an ASIP design flow. The profiler software is
based on a static-dynamic workflow where data is assembled from both static
analysis and dynamic execution of the program and then analyzed together in
an specially made analysis software.

Nyckelord
ASIP, static-dynamic, profiling, instrumentation, co-design
Abstract

As the embedded devices has become more and more generalized and as their product cycles keeps shrinking the field has opened up for the Application Specific Instruction set Processor. A mix between the classic generalized microcontroller and the specialized ASIC the ASIP keeps a set of general processing instructions for executing embedded software but combines that with a set of heavily specialized instructions for speeding up the data intense application core algorithms. One important aspect of the ASIP design flow research is cutting design time and cost. One way of that is automation of the instruction set design. In order to do so a process is needed where the algorithm to be ASIPed is analyzed and critical operations are found and exposed so that they can be implemented in special hardware. This process is called profiling. This thesis describes an implementation of a fine grained source code profiler for use in an ASIP design flow. The profiler software is based on a static-dynamic workflow where data is assembled from both static analysis and dynamic execution of the program and then analyzed together in an specially made analysis software.

Keywords: ASIP, static-dynamic, profiling, instrumentation, co-design
Acknowledgements

I would like to especially thank Dake Liu for endless support in my work and for always keeping a positive spirit. One could not wish for a better supervisor. Many thanks also to Alexey Smirnov for support on GEM and for answering many questions concerning the inner workings of GCC. Finally I would like to extend my gratitude to Christopher Clark who selflessly published his hash table implementation for public use.
Abbreviations and explanations

ADL Architecture Description Language
AIS Application Instruction Set
API Application Programming Interface
ASIC Application Specific Integrated Circuit
ASIP Application Specific Instruction set Processor
Basic Block A piece of code with no branch instructions
C The C programming language, a platform independent systems programming language.
CFG Control Flow Graph
DCT Discrete Cosine Transform
DSE Design Space Exploration
DSP Digital Signal Processing
EDA Electronic Design Automation
GCC The GNU Compiler Collection (Sometimes GNU C Compiler)
GEM GCC Extension Modules, a code plugin system for the GNU Compiler Collection
Gprof The GNU Profiler
GUI Graphical User Interface
IR Intermediate Representation
IS Instruction Set
JPEG Joint Photographic Experts Group
MAC Multiply ACcumulate
Relief Profiler The tool developed in the scope of this thesis
RGB Red, Green, Blue color space
RLE Run Length Encoding
RTL Register Transfer Level, a low level abstraction in digital systems design.
SSA  Single Static Assignment
Qt  Platform independent GUI API
UML  Unified Modeling Language
WCET  Worst Case Execution Time
XML  eXtensible Markup Language - A text based human readable tree representation
YUV  Luminisence, Chrominence color space
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td>Abbreviations and explanations</td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Preamble</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Method Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Assumed Prior Knowledge</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Thesis Overview</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Motivation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 ASIP design flow</td>
<td>5</td>
</tr>
<tr>
<td>2.3 The role of the profiler</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Related work</td>
<td>8</td>
</tr>
<tr>
<td>2.4.1 Coarse grained software profiling</td>
<td>8</td>
</tr>
<tr>
<td>2.4.2 Fine grained ASIP profiling</td>
<td>8</td>
</tr>
<tr>
<td>2.4.3 Related fields</td>
<td>8</td>
</tr>
<tr>
<td><strong>3 Profiling theory</strong></td>
<td>11</td>
</tr>
<tr>
<td>3.1 Profiling in general</td>
<td>11</td>
</tr>
<tr>
<td>3.2 The static profiler</td>
<td>11</td>
</tr>
<tr>
<td>3.2.1 Static profiling on different abstraction levels</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Code instrumentation</td>
<td>13</td>
</tr>
<tr>
<td>3.3 The dynamic profiler</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 Some examples</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Hardware dependencies</td>
<td>15</td>
</tr>
<tr>
<td>3.5 Profiling for instruction set design</td>
<td>16</td>
</tr>
<tr>
<td>3.6 Introduction to compiler theory</td>
<td>18</td>
</tr>
<tr>
<td>3.6.1 The front end</td>
<td>18</td>
</tr>
</tbody>
</table>
5.3.3 Basic block entry .......................... 48
5.3.4 Basic block exit .......................... 48
5.3.5 Program de-initialization ................. 49
5.3.6 Memory efficiency in host machine ....... 50
5.3.7 Relevant source files ...................... 50

5.4 Post Execution Analysis .................. 50
5.4.1 Introduction ............................ 50
5.4.2 Implementation details ................... 51
5.4.3 Software architecture .................... 51

5.5 Discussion ................................ 52

6 Results and conclusions .................... 54
6.1 Field test: libJPEG ....................... 54
6.1.1 Overview of the JPEG algorithm ........... 54
6.1.2 The libJPEG software package .......... 55
6.1.3 Test purpose ............................ 55
6.1.4 Test preparations ......................... 56
6.1.5 Running the test cases .................... 57
6.1.6 Post execution analysis .................. 57

6.2 Profiling results ......................... 57
6.3 Conclusions .............................. 57
6.3.1 Project results .......................... 57
6.3.2 Problems with the software ............... 58
6.3.3 Future work ............................. 60
6.3.4 Final words ............................. 60

Index ........................................ 61

List of figures ............................... 62
List of tables ................................. 62
Listings ...................................... 63
References ................................... 65

A Optimizing suggestions from 18 different encoding runs 68
Chapter 1

Introduction

The following chapter introduces the field and motivates the work that has been performed and formalizes the goals of the project.

1.1 Preamble

In the world of embedded computing such as in digital cameras, mobile phones and portable music and video players the demands on performance is skyrocketing. But that is not enough, in order for the devices to be usable they need to conserve power if the batteries are to last longer than an hour. In order to accommodate both these requirements designers use custom hardware. For a long time most designs were based on the Application Specific Integrated Circuit (ASIC). A highly customized chip performing one thing and one thing only with very high performance and low power consumption.

As the embedded devices has become more and more generalized and as their product cycles keeps shrinking the ASIC has become harder and harder to keep updated on constrained budgets. The field has instead opened up for the Application Specific Instruction set Processor (ASIP). A mix between the classic generalized microcontroller and the specialized ASIC the ASIP keeps a set of general processing instructions for executing embedded software but combines that with a set of heavily specialized instructions for speeding up the data intense application core algorithms.

This thesis deals with the first part of the ASIP design flow. In this part the desired algorithm is taken apart and analyzed in order to find which parts of it is suitable to be implemented as custom instructions or accelerators. The end product of this work is an instruction set (IS) and it is therefor named Instruction Set Design. Performing the instruction set design by hand is painstaking and time consuming work. Therefore, any tool that can simplify and or automate this process is a welcome addition to the designers that face these problems every day.
1.2 Objectives

The project described in this thesis is a part of the ongoing system design automation research at the Computer Engineering Group at the Department of Electrical Engineering at Linköping University.

ASIP design has been the focus of the division since August 2001. Different from general purpose microprocessors, ASIPs give much higher performance, much lower power, and much lower silicon costs because of the optimization of the application specific instruction set and the corresponding micro architecture. One important aspect of the ASIP design flow research is cutting design time and cost and that is where this thesis comes in.

One way of reducing the design time is automation of the instruction set design. In order to do so a process is needed where the algorithm to be ASIPed is analyzed and critical operations are found and exposed so that they can be implemented in special hardware. This process is called profiling.

The objective of this project has been twofold:

- To evaluate the research field of profiling with a focus on profiling as a tool in hardware design automation. What research has there been, what results has been obtained and what is the next step?

- To develop and test a new form of source code profiler. The purpose of this profiler is to analyze a program from the perspective of a ASIP designer and expose potential application specific instructions that will generate the highest increase in execution speed of the profiled application if implemented.

Since the specific field of research we were looking for turned out to be very badly covered the main focus of the project shifted towards the development of a new piece of research profiling software. The intention is that this new profiler will take a fresh approach to the field by combining input from both static code analysis and dynamic execution data from running the algorithm. This we hope will give a new a more complete view of the profiled algorithm.

The rather vague wish that the profiler be aimed at ASIP design is embodied in the more concrete requirement that the profiling granularity be on a basic block level so that potential special purpose instructions be exposed.

In order to facilitate project evaluation, the following requirements was set up for the profiler.

- The profiler shall be able to analyze both static code and dynamic execution data and print the relevant results in a readable fashion.

- The profiler shall have a sub function granularity so that all program control flow is exposed.
1.3 Method Overview

The project has been performed in two separate steps. Firstly the literature study was performed by searching relevant sources such as the IEEE website IEEExplore.com and the ACM counterpart as well as the Google Scholar academia web search for relevant articles concerning software profiling.

The second step was implementing the software profiler. This was done by evaluating the different models of profiling that were available and were based on open source. Writing the entire system from scratch was not a viable option if the objectives was to be met so searching for open source software that could do complex tasks such as code parsing was the natural start.

When the decision had fell on the GCC and the GEM module system the development was fairly straight forward. The libJPEG software package used for testing has been continously used for instrumentation testing for as long as the code base has been mature enough to handle it. Before that only very small and basic programs was used.

1.4 Assumed Prior Knowledge

This report should be readable by any graduate of a computer science or computer engineering program on a university level. The reader is assumed to be familiar with the C language and programming concepts in general as well as basic data structures such as stacks and hash tables. A familiarity with basic compiler architecture is beneficial but is not required. Instead there is a quick recap of the different parts of a compiler in section 3.6. Furthermore the reader is assumed to have basic knowledge in the field of computer architecture.

1.5 Thesis Overview

The listing below gives a brief presentation of the remaining chapters of this report.

Chapter 2 contains the introduction and background to the field of software profiling. Profiling as a concept is introduced and then divided into static and dynamic counterparts. Profiling in software development is also contrasted against profiling as a tool for ASIP design.

Chapter 3 is a detailed technical introduction into the world of software profiling. Here the background information is connected with the intentions of this project.

Chapter 4 is a detailed walkthrough of the developed software which has been named Relief from the point of the user. The profiler is covered
from installation, through building of the software to be profiled and finally analysis of the results. It may seem unusual to have a user’s manual before the technical specification of the internals but it serves as a good introduction to the tool and the reasoning in the implementation chapter will be easier to follow if the end result is known in advance.

Chapter 5 gives an iterative description of the Relief profiler implementation. First, the fundamental division into an instrumentation- and profiler-library is introduced. Then in depth specification of each library and the following analysis software is presented.

Section 5.2 explains the ins and outs of the code instrumentation library. Basic block instrumentation is explained in detail. The static profiling output as XML-files containing statistics and graph visualization are also detailed.

Section 5.3 walks the user through the operation of the profiler library. Program as well as function and basic block entry and exit is described in detail.

Finally 5.4 explains the software used to combine the outputs of the static and dynamic profiling into usable information on unveiling where the true potential for application specific instruction.

Chapter 6 details the results of the profiler when used to profile the free libJPEG implementation of the standardized JPEG program.

It also finishes off the report by discussing the results of the project, what has been good and bad and finally depicts some possible future work.
Chapter 2

Background

This chapter paints the big picture around the very specific field that is application profiling as it appears in this thesis.

2.1 Motivation

As the research direction in the field of computer architecture shifts its weight from performance to performance per watt the number of custom designs in products increases. The rapid pace of electronics development still remains though and thus the pressure on designers to develop stable yet highly customized designs increases.

As a consequence, for the last couple of years, design automation has arisen as a new field of research. High level alternatives to full custom designs such as core driven and functional cell design enables computer powered implementations while minimizing “hands on” engineer involvement [16].

The most powerful trend is in the field of ASIPs or Application Specific Instruction set Processors. These are, in essence, programmable processors with extra custom designed instructions designed to run specific applications such as baseband or mp3 coding algorithms [16].

