Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2011-07

M.Sc. Thesis

A Memory Access and Operator Usage Profiler Framework for HLS Optimization

C. Feenstra, B.Eng.

Abstract

As reconfigurable hardware such as FPGA's become bigger and bigger, large and complex systems can be implemented in such devices. It becomes a challenge for engineers to manually convert an algorithm in an HDL, considering the pushing time-to-market constraints. High Level Synthesis tools are developed to make this process less laborious. HLS tools use the original source code and transforms this to a hardware description. The quality of the original source code is of great influence for the resulting hardware.

In many data intensive applications, memory accesses form a bottleneck. To improve the performance of the hardware implementation, the execution behavoir of these accesses must first be optimized in the software source code. While doing this, an analyzer providing crucial information about the algorithm itself helps reduce engineering time.

This thesis work presents a framework which is capable of providing information about memory accesses and operations executed within an algorithm. The reports containing this information can be generated on a per function or per loop basis. This enables the engineer to find loop specific information, which can be used to optimize the algorithm and to provide crucial pipeline information to the HLS tool. An Optical Flow algorithm is used as case study to demonstrate the functionality of the framework. A massive speedup of a factor of 13.7 was achieved while the area increased only with a factor of 1.47. This demonstrates the effectiveness of the presented framework.

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

A Memory Access and Operator Usage Profiler Framework for HLS Optimization

Using the Lucas Optical Flow Algorithm as Case Study

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

C. Feenstra, B.Eng.
born in Dordrecht, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"A Memory Access and Operator Usage Profiler Framework for HLS Optimization"** by **C. Feenstra, B.Eng.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 16 December 2011

Chairman: ______________________
prof.dr.ir. A.J. van der Veen

Advisor: ______________________
dr.ir. T.G.R. van Leuken

Committee Members: ______________________
dr. K.L.M. Bertels

____
ir. A.C. de Graaf

Abstract

As reconfigurable hardware such as FPGA's become bigger and bigger, large and complex systems can be implemented in such devices. It becomes a challenge for engineers to manually convert an algorithm in an HDL, considering the pushing time-to-market constraints. High Level Synthesis tools are developed to make this process less laborious. HLS tools use the original source code and transforms this to a hardware description. The quality of the original source code is of great influence for the resulting hardware.

In many data intensive applications, memory accesses form a bottleneck. To improve the performance of the hardware implementation, the execution behavoir of these accesses must first be optimized in the software source code. While doing this, an analyzer providing crucial information about the algorithm itself helps reduce engineering time.

This thesis work presents a framework which is capable of providing information about memory accesses and operations executed within an algorithm. The reports containing this information can be generated on a per function or per loop basis. This enables the engineer to find loop specific information, which can be used to optimize the algorithm and to provide crucial pipeline information to the HLS tool. An Optical Flow algorithm is used as case study to demonstrate the functionality of the framework. A massive speedup of a factor of 13.7 was achieved while the area increased only with a factor of 1.47. This demonstrates the effectiveness of the presented framework.

Acknowledgments

It is a pleasure to thank all people who made this Master thesis possible.

First of all, I want to thank my advisor dr.ir. Rene van Leuken for his great help during every phase of this thesis. In our first meeting you showed me all kind of subjects available for a Master thesis within the Circuits and Systems group. In this meeting you took all the time to find the right project for me. This dedication never changed. In all our meetings where we discussed progress and results, you always invested time and provided excellent feedback by asking the right questions or popping up ideas. Many thanks for reviewing this thesis report.

The second person I want to thank is my classmate Arjan Kodde. Arjan, thank you for all lunchtime walks where we discussed each others progress and results and for discussing all kind of things nothing to do with any thesis. You also took the time to review this thesis work in great depth, providing excellent feedback.

Finally, I want to thank my family for supporting me during this thesis work. Both my parents encouraged me in times I needed it. My brothers Timon and Elian always provided the distraction I so longed for sometimes. Thank you all for being there.

C. Feenstra, B.Eng.
Delft, The Netherlands
16 December 2011

Contents

List of Figures

List of Tables

Introduction 1

This thesis work presents a framework which can be used to analyze algorithms that are to be implemented in hardware using High Level Synthesis.

In the first part of this chapter, a short introduction to High Level Synthesis as well as the common challenges a designer faces when implementing an algorithm with High Level Synthesis are given. The need for analyzing algorithms will become clear.

The second part of this chapter presents a short introduction to Optical Flow algorithms. These algorithms are computational intensive and they involve a lot of memory accesses. Hence, they are good candidates to implement in hardware using High Level Synthesis. An Optical Flow algorithm has been used as a case study for the presented framework. In this case study, the framework proved to be very effective. An overall speedup of 13.7x was achieved, while the used area increased with a factor of only 1.47.

Finally the goals and contributions are presented.

1.1 High Level Synthesis

Hardware can be described in a number of languages e.g. *VHDL*, *Verilog* and *SystemC*. When describing hardware, all information has to be provided in such a language. For example, the data path has to be described completely and the operations must be given. The data path and operations which have to be done on the data are then combined in a certain schedule. Designing a schedule is a challenging job. The way an engineer schedules has direct consequences on the maximum clock frequency, latency, throughput, etc. When designing hardware using these languages, one has to keep the destination platform in mind constantly. This is exactly where the biggest problem arises. Not many algorithms are specially designed for hardware implementation. Often they are programmed in *C* or *Matlab* code.

C code can describe an algorithm very well, but it contains no information about scheduling. This is because most *C* code is written for use with a sequential processor. Translating such an algorithm to real hardware is quite challenging and involves a lot of labor. In most cases, a good start is finding the data path in the algorithm. When the data path is found, the operations have to be defined. Finally, the operations must be scheduled in a way in which the resulting hardware meets the requirements the best.

Since a couple of years, experiments with High Level Synthesis (HLS) are published. The main goal is to be able to describe the behavior of the algorithm and let a synthesis tool do the complex work of determining the data path, operations and finding a suitable schedule. The algorithms can now be described in plain *C* code. The HLS tool can convert the algorithms to a hardware description language (HDL). Today, a couple of tools are actually developed to do this. One example is *Catapult C* from *Mentor Graphics*. This tool can translate given *C* code to a hardware implementation. Although this sounds as great as the invention of HDL's themselves, there are some challenges. The generated solution is only as good as the provided *C* code. Therefore, the *C* code has to be optimized for hardware use, to improve the quality of the hardware solution.

The biggest bottleneck in many data intensive applications will be memory accesses. If for instance the *C* code iterates over an array sequentially, the hardware cannot run that code concurrently without doing any special optimizations, since the number of ports to the RAM module is limited and in many cases only one. If the memory has one port and can provide data at each clock cycle, then for n iterations, the algorithm needs at least n clock cycles.

Another challenge is the usage of operators. When certain code runs sequentially, only one operator runs at a time (neglecting coprocessors, such as floating point processors as the MIPS architecture has). Therefore, one adder, one multiplier, etc. might be sufficient. When the algorithm is implemented in hardware, more operators can run at the same time. The challenge is how to find where these operators are used and what number of operators are needed to best meet the requirements. Other important subjects are data types. In *C* code one can represent an integer in only a couple formats. Among them are `char`, `short`, `int` and `long` consisting in many cases of 8, 16, 32 and 64 bit respectively (depending on the targeted architecture). In hardware, all number of bits are possible and in fact desirable. Area is saved by only using the minimal amount of bits needed.

It is clear that finding information about an algorithm is very important when that algorithm is to be implemented in hardware using High Level Synthesis.

1.2 Optical Flow Used Everywhere

Humans and almost all other living creatures use a form of vision to interpret the world around them. In most cases this information is used for finding danger, which for example allows these creatures to prevent them from getting eaten or falling off a cliff. In the case of humans, vision can be a form of entertainment as well. A lot of things we do to enjoy ourselves are based on vision, e.g. watching television.

Translating images to a perspective of what happens around us is not as simple as it might appear. Our brain does an incredible job when it comes to finding movements and the direction and speed of these movements. We can focus on an object and calculate this information real time. For computers however, this is not that straight forward. Detecting movement in a couple of images might not be too hard to implement, but finding the speed and direction of the motion is. When computers use this kind of information, it is called Computer Vision. One of the most obvious applications might be robot eye sight. A lot more applications are using Computer Vision, as will be discussed in the next section.