2.2 ASIP design flow

Figure 2.1 demonstrates an overview of the process of designing an ASIP. The very first step transferring the informal specification such as “We need an image compression chip for our new digital camera phone”. The performance requirements of the chip is acquired from market and technical analysis: “The next generation camera phones will have to take 5 megapixel images so the chip will need to compress such an image in time so that it does not bother the consumer as slow”. After that the application is implemented in a specifica-
Chapter 2. Background

Figure 2.1: A rough overview of the ASIP design flow.
tion language which can be a system independent Architecture Specification Language (ADL) or machine executable C code or an anything inbetween.

From this specification a co-design analysis is performed in which the decision must be taken which parts of the system is to be executed in software and which parts should be implemented as hardware modules.

This co-design step can be implemented in several different ways depending on a multitude of previous decsions like what kind of specification language has been used and the nature of the application to be designed. In the case of using C as the specification language a profiler software is very suitable as means of performing this sorting into hardware and software.

2.3 The role of the profiler

A software profiler may be defined as a piece of software who’s purpose is to give information on a program or a piece of code. This is illustrated in figure 2.2. The definition is intentionally weak on what kind of information the profiler must supply and in what way it obtains this information. As we shall see in chapter 3 there are two distinct ways in which modern profilers work. However, room should be left for new ways to interpret programs.

The classical use of a software profiler is as a software development tool. The developer has a piece of software with insufficient performance. The so called 90-10 locality rule depicts that 10% of the code in aforementioned software will be in execution during 90% of the run-time while the other 90% of the code only deals with exceptions, error handling and rare events in general. Naturally the developer’s only interest is in the first 10% because that is where the difference will be made. The profiler is the software that will tell the developer which of the millions of lines of code are the ones to focus on when looking for optimizations.

As already noted, another way of utilizing profiling technology is in the design process of application specific instruction set processors. The main question is still the same: Which parts of the software are executed the most? However, circumstances differ. The ASIP designer is usually not the same engineer as the software designer. Instead the ASIP designer will receive an already optimized piece of code from the software developer. Neither does the
ASIP designer necessarily have deep insights into ASIP or DSP algorithms in general. This means that the profiling software must be as precise as it possibly can in the information it delivers to its user in order to minimize the amount of code the user will have to plow through in order to find the desired code.

2.4 Related work

2.4.1 Coarse grained software profiling

In the profiling field the first thing to consider is existing profilers and code instrumentation tools. The most prominent professional profiler in the free software world is the GNU Profiler or Gprof [11] which works in conjunction with the GCC compiler. The Microsoft Visual C++ compiler and the Sun Java framework has support for automatic code instrumentation [17][19]. All these alternatives enable profiling only on a function level which renders them subpar in the comparison.

2.4.2 Fine grained ASIP profiling

Gschwind presents the argument that the co-design nomenclature can be applied to ASIP design [13]. Instead of partitioning the system into a set of co-processors and interconnects as Wolf does in [26] he suggests starting off with a simple general purpose microprocessor and adding application critical special instructions with the goal of reaching a given metric.

Another alternative is the µProfiler developed by Faruque et. al. at Aachen University of Technology [2]. The µProfiler instruments on a very low intermediate representation (IR) level which gives it very high control of the workings of the profiling at a performance penalty. Furthermore the µProfiler does not fully combine static analysis with the dynamic and instead focuses on pattern matching with patterns supposedly supplied by the designer.

Ravasi and Mattavelli has a tool with quite similar goals to the one developed in this project but they too has very little feedback between the static analysis and the dynamic execution data from what conclusions can be drawn from the papers[21][22].

2.4.3 Related fields

Electronic Design Automation (EDA) and Design Space Exploration (DSE) are hot terms. They represent a wide field of different tools and techniques, all with the aim of increasing productivity of electronic systems engineers. Being so wide, the terms also fall on their own strength since they are as relevant to the matters being discussed in this thesis as they are to for example register
transfer level (RTL) or gate level design optimization in a system integration article.

Still, all these matters are more or less related and the work done here must be understood in a context of work done by others. Since the code profiler is the first step in the ASIP design flow what is most relevant when reading this thesis is work done on this very matter but also its relation to the work done in the following steps of the design flow.

**Hardware/Software co-design**

HW/SW co-design is a collective label for a set of tools and methods used to split a functional specification of an embedded system into a hardware/software architecture [27]. The term is generally used as very closely related to EDA but is not as wide and only refers to the early design stages before the implementation and testing stages have begun.

The task complexity of systems co-design is exponential and lies in deciding which parts of the specification are suitable for hardware implementation and which are to be implemented in software. Often there are also several different subdecisions to be made like on which processor a software function should be run in multi processor systems and in which way a hardware function should be implemented [26].

There have been multiple heuristics developed in order to get around this problem [6, 26] which have proved fast enough and still constantly producing near optimal solutions to design problems given the circumstances.

Gries argues that the importance of automated design space exploration is not only in design time but also in design quality. When the decision is experience based, decisions that worked out in the last design will get priority over considering new approaches if there is not quantitative data to support a new approach [12].

**Worst case execution time analysis**

Worst case execution time-analysis or WCET-analysis is a static code analysis method in which operation counting and loop boundary calculations are used to find an upper limit for program execution time [10]. This is not a tool for optimizing performance but rather one to guarantee the correct behavior of real time systems.

**Model based performance estimation**

Simonetta Balsamo reviews the concept of model based performance estimation. The idea in this field is that performance issues should be accounted for although the design process of software development. Therefore high level software modeling tools such as stochastic Petri nets or queueing networks is used to model the software in the very early stages of development such as
during requirements specification process. This model is then formally analyzed in order to retrieve a rough performance estimation before the project moves into the actual design and implementation stages where the cost of architectural and requirement changes become very large\[4\].

Now this may seem like a far step away from the performance estimation methods that we present in this project but it actually is not. They both differ from classical performance tuning software profiling in the sense that they are performed in the early stages of design, in order to pre-implementation test the feasibility of the requirements against the resources.

**Hardware specific profiling**

Many firmware development kits come with a profiling tool specific for that particular chip. These profilers however are only usable as development tools for that particular system as they require that the architecture is fixed and should therefore be placed in the same group as the other software development profilers such as Gprof. Two examples of hardware specific profilers are ATOM \[24\] and Spix\[7\].
Chapter 3

Profiling theory

The following chapter will cover the majority part of the profiling field in rough detail. Some parts will be covered on a more detailed level based on relevance for the work in coming chapters.

3.1 Profiling in general

Profiling was defined in very broad terms in section 2.3. Now it is time to go into more detail. In this thesis, the term profiling implies a technique for estimating run time or memory cost for executing application source code. A program for analyzing programs if you will.

The result of the profiling is quantitative information on the program. Since the uses are so many, the information can be of many different kinds such as execution time, subroutine call statistics, operations used and longest execution path taken. The main use of profiling is exposing performance characteristics of the program. In what way does the program achieve its results?

The way in which the profiler attains this information also varies and we will be studying them in the coming sections.

3.2 The static profiler

The static code profiler works by analyzing a representation of the program code as it is and without running it. The non runtime environment gives the possibility to go into great detail in the analysis but also places restrictions on it. Non deterministic properties such as recursion, dynamic data structures and non bound loops, of the program can not be estimated without given data on the input which in turn introduces dynamic and possibly insecure profiling.
Static program analysis is a big field of research but does not have its focus on code profiling. Instead it is used primarily for analysis of security properties of network applications and for verification of hard realtime applications such as [10].

### 3.2.1 Static profiling on different abstraction levels

It was mentioned earlier that the static profiler studies a representation of the program. It is important to recognize that the program may exist in several different representations during its existence. First there is the source code representation in which the developer writes it. Then there is the binary representation that is executed. As we will see there is also a third representation inside the compiler when the compiler is in the process of translating the source code into binary executable code known as Intermediate Representation (IR). The result of the profiling varies depending on which of these representations are studied as information may be missing in some stages. While profiling on binary code level might give the most detailed information and the highest accuracy on exactly which instructions are being executed it is also quite a narrow approach. Much of the structural information on the program has been lost in the compiler which is a loss. In that case profiling the source code is a much better idea. No, even low level programming languages such as C will hide several kinds of memory accesses and operations, resulting in a lower profiler accuracy.

### Limitations of static analysis tools

Another important notice is that the profiling accuracy depends, not only on what is measured i.e. the instrumentation but also on the implementation language of the software we are trying to profile. A high level dynamic language such as Matlab or the modern web languages Python and Ruby very heavily disguises memory access and complex operations in code. For example a simple minus character in Ruby code may actually be interpreted as a set operation if the operands happen to contain arrays during execution. Something which the profiler may not be able to determine during static analysis. Figure 3.1 shows the relation between code efficiency(abstraction level) and profiling accuracy for some common code representations.

There are fewer occasions where problems like these may occur in statically typed and compiled languages such as C but one sould be aware that it does happen. Such an occasion is the function pointer construct in which a void pointer variable can be pointed at the symbol of a function and then called. In this case, a call is unless the parser is very advanced, not recognized as a call to this function. Consider the following example in listing 3.1 in which syntactically, no call is ever made to sqr.
3.2. The static profiler

![Figure 3.1: Profiling accuracy depending on language.](image)

```c
int sqr(int x) {
    return x*x;
}
int main() {
    int (*p)(int);
p = sqr;
p(3);
}
```

Listing 3.1: Function pointers may be deceptive when syntactically parsed.

3.2.2 Code instrumentation

Aside from analyzing the code, static analysis might be equipped with altering the program it is analyzing. In this case code to measure and alter the dynamic behaviour of the program by inserting new code. This processing is called code instrumentation because of its focus on probing and measuring code. Altering the behaviour of the code in any way is not to the authors knowledge a very usable feature of code instrumentation. In the same manner as the analysis the code instrumentation can be performed on several different levels of abstraction:

- **Source code instrumentation** Involves inserting code directly into the source code files before they are compiled.

- **Intermediate level instrumentation** Inserting code into some form of compiler internal representation of the code. Can be instrumentation of the abstract syntax tree or three operand assembly code. The Relief profiler depicted in this thesis as well as the GNU Gprof profiler both use this method albeit in very different ways.
**Executable instrumentation** Instrumentation of the compiled binary.

**Run time instrumentation** Similar to the executable instrumentation but is performed during run time. One example of run time instrumentation is the profiling framework Valgrind [23] for the linux platform. Run time instrumentation drastically severes the execution performance of the program but it also unifies the instrumentation and profiling activities not possible in other forms.

### 3.3 The dynamic profiler

Dynamic analysis naturally is the opposite of static ditto. Instead of analyzing the program code, the profiler executes it. During execution the profiler logs which code is executed and whatever properties of the execution is deemed interesting.

The dynamic profiler cannot give the engineer as profound information on the code as the static profiler can but in return it can in detail report what happens during execution of the program for a well defined set of inputs. As an example, the dynamic profiler can measure such things as average execution time of a function or branch execution statistics for the conditional jumps in the code.

As there are algorithms whose performance is highly dependent on input the results produced by the dynamic profiler need to be read with great care. However, it is also important to bear in mind that software profiling for the most part has taken the dynamic road still being very useful.

#### 3.3.1 Some examples

There is a multitude of profilers on the market. Each with its own modus operandi. Here are two often used open source examples:

**The GNU Profiler**

GNU profiler or Gprof for short is the GNU projects standard profiler. It is invoked by compiling the software with the \(-g\) flag when compiling with GCC. The program will then output the value of the CPU program counter to file with static intervals invoked by interrupts. When program execution has ended the `gprof` program can analyze the data against a mapping between program counter values and function memory locations. The number of times the program counter happens to be in each function is counted and statistics is created for where most time has been spent is created and then presented to the user [11].
Valgrind

Valgrind is an open source profiling framework consisting of a virtual processor and a plugin system. The extra abstraction layer that the software processor constitutes serves as a looking glass into the execution of the program and a set of functions for surveillance and data collection makes the Valgrind a great tool for building many different kinds of software profiling systems [23].