The algorithms which translate information in images to speed and direction of motions are called Optical Flow algorithms. In many cases the used images are in fact frames of a video generated with a camera. But other techniques can be used as well, e.g. radar. The main challenge in Optical Flow algorithms is finding the correlation between a number of images. In general, Optical Flow algorithms use direct pixel information to find this correlation. As a result, two main limitations will follow [1]. The first one is the so called lumination constraint. This constraint states that the lumination in different images should be constant. This is necessary because pixels in two different images should match each other if they represent the same object in the image. The second limitation is the speed of the movement. This speed can be expressed in pixels / frame. Because the search space per pixel to its related pixel in another image is often limited, the pixel speed should not exceed this search space. Both of these constraints are further explained in Appendix A.

1.2.1 Example Optical Flow Applications

As mentioned above, Optical Flow is used a lot these days. Some examples of the everyday use of Optical Flow algorithms are given below.

- Song and Huang [2] proposed an algorithm for their robot guide dog for blind people. The robot searches for obstacles and gives information to its user about them. An Optical Flow algorithm is used for finding the obstacles and estimating depth.
- Another application of optical flow has been developed by Gern et al. of Daimler-Chrysler [3]. Cars can be equipped with a lane recognition system. This system can warn the driver of a vehicle in case the vehicle is moving outside the lane it is in. An even more sophisticated use of this system can be the autonomous driving of the vehicle. In this case, the vehicle steers itself to keep within the current lane. Most of these systems use some kind of lane markings recognition. But in severe weather, the lane markings may not be visible enough. Gern et al. suggest to make use of horizontal Optical Flow. Doing this, all kind of structures parallel to the lane (e.g. crash barriers or oil traces) can be used to determine the position of the vehicle with respect to the lane.
- Monteiro et al. [4] proposed a solution for finding wrong way drivers. They use the Lucas & Kanade [5] algorithm for detecting the driving direction. The proposed algorithm first learns the normal (intended) driving direction. When the learning process is completed, the system is ready for detecting wrong way drivers. If a moving object is discovered moving in the opposite direction of what was learned earlier, a validation process is started. When the movement is indeed found to be a wrong way driving vehicle, action can be taken.
- In a soccer game, group action is a subject of interest to viewers. Kong et al. [6] proposed a way to determine group action out of video images. The images captured by one camera are used. This camera may only pan and tilt to focus

on the ball. The group action is represented by moving players at the field (local movement). Besides that, the camera is moving as well (global movement). Kong et al. present a way to extract the global movement from the sequence while the local movement (i.e. group action) is preserved.

- As the number of security cameras grows in public buildings or in cities, automated crowd density detection is becoming more wanted. Kim et al. [7] proposed a system which is able to estimate the crowd density. Kim et al. use an Optical Flow algorithm (Horn & Schunck [8]) and an edge detection algorithm to estimate the number of people in a certain scene.

1.3 Combining Optical Flow and HLS

Optical Flow algorithms are complex algorithms. In many cases these algorithms are already programmed in the *C* or *C++* languages. Because these optical flow algorithms involve a lot of calculations on a large data stream, powerful systems are required to be able to provide enough throughput. In this case, throughput will translate to frames per second (or FPS).

Optical Flow algorithms are divided into a couple of stages. Each stage is doing certain calculations on the data to prepare it for the next stage. Most stages are doing certain calculations on all pixels. More about the working of Optical Flow algorithms and more information about different stages within these algorithms are given in Appendix A. It might be clear that a lot of parallelism can be implemented. Functional oriented parallelism can be used to run different stages concurrently, while data oriented parallelism can be used to run multiple instances of a stage core. An HLS tool like *Catapult C* from *Mentor Graphics* can be used to 'easily' convert the given *C* source code of the algorithm to actual hardware. Of course, the limitations stated in Section 1.1 must be taken into account.

1.4 Goals

The main goal of this thesis work is to provide a framework that allows an HLS designer to retrieve useful information about an algorithm which has to be implemented into hardware using HLS. The information presented by the framework should provide the insight to the algorithm being analyzed. This enables the engineer to improve an algorithm efficiently. Two sub goals results from this:

- The framework must be able to retrieve memory access and operator usage information. It must be able to present this information on a per function and per loop basis.
- To proof the correct functionality and the effectiveness of the framework, it must be used on a real use case: the Lucas Optical Flow [5] algorithm. This algorithm is provided and must be optimized using the framework to develop.

1.5 Contributions

The following contributions were achieved by implementing the framework presented in this thesis:

- The presented framework can provide essential information about an algorithm written in *C* using a GCC PlugIn.
- The framework is able to retrieve memory access and operator usage information. The user can request this information interactively or by passing commands as arguments to the analyzer.
- It is possible to easily expand the framework with new types of reports.
- The framework has the ability to retrieve loop information from an algorithm. Memory access and operator usage reports provide the option to report only information from a certain loop.
- All access reports provide location information about where the statements exist that caused a memory access or that used an operator in the original source code.
- Arrays are detected by the framework. This is of great importance, since these are often mapped to memory in practice.
- The correct working and capabilities of the framework are presented by analyzing and optimizing the Lucas Optical Flow algorithm. A massive speedup of 13.7x has been achieved.

1.6 Outline

The rest of this thesis report is organized as follows. Chapter 2 describes the background of HLS. What optimizations can be done, the related work on HLS optimizations and how to apply them using *Catapult C* is covered in this chapter. Chapter 3 discusses how information about an algorithm can be retrieved and a choice is made on what method to use for the framework. Chapter 4 describes the design of the framework in great detail. From original algorithm till the generation of the reports is covered. Chapter 5 uses the Lucas Optical Flow algorithm to demonstrate the usage and capabilities of the framework. A great performance improvement compared to a previous implementation of this algorithm is presented. Chapter 6 concludes the work and gives suggestions about future work.

Appendix A provides background information about Optical Flow. Different methods of calculating Optical Flow along with different Optical Flow algorithms are presented. Appendix B defines the format used in the log files. Finally, Appendix C provides the description of the Analyze Library designed for use with the framework.

HLS Background and Related Work 2

High Level Synthesis tools like *Catapult C* are very complex tools. Most steps to convert the input *C* code to an RTL level output are done automatically. In some cases, the engineer can change parameters to influence the resulting RTL output. An example of this is the scheduling part of *Catapult C*. The engineer can tune the schedule to meet the design goals. Although *Catapult C* does most of the job automatically, some input must still be provided by the engineer. Examples are loop unroll factor, pipeline initiation interval, and word width of variables. In some data intensive parts of an algorithm, the memory access profile can have a great influence on the overall performance. Making some changes in the design to help decrease the needed memory bandwidth can help the scheduler to produce better (faster) results. In this chapter, on most of the tasks that have to be done manually will be zoomed in to see what others have done in the past.

2.1 Loop Unrolling

Loop unrolling is one of the most commonly used techniques for loop optimization. An example of loop unrolling is given in figure 2.1.

```
for (int i = 0; i < N; i++)
{
  A[i] = B[i] + C[i];
}
```

```
for (int i = 0; i < N; i+=2)
{
  A[i] = B[i] + C[i];
  A[i+1] = B[i+1] + C[i+1];
}
```

Figure 2.1: A simple for-loop (left) and its unrolled variant (right)

When using loop unrolling, the body of the loop is replicated u times, where u is called the unroll factor. When u is not a multiple of the loop bound N, the transformed code need to include an epilogue that handles the iterations not included in the unrolled part. If the unroll factor is equal to 1, the resulting loop will be the same as its original version, i.e. the loop is not unrolled. When the unroll factor is equal to the loop bound, the loop is known to be fully unrolled. In this case the body of the loop is replicated as many times as the number of iterations of the loop. Therefore, a loop can only be fully unrolled if the number of iterations is known at compile time.

Initially, loop unrolling was used to reduce control overhead. But as processing units get faster, the number of cycles used per operation differs more. Especially reading from memory or complex arithmetic operations consume a lot of time. This will result in stalls, i.e. one instruction waits for the previous one to finish because of data dependencies. By unrolling loops, independent instructions can be executed while other instructions must wait for data to become available. This results in less stalls per loop compared to the not unrolled variant. This phenomenon is called instruction-level parallelism (ILP). Because ILP can dramatically affect the overall number of cycles needed to execute a loop, it often outperforms the performance gained by minimizing control overhead.

Loop unrolling has been used for decades within software compilers to minimize loop control overhead and reducing branch penalties. Therefore, most literature on loop unrolling is focused on software (i.e. using some kind of processor). Cardoso and Diniz [9] have written a nice book about compiling source code to design reconfigurable hardware, such as FPGA's. A lot of optimization techniques are covered, including loop unrolling.