The Valgrind distribution comes with several different kinds of tools:

**Memcheck** A memory checking utility. It checks all reads and writes of memory and calls to malloc/new/free/delete are intercepted. As a result, Memcheck can detect the following problems:

- Use of uninitialised memory
- Reading/writing memory after it has been free’d
- Reading/writing off the end of malloc’d blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks - where pointers to malloc’d blocks are lost forever
- Mismatched use of allocation and freeing of memory.

**Cachegrind** A cache profiling tool. Cachegrind uses the processor emulation of Valgrind to run the executable, and catches all memory accesses, which are used to drive a cache simulator. The program does not need to be recompiled, it can use shared libraries and plugins, and the profile measurement doesn’t influence the memory access behaviour. The trace includes the number of instruction/data memory accesses and 1st/2nd level cache misses, and relates it to source lines and functions of the run program. A disadvantage is the slowdown involved in the processor emulation, around 50 times slower.

**Callgrind** Callgrind is a Valgrind tool for profiling programs. The collected data consists of the number of instructions executed on a run, their relationship to source lines, and call relationship among functions together with call counts. Optionally, a cache simulator (similar to Cachegrind) can produce further information about the memory access behavior of the application.

### 3.4 Hardware dependencies

A profiler may take into account the target architecture. In that case the result of the profiling is the resources demanded in terms of computing elements as well as run time of the hardware for running this particular program. This differs from the hardware independent profiler which has no previous information on the hardware the code will be executed on and therefore can only
respond with the kind and number of operations as well as the amount of memory needed to finish the task. The number of clock cycles and run time depends on the unknown hardware and can therefore not be predicted.

Please note that as the topic of this report is the design of hardware we by definition do not know what the execution platform will be and so all profiling will be of the hardware independent kind.

3.5 Profiling for instruction set design

Using a source code profiler for hardware design use places somewhat different demands on the profiler than from software development. As mentioned in the introduction in section 2.3 the user of the profiler will likely benefit from any increase in the accuracy of the information given by the profiler.

First of all the granularity is increased from functions to basic blocks. A basic block is a series of instructions which has only one entry at the beginning and one exit at the end. Therefor, if one instruction in a particular block is executed once, every other instruction in the block is executed once, and only once. Increasing the granularity to a basic block level thereby means that instead of taking note of every function entry, this will be done at every basic block entry instead.

The above definition of a basic block naturally means that any function that has any form of flow control such as loops of if-statments contains more than one basic block. This is why we say that the granularity increases.

The reason for this increase in granularity is that when profiling on function level functions which contains several loops must be analyzed in detail by hand in order to find which inner loops may be optimized or implemented in hardware. Such a profiling result can of course still be of use but for a hardware designer the granularity needs be on a basic block level in order to attain the details of the software with a minimal hands on effort.

Another field of interest for the hardware designer not present in classic profiler tools is argument sizes in function calls. Here we are not talking about the bit width of the type used but of the actual size of the values it contains. This matters in that the implementation bit width of the application specific instructions selected and thereby the chip area needed for the implementation varies with it. If the implementation width is cut by half the chip area needed will go down by 75%.

But the most important feature of ASIP design tool would be a 90-10 exposure feature. The idea is that the part of the program most suitable for optimization is not necessarily the part executed the most times (although that is quite often the case) but rather the part that has the highest number of operations removable. Let’s say that every basic block in the program in one way or another can be implemented in hardware to take exactly one clock cycle, possibly through pipelining.

If \( n \) is the number of executions of the block and \( O_p \) is the number of clock
cycles that the basic block will take for one execution on a purely general processor. In that case

\[ C = n \cdot O_p - n \]

is the number of removable clock cycles if the block was to be implemented in hardware. The implication of this is that blocks containing very few operations will not be as attractive for hardware implementation. In many cases software loops may be heavily parallelized in conjunction with a hardware implementation so this theory should not be considered law but larger blocks generally has more optimization potential then smaller ones.

Now, let’s say we have found the most suitable code for execution. What more can be done? Well, first of all chip area is expensive we would like to minimize the overlap between the different hypothetical special instructions. In other words, if the same sequence of code might show up in two different parts of the program.

Let us look at an example. A common complex instruction used in ASIPs is the Multiply Accumulate (MAC) instruction used in convolution functions common in signaling applications.

Now consider that instructions executed in the most commonly executed basic block must be compared not only to the second most commonly executed block but to the sum of complex instructions executed in total. Let us say that the most common block contains this code:

```
tmp01 = x * x;
tmp02 = tmp01 * x;
return tmp02;
```

Listing 3.2: Square block.

The conclusion would then be that the cube function should be implemented in hardware. But a further inspection concludes that the following piece of code exists in several basic blocks, the sum of which is executed more often than the first snippet of code.

```
tmp01 = x * c;
tmp02 = y + tmp01;
return tmp02;
```

Listing 3.3: MAC block

Here we have one addition and one multiplication which would then be preferred for hardware implementation. This of course is a trivial example, With such simple functions it would naturally be preferred to implement both but in a less simple reality the code blocks are more and considerably more complex.

But what if chip area is more expensive than execution time? There may not be enough room to have more than one single multiplication unit. In that
case special blocks containing multiplication operations would have to be split and take two or more clock cycles instead of the ideal one. Something the profiler would ideally take into account.

3.6 Introduction to compiler theory

The Releief profiler implementation in this project is as we will see later heavily dependent on the GCC compiler. In order to understand the later chapters an introduction to how a compiler works is necessary.

A compiler implementation is a complex piece of software. In the following section a brief introduction will be given to the different parts of a normal compiler. In cases where there are differences in strategies and opinions the GNU GCC C-compiler will be given as an example. Please note that what follows is a very brief and shallow description and that the interested reader will find several unanswered questions.

The process of translating human readable source code into binary executable code in a modern compiler goes through three distinct phases. These phases are commonly called front end, middle end and back end[1].

3.6.1 The front end

The purpose of the front end is to generate an internal representation of the program suitable for algorithmic analysis and transformation of the program. The most common representation here is the syntax tree. This phase has in itself a series of sub-parts. Firstly, since reserved words in the programming languages such as if and then as well as function- and variable names are atomic in nature, the series of characters from which they consist is transformed into atomic structures called tokens in a process called lexical analysis. In the example below each lexical entity is transformed into a data structure which is easier for the program to handle.

\[
x = \text{funcall} (x*y+3) \\
\Rightarrow <x><==><\text{funcall}><(><x><==><y><==><3><==>)>
\]

The generated stream of tokens is then analyzed again and a tree structure is built which represents the code and unambiguously depicts in which order the code is to be executed. This process is known as parsing.

This so called syntax tree structure is the output of the compiler front end. Each node in the tree can be either an operation such as +,− or =; or an operand such as x or funcall. The resulting syntax tree from the previous string of tokens is showed in figure [3.2].

In GCC the first kind of generated syntax trees are known as GENERIC trees. We will see why in the next section.

Since writing lexical analyzers and parsers is tedious and very error prone work there are special generator software packages for this work for which the
3.6. Introduction to compiler theory

<funcall><x><*><y><+><3><>

= x

funcall arg_list
+
3*
y

Figure 3.2: The list of tokens is converted into a tree

developer specify only the details specific for the particular language being developed.

3.6.2 The middle end

The central point of the middle end is that the format of its input is the same as its output. In later versions of GCC ($\geq 4.0$) the middle end is used to transform the parse tree from its GENERIC form into a much more restricted form of trees named GIMPLE trees. The GIMPLE form is then transformed according to a series of code optimization algorithms in order to reduce code size and execution time.

For a simple compiler the middle end is not a necessary part but all modern compilers heavily optimize the code and this process has been moved into a separate middle end.

3.6.3 The back end

The optimized code from the compiler middle end is now handed over to the back end. The back end is concerned with transforming the internal representation of the code into actual machine readable code. The reader might not be surprised to learn that this is done in several steps:
Firstly the trees are translated to a general form of assembler instructions that is not executable on an actual processor but can be used for even further code optimizations. In GCC this is done in accordance with a rule called Static Single Assignment or SSA where any variable only can be assigned a value a single time and must then keep this value for the remainder of its existence. One central advantage of the previously mentioned GIMPLE trees is that they facilitate this transformation.

The SSA intermediate code is now in the final step translated into the assembler language of the computer architecture for which we are compiling which is then assembled into the binary executable file.

3.7 Final words

In chapter 5 we will cover the technical details of the profiler implementation. What will then be important from this chapter is that inside the compiler the program is represented as basic blocks which consists of among other things a list of statements and that each statement in that list is a tree of operations and operands.
Chapter 4

Introduction to the tool

The following chapter is an introduction to the Relief profiler tool from a users point of view. In some cases such a text would be placed as an appendix to the thesis but in this case it will serve as an introduction to the profiling process in practice. This introduction will then be backed up by the implementation details in chapter 5.

4.1 User workflow

The Relief profiler is a combinational static-dynamic profiler. This means that execution independent fine grained program data is combined with dynamic execution statistics in order to attain a complete view of the program.

There are four major steps that have to be taken in order to profile a software with the Relief profiler. Figure 4.1 shows a schematic view of the process.

**Build configuration** Configure the software build process so that the instrumentation enabled compiler is used and the software is linked against

![Figure 4.1: A schematic view of the Relief workflow for the user](image-url)
the dynamic profiling library.

**Build the software**  This is the static analysis part of the process where execution independent data is extracted and profiling enabling instrumentation is performed.

**Run the software**  This is the dynamic profiling step. The software is simply executed and the dynamic data is outputted. This part may be repeated with a wide variety of inputs since dynamic data will vary with different inputs.

**Data analysis**  The data produced in part two and three are combined in a special analysis software so they may be considered as a whole.

### 4.2 Installation Overview

Due to the experimental nature of the Relief profiler, installation and usage is quite a complicated matter. No focused testing has been conducted so the manner in which the process is expected to flow is in no way a guarantee that it will do so in the real world.

The guide that follows is therefore a description of the expected behavior and process needed to get the system up and running.

### 4.2.1 Prerequisites

The Relief profiler has four distinct parts that all need to be retrieved and compiled in order for the profiler to work.

**The GCC C-compiler**  GCC is the GNU Compiler Collection, the C-compiler is at the core of the GCC and also of the profiler.

**The GEM Framework**  GEM is the GCC Extension Modules, a framework for extending the GCC C-compiler with instrumentations or optimizations.

**The Instrumentation library**  is a GEM-module that instruments GCC compiled code with time-taking and general profiling.

**The Profiler library**  contains the definitions of the function calls inserted by the instrumentation library. It will be linked into the compiled software.

The Relief profiler has been developed on a PC running Gentoo Linux-x86-64. It is expected to run on most PCs running Linux and with more or less effort it will probably run on most systems for which there is an official release of GCC.
4.3 Installing GCC-GEM

4.3.1 Downloading GEM

The GEM installation script is located at http://www.ecsl.cs.sunysb.edu/gem/. Click and download the latest (at the time of writing 1.6) version which consists of a small .tar.gz file. Untar the file in the location where you want the profiler to be located with the command.

```
[ ~ ] $ tar -xvz gem-1.6.tar.gz
```

Enter the folder gem-1.6 and edit the file Makefile with for example emacs. Make sure that the two top lines look as follows:

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

In other words that we will be compiling GCC version 4 and not version 3.

4.3.2 Downloading GCC

Now download and patch the GCC release by typing

```
[ ~/gem-1.6 ] $ make gcc-download
```

This process may take a couple of minutes. If download is too slow, abort the process and find the file gcc-core-4.1.0.tar.gz on an ftp closer to you. ftp.sunet.se/gnu/gcc/releases is a good alternative if you are sitting on a Swedish university. Then enter its address at the appropriate place in the Makefile and retry.