2.1.1 Unroll Factor

Although loop unrolling was introduced for optimizing software related loops, it can easily be adopted for hardware. *Catapult C* offers an unrolling feature for each loop. The engineer must provide the unroll factor on a per loop basis. Finding the best unroll factor is not an easy job and depends mainly on three parameters. First of all, loop unrolling will naturally result in more hardware, since the body of the loop is replicated u times. Another obstacle is the memory access. In most data oriented algorithms, memory access will most likely be the bottleneck of the overall performance. Unrolling a loop will result in doing more in less time, hence more memory accesses occur in less time. The third parameter to keep in mind is the gained speedup. If unrolling a loop (further) will not provide much speedup (e.g. all available memory bandwidth is used), the used area increases at minimal performance increase, which probably results in unfair costs. Dragomir, Moscu-Panainte, Bertels and Wong [10] have proposed a methodology to determine the unroll factor given the above three parameters. The basic idea is to find an unroll factor bound per parameter. By later on combining these bounds, the optimal unroll factor can be found. The following sections describe the idea proposed in [10]. Some of the writers of [10] have worked out their work even more by including loop shifting. More information on that can be found in [11]. Because the research in finding an optimal unroll factor in this thesis is driven by the need of providing such a value to *Catapult C*, loop shifting was not further investigated.

The next couple of sections describe the three constraints involved when unrolling loops for use in hardware as described in [10].

2.1.1.1 Area Constraint

As mentioned earlier, by unrolling a loop, the consumed area grows. By only taking into account the area constraint and not the shape of the design, an upper bound can be found by:

$$u_a = \left\lfloor \frac{Area_{(available)}}{Area_{(K)} + Area_{(interconnect)}} \right\rfloor \tag{2.1}$$

where $Area_{(available)}$ is the amount of area available for this loop. $Area_{(interconnect)}$ is the area necessary to connect one kernel to the rest of the design (the assumption has been made that the overall interconnect area grows linearly with the amount of kernels). $Area_{(K)}$ is the area initialized by one instance of the kernel, including the storage space for the values read from the shared memory.

2.1.1.2 Memory Accesses Constraint

Let's consider T_r, T_w and T_c to be the times necessary to read memory, write memory and do the computations on hardware for kernel K respectively. The total time used by K then is $T_r + T_w + T_c$. The assumption is made that memory reads are performed at the beginning and memory writes at the end. When considered is that $T_w \leq T_r < T_c$ and $T_r + T_w > T_c$, a new instance of kernel K can only be started after T_r time. Furthermore, the condition that memory access request from different kernels should not overlap, sets another bound (u_m) for the unroll factor:

$$u \cdot \min(T_r, T_w) \leq \min(T_r, T_w) + T_c \Rightarrow u \leq u_m = \left\lfloor \frac{T_c}{\min(T_r, T_w)} \right\rfloor + 1 \tag{2.2}$$

The time needed for running u instances of K is:

$$T_{K(hw)}(u) = \begin{cases} T_c + T_r + T_w + (u-1) \cdot \max(T_r, T_w) & \text{if } u < u_m \\ u \cdot (T_r + T_w) & \text{if } u \geq u_m \end{cases} \tag{2.3}$$

Obeying the constraint $u < u_m$, the time needed for u instances of K can be derived from 2.3:

$$T_{K(hw)}(u) = T_c + \min(T_r, T_w) + u \cdot \max(T_r, T_w) \tag{2.4}$$

2.1.1.3 Speedup Constraint

Before the speedup constraint can be discussed, the speedup calculations must be defined. To calculate the time it takes to execute the loop without loop unrolling, the following formulas can be used:

$$T_{loop(\text{with unrolling})}(u) = T_{loop}(u) = (T_{sw} + \max(T_r, T_w)) \cdot N + (T_c + \min(T_r, T_w)) \cdot N(u) \tag{2.5}$$

$$T_{loop(\text{without unrolling})} = T_{loop}(1) = (T_{sw} + T_r + T_c + T_w) \cdot N \tag{2.6}$$

where T_{sw} is the time the unparallelizable part of the loop takes to execute, N is the number of loop iterations and $N(u)$ is the number of loop iterations per kernel K for the given unroll factor u. As can be seen, equation 2.6 can be derived from equation 2.5 by using unroll factor $u = 1$. The speedup at loop nest level now is:

$$S_{loop}(u) = \frac{T_{loop}(1)}{T_{loop}(u)} \tag{2.7}$$

While still satisfying $u < u_m$, $T_{loop}(u)$ is a monotonic decreasing function. Because $T_{loop}(1)$ is a constant, $S_{loop}(u)$ is a monotonic increasing function. Now a new parameter is introduced: the calibration factor F, a positive number decided by the application designer. F determines a limitation of the unroll factor according to the desired trade-off. This helps prevent the area to increase much more than the speedup would increase. A new formula can be constructed with this factor:

$$\Delta S(u+1,u) > \Delta A(u+1,u) \cdot F \tag{2.8}$$

where $\Delta S(u+1,u)$ is the relative speedup increase between the unroll factors u and $u+1$:

$$\Delta S(u+1,u) = \frac{S(u+1) - S(u)}{S(u)} \cdot 100[\%] \tag{2.9}$$

and $\Delta A(u+1,u)$ is the area increase. Since all kernels are identical, the consumed area increases linearly with the number of kernels used. Therefore, $\Delta A(u+1,u) = Area_{(K)}$. This results in a new unroll factor bound:

$$u_s = \min(u) \text{ such that } \Delta S(u+1,u) < F \cdot Area_{(K)} \tag{2.10}$$

Local optimal values for unroll factor u could appear if u is not a divisor of N, but $u+1$ is. To avoid this situation, another condition for u_s is added: $\Delta S(u_s+2, u_s+1) < F \cdot Area_{(K)}$.

By using formulas 2.5, 2.6 and 2.7, the total speedup can be calculated by:

$$S_{loop}(u) = \frac{\frac{T_{sw}+T_{K(sw)}}{\max(T_r,T_w)+T_{sw}}}{1 + \frac{T_c+\min(T_r,T_w)}{(\max(T_r,T_w)+T_{sw})\cdot N} \cdot N(u)} \tag{2.11}$$

Given speedup formula 2.11 and the fact that the maximum unroll factor equals the number of iterations N (fully unrolled), u_s can be found in $O(\log N)$ time using binary search.

2.1.1.4 Combining Area, Memory Accesses and Speedup Constraints

Using the three constraints (u_a, u_m and u_s), the optimal unroll factor can be found by:

$$u_{optimal} = \begin{cases} \min(u) \text{ such that } u_s < u \leq \min(u_a, u_m) & \text{if } u_s < \min(u_a, u_m) \\ \max(u) \text{ such that } u \leq \min(u_a, u_m) & \text{if } u_s \geq \min(u_a, u_m) \end{cases} \tag{2.12}$$

The resulting unroll factor can be used when unrolling the loop with an HLS tool like *Catapult C*.

2.1.2 Code Style Recommendations

Fingeroff [12] wrote a blue book about High Level Synthesis (and actually called it like this). This work focuses more on the consequences the original *C* code has on the resulting RTL. When it comes to loops, a lot can be said. It is best to use constant bounds wherever possible. This means that the initialization of the iterator should be a constant, the test condition should be against a constant and the iterator should be incremented with a constant. When doing this, the synthesis tool chain will be able to find all bounds. When for example the test condition is done against a variable of type `int`, the synthesis tool chain will not be able to determine the bounds and therefore chooses the worst case bounds, which will be the full range of an integer (usually 32 bits). This issue can be coped with by providing the real worst case bound in the test condition. Then, by using a conditioned break statement, the execution of the loop can be terminated.

Nested loops can be unrolled as well. Normally, nested loops which are still rolled and not pipelined have a lot of overhead. This is caused by the condition checks of the outer loop(s). If for example the outer loop exists of 2 iterations and the inner loop exists of 4 iterations and assuming that the calculations are only done in the inner loop and take one cycle, the total number of cycles actually doing the calculations is 8. But the outer loop needs 2 additional cycles each iteration to increment the iterator and to check the exit condition. Also 2 additional cycles are needed by the main loop (i.e. the function itself). A total of 14 cycles are needed for this nested loop to complete $(2 + 2 \cdot (2 + 4 \cdot 1))$. In this example 43 % of the cycles are outer loop overhead. It is recommended to first unroll the inner loop, because this can be done quite easily. Unrolling the outer loop (and leaving the inner loop rolled) will result in more control overhead. This is why Fingeroff [12] recommends to work from the inside out. The performance of nested loops can probably be increased more by unrolling the inner loop and pipelining the outer loop(s). This will further be explained in section 2.3.