4.3.3 Compiling GCC-GEM

If all above has gone well the build process for the patched GCC is started with the command:

```
[ ~/gem-1.6 ] $ make
```

Please note that the GCC build process is a very complicated one with several so called bootstrap stages and a myriad of software components and dependencies. It might very well work fine but do not be surprised if there is error. In that case also be aware that there is a good chance that there is a solution somewhere on the Web. Most problems can be resolved by giving directives to the configure process. Google usually knows more.
4.4 Building the libraries

Building both libraries can be fully automated but for the sake of clarity the process will be split up into two separate steps here.

4.4.1 Building the instrumentation library

Enter the instrumenter folder in the source tree and invoke make:

```
[˜/gem−1.6]$ cd ..//relief/code/instrumenter

[˜/relief/code/instrumenter] make
```

The make process should pass without errors or warnings.

4.4.2 Building the profiler library

Go to the profiler folder and invoke make again.

```
[˜/relief/code/instrumenter] cd .. // profiler

[˜/relief/code/profiler] make
```

Again, the make process should not result in any warnings or errors. The profiler folder also has a shell script which invokes a complete remake of both libraries which can be rather useful during development.

```
[˜/relief/code/profiler] $ ./build
```

4.5 Static Analysis - Compiling your software

In order for your software to be profiled it needs to fill three separate requirements.

- The compiler must be the special GCC compiler you built earlier.
- The instrumenter library ci.gem must be given as an argument.
- The software must be statically linked with the profiler library libprofile.a

In order to achieve this for a single file program the compile command might look something like this.

```
[˜/test]$ ˜/gem−1.6/gcc−4.1.0/gcc/bin/gcc −fextension −module=

˜/relief/code/instrumenter/bin/ci.gem −L˜/relief/code/profiler −lprofile test1.c
```

Listing 4.1: The build command invokes the instrumenter library
As can be seen, the newly built special GCC program is being used, the extension-module command is given the file `ci.gem` as argument, the `-L` argument is used to inform gcc where extra libraries can be found and `-l` links in the profiler library.

Naturally most software complex enough to actually be considered for profiling consists of several files and uses some sort of build system. In that case the following is a general guide to getting the profiler to work.

- Make sure the compiler is invoked with a variable and that that variable is changed to the modified GCC. In a GNU Makefile this may look like `CC=/gem-1.6/gcc-4.1.0/gcc/bin/gcc`.

- Each compiled file should have a standard set of arguments in at least one variable. In the suitable variable definition therefor add the extension argument `extension-module=/relief/code/instrumenter/bin/ci.gem`

- During linking there should also be at least one variable with library locations and included libraries. Make sure `-L/relief/codeprofiler -lprofile` is invoked during linking.

### 4.6 Instrumentation configuration

It is not rare to want to profile only specific parts of a program. For this purpose the instrumentation process may be guided by a configuration file. There are three ways in which the profiling may be configured to accommodate for different needs.

#### 4.6.1 The configuration file

Before the instrumentation of a file during compilation a configuration file is read from file. The instrumentation library first and foremost looks for a filename in the system environment variable `RELIEF_CONF`. If this variable is not defined or defined to a non-existing file it looks for a file named `instrumenter.conf` in the build directory.

The configuration file may contain three different sections in no particular order, one for each of the configuring types. These are the instrumentation section, the no instrumentation section and a patterns section.

In addition to this single lined comments may be used. The comments are begun with the characters `//`.

#### 4.6.2 Opt-in profiling

If the algorithm needed to be profiled is well defined in terms of which functions should be included opt-in profiling is the way to go. This means that the
functions that are to be profiled are specified and all others are disregarded. An opt-in configuration starts with an \texttt{instrument:} label and then a list of functions to be included in the profiling. Listing 4.2 is an example of opt-in profiling configuration.

```
// This is a comment.
instrument:
   my_sort
   my_swap
   my_gen
```

**Listing 4.2: Opt-in profiling configuration**

### 4.6.3 Opt-out profiling

For quick and dirty profiling or when only information on which functions are definitely not to be included there is also an opt-out configuration mode. This is invoked with the \texttt{no_instrument:} label and then a list of functions not to be included in the profiling. All other functions will be profiled as usual. An opt-out configuration might look as in listing 4.3.

```
// Don't instrument known config functions.
no_instrument:
   main
   init
   my_print
```

**Listing 4.3: Opt-out profiling configuration**

There is no explicit error in having both \texttt{instrument:} and \texttt{no_instrument:} sections in the same configuration file. However, as they are implicitly defining each other using both increases complexity and thereby the likelihood of errors it is not recommended.

There may however be necessary to re-analyze the program in several steps as more and more information is exposed from earlier analysis. For example the designer might start with only a vague notion of what not to profile so the configuration is set to disregard certain information but after the first couple of dynamic executions and post analysis the core set of functions is uncovered and the program is recompiled with a specific instrument-section to focus the analysis on one separate algorithm.

### 4.6.4 Experimental patterns profiling

There is a feature which was not completed due to time constraints and this is the patterns matching feature. The idea is that the designer in the configuration file specifies a list of operations which are suspected to be suitable for
implementation as a custom instruction in the ASIP. During instrumentation
the IR code is matched against this pattern and matches are noted so that an
estimation of the possible speedup can be estimated during post execution
analysis.

A pattern is specified in the configuration file with the keyword pattern:
and then a name of the pattern followed by a list of operations in the pattern.
For example:

```
% my first pattern
pattern: MAC
  ARRAY_DEREF
  MULT
  ADD
```

Listing 4.4: A simple instruction pattern

The keywords available for different operations are as follows: ADD,
SUB, MULT, DIV, ARR_DEREF, SHIFT_LEFT, SHIFT_RIGHT, BITWISE_OR,
BITWISE_AND, BITWISE_XOR, AND, XOR, OR.

As mentioned this feature was not completely implemented. Patterns can
be specified in the configuration file and if so, they will be matched against
the profiled code. However nothing will happen upon a match.

```
<?xml version="1.0"?>
<dyn_file>
  <environment>
    <executable>./cjpeg</executable>
    <pid>13969</pid>
  </environment>
  <basic_block>
    <name>jpeg_write_scanlines_10</name>
    <locus>
      <file>jcapistd.c</file>
      <line>107</line>
    </locus>
    <executions>682</executions>
    <time>
      <time_sec>59</time_sec>
      <time_usec>256701</time_usec>
    </time>
    <previous_blocks/>
  </basic_block>
  ...
```

Listing 4.5: Cutout from a dynamic data file
4.7 Dynamic Profiling - Running your software

When successfully built, the static analysis results will be written to XML-files in the build directory. Each XML-file will carry the name of the source file to which it corresponds appended with .stat.xml. The program will work as normal but at exit it will write the collected dynamic statistics of execution to standard output. Furthermore it will write another XML-file named after the executable postfixed with the process-id of the runtime and .dyn.xml so an execution of the file test1 might produce test1-2654.dyn.xml. En example cut-out of such a dynamic data file is listed in listing 4.5.

4.8 Post execution analysis

The following manual details the intended procedure for using the post execution analyzer software. In order to benefit from it the user needs to have both static and dynamic profiling data from the software that is to be profiled. These are obtained during recompilation and running of the profiled software using the instrumentation and profiling tools described in the previous sections of this chapter. The output files needed are .stat.xml for the static data, .dyn.xml for the dynamic data and .dot for the CFG-visualization data files. If the previous steps have been run, these files should have been produced in the build directory automatically.

4.8.1 Getting started

Since the Python code is interpreted dynamically no compilation step is needed in order to run it. To start the program, go to the /code/analyzer/ directory and run the file analyzer.py

```
˜$ cd relief/code/analyzer/
˜/relief/code/analyzer $ ./analyzer.py
```

The program boots and an empty window pops up. This is the analyzer main window. It is split into two separate subwindows. We shall soon see what they do. In order to do anything useful however, we need to load the profiling results file. Go to File→Open. A standard dialog for opening files will appear. Now locate the dynamic statistics data file that was produced during the run of your profiling session. The file you are looking for is named according to the pattern <Binary name>--<Process ID>.dyn.xml. For example an JPEG encoding test run resulted in the file cjpeg-13072.dyn.xml. As the process id is different for each time a program is run running the instrumented program several times will result in a separate dynamic data file for each run time. This file is highlighted in the opening window. You will notice that static data files are greyed out as in
Figure 4.2: The open window, only dynamic data files are allowed to be opened.

This is because they are instead referenced from the dynamic data file and are opened and read when necessary.

The first time a dynamic file is opened the full CFG-graph representation of that program is produced by aggregating the .dot-files that each represents the graph of a single source file in the program. This may take some time. During the process information is written to the system console:
Aggregating to
"/jpegtest/cjpeg -13072.dyn.xml.dot
aggregating "/jpegtest/jcapistd.c.dot
aggregating "/jpegtest/jcmainct.c.dot
aggregating "/jpegtest/jccoefct.c.dot
aggregating "/jpegtest/cjpeg.c.dot
aggregating "/jpegtest/jcprepc.t.c.dot
aggregating "/jpegtest/rdbmp.c.dot
aggregating "/jpegtest/jcsample.c.dot
aggregating "/jpegtest/jcdctmgr.c.dot
aggregating "/jpegtest/jccolor.c.dot
aggregating "/jpegtest/jfdctint.c.dot
aggregating "/jpegtest/jcmaster.c.dot
aggregating "/jpegtest/jchuff.c.dot
aggregating "/jpegtest/jdatadst.c.dot
aggregating "/jpegtest/jcinrt.c.dot
aggregating "/jpegtest/jcdctmgr.c.dot
aggregating "/jpegtest/jcmarker.c.dot
aggregating "/jpegtest/jcapimin.c.dot
aggregating "/jpegtest/jcomapi.c.dot
aggregating "/jpegtest/cdjpeg.c.dot
aggregating "/jpegtest/jerror.c.dot
aggregation finished

Listing 4.6: The output from the graph aggregation process

After that the xml-data files are opened and read into the program. This also takes some time and no data is outputted so the system may feel sluggish or even hung up during this phase.

When the opening phase is completed, data will be written to the main window as seen in figure 4.3.

4.8.2 Left window

The leftmost window contains all the basic blocks in the program sorted by name. The name as mentioned before is an aggregation of the name of the function that the block belongs to and an index number since most functions consists of several basic blocks.

Each block in the left window can be expanded to reveal a multitude of information concerning that particular block as seen in figure 4.4. The information consists of:

- Total execution time in seconds and microseconds (dynamic data)
- Number of operations consumed by the block each time it is executed. (static data)
4.8. Post execution analysis

Figure 4.3: The post analysis in action, the left frame contains all the blocks sorted by name and the right is a printout of the blocks sorted by operations consumed during the execution denoted by the dynamic data xml-file.

- Source location of the block, file and line number (static data)
- number of iterations during the run. (dynamic data)
- Total number of consumed cycles. This is a multiplication of total number of operations and iterations and should be considered static-dynamic data.
- A list of branch statistics from previous blocks. The prev_block folder contains a number of times that branch was taken and a percentage on how many times that particular branch was taken out of the times the block was entered.

Please note that all the information concerning the block available to the profiler at the given moment is located in the tree. This means both dynamic information from the currently profiled execution as well as the static information collected during the compilation step.

4.8.3 Right window

The right section of the window also contains the information of the blocks but this time they are sorted by the previously mentioned total number of clock cycles consumed.
Figure 4.4: A printout of an expanded basic block tree.
4.8. Post execution analysis

The top line of the right window displays a number which is the sum of all operations consumed of all basic blocks. This means that it corresponds to all operations consumed during the entire execution of the program at the current optimization level. After that comes a percentage number which is the previous number divided by the total number of operations needed if the program was completely unoptimized. We call this the optimization level.

When our file is newly opened we have not specified any blocks to be optimized so naturally the optimization level is at 100% as seen in figure 4.5. The block which consumes the highest number of operations is at the top of the right window. In our case it is the block rgb\_ycc\_convert\_2. We can also see that it consumed as much as 35% of the total operations consumed by the program during our test run.

What would happen if we tried to implement all the operations consumed by this block as an application specific instruction in an ASIP? Since the basic block by definition is just a set of operations without conditional jumps the result should be an operation that takes only one clock cycle and the total number of cycles consumed by the block will be reduced from \( \text{iterations} \cdot \text{operations} \) to just \( \text{iterations} \).

4.8.4 Simulating optimizations

We can simulate the effects of implementing the block rgb\_ycc\_convert as an application specific instruction by scrolling down and selecting it in the left window and choose Optimize→Optimize in the menu.

This will take a short moment of computing and then the right window will update.
As you can see from figure 4.6, the optimization level is now at 64% since a lot of operations has been condensed into a single clock cycle. This means that if we implemented a processor with such an instruction, the number of cycles it would need for this program run would be only 64% of the number of cycles needed by a processor with only primitive operations.

### 4.8.5 Target optimization

Since a real world situation for a designer would likely be a program and a clock cycle requirement such as “Make this algorithm run in 20 milliseconds in 500 MHz”, and the process would more or less be just selecting the top block and selecting it for optimization until the requirement is met, there is a target optimization mode.

In order to use it select **Optimize→Target Optimize** in the menu bar. The dialog box requests a desired value of the optimization level. If we would want to make this particular situation 5 times faster we enter 20 (for 20% of original cycle count) and press enter. The program will now optimize the most cycle consuming basic block and move on to the next until the target is reached or there are no more blocks to optimize.
Chapter 5

Implementation

The following chapter depicts the implementation of a source code profiler with a combined static-dynamic analysis. The text takes an iterative approach with the first part describing the large scale division into two libraries and an analysis program with separate responsibilities. Sections 5.2 and 5.3 then takes the reader through the internals of each library and section 5.4 details the analysis software.

5.1 Introduction

5.1.1 Architectural overview

![Diagram of relations between source, compiler, program and the two libraries.]

Figure 5.1: Relations between source, compiler, program and the two libraries.

The profiler itself has two major parts. One part is the code instrumentation library which performs the code instrumentation and analysis. The other one has been named the profiler library and it performs the runtime analysis. Basically the instrumentation library contains the static part of the profiler and the profiler library contains the dynamic part. In addition to this an analysis software is required to interpret the outputs of both the static data generated by the instrumentation library and the dynamic data produced by the profiler library.
As mentioned in 4.5, a program that is to be profiled must be separately compiled with the special GCC implementation and the code instrumentation library. During compilation, function calls for the dynamic profiler is inserted at the beginning and end of every basic block in the program. The profiler library is then statically linked to the program. Figure 5.1 shows how the libraries relate to the program and the process of compiling it.

The instrumentation library besides inserting function calls also counts and stores the instruction statistics for the basic blocks. The instruction account contains the numbers on each kind of operation to be executed for it. Memory reads and writes, arithmetic operations, array references and so forth. This information are then stored in XML-files on disk. Finally if the compilation succeeds, a binary executable with the new function calls is created.

The function definitions for the calls inserted are implemented in the next major part of the profiler. The profiler library. So now when the program is executed, every time a new basic block is entered the time is taken and stored on a stack called the open environments stack. When this block is exited the open environments stack is popped and the time spent executing the block is calculated. This time is then stored on another stack called the closed environments stack. This way the program is executed until the end of the main() function is reached. At this time, all the elements on the closed environments stack are one by one being inserted into a hash table with the basic block name as a key, thus adding up all the time spent in each separate basic block. With the times added, the elements are finally sorted by total execution time and printed both to standard output and in XML-format to disk.

5.1.2 Profiling workflow

Consider figure 5.2 which depicts the complete process of profiling a program. The source code for the program to be profiled starts untouched at the top of the flowchart. The first three boxes Lexical analysis, Parsing, and Parse tree optimization together constitutes the compiler front- and middle-end such as discussed in chapter 3. From the parse tree produced GCC itself extracts the Control flow graph or CFG. The CFG consists of a set of basic blocks and edges between these which represent the various control flow jumps in the source code. The CFG is fed to the Relief instrumentation library which instruments the blocks and extracts the basic block costs. See the next chapter for further details on this matter.

The instrumented CFG is then fed onward to the compiler back end where binary code is generated. During the linking the various object files of the software is linked and in this process the profiler library is linked in. Currently this must be explicitly entered into the build configuration (Makefile) by the developer or person performing the profiling.

When the binary executable is written to disk it can be executed normally.
Fed with input it produces its normal output and when exiting it prints execution statistics to disk and standard output for that particular run.

5.1.3 Example run

As mentioned no special steps are necessary when performing the dynamic profiling. The program is executed in a completely normal fashion and the actual profiling process is fully automated.
The output in listing 5.1 is from the example run of a program compiled with the profiler-compiler. Each line in the table corresponds to one basic block in the code. The lines are sorted by the total execution time spent within them.

The first tab cell contains a simple line numbering, the second the name of the basic block which is the name of the function which the block resides in combined with an index number since a function usually contains several basic blocks. The third tab is the file and line number in the code where the block starts. The forth cell is the number of times the block was executed during the execution and the last one is the total execution time in seconds.

<table>
<thead>
<tr>
<th>Profiling results: pid: 25923</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>from: my_sort_1</td>
</tr>
<tr>
<td>from: my_sort_5</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>from: my_sort_3</td>
</tr>
<tr>
<td>from: my_swap_0</td>
</tr>
<tr>
<td>from: my_sort_2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>from: my_sort_6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>from: my_sort_2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>from: my_sort_4</td>
</tr>
</tbody>
</table>

Listing 5.1: Output from a test run of a profiling run.

Now, aside from the executable, the compiler also writes an xml-file for each source .c-file it compiles. Therefore if we look inside test1.c.stat.xml we can find the following for the top block in the list above.
<basic_block>
  <name>my_sort{2}</name>
  <locus>
    <file>test1.c</file>
    <line>21</line>
  </locus>
  <mem_read>30</mem_read>
  <mem_write>11</mem_write>
  <func_call>2</func_call>
  <addition>3</addition>
  <multiplication>2</multiplication>
  <greater>1</greater>
</basic_block>

Listing 5.2: Example of static XML data in an .stat.xml file

Inside the XML-file each basic block is represented by a tag like this one. The tag contains, as well as the name and location in the source file, an accounting of the number of operations of each kind which is executed each time the block is executed. Since the basic block per definition is executed in its entirety (one entry and one exit point) these numbers are not dependent on the dynamic nature of the execution.

5.2 The code instrumentation library

The following section describes the architecture and design of the code instrumentation library which is in charge of the static parts of the profiling process. Because of the static nature of it, every task performed and all the information collected by the code instrumentation library is independent of when, where or how the program is later executed. The input data to the algorithm also does not matter to this process as we will see.

5.2.1 Overview

The basis of the code instrumentation library is a plugin system to the GCC called GCC Extension Modules or GEM for short. Simply put, a set of function call hooks is inserted into the GCC C-compiler source code. It is then up to the GEM user to define the function definitions for these hooks and compile them into a dynamic library to be run alongside GCC when it is executed.

The GEM plugin created in this project is tasked with three separate but related objectives. First, the program code being compiled must be instrumented with profiling calls in order for the dynamic part of the profiler to work. Secondly the relevant information about the code is collected and writ-
ten to XML-files on disk and finally, a representation of the code dependency graph for each file is written to separate files.

Each of these tasks are performed per source file so every .c-file that is compiled will end up with one .stat.xml and one .dot file in which basic block data and GraphViz-data are written respectively.

Furthermore a compile time static analysis of each basic block has been added. The block analysis basically consists of instruction counting. One particular block may consist for example of two memory accesses, three multiplications and six additions. Combined with execution statistics this data can with the correct post execution analysis give a fairly accurate view of the software execution and what parts to focus your design efforts on.

5.2.2 GEM hooks used

The profiler uses the following GEM hook function calls inside the GCC source code.

**gem_common_nodes_and_builtins** - Called at the beginning of each source file. In order for the function definitions to be found, their respective declarations are inserted here. Also the XML and graph visualization files to be written are opened.

**gem_finalize_compilation_unit** - From this call the .stat.xml and .dot files to which operation stats and graph specifications are written are closed.

**gem_annotate_cfg** - This hook call is called for each basic block in the source file. From here the static analysis of the actual code is performed, the XML and graph visualization-data is written to disk for each basic block and finally the basic block profiling instrumentation is inserted.

**gem_finish_function** - Every time a function body is parsed this function is called and the function level instrumentation is performed. More specifically the function argument profiling calls are inserted.

5.2.3 Code instrumentation

Inserting alien code into a preexisting piece of software is complicated and error prone matter. Therefore the code instrumentation of the Relief profiler is kept to a minimum. Instrumentation is done on functional level for argument size profiling and on basic block level for execution path profiling.

**Function instrumentation**

Each function entry is instrumented with at call to arg_collect() for each integer argument the function takes. The purpose of this is in the end to
collect and profile the maximum size of the argument values received. This will be discussed more in the next chapter.

**Basic block instrumentation**

Basically each non-empty basic block is instrumented at the beginning and the end with calls to `bb_preface()` and `bb_appendix()` respectively. Figure 5.3 illustrates this. Also, the `main()` function is instrumented at the beginning and end with calls to `init_instrument()` and `finish_instrument()`. In order for all of these function calls to work, their respective declarations is also inserted at the beginning of each compiled file.

The definitions of these functions reside in the profiling library and is thereby a part of the dynamic profiler. See subsection 5.3.2-5.3.5 for details.

**Instrumentation configuration**

The instrumentation process can be guided by configuration files. The parsing of these configuration files are done by grammars specified in GNU Flex [20] and Bison [8] tool formats. The lexical specification is located in `conf_lexer.lex` and the bison parser specification is located in `conf_parser.y`.

The specifications of the configuration file is found in section 4.6. The grammar for the specification is fairly self-explanatory in the source files if the reader is familiar with the tools.

**5.2.4 Instrumentation examples**

**Inserting declarations**

Listing 5.3 exemplifies inserting a function declaration into a source file. This is necessary for the call instrumentations that is inserted in the file to be executed correctly. Declarations are kept on a stack so the way it works is the `t_decl` which is a pointer typedef to a tree structure is created and pointed to a declaration tree created by `build_decl()`. The function `get_identifier()` is a GCC internal function which designates identifier values for declared symbol names. Following that some properties are set.
to make the declaration reachable by calls in the code. `make_decl_rtl()` converts the declaration to an intermediate representation form and finally it is pushed to the declaration stack.
5.2. The code instrumentation library

Listing 5.3: insert a function declaration into a source file

tree t_decl;
t_decl = build_decl(FUNCTION_DECL,
    get_identifier("bb_prelude"),
    t_func_type);
DECL_EXTERNAL(t_decl)=1;
TREE_PUBLIC(t_decl)=1;
make_decl_rtl(t_decl);
pushdecl(t_decl);

Listing 5.4: insert a function call into a source file

Inserting a function call

Listing 5.4 illustrates a simplified version of the code which instruments basic blocks with the call to bb_prelude(). The procedure consists of three steps. Fetching the function symbol, creating the call and inserting it into the statement list. find_symtab() is used to get a declared function symbol from a string with its name. build_function_call_expr() creates a call expression. The expression is considered a statement when it has only side effects and no value so that is what we set it to. Finally the new statement is inserted with tsi_link_before. The block statement iterator is used to point out where to insert the call, tsi stands for tree statement iterator which is closely related form of iterators.

tree t_func_decl;
tree t_body;
tree t_new_stmt;

block_stmt_iterator bsi = bsi_start(bb);

find_symtab(&t_func_decl, "bb_prelude");
t_new_stmt = build_function_call_expr(
    t_func_decl, NULL_TREE);
TREE_SIDE_EFFECTS(t_new_stmt) = 1;
TREE_TYPE(t_new_stmt) = void_type_node;

Listing 5.4: Insert a function call into a basic block

In the actual source code, the function call also has several arguments which are created as a list and passed to the call to build_function_call_expr(). Furthermore the real world code must insert the call not at the very beginning but after branch labels which reside at
the beginning of the block. Otherwise entrances to the block via these labels will miss the call when it lies before the label.