A special case of (nested) loops are window operations which are computationally intensive and data intensive. These window operations are commonly used in image processing and digital signal processing. Optical Flow algorithms are a typical example of image processing and they contain in many cases window operations. Generally, a window operation is a 1D or 2D loop program. In case of more than 1 dimension, the loop is in fact a nested loop. In many cases a certain number of inputs are used to calculate one output value. In case of image processing, a number of pixels are used for the calculations and the resulting output is one pixel (often in the center of the selected input pixels). The selected input pixels are called the window. Because this window slides over the input array, the input pixels are read more than once. If the window has a size of 5 pixels, an input pixel may be read 5 times within 5 adjacent iterations. By unrolling such loops, data reuse becomes possible. Dong et al. [13] have done some research on the consequences of unrolling on area, throughput and clock frequency, when unrolling either the inner or outer loop of these window based nested loops. They conclude that unrolling the inner loop increases the controller complexity and area compared to unrolling the outer loop, while unrolling the outer loop requires more memory for data reuse compared to unrolling the inner loop.

2.2 Memory Access

Memory accesses can be very time consuming, especially on high data rate applications. Many operators of the same type can easily be scheduled in parallel, while accesses to a certain memory can often not be executed concurrently. Therefore, it is very important how the data is mapped to the memory and how it is read from memory. This section describes how memory accesses are influenced by the way they are passed as argument to a function. Also different techniques are discussed to reduce memory accesses in general.

2.2.1 Passing Arguments (IO)

How arguments are passed is very important. A lot of information given in this section is *Catapult C* dependent. More can be found in the main source of this section, which is the Blue Book by Fingeroff [12], who is a technical marketing engineer at *Mentor Graphics*.

In general, *C* has two ways of passing arguments, namely: *by reference* and *by value*. When passing an argument by reference, the address of the data is provided to the function. Any reads or writes will be done from or to this address. This is useful if a function must be able to edit the value. When passing data by value, a local copy will be made on the stack. Only when making this copy, the original data will be read. When data is written from within the called function, only the local copy will be updated.

When only concerning sequential implementations (as *C* applications often are), the above definition covers all cases. But as different pieces of code run in parallel, the pass by reference method will become more complex. Because every read from within the called function reads from the original address, it is possible to change the data at this address between two reads of the called function. As hardware solutions are often designed for combinational use, HLS treats the two methods of passing arguments different as well. Besides that, a passed memory can either be accessed conditionally or unconditionally. A conditional access to a memory is an access only executed if a certain condition was met. In HLS tools, this also is of influence to memory accesses in practice.

2.2.1.1 Unconditional IO

Unconditional IO is considered to be an interface mapped to a 'wire' type resource. No handshaking is involved. HLS is free to move the IO to different clock cycles and into or out of conditions to reduce area. Therefore, it is not known when the memory accesses will occur before scheduling is completed. This implies that the IO must be ready to read or write before the function starts and during the execution of the function. Unconditional IO is often used for control signals (which do not change during execution) or in designs which are pipelined with II = 1 and where the IO is read or written every clock cycle. More about pipelining and the definition of II can be found in section 2.3.

Pass by reference used with unconditional IO will result in reads and writes whenever they are scheduled. When iterating over an array, the by the iterator pointed element of the array will be accessed. When non-array variables are used as well in these iterations, they will be read every iteration.

Pass by value used with unconditional IO will read all needed data at the beginning of the function, whether or not they are actually used in the iterations. Note that arrays are not read at the beginning of the function, because an array is a pointer to the first element of that array and therefore always passed by reference in languages like *C*. The advantages of pass by value on unconditional IO is that the data is read only once (which saves memory accesses) and the IO does only have to be held stable at the beginning of the main loop.

2.2.1.2 Conditional IO

Conditional IO is considered to be an interface mapped to a resource that has hardware handshaking. This can be a *ready to send / ready to receive* data or a *ready / acknowledge* behavior. Conditional IO cannot be moved to different clock cycles or conditions by HLS.

Pass by reference used with conditional IO works a lot like it does on unconditional IO, except it uses now handshaking to synchronize the data on reads and writes. In case the IO is only used when certain *C* conditions are met, the actual memory access is only started when these conditions did met.

Pass by value used with conditional IO is always read at the beginning of the main loop. While pass by reference only accesses the data inside a *C* condition when this condition is true, pass by value always reads the data, regardless where the condition evaluates to.

2.2.1.3 Merging IO

When a pipelined loop iterates over an array and the array reads are not conditioned, *Catapult C* automatically merges the array. The merged array is now read as a whole. When the array reads are conditioned (e.g. if (condition) then read array element) the array will not be merged automatically. This implies that the reads are done during the execution of the iterations. When the loop was pipelined with II = 1, every clock cycle a new iteration starts. If the array contains four elements, then four reads from that array may be initiated at the same time, which is not possible. A solution to this is making the array reads unconditioned by copying the array to a local array. Let the local array be read under certain conditions. Now, *Catapult C* will merge the original array again and therefore will be able to pipeline the loop successfully.

2.2.2 Memory Architecture

In streaming data applications, such as Optical Flow applications, a lot of memory accesses occur. Most memories are so called *true single port memories*. This means that the memory has only one port to read and write data. If a memory access can be completed within one clock cycle, each and every clock cycle can only do one memory access on a certain memory. This can impact the performance drastically when applications need to access the memory very often. Different techniques are developed in the past to improve performance in these situations. Among them are memory interleaving, widening of word widths of memories and caching. In the next three sections, these techniques are further explained.

2.2.2.1 Memory Interleaving

Memory interleaving is a technique in which a single chunk of memory space is divided over multiple physical memories. Because the multiple memories can be accessed at the same time, the memory performance increases. This technique is used for over 40 years. In 1968, SIMD processors including ILLIAC IV [14] already used interleaving for reading multiple words in one access cycle. Other implementations used interleaving to be able to access one word per cycle, such as the CDC 6400 [15]. When considering hardware implementations, memory interleaving can be used to access different memory locations needed by a certain computation at the same time. To determine how the memory elements, such as arrays, can best be mapped to the physical memories, the data access pattern must be investigated.

VanCourt and Herbordt have proposed a technique on planning the memory interleaving on grid computations [16, 17]. Some Optical Flow algorithms make use of grid computations, e.g. smoothing filters. Grid based computations with respect to image processing take a number of input pixels, do the computations and save the resulting output pixel. Because it is known at design time which pixels are used as input for a certain output pixel, all data reads from memory are known as well. When mapping the image to memory in such a way that every input pixel is physically stored on a separate memory, all input pixels can be read in a single clock cycle. The input pixels usually are positioned around the output pixel. VanCourt et al. [17] proposes to make use of the LSBs of the array indices to address the memory bank used for a certain pixel. This works very easy when a rectangle of pixels is used and where the number of pixels is a power of two. In other cases, the memory address controller becomes more complex.

Catapult C has an option for automatically interleaving memory, which has to be turned on first. According to Fingeroff [12] this automatic interleaving can be sufficient, but especially when interleaving with a factor other than two, better performance results can be achieved when manual interleaving the memory.

2.2.2.2 Widening Word Width

Another technique is widening the word width. By combining more data words into a new data word, more data can be read en written from or to memory in the same clock cycle. The algorithm must read or write the data sequentially to really improve performance. *Cataput C* can automatically widen the word widths, but for non power of two widening, manually widening might provide better results (if the HLS tool could do the job anyway). The main bottleneck on widening word widths is the physical word width of the memories available. When the original word width is already large, the memories available might not be able to provide double the original word width.

2.2.2.3 Caching

Caching is saving data in a faster (but often smaller) memory to decrease access times. Back in 1968, the developers of the IBM System/360 Model 85 needed a large main memory. This main memory was relatively slow compared to the CPU. The solution was to make use of a memory hierarchy. A cache memory was placed near the CPU. This memory was not directly addressable, since it cached data from the main memory. The cache memory was a fast memory compared to main memory, but it was much smaller. This great invention is published in a journal written by Liptay [18].