5.2.5 Generation of static code statistics

In order to fully understand the execution of a program the profiler must collect information not only on which parts of the code is executed when but also what these parts consist of. In order to do so, the instrumentation library before each basic block is instrumented, counts the number of different operations in the block and writes an xml-tag into a file named after the source code file but with the postfix .stat.xml so that if function my_fun() resides in my_file.c we might find the following in my_file.c.stat.xml:

```
<basic_block>
  <name>my_fun_0</name>
  <locus>
    <file>my_file.c</file>
    <line>47</line>
  </locus>
  <mem_read>9</mem_read>
  <mem_write>3</mem_write>
  <func_call>5</func_call>
</basic_block>
```

Listing 5.5: Static data from a .stat.xml file

The code for this xml output resides in the file static_analysis.c. It is very primitive in its nature and does not rely on any form of professional xml-writing tools since the xml written is no more advanced then a series of tags such as above.

5.2.6 Dependency graph visualization

As the internal control flow graph representation of GCC for each basic block contains notations of following and preceding blocks a simple dependency graph description is outputted to files postfixed with .dot in same manner as for the .stat.xml files. In test1.c.dot we find this outtake:

```
subgraph my_gen_0 {
  my_gen_1 -> my_gen_2;
  my_gen_2 -> my_gen_1;
  my_gen_0 -> my_gen_2;
  edge [color = firebrick];
  my_gen_0 -> srand_0 [label = srand ];
}
```

Listing 5.6: A subgraph from a .dot-file
Figure 5.4: Dependency graph visualization generated by dot based on basic block dependencies in the code.

Each line contains either an edge description or a command to change the edge colors. Red colored edges denote function calls while black edges are other control flow jumps such as loops or if-constructs. Nodes are not defined but instead defined implicitly by the edges. If an edge begins or ends in a node not yet defined a new node is created.

The .dot-files are compliant with the open source GraphViz toolkit. There are several programs in this kit which all render the graphs but with different layout priorities. We recommend using the dot software. Figure 5.4 is a render of the dependency of a simple single file program using dot.
5.2.7 Relevant source files

- `instrumenter.c` - Contains function hooks for the GEM framework.
- `ci_support.c` - Major support functions for the code instrumentation.
- `cfg_support.c` - actual code instrumentation implementation.
- `static_analysis.c` - static analysis of basic block and xml-output.
- `graph_writer.c` - GraphViz compliant dependency graph output.
- `conf_lexer.lex` - Flex configuration file
- `conf_parser.y` - Bison parser generation configuration file
5.3 The profiler library

This chapter depicts the internal structure of the dynamic part of the Relief profiler by walking through the execution of a program.

5.3.1 Architectural overview

The profiler library is organized in a collect → sort fashion in which data is first collected in the fastest possible manner and sorted before exit. Figure 5.5 shows the profiling process from a helicopter perspective. The process starts off with program initiation. Data is then collected within the bb_preface() and bb_appendix()-functions and finally sorted in the finish_instrument()-function call.
5.3.2 Program initiation

The main() -function has aside from the instrumentation’s we will be discussing later also two additional function calls inserted. These are init_instrument() and finish_instrument(). The main purpose of these functions are memory allocation and deallocation for the data structures discussed. In the case of init_instrument it allocates memory for two stacks, a hash table and a list. The use of which will be detailed below.

5.3.3 Basic block entry

Each time a new basic block is entered the instrumented code calls the function bb_preface() with the basic_block name and the source code location of that basic block as arguments. This function stores the time on a microsecond granularity level together with the function arguments in a struct named time_item and puts it on a what is called the open environments stack or open_envs. The reason for using a stack structure is twofold but first of all the push operation is constant time and extremely fast. Consider that any time taken outside of the actual execution of the program from that the time is stored in memory will skew the execution time results. In the beginning of the project this was a hash table as that too has constant time insertion. However, as the hash function could take several hundred operations for a basic block that represented only three instructions it proved unusable.

The bb_preface() function also reads the global variable last_block_name, which holds a string with the name of the last block that was executed, stores that information in the time_item-structure in a variable called prev_blocks and writes its own name to the last_block_name-variable. This will aid in constructing a branch statistics table between the blocks.

5.3.4 Basic block exit

Remember that we defined a basic block as a series of instructions which had only one entry and one exit point. This means that when we reach the inserted bb_appendix, no other structure than the last inserted can be on the top of open environments stack. Thus bb_appendix() pops the stack confident that the top represents the time_item inserted at the beginning of this particular basic block. The clock is now read again and the old time from the stack is subtracted from the current time giving us the execution time of the basic block. This time replaces the start time in the time_item which is now moved over to another stack called the closed environments stack or closed_envs.
In this process the variable prev_blocks which held the name of the block executed before the current block is converted into a list structure so that the names of all blocks executed before it may be contained. This is illustrated in figure 5.6. Notice how for each entry branch how we can get a percentage number by dividing its times_taken value by the executions-variable of the time_item.

5.3.5 Program de-initialization

It was mentioned in subsection 5.3.2 that the main purpose of the finish_instrument() function was deallocation of the memory used for the various data structures. That is true but before it does that it also does some additional computing. One thing that is important to remember here is that by the time we reach the finish_instrument() function the execution of the program itself is over. This means that the previous notice that execution of code outside of the program would skew the results no longer holds and the use of for example a hash table is now perfectly acceptable.

Each time a basic block is exited a time_item is added the closed environments stack. This means that at the end of the program we will have a large array of execution times for different basic blocks. Many of them will also have been executed several times which means that they will have several copies on the stack. For the purpose of adding up the execution times and statistics of all these time_items a loop will take all the elements of the closed environments stack and one by one enter them into a hash table hashed by the unique basic block name. If the names match, the execution times of the two time_items will be added together, their prev_blocks-lists will be concatenated and their executions added.

Ideally the hash table would now be sorted and outputted to whatever media is desired. Unfortunately the hash table implementation used does not
support sorting and therefore the elements are sorted by moving them to yet another data structure; this time a list and the list is then printed to standard output.

5.3.6 Memory efficiency in host machine

Since one verb+time_item+ structure is produced for every time a basic block is executed and quite a few blocks are executed during a program run the memory utilization of a profile run may become staggering for complex programs. In order to reduce the memory requirements of the profiler the algorithm used in finish_instrument() to copy stack elements to the hash table where they are also compacted to one copy per existing basic block instead of one per time the block is executed memory usage is severely reduced this way.

5.3.7 Relevant source files

- profiler.c - Function definitions for the function calls inserted by the instrumentation library.
- time_stack.c - Stack implementation for the open and closed environment stacks.
- hashtable*.c - Hashtable implementation. This code is courtesy of Christopher Clark.
- timeval_support.c - List implementation for the sorting and printing of final results as well as support such as addition and subtraction of the timer values.

5.4 Post Execution Analysis

The following section motivates and describes the implementation of a post execution analysis software in the Relief profiler software suite. A manual for using it is included.

5.4.1 Introduction

In the previous chapters we have described the tools used to profile both the static properties and the dynamic behavior of a program. This has produced two separate data sets for us to utilize when trying to get a good understanding of it. However, in order to quantitatively measure the impacts of application specific instructions on the software both these data sets need to be analyzed together.
5.4. Post Execution Analysis

The static analysis has given us the properties of the basic blocks of which the program consists and thereby the number of operations each block will consume. At the same time the dynamic profiling has given us the number of times each block is executed. Combined the two of them tells us how many operations the block has consumed in total during the execution.

So what is needed is a tool that takes all the collected data, combines it and sorts the blocks by the number of total operations consumed which will then be a list of blocks most receptive to implementations as custom instructions.

Such a tool has therefore been developed. It is still in an alpha state but constitutes a useful proof-of-concept.

5.4.2 Implementation details

As all profiling data is written to disk in XML-format the only external requirement on the analysis software is that it can read and interpret these XML-files and display the data in a usable fashion. With this in mind it has been implemented in the Python language using the Qt4 GUI software toolkit.

Python is a fully dynamic interpreted object oriented programming language. It has support for several different data structures such as lists and hashes as well as algorithms. This makes it highly suitable for data analysis and treatment. Python interpreters exist for Linux, Windows and the MacOS X platforms which makes for highly portable programs [18].

Qt is a cross platform GUI development platform written in C++. In addition to GUI components it includes tools for interpreting XML. In order to access the C++ classes of Qt from Python the PyQt software package is used[5].

As a result of this the post analysis software is portable between all major operating system platforms and has compact and easy to read code for further development.

5.4.3 Software architecture

The software consists of the following set of Python classes, a UML class diagram is provided in figure 5.7.

MainWindow  Keeps references of all the instances of the different parts of the program.

AppMenu  Represents the program menu bar.

Console  Represents the data console to the right in the main window.

TreeDisplay  Represents the basic block list to the left in the main window.

DynXmlParser  Reads and parses dynamic basic block data from a dyn.xml-file into BasicBlock-instances. Each reference to a stat.xml-file is referred to the StatXmlParser.
Figure 5.7: Basic architecture of the post analysis software.

**StatXmlParser**  Reads and parses static basic block data from a stat.xml-file into BasicBlock instances.

**BlockAnalyzer**  Keeps track of all BasicBlock-instances in addition to extracting useful statistics.

**BasicBlock**  Keeps composite data from both static and dynamic stats of a basic block.

**DotAggregator**  Responsible for aggregating data from all .dot-files into one single file which will illustrate the whole program instead of each source file by itself.

**TargetOptimizeDialog**  Represents the Target optimize dialog box.

**ShowOptimizeDialog**  Represents the Optimizing results dialog box.

### 5.5 Discussion

As mentioned in 3.2.1 the use of function pointers may disrupt the static analysis of the code. Especially the graph visualization will be skewed and for example give the impression that some function trees are never called. This is not the case however. The calls to these trees are just made through another symbol which masks the actual code. Therefor it is recommended, as there are fairly good alternatives to function pointers in the cases where they are
considered, to aid in the profiling by using only direct calls to functions in the algorithm modeling when there is a choice to be made.

Another weakness in the implementation are the focus on time measurements. In the beginning it was unclear whether or not the actual execution time of a block was a usable feature. Therefore there is at the moment an unproportional focus on execution time in addition to the fact that the time measurements are severely skewed by library execution time added after the focus on time was dropped. Nevertheless it may be good to know that as executing the `bb_prelude()`- and `bb_appendix()`-functions takes time there will be a skew in the measurements in which small blocks executed several times will have a significant amount of its execution time coming from the profiler library. Large blocks with few iterations will at the same time have a very small skew. This is not a big problem since execution time is not the main source of information in the design flow which is instead number of iterations and collective number of operations in the block and those numbers are not skewed. Still, designers using the tool should be aware of this fact.
Chapter 6

Results and conclusions

This is the final chapter of the thesis. It contains the details of a field test of the Relief profiler which was performed on the libJPEG image compression software package which is an implementation of JPEG standard provided by the Independent JPEG Group. The final sections discuss the results of the field test in a larger context and applicability of the Relief profiler in the modern field of ASIP design.

6.1 Field test: libJPEG

In order to prove that the Relief profiler works as intended it has been put to the test in a setting where the results are known. A well tested and implemented algorithm is the JPEG image compression format.

6.1.1 Overview of the JPEG algorithm

The JPEG algorithm consists of a series of compression steps with different goals. In its essence it consists of color space transformation, down sampling, discrete cosine transform and entropy coding.

**RGB to YUV** This is a color space transform where the Red, Green and Blue components of the RGB coded image is transformed into a more space efficient color space which consists of the Luminisence (Y) and Chrominence (U and V) channels.

**Image downsampling** The idea of the YUV color space is that the eye is not equally sensitive to all frequencies of light so the representation of the colors does not have to take as much data for all colors. In this stage the chrominence channels are downsampled to preserve space.