Nowadays caches are used very often in general purpose processors. The performance increase depends on the application running on the CPU in combination with the type of cache used. There exist different kinds of caches. Direct mapped caches map certain data from main memory to one single location in cache. Often a modulo operation is used to determine the location in cache for a certain address in main memory. Direct mapped caches are easy to control and the addresses are easily transformed to cache locations. On the other hand, these caches are not very flexible. If two addresses are used that are mapped to the same location in cache, the cache is not working optimal. A more complex type of cache is a fully associative cache. This cache can use any location for any data. The control logic and power consumption become worse compared to direct mapped cache, but this cache is more flexible. The third type of cache is set associative cache. Here n sets (or locations) in cache can be used for a certain address from main memory. This solution lays between direct mapped and fully associative cache. Which cache can best be used depends on the design constraints and the application. More about these commonly used caches can be found in literature, for example in [19]. Gil et al. [20] have proposed a cache implementation in an FPGA which can easily be reconfigured. Now different kind of caches can be selected fast. This solution can be of great use when using embedded processors.

In case of a full hardware design of an application (as this thesis is about), most memory accesses are known or can be determined at design time. This opens more possibilities for caching data. When the memory access patterns are known, the designer can decide where to implement a cache and what exactly will be cached. An example of this is given by Fingeroff [12] and involves image applications where pixels are read and written row after row and column after column. Many image applications use different pixels close to each other to calculate a value. The pixels used as input are called the window. Often, this window is moved pixel by pixel to calculate each value needed. Window n and window $n - 1$ usually differs only in a minor number of pixels, i.e. many pixels are used again. By caching these pixels, a lot of memory accesses can be saved. By creating a custom cache design for these applications, the cache efficiency will by maximal.

2.3 Pipelining

Besides loop unrolling, *Catapult C* can do pipelining on loops as well. With loop unrolling, the designer must provide a value. In case of pipelining, this value is called the Initiation Interval (II). This II value determines how many cycles are taken before starting the next iteration. An II value of 1 will start a new iteration each clock cycle. Low values for II will cause more iterations to run at the same time. This can prevent resource sharing and therefore may result in more area consumption.

Nested loops are interesting to pipeline. Fingeroff [12] clearly explains what happens when loop pipelining with *Catapult C* is used. When nested loops are pipelined together, the loops are flattened into one loop. The initiation interval now is used on this new loop. Pipelining the outer loops have good results on performance (latency and throughput), but will result in complex control logic.

2.3.1 Pipelining and Function Arguments

When and how many times arguments passed to a function are read, depends on how these arguments are passed to the function. This is covered in more detail in Section 2.2. The performance increase resulting from loop pipelining can depend heavily on how these arguments are passed. As is explained in section 2.2.1, when passing arguments by value, the data is read at the beginning of the function. If the main loop of this function is pipelined, more of these reads may be initiated at the same time. Because the memory interface can often handle only one data transfer per clock cycle (or a few on multi-port memories), the pipeline needs to stall. This of course influences the performance in a negative way. Another important consequence of pipelining the main loop with II = 1 in combination with conditional IO is the ramp-down of the loop. If all data has been read, the pipeline will be stalled because the conditional IO indicates that there is no data available for read. This stalling prevents the last iterations to write the results back. This unwanted behavior can be prevented by only pipelining the inner loops and leave the main loop unpipelined or by manually flushing the pipeline. The pipeline can be manually flushed by implementing an acknowledge signal within *C*. By using this acknowledge signal as condition on memory reads within the loop, reads are only done as long as the acknowledge signal is high. When no more data is available, the acknowledge signal becomes low and no more data requests are done. Now, the pipeline will not stall, so all data can be flushed.

2.3.2 Conclusion

This chapter provided different opportunities to improve the performance of an application implemented in hardware using High Level Synthesis. All improvement methods need information about the algorithm in order to be implemented effectively. This emphasizes the need of an analyze tool providing such information. Chapter 3 describes different methods on how information can be extracted from an algorithm.

3 Extracting Information from an Algorithm

When implementing an algorithm in hardware, information about memory accesses is always one of the most important topics. Not only for the use in HLS solutions, but also for other variants of hardware implementations. An example of this is a System-on-Chip (SoC) design where multiple processors exist on a single chip, all accessing memories. Since the applications form the basis of such designs, the need to tune the underlying architecture for extracting maximal performance from the software code becomes imperative [21]. As the size of reconfigurable devices, such as FPGA's, grows, not only the logic is implemented in these devices, memory can be placed there as well [22]. This makes sense, because if the memory exists close to the user of this memory, latency can potentially be minimized. This minimizes the gap between processing and memory, but the need of analyzing memory accesses is still an inevitable demand.

When implementing an application fully in hardware (and not using any processors), the amount of operations is very important as well. In case of High Level Synthesis, most parallelizations can be achieved by pipelining loops. These loops come directly from the originating source code. Bringing memory accesses and operators into relation with these loops is essential with HLS. This chapter will focus on retrieving memory access and operator information from an algorithm.

3.1 Static or Dynamic Analysis

All analysis methods can basically be grouped in two types of analysis, namely: static analysis and dynamic analysis. The goal of both methods is to find certain information about an algorithm.

Static analysis analyzes the algorithm at compile time. Therefore, the algorithm does not have to be completely compiled prior to analysis. The information (e.g. memory accesses or operator counts) is retrieved by statically reviewing the source code. There are different methods to do this. The static analysis of a function without any loops or conditions is clear, since retrieving is not too hard to do. Also loops with clearly defined bounds can easily be analyzed. Things become more complex when the bounds of a loop are not so clear at compile time. This can happen for instance if the loop depends on external data. Another problem arises when parts of the code are conditionally executed. If the condition cannot be evaluated at compile time, it cannot be determined if the code within the condition is actually executed. These limitations are a major disadvantage of static analysis. In the past, some analyzers have been build using the static analysis approach. An example is the Volta project [23], which is a *Java* analyzer. Volta focuses on the Worst Case Execution Time (WCET) of algorithms to be used in real-time applications. When using Volta, loop bounds must be provided by the developer using annotations.

Dynamic analysis does the analysis in a dynamic way. This means that the algorithm is actually executed and in some way monitored. The drawback is that the algorithm has to be compiled and that the input data (if applicable) has to be available. This can be a problem if the algorithm itself is still in development or the input data is not yet known. The major advantage of dynamic analysis over static analysis is of course that the algorithm is monitored step by step. This implies that the loop bounds and conditions are not needed to be known in advance.

3.2 Dynamic Analysis

In order for static analysis to provide the execution order of an algorithm, loop bounds have to be manually assigned. Beside that, it cannot be determined if conditional memory accesses are done or not without knowing how the condition will evaluate. This makes static analysis impractical, not to say impossible, when the exact memory access order is of interest. Hence, it is clear that a solution has to be searched within the set of dynamic analysis solutions. Different possibilities were investigated. The following sections will present these.

3.2.1 Previous Work and Existing Tools

Analyzing algorithms is not new. For many years, engineers are analyzing algorithms to improve performance, reduce the needed amount of resources or minimizing power consumption. Most of these analysis tools or frameworks are focusing on software, to improve the implementation with some kind of processor. A few solutions for hardware purposes have been proposed.

Tools that are analyzing the run-time behavior of an application in order to improve the performance of that application are called profilers. A profiler generally analyzes where an application consumes resources, whether that is processing or memory access. General profiling tools like *gprof* [24] can provide function-level execution statistics to find application hot-spots. Unfortunately, no distinction is made in computational and memory access time. Therefore, these tools cannot be used for analyzing memory specific analysis. MemSpy [25] is a tool that can help find memory bottlenecks in both sequential and parallel programs. This tool focuses mainly on memory accesses in single or multi processor architectures, not in fully custom hardware designs. Also operator analysis is not possible. QUAD [26] is a tool designed for memory access and dependency analysis. This tool analyzes an algorithms by working with the executable itself. No special compilation is needed. All memory accesses are monitored and used for the analysis. However, QUAD does not provide information about loops. Hence, no memory access analysis within certain loops can be done. This is for HLS important, because HLS tools like *Catapult C* depend heavily on loops for easy pipelining a design.

3.2.2 Alternative options

Existing tools are mainly focused on finding bottlenecks in memory accesses. Other tools do provide a more complete memory access analysis, but lacks the operator usage analysis. Most tools do not take loops from the source code into account. When code is executed on a sequential processor, loops are not interesting. The actual accesses to the memories are of interest then. But High Level Synthesis tools do use loops to implement an algorithm into hardware. Optimizing an algorithm loop based will improve the final result of the HLS tool. Loop based memory access and operator analysis is therefore desirable. To build a tool capable if this, a number of solutions were investigated.

3.2.2.1 Valgrind

One of the first dynamic analysis options a software developer will think of is probably Valgrind. Valgrind is a powerful tool which is used a lot to find memory leaks and debugging segmentation faults. Documentation is available online [27].

Valgrind consists of a number of tools [28]. Memory leaks and segmentation faults are found by using Memcheck. Memcheck intercepts calls to `malloc`, `free`, `new` and `delete`. By doing this, Memcheck is able to find memory violations, e.g. invalid reads or writes. Another tool provided by Valgrind is Cachegrind. Cachegrind can be used to determine the cache usage on a system. It will provide the number of cache misses and pinpoints the sources of these misses. A third tool of Valgrind is Callgrind. Callgrind is an extension to Cachegrind. It will provide the same information as Cachegrind does and in addition it provides information about different calls within the application under investigation. The graphical user interface *KCacheGrind* provides the results in a very human readable way. The information gathered about calls will result in profiling information.

Although Valgrind provides a lot of other tools, none of them are stable tools which can provide the memory access information this thesis is interested in. An option can be developing our own Valgrind tool. A lot of research on Valgrind would be necessary for this to be successful. It must be investigated if Valgrind does provide the support to produce loop based information and operator usage information. A number of tools available with Valgrind are actually developed by third parties. Important to mention is that the source code should be compiled with debugging information (often `-g` option) in order for Valgrind to provide any information.

3.2.2.2 GDB - GNU Debugger

Another option is to use GNU Debugger or GDB [29]. GDB can be used to debug applications. Common debug techniques like breakpoints are supported by GDB and it has been used in some IDE's. The Eclipse CDT IDE which is a C/C++ development environment uses GDB as debugger for example [30]. However, normally a developer places a breakpoint at one or more points of interest within the source code. The point of interest of this thesis can involve the whole application. Placing a breaking point at the start of the main function and then stepping through the whole application can be done in order to find all information of interest in a dynamic way. There exists however a major drawback to this method. This will be illustrated in the example below (figure 3.1). In the example, only three different GDB commands were used, namely `b 1`, `r` and `s`. These commands set a breakpoint at the first line available, start running the application and steps to the next line in source code, respectively.

```
void main (void)
{
  int i = 89;
  i+=7;

  i -= 44 + 8;
}
```

```
Reading symbols from a.out...done.
(gdb) b 1
Breakpoint 1 at 0x804839a: file main.c, line 1.
(gdb) r
Starting program: a.out

Breakpoint 1, main () at main.c:3
3 int i = 89;
(gdb) s
4 i+=7;
(gdb) s
6 i -= 44 + 8;
(gdb) s
7 }
```

Figure 3.1: Example C code (left) with its GDB debug output (right)

The different coding styles (the use of spaces) in the source code (left) is on purpose. As can be seen in the GDB debugger output, debugging is done line by line. At each line, the original source code of the current line is provided. This is very helpful when debugging an application, because it is clear from the debugger output what code was executed exactly and one can find the same code in the original source easily. However, for this thesis project, this is not so helpful. Because GDB outputs the exact line of code, all coding styles and variations have to be supported in the parser that parses the GDB output. Developing such a parser will be a very time consuming job.

As with Valgrind, GDB needs the application to be compiled with debug information (often `-g` option) to be able to provide information.

3.2.2.3 Variable Replacement Using Macros

Another way to investigate memory accesses for instance, is to change the source code itself. The changes should be able to log everything of interest in order to be able to analyze later. These changes in source code have to be done with as little human intervention as possible, because the system should save time for the developer and not take too much time. The basic idea is to design a custom data type, which is a *C++* class. By operator overriding, all kind of operators are supported and monitored. Using a Macro, the original data types are replaced by the custom data type. An example of this is given below in figure 3.2 the corresponding output is provided in figure 3.3.

Although the provided example does only include the basic information, i.e. the type of operation and the size of the variable in bytes, the output looks promising.

```
#define char Analyze<char>
#define long Analyze<long>

int main (void) {
  char a = 60;
  char b;
  char c;
  long d,e;

  b = 5;

  c = a + b + 1;
  d = 8;
  e = d + 9;
  return 0;
}
```

```
template<typename T>
class Analyze {
public:
  T value;

  Analyze(T j = 0)
  {
    value = j;
    cout << "Analyze assignment" << endl;
    cout << "Analyze size = "                    //
 << sizeof(value) << endl;
  }

  Analyze<T> operator+
    (const Analyze<T> &other) const
  {
    cout << "Analyze addition" << endl;
    return Analyze (value + other.value);
  }
};
```

Figure 3.2: Example C code including the macro (left) and the custom Analyze class (right)

However, there is a major drawback of this system. In practice, a developer is most likely interested in arrays. This is because arrays are often used in loops and therefore are candidates to run in parallel. Furthermore, arrays are typically mapped to memories. Data dependencies and the order of execution should be clear from the log file. This is where a major shortcoming of data type replacement comes in. Imagine for a moment that an array of type `char` is replaced with an array of the custom class `Analyze<char>`. If a data access occurs within the array, the corresponding function called in the Analyze class is called. However, this class has no information about being an array element and also array indices are unknown to this function. In fact, from the point of view of the functions within the Analyze class, all elements of an array are independent variables. This problem can be solved by overwriting arrays with another analyze class. If an access to an element in the array is accessed, the value of the index is known by using the subscript operator (`operator[]`). Unfortunately, there is no easy way of replacing arrays with another data type. This would involve manual changes in the source code, or a sophisticated parser that can parse the source code and replace all arrays with the custom data type. Hence, one of the major points of interest is not practically analyzable by this system.

3.2.2.4 Pin

Pin [31] can do analysis to memory accesses and operations by working with the executable code of an application. This has two major advantages. The first is that the language used in the source code does not matter, since Pin uses the executable file and does not interfere with the source code. The second advantage is that Pin makes

```
Analyze assignment
Analyze size = 1
Analyze assignment
Analyze size = 1
Analyze assignment
Analyze size = 1
Analyze assignment
Analyze size = 4
Analyze assignment
Analyze size = 4
Analyze assignment
Analyze size = 1
Analyze assignment
Analyze size = 1
Analyze addition
Analyze assignment
Analyze size = 1
Analyze addition
Analyze assignment
Analyze size = 1
Analyze assignment
Analyze size = 4
Analyze assignment
Analyze size = 4
Analyze addition
Analyze assignment
Analyze size = 4
```

Figure 3.3: Resulting output of fig. 3.2

the analysis compiler independent. What compiler was used to generate the executable file is not important. No special compiler options have to be set when the algorithm is compiled, making the compilation process as transparent as possible. QUAD [26] uses Pin to dynamically extract information from an algorithm.

Pin is great when the actual memory accesses are of interest. According to [31], analysis should be able to report on a basic block level. This implies that loops should be detectable. More information on basic blocks can be found in Section 4.2. The algorithm should be compiled with debug information to be able to retrieve information about variable names, line numbers, etc. Compiler optimizations should be turned off completely to prevent the resulting executable from changing the execution order of memory accesses and operations. That would cause a difference in the generated analysis reports compared to the original source code.

3.2.2.5 GCC PlugIns

GCC stands for GNU Compiler Collection [32] and is one of the most commonly used open source compilers. It is evident that it should be possible to make GCC do static analysis (changes in source code of GCC might be needed). But is it possible to do dynamic analysis using GCC? Let's first take a look on how dynamic analysis can be done manually.

Manual Dynamic Analysis

Manual dynamic analysis can be done by calling a function on every point of interest, e.g. on every memory access, operation, loop, etc. By providing the called function with important information using arguments, the called function should be able to log all events. By analyzing this log later on, all kind of information can be provided to the developer. After inserting all function calls, the new source code has to be compiled and executed in order to generate the log file. The main advantage of this is that every event can be logged as long as a function call can be inserted. The enormous manual labor is of course too much to make this idea reasonable. The question is: is there a way to automatically insert the correct function calls, to avoid as much manual labor as possible?

GCC Basic Working

Before we answer the question above, let's take a brief look on how GCC works under the hood. More detailed information is provided in Section 4.2. GCC is able to compile a number of different programming languages, these include but are not limited by *C*, *C++* and *Fortran*. Also many architectures are supported, for example ARM, AVR, MIPS and of course x86. To make the optimizations independent from the input language (Front End) and the target architecture (Back End), an intermediate representation (IR) or intermediate language (IL) is used. The Front End translates the input source code into the IR. The Back End translates the IR into the machine code for the targeted hardware. Optimizations can be run on the IR. It is possible to make changes in GCC to make it able to insert function calls into the IR level of the compiler.