**Discrete Cosine Transform** DCT is a frequency transformation not very different from the discrete Fourier transform. The input of this part is a
6.1. Field test: libJPEG

function $A(i, j)$ where $A$ is the pixel color values (now in YUV-format) in row $i$, column $j$ into a function $B(k_1, k_2)$ where $B$ is the DCT coefficient of row $k_1$, column $k_2$.

**Entropy coding** This part consists of Run Length Encoding (RLE) and Huffman Coding. The RLE compacts long runs of equal values. In essence it works like humans would relate to a sequence such as \{1, 0, 0, 0, 0, 1, 1, 1, 1\}, instead of describing it value by value, we would probably describe it as *One one, four zeroes and then five ones* which is the same information in a more compact form.

Huffman coding works by encoding the most commonly occurring sequence of data with a short string and then using increasingly longer strings of data to replace less commonly occurring strings [3].

The JPEG decompression is performed by running the inverses of the different steps in reverse order so you start by decoding the entropy coded data, then performing the reverse DCT and so forth.

### 6.1.2 The libJPEG software package

Since JPEG is a standard there exists several implementations of it but profiling the algorithm requires access to the source code so an implementation with freely distributable source code was required.

The independent JPEG groups implementation libJPEG is the most commonly used JPEG encoder in the Linux operating system distributions. It is also distributed under an open source license so it is very suitable for our purposes.

Unfortunately running Huffman coding is highly data dependent. The time requirements of the Huffman coder can double between executions based on what image is compressed/decompressed. Since this thesis does not deal with optimizing Huffman coding in particular the entropy coding part of the libJPEG software will have to be disregarded during profiling or the results will not be viable in a larger context.

### 6.1.3 Test purpose

The purpose of the tests where manifold. The profiling results must be:

**Correct**: Optimizing suggestions should be as expected based on the methods used.

**Stable**: Profiling results should accurately mirror the algorithm and not the data.

**Usable**: The software should in its current or future form be able to produce results that are relevant. The methods used today may not be enough
but they should in that case work as a platform for future work with potential to remedy that weakness.

6.1.4 Test preparations

The libJPEG distribution came with a set of two test programs. One for encoding and one for decoding images between BMP and JPEG formats.

The supplied test program for encoding JPEG images from bitmaps was selected for the field tests. In order to test the stability of the profiling many profiling runs with different input data was required. For that purpose a set of 20 random images was downloaded from the internet. No particular attention was paid to the images suitability in straining the encoding algorithm but the image sizes was in and around the area of 800 by 600 pixels so that the data treatment algorithms would be clearly separated from supporting code which is not data dependent such as initialization and setup procedures.

The makefile delivered with the software was altered in accordance with the steps depicted in 4.5 so that the programs would be instrumented and linked to the Profiler library.

Since the branch statistics feature of the profiling library (5.3.3, 5.3.4) was completely unoptimized it had to be shut off for the field test in order to preserve memory.

In order to have the Huffman coding code disregarded by the profiler a instrumenter.conf file was created which listed all the functions in the file chuff.c as not to be profiled. The result is showed in listing 6.1.

```
no_instrument:
    start_pass_huff
    finish_pass_huff
    jinit_huff_encoder
    jinit_phuff_encoder
    add_huff_table
    encode_mcu_huff
    jpeg_make_c-derived_tbl
    std_huff_tables
    start_pass_huff_decoder
    jpeg_huff_decode
    jpeg_make_d-derived_tbl
    jinit_huff_decode
    decode_mcu
    jpeg_fill_bit_buffer
    emit_bits
    encode_one_block
```

Listing 6.1: instrumenter.conf
The library and related test program were built using the `make` command. The build process worked well and everything was correctly built.

### 6.1.5 Running the test cases

The test cases were executed with the command

```
./jpeg -dct int -outfile <num>.bmp <num>.jpg
```

Where `<num>` varied between '1' and '20'. Each test took about five minutes to execute and resulted in the expected output of `.dyn.xml`-files. Two of the files were too big and the profiling was aborted by the operating system when the virtual memory usage exceeded 4 GiB. More on this in section [6.3.2](#).

### 6.1.6 Post execution analysis

The resulting 18 XML-data files were loaded into the post execution analysis software one by one. Target optimization was selected with the goal of finding an optimization level were 10% of the original cycle count which was achieved on all the available data.

### 6.2 Profiling results

The resulting optimization level was noted in a text file along with the list of optimized basic blocks for each profiling run. The complete listing of this text file is available in Appendix [A](#).

The results of the tests showed that the same set of 10 optimizations would all result in an increase of more than 10 times for the profiled algorithms given that they may be implemented to take exactly one clock cycle. The block to be optimized are listed in table [6.2](#). For some images different parts of the algorithms took more or less effort than others but that it was always the same 10 blocks that resulted in the tenfold increase in speed.

### 6.3 Conclusions

#### 6.3.1 Project results

How secure are these implementation recommendations? An educated guess is that they are quite conservative. What is important to remember is that the possibilities for parallelization have not been investigated but for example data reading may usually be parallelized far beyond the simulated one execution per clock cycle here given a wider data bus. Since this goes for most data treatment there is a good reason to believe that it is possible to go much
Table 6.1: Optimization suggestions from the post analysis of the JPEG profiling.

<table>
<thead>
<tr>
<th>Block name</th>
<th>File</th>
<th>Line</th>
<th>Part in algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>rgb_ycc_convert_2</td>
<td>jccolor.c</td>
<td>149</td>
<td>Color space Transform</td>
</tr>
<tr>
<td>jpeg_fdct_islow_4</td>
<td>jfdctint.c</td>
<td>220</td>
<td>DCT</td>
</tr>
<tr>
<td>jpeg_fdct_islow_1</td>
<td>jfdctint.c</td>
<td>155</td>
<td>DCT</td>
</tr>
<tr>
<td>get_24bit_row_1</td>
<td>rdbmp.c</td>
<td>170</td>
<td>Data read</td>
</tr>
<tr>
<td>forward_DCT_5</td>
<td>jcdctmgr.c</td>
<td>232</td>
<td>DCT</td>
</tr>
<tr>
<td>h2v2_downsample_2</td>
<td>jcsample.c</td>
<td>272</td>
<td>Image downsampling</td>
</tr>
<tr>
<td>forward_DCT_2</td>
<td>jcdctmgr.c</td>
<td>203</td>
<td>DCT</td>
</tr>
<tr>
<td>preload_image_6</td>
<td>rdbmp.c</td>
<td>212</td>
<td>Data read</td>
</tr>
<tr>
<td>forward_DCT_13</td>
<td>jcdctmgr.c</td>
<td>260</td>
<td>DCT</td>
</tr>
<tr>
<td>preload_image_4</td>
<td>rdbmp.c</td>
<td>210</td>
<td>Data read</td>
</tr>
</tbody>
</table>

Further. But the central point here is not how much we can actually increase the performance of the application. It is that we have a conservative estimate of how much is possible and at the very core of the issue: Where we should focus our efforts. Therefore the project is considered a success even though the software itself is far from finished.

As for the development of the profiler each new function added unveils a myriad of functions needed in order for the profiler to become usable. However, it is also my distinct impression that it is work that is work well spent. The Relief profiler at the moment is, more than anything else a good platform to stand on when developing the system co-design tools of tomorrow.

### 6.3.2 Problems with the software

There are three main areas in which the Relief profiler lacks today. These are listed below along with an estimation on the possibilities of resolving them.

#### Massive memory usage

As mentioned an image about the size of 1000 by 1000 pixels or larger will cause an instrumented JPEG encoder to use more than 4 GiB of memory. It is not impossible to use virtual memory for this but an ordinary Linux system will kill the process when it has allocated 2GiB and an x86-64 Linux kernel which was used in this case will kick the process out when memory allocation reaches 4 GiB.

This immense memory usage is clearly not acceptable if the profiler is to be used for profiling higher complexity algorithms such as the “264” family high definition video codecs.

The problem in this case is the Profiling library which produces execution
information on each execution of a basic block. Naturally this will consume massive amounts of information but it does not have to be as much as it is at present. The focus of the information collection algorithms was during the beginning of the project on speed. This decision will have to be reconsidered when continued development is discussed. Speed may also be preserved with a clever enough design of the library. Also, since memory usage in the vicinity of 4 GiB will require virtual memory on disk on most systems, the profiler might as well just print its data to file where there is space in abundance.

**Lack of pattern matching**

As mentioned in section 3.5 implementing a recurring set of operations usable in several high use blocks is more effective and cheaper than to implement several very block specific instructions.

However, finding the longest and most heavily used pattern of operations is very likely quite a complex combinatorial optimization problem and requires its very own research project. Instead this part will for the moment rely on the designers experience in finding likely patterns of operations that can then be tested against the code to see what the increase will be if they are implemented. This function has not yet been implemented. The patterns can be specified in the instrumentation configuration file and they will be tested against the code so the only thing missing is saving and analyzing the results of such a match.

An interesting variation to the pattern issue is taking patterns from the highest ranked blocks found in the profiling such as in the field test earlier and matching just their patterns or subpatterns against each other and possibly against the rest of the application. This would require extracting further data from the instrumentation phase for use in the analysis phase but that is fairly easy to introduce. It would be a key component when trying to minimize functional units but also in generally minimizing the silicon cost of the desired chip. This feature is also expected of the future development of the tool.

**Lack of automation**

Loading and optimizing the 18 test cases into the post execution analysis software turned out to be quite a slow and tedious task. Since having several execution cases is pretty much a requirement when dealing with dynamic profiling data this process needs to be further automated. Preferably by providing a terminal interface to the post analysis so that it can be scripted in a shell script thereby automating it. The output of these analysis runs should probably they too be automatically compared against each other in yet another step.
6.3.3 Future work

All the problems mentioned in 6.3.2 should naturally be dealt with in the future development of the Relief profiler.

One can imagine that at the heart of a future version of the post analysis profiler would be the Target Optimize feature. Today it is concerned only with total number of clock cycles required to finish the execution. Another central part of the ASIP design flow is minimizing silicon usage. Naturally therefore a future target would include such things as maximum number of functional units such as multipliers.

The next really big thing to introduce is a memory profiling technologies. One very important feature of custom ASIP design is the possibility to hand craft a memory subsystem, tailor made for the application and this process can highly benefit from an automated generation of understanding of the application.