GCC PlugIn

Actually changing the source of GCC would result in a custom GCC version, which inherits it main source from one particular version of GCC. If GCC releases a new version, the custom made GCC version is outdated and may be developed once again. Since the GCC release of version 4.5.0 [1], PlugIns are supported. A PlugIn is a custom made module which can be used by GCC while compiling. The PlugIn is able to make changes in the code being compiled. If a new version of GCC is released, the PlugIn still works, provided that the changes in GCC did not affect the used options by the PlugIn. By using PlugIns, the custom code is less likely to get outdated. However, there is a major drawback by doing this. The GCC documentation [33] is not always as clear and complete as one would like. The GCC PlugIn documentation (chapter 23 of [33]) is very incomplete and only covers the very basics. On top of that, since PlugIns for GCC are relatively new, not much work has been published on this yet. Therefore not many examples are available.

[1] http://gcc.gnu.org/gcc-4.5/changes.html

3.3 Conclusion

Replacing variables with macros is an easy to implement but incomplete solution, and therefore not suitable for this thesis. Valgrind, GDB and Pin need the algorithm to be compiled with debug information enabled and the GCC PlugIn requires the algorithm to be compiled with the PlugIn enabled. GDB would involve the development of a parser that is able to parse all *C* (or *C++*) style code. Valgrind does not provide a suitable stable tool which can provide all needed information. A custom tool must be developed when Valgrind is to be used. Besides that, it is not known if Valgrind will be able to report all loops, operations and memory accesses and operations within a certain loop. Because of this, Valgrind was not used to develop the analyzer with.

A GCC PlugIn and Pin can probably provide all necessary information. The GCC PlugIn will run on the original source code. This causes the analysis reports to accurately match the original source code. The poor documentation and the obligation to use the GCC compiler (compiler dependent) are drawbacks. Pin is compiler independent, but operates on the compiled source code. Changes in execution order caused by optimizations of the compiler will influence the report results. This is not desirable.

Both options will probably be suitable for analyzing algorithms and providing loop based information. It was decided to use a GCC PlugIn to retrieve the information from the source code. Chapter 4 presents the system design using a GCC PlugIn.

System Design 4

This chapter describes the presented framework system design. First, a basic overview of the entire system and a more detailed description about GCC are given. Then, both components (PlugIn and Parser / Analyzer) of the system itself will be discussed in great detail among with the reports that can be generated.

4.1 Basic Overview of the System

As concluded in the previous chapter, a GCC PlugIn is used to retrieve all information of interest. By compiling an algorithm with GCC and the developed PlugIn for GCC, function calls are inserted into the Abstract Syntax Tree (AST) representing the original source code. On every event, such as a memory access or an operation, a function call is inserted. The function definitions are provided in a separate file (`analyze.cpp` and its header file `analyze.h`). This file is compiled to an object file, as all other source files are. The linker finally combines all object files into a single executable file.

By executing this file, a log file will be generated. The size of this log file is heavily depending on the number of operations and memory accesses within the original source code. More about the size of the log file is mentioned later on. The log file can be read by a parser. This parser reads the file and generates a tree in memory from it. We call this tree a Data Flow Tree, or DFT. When the DFT has been completely built, a command prompt is provided to the user. The user is now able to generate reports. These reports are directly generated from the DFT. Reports can be shown on the screen or saved in a file.

Figure 4.1 provides a schematic representation of the whole system. The definition of the log file format can be found in Appendix B. The presented framework itself consists of the GCC PlugIn and the Parser / Analyzer.

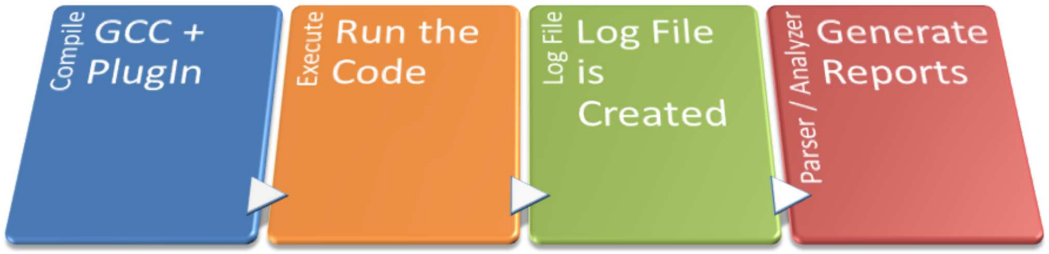

Figure 4.1: System overview

4.2 Basic Working of GCC

To understand how the presented GCC Plugin works, a good understanding of the basic working of GCC is needed. Figure 4.2 illustrates the basic steps of GCC. The first step is the Front End. For each supported input language, a Front End is available (the figure only shows the *C* language as example). This Front End translates the source code into en generic representation, called *GENERIC* [34]. This representation is further simplified into a *GIMPLE* [34] representation. The *GIMPLE* representation has been heavily influenced by *SIMPLE IL* [35]. At the other side of the compiler the Back End is present. The Back End translates the *GIMPLE* code into the machine code for the targeted architecture. As can be seen, the design is modular. Adding a new language to the compiler can be achieved by developing a new Front End. To support a new target architecture, only a new Back End has to be made.

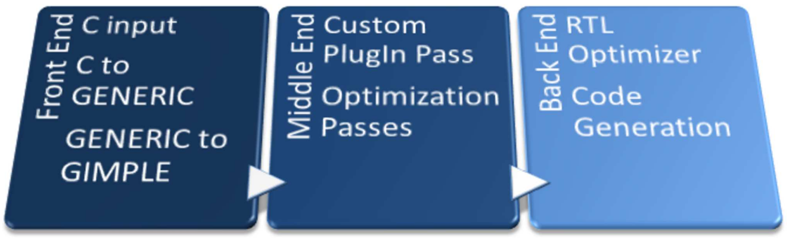

Figure 4.2: Basic GCC overview

The representation and language used between the Front End and the Back End is called the Intermediate Representation (IR) and Intermediate Language (IL). The IL used by GCC is *GIMPLE*. Between the Front End and the Back End, the Middle End exists. In this part of GCC, most commonly performed operations are executed, such as code optimization. To do this, GCC uses passes. Once the *GIMPLE* representation is complete, the Pass Manager is started. All passes which are to be executed are known to this Pass Manager. By developing our own pass, and registering it to the Pass Manager, we are able to make changes to the Abstract Syntax Tree.

GCC compiles each function separately. This means that each function results in its own *GIMPLE* representation. The *GIMPLE* representation forms the Abstract Syntax Tree (or AST) of this function. For each function, the Pass Manager executes all registered passes. The custom pass is therefore executed for each function.

In *GIMPLE*, all operations are grouped into Basic Blocks. Each function therefore contains at least one Basic Block. Every Basic Block can contain one condition. This condition is located at the end of the block and determines what block should be executed next. A conditional branch will therefore result in a new Basic Block. It is not possible to jump to code within a Basic Block. A simple for-loop without any additional branches results in three Basic Blocks. The first block sets the initial value of the iterator. The second block checks the condition, and the third block contains the actual code executed within the loop and the incremental operation of the loop iterator.

4.3 GCC Plugin

In GCC, all functions will be handled separately and will result in separate AST's. Not only the functions provided in the user's source code, but also system functions. If for example a call is made to `printf()`, the `printf()` function will also be passed to the Pass Manager. A log file can become quite big. It is important to only analyze the function(s) of interest. System functions are most likely not important to be analyzed. With large projects to analyze, the engineer is probably working at optimizing function after function. This also calls for the need to be able to provide the functions to be analyzed. By minimizing the number of functions to analyze, compile time is improved, but more important, the execution of the algorithm is as fast as possible and the resulting log file(s) are as small as possible. Small files are later on faster to parse by the parser. As can be seen, it is very attractive to analyze as less functions as possible.

By passing an argument to the plugin, the functions to analyze are provided to the plugin. The argument to use is `-fplugin-arg-plugin-functions={Functions}`. This argument can be entered as any other argument directly after GCC (e.g. `g++ -g -Ipath-to-analyze-h/ -fplugin=path-to/plugin.so -fplugin-arg-plugin-functions="functions to analyze" -c -o object.o source.cpp`). The plugin itself must also be provided to GCC using the `-fplugin=path-to/plugin.so` argument.