6.3.4 Final words

So does the future of automated instruction set design spell profiling? There is not yet enough concrete evidence to make that claim but there is certainly such a considerable lack of evidence to the contrary to motivate further explorations of the field.
Index

µProfiler, 8
ASIC, 1
ASIP, 1
automaton, 2
Basic block, 16
co-design, 7
Compiler
Front end, 18
Lexical analysis, 18
Middle end, 19
Parsing, 18
Phases, 18
Syntax tree, 18
DCT, 55
design automation, 5
design flow, 5
Discrete Cosine Transform, 55
EDA, 8
Entropy Coding, 55
Function pointer, 13
GNU Profiler, 14
GNU profiler, 8
Gprof, 8
Instrumentation, 13
JPEG, 54
libJPEG, 54
Profiler
Definition, 7
Abstraction level, 12
Dynamic, 14
Granularity, 16
Static, 11
Profiling, 11
Accuracy, 12
HW specific, 10
Model based, 10
requirements, 2
RLE, 55
Run Length Encoding, 55
Valgrind, 14
WCET, 9
YUV, 55
List of Figures

2.1 A rough overview of the ASIP design flow. . . . . . . . . . 6
2.2 A flow chart representation of the profiler definition. . . . . 7

3.1 Profiling accuracy depending on language. . . . . . . . . . . . 13
3.2 The list of tokens is converted into a tree . . . . . . . . . . . 19

4.1 A schematic view of the Relief workflow for the user . . . . . . . 21
4.2 The open window, only dynamic data files are allowed to be
opened. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.3 The post analysis in action, the left frame contains all the
blocks sorted by name and the right is a printout of the blocks
sorted by operations consumed during the execution denoted
by the dynamic data xml-file. . . . . . . . . . . . . . . . . . . . . . 31
4.4 A printout of an expanded basic block tree. . . . . . . . . . . 32
4.5 A printout of the top part of the right window. . . . . . . . . . 33
4.6 A printout of the top part of the right window after performing
an optimization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5.1 Relations between source, compiler, program and the two li-
braries. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2 Large scale design flow overview . . . . . . . . . . . . . . . . . 37
5.3 Schematic of the basic block code instrumentation. . . . . . . 41
5.4 Dependency graph visualization generated by dot
based on basic block dependencies in the code. . . . . . . . . . . 45
5.5 Basic functionality of the profiler library in cooperation
with program instrumentations. . . . . . . . . . . . . . . . . . . . . 47
5.6 Illustration of the time item structure. . . . . . . . . . . . . . . 49
5.7 Basic architecture of the post analysis software. . . . . . . . . . 52
List of Tables

6.1 Optimization suggestions from the post analysis of the JPEG profiling. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Function pointers may be deceptive when syntactically parsed.</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Square block.</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>MAC block.</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>The build command invokes the instrumenter library</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Opt-in profiling configuration.</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Opt-out profiling configuration.</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>A simple instruction pattern.</td>
<td>27</td>
</tr>
<tr>
<td>4.5</td>
<td>Cutout from a dynamic data file.</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>The outout from the graph aggregation process.</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Output from a test run of a profiling run.</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of static XML data in an .stat.xml file.</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>insert a function declaration into a source file.</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Insert a function call into a basic block.</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Static data from a .stat.xml file.</td>
<td>44</td>
</tr>
<tr>
<td>5.6</td>
<td>A subgraph from a .dot-file.</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>instrumenter.conf.</td>
<td>56</td>
</tr>
</tbody>
</table>
References


Appendix A

Optimizing suggestions from 18 different encoding runs

Test1 pid: 13963
Consumed cycles: 8632048 or 9.15744078027%

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jc dctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
forward_DCT_2: jc dctmgr.c : 203
preload_image_6: rdbmp.c : 212
forward_DCT_13: jc dctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test2 pid: 13969
Consumed cycles: 48143004 or 8.37916938004%

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jc dctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
preload_image_6: rdbmp.c : 212
forward_DCT_2: jc dctmgr.c : 203
forward_DCT_13: jc dctmgr.c : 260
preload_image_4: rdbmp.c : 210
Test3 pid: 14002
Consumed cycles: 48339440 or 8.4104830979%

rgb_ycc_convert_2: jccolor.c: 149
jpeg_fdct_islow_4: jfdctint.c: 220
jpeg_fdct_islow_1: jfdctint.c: 155
get_24bit_row_1: rdbmp.c: 170
forward_DCT_5: jcdctmgr.c: 232
h2v2_downsample_2: jcsample.c: 272
preload_image_6: rdbmp.c: 212
forward_DCT_2: jcdctmgr.c: 203
forward_DCT_13: jcdctmgr.c: 260
preload_image_4: rdbmp.c: 210

Test4 pid: 14053
Consumed cycles: 50188719 or 8.40244492778%

rgb_ycc_convert_2: jccolor.c: 149
jpeg_fdct_islow_4: jfdctint.c: 220
jpeg_fdct_islow_1: jfdctint.c: 155
get_24bit_row_1: rdbmp.c: 170
forward_DCT_5: jcdctmgr.c: 232
h2v2_downsample_2: jcsample.c: 272
preload_image_6: rdbmp.c: 212
forward_DCT_2: jcdctmgr.c: 203
forward_DCT_13: jcdctmgr.c: 260
preload_image_4: rdbmp.c: 210

Test5 pid: 14087
Consumed cycles: 62409514 or 8.55408790393%

rgb_ycc_convert_2: jccolor.c: 149
jpeg_fdct_islow_4: jfdctint.c: 220
jpeg_fdct_islow_1: jfdctint.c: 155
get_24bit_row_1: rdbmp.c: 170
forward_DCT_5: jcdctmgr.c: 232
h2v2_downsample_2: jcsample.c: 272
preload_image_6: rdbmp.c: 212
forward_DCT_2: jcdctmgr.c: 203
forward_DCT_13: jcdctmgr.c: 260
preload_image_4: rdbmp.c: 210

Test6 pid: 14131
Appendix A. Optimizing suggestions from 18 different encoding runs

Consumed cycles: 32878291 or 8.34313688071%

---

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
preload_image_6: rdbmp.c : 212
forward_DCT_2: jcdctmgr.c : 203
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test7 pid : 14148
Consumed cycles: 33411929 or 8.30290632184%

---

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
forward_DCT_2: jcdctmgr.c : 203
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test8 pid : 14161
Consumed cycles: 43080598 or 8.61961864799%

---

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
forward_DCT_2: jcdctmgr.c : 203
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test9 pid : 14185
Consumed cycles: 66081760 or 8.34809876609%
rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
preload_image_6: rdbmp.c : 212
forward_DCT_2: jcdctmgr.c : 203
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test10_pid: 14258
Consumed cycles: 36082780 or 8.78624794376%

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
forward_DCT_2: jcdctmgr.c : 203
preload_image_6: rdbmp.c : 212
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test11_pid: 14278
Consumed cycles: 36647082 or 8.68986874213%

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
jpeg_fdct_islow_1: jfdctint.c : 155
get_24bit_row_1: rdbmp.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2_downsample_2: jcsample.c : 272
forward_DCT_2: jcdctmgr.c : 203
preload_image_6: rdbmp.c : 212
forward_DCT_13: jcdctmgr.c : 260
preload_image_4: rdbmp.c : 210

Test12_pid: 14295
Consumed cycles: 65377057 or 8.32770768469%

rgb_ycc_convert_2: jccolor.c : 149
jpeg_fdct_islow_4: jfdctint.c : 220
<table>
<thead>
<tr>
<th>Function</th>
<th>Source File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg_fdct_islow_1</td>
<td>jfdctint.c</td>
<td>155</td>
</tr>
<tr>
<td>get_24bit_row_1</td>
<td>rdbmp.c</td>
<td>170</td>
</tr>
<tr>
<td>forward_DCT_5</td>
<td>jcdctmgr.c</td>
<td>232</td>
</tr>
<tr>
<td>h2v2_downsample_2</td>
<td>jcsample.c</td>
<td>272</td>
</tr>
<tr>
<td>preload_image_6</td>
<td>rdbmp.c</td>
<td>212</td>
</tr>
<tr>
<td>forward_DCT_2</td>
<td>jcdctmgr.c</td>
<td>203</td>
</tr>
<tr>
<td>forward_DCT_13</td>
<td>jcdctmgr.c</td>
<td>260</td>
</tr>
<tr>
<td>preload_image_4</td>
<td>rdbmp.c</td>
<td>210</td>
</tr>
</tbody>
</table>

Test13 pid: 14344
Consumed cycles: 65273474 or 8.35601997255%

<table>
<thead>
<tr>
<th>Function</th>
<th>Source File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg_fdct_islow_4</td>
<td>jfdctint.c</td>
<td>220</td>
</tr>
<tr>
<td>jpeg_fdct_islow_1</td>
<td>jfdctint.c</td>
<td>155</td>
</tr>
<tr>
<td>get_24bit_row_1</td>
<td>rdbmp.c</td>
<td>170</td>
</tr>
<tr>
<td>forward_DCT_5</td>
<td>jcdctmgr.c</td>
<td>232</td>
</tr>
<tr>
<td>h2v2_downsample_2</td>
<td>jcsample.c</td>
<td>272</td>
</tr>
<tr>
<td>preload_image_6</td>
<td>rdbmp.c</td>
<td>212</td>
</tr>
<tr>
<td>forward_DCT_2</td>
<td>jcdctmgr.c</td>
<td>203</td>
</tr>
<tr>
<td>forward_DCT_13</td>
<td>jcdctmgr.c</td>
<td>260</td>
</tr>
<tr>
<td>preload_image_4</td>
<td>rdbmp.c</td>
<td>210</td>
</tr>
</tbody>
</table>

Test14 pid: 14380
Consumed cycles: 57058839 or 8.38528077117%

<table>
<thead>
<tr>
<th>Function</th>
<th>Source File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg_fdct_islow_4</td>
<td>jfdctint.c</td>
<td>220</td>
</tr>
<tr>
<td>jpeg_fdct_islow_1</td>
<td>jfdctint.c</td>
<td>155</td>
</tr>
<tr>
<td>get_24bit_row_1</td>
<td>rdbmp.c</td>
<td>170</td>
</tr>
<tr>
<td>forward_DCT_5</td>
<td>jcdctmgr.c</td>
<td>232</td>
</tr>
<tr>
<td>h2v2_downsample_2</td>
<td>jcsample.c</td>
<td>272</td>
</tr>
<tr>
<td>preload_image_6</td>
<td>rdbmp.c</td>
<td>212</td>
</tr>
<tr>
<td>forward_DCT_2</td>
<td>jcdctmgr.c</td>
<td>203</td>
</tr>
<tr>
<td>forward_DCT_13</td>
<td>jcdctmgr.c</td>
<td>260</td>
</tr>
<tr>
<td>preload_image_4</td>
<td>rdbmp.c</td>
<td>210</td>
</tr>
</tbody>
</table>

Test15 pid: 14431
Consumed cycles: 54872335 or 8.49903443879%
<table>
<thead>
<tr>
<th>Test</th>
<th>PID</th>
<th>Consumed Cycles</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>14489</td>
<td>17,224,476</td>
<td>8.53%</td>
</tr>
<tr>
<td>17</td>
<td>14502</td>
<td>57,790,946</td>
<td>8.48%</td>
</tr>
<tr>
<td>18</td>
<td>14538</td>
<td>28,679,559</td>
<td>8.71%</td>
</tr>
</tbody>
</table>

```c
forward_DCT_5: jcdctmgr.c : 232
h2v2.downsample_2: jcsample.c : 272
preload_image_6: rd BMP.c : 212
forward_DCT_2: jcdctmgr.c : 203
forward_DCT_13: jcdctmgr.c : 260
```

```c
Test16 pid: 14489
Consumed cycles: 17224476 or 8.53317658299%
```

```c
rg b_y cc_con verts_2: jcc olor.c : 149
jpeg_fdct_is low_4: jfdctint.c : 220
jpeg_fdct_is low_1: jfdctint.c : 155
g et_24bit_row_1: rd BMP.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2.downsample_2: jcsample.c : 272
preload_image_6: rd BMP.c : 212
```

```c
Test17 pid: 14502
Consumed cycles: 57790946 or 8.48854536484%
```

```c
rg b_y cc_con verts_2: jcc olor.c : 149
jpeg_fdct_is low_4: jfdctint.c : 220
jpeg_fdct_is low_1: jfdctint.c : 155
g et_24bit_row_1: rd BMP.c : 170
forward_DCT_5: jcdctmgr.c : 232
h2v2.downsample_2: jcsample.c : 272
preload_image_6: rd BMP.c : 212
```

```c
Test18 pid: 14538
Consumed cycles: 28679559 or 8.71197763612%
```

```c
rg b_y cc_con verts_2: jcc olor.c : 149
jpeg_fdct_is low_4: jfdctint.c : 220
jpeg_fdct_is low_1: jfdctint.c : 155
g et_24bit_row_1: rd BMP.c : 170
```
forward_DCT_2: jcdctmgr.c : 203
preload.image_6: rdbmp.c : 212
forward_DCT_13: jcdctmgr.c : 260
preload.image_4: rdbmp.c : 210
Copyright

Svenska
Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära 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 icke-kommersiell 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.

Uphovsmannens ideella rätt innefattar rätt att bli nämnad som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd 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/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time 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/

© Björn Skoglund
Linköping, February 27, 2007