The GCC Plugin will insert function calls to the `analyze.cpp` file. It is therefore necessary that the header file is included in the source. If all source files point to a global header file, it is efficient to place the include in this global header file. The header file must be included before any function definition is given, because the functions to be inserted are not known otherwise. The following code fragment can be used:

```
#ifdef ANALYZE
#include <analyze.h>
#endif
```

Figure 4.3: Include `analyze.h` file

By passing the argument `-DANALYZE` to GCC, the `analyze.h` file will be included. Do not forget to provide the location of the analyze header file to GCC. To make sure all functions of the analyze library are loaded correctly, the existence of them will be checked by the custom pass before a function will be analyzed. If not all analyze functions are found, an error is returned to GCC and will be shown to the user.

4.3.1 Loading the Analyze Prototypes

By including the header file of the analyze functions (i.e. `analyze.h`), the prototypes of these functions are available to the application to compile. The prototypes are saved in the global namespace.

The first function of the PlugIn loaded is the `plugin_init()` function. Here, the custom pass is registered to the Pass Manager and the names of all analyze functions are set. Later on, when a function is analyzed, it will search for all functions defined during the initialization phase. An overview of all these functions and their purpose can be found in Appendix C.

4.3.2 Placement of the Custom Pass

One of the first steps is to determine where the custom pass has to be inserted. Important is that this pass has to be one of the first passes to be executed. Once the code has been optimized, the execution order of different statements may have changed. Also variables within the source code may have been optimized away. This may result in unexpected logs. One of the first passes is the SSA pass. This pass converts the *GIMPLE* representation into an SSA form. In SSA form, a variable may be read as many times as needed, but may be written only once. If a variable is written more than once, the SSA pass will insert as many new variables as needed to make sure that all variables are written once. These new variables can get unexpected (or no) names. This makes it very hard to generate an accurate log file, which provides information about all memory writes. It is clear that the new pass has to be inserted before this SSA pass.

4.3.3 Managing the Log File

To guarantee that the log file has been opened prior to writing new log data to it, the function call to open the log file is inserted at the top most position of the main function. The function call to close the log file will be inserted at the very last position of the main function.

32 bit versions of Linux support files up to 2 GB. To keep the generated log files well within this file size limit, the size is monitored during the execution of the algorithm. Every time a Basic Block is opened or closed, the file size will be checked. If the size exceeds 1 GB, the file is closed and a new log file is created. When analyzing an algorithm with many executed statements (for example a large nested loop), more log files can be expected to be generated.

4.3.4 Logging Basic Blocks

GCC assigns a unique number to each Basic Block. By logging this unique number when a block is entered and left, later on it will be possible to determine the statements which belong to this Basic Block. Also the number of passes of the Basic Block can be found which is needed for detecting loops.

A `gimple_statement_iterator` is used to iterate over all statements within a Basic Block. When new function calls to log functions have to be inserted, sometimes they are inserted before the current statement iterator, and some other times, the function calls are inserted after the current statement iterator. The reason for this is explained in Section 4.3.5.1. The inserted function calls are new statements which are inserted into the Basic Block. Unfortunately, in some situations a statement cannot be inserted after the current statement iterator. This occurs in three situations:

- The Basic Block contains no statements. This occurs when a function is empty.
- The statement iterator points to a `RETURN` statement. After a return statement, no other statements can exist, because they can never be executed.
- The statement iterator points to a `CONDITION` statement. As mentioned earlier, a Basic Block can only contain one condition, because no branches can exist within a Basic Block. The condition determines which Basic Block should be executed next. Therefore, if code is inserted after a `CONDITION` statement, this code will never be executed.

To solve this problem, if one of the above situations occurs, a `NOP` (no-operation) is inserted at the very top of the Basic Block.

4.3.5 Logging Statements

Only three types of *GIMPLE* statements will be analyzed:

- `ASSIGN` Statement: All memory accesses and operations.
- `CALL` Statement: Function calls.
- `COND` Statement: Conditional branches.

4.3.5.1 Logging Assign Statements

The most complex statement which can be logged is the `ASSIGN` statement. Assign statements read one or more variables, does an operation and saves the result in a variable. The variable in which the result is saved is the variable on the left hand side of the operation in most programming languages (e.g. *C*). In GCC this variable can be addressed by the `gimple_assign_lhs()` function. The number of input variables can be either one or two, depending on the kind of operation. The first of these variables can be found by using the `gimple_assign_rhs1()` function. If a second input variable exists, it can be accessed by the `gimple_assign_rhs2()` function. There are three kind of operations, namely:

- Binary Operation: An operation using two input variables. Examples are the addition or multiplication operations.
- Unary Operation: An operation using only one input variable. Examples are the negate or convert operations.

- Single Operation: An operation using only one input. This type of operation assigns values to variables or pointers. The difference with unary operations is that no particular calculation is done.

In the PlugIn, the function `processGimpleAssign()` determines what kind of operation is being executed and depending on that, calls the appropriate log functions. These log functions insert the function calls into the AST. When a function call is inserted, it can be inserted just before of just after the statement it refers to. In case of variables which are used as input to an operation, the function call is inserted before the original statement. This is necessary because only then it is guaranteed that the value of the variable is the same as it was when passed to the operation. The function calls representing the variables which are used as output of an operation are inserted after the original statement for similar reasons. The example below illustrates this.

```
index = index + 2;
```

Figure 4.4: Insert the log function call before or after the statement?

In this example, the value of variable `index` is different before and after the statement. Function calls to log functions which logs the access of the first input variable (i.e. the `index` after the assign operator) has to be inserted before this statement, otherwise the value is overwritten. For the same reason the output variable has to be logged after the statement. Hence, the process of inserting function calls is done in the following order:

1. Log the access(es) to the input variable(s).
2. Log the operation.
3. Log the access to the output variable.

Value or Pointer

When normal variables are used, logging the access is not too hard. But pointers can also occur in assign statements. Normally, pointers are not desired in algorithms which are to be transformed into HDL's. But sometimes, array accesses are implemented as pointer accesses in the AST. This happens for example when an array is passed as argument to a function and within this function, the array is accessed. Because passing an array to a function is basically passing the pointer of the first element of the array to the function, accesses to this array are always implemented as pointer accesses by GCC.

Because of the above, chosen was to fully support pointers in the GCC PlugIn. This makes the logging process a bit more complex, because an access to a pointer can be either an access to the value where the pointer points to, or an access to the pointer itself (i.e. the value of the address the pointer contains). This can be determined by checking the `TREE_TYPE` of the variable:

```
if (TREE_CODE(TREE_TYPE(gimple_assign_rhs1(stmt))) == POINTER_TYPE)
  /* save access to the address of the pointer */
else
  /* save access value of the pointer or the variable */
```

Figure 4.5: Determine whether the address or the value is accessed

Casts

A cast is a translation from one data type to another. The value of one variable is translated to the data type of the other variable and after that, saved in this other variable. Often, casts are used to change the number of bits (e.g. from `char` to `int`), or to change from an unsigned representation to a signed representation (or the other way around). Both of these examples are changes of interpretation of the bits or just adding or removing most significant bits. No change in binary code is actually done. Of course, this is not always the case. Casting between integers and floating point numbers will change the bits. In hardware, casts do not always imply the need of a new variable (or signal). To be able to see the cast relation between variables while analyzing an algorithm, casts must be logged. GCC uses a unary operation for casts, namely a `NOP` operation with the CAST flag set. This can be checked by:

```
if (gimple_assign_rhs_code(stmt) == NOP_EXPR && gimple_assign_cast_p(
    stmt))
  /* this is a cast operation */
else
  /* this is not a cast operation */
```

Figure 4.6: Detect cast statements

Pointers can also be casted. A pointer cast is a special cast and can be found by checking the `TREE_CODE` of the `lhs` and the `rhs1` nodes of the assign statement. If both nodes are of type `POINTER_TYPE` and the cast check above returned true, then the statement is a pointer cast.

Accessing Pointer Addresses

As mentioned earlier, pointers can have more types of accesses than other variables. When the operation is of type Single Operation, the address of a pointer may be accessed. This can be determined by checking the `TREE_TYPE` of the `lhs` node. If this `TREE_TYPE` is `POINTER_TYPE`, then the address where the pointer points to is changed. The new value of this address can be either an expression or the address of another pointer. In the latter case, the `TREE_TYPE` of the `rhs1` node must be `POINTER_TYPE` as well. If the new address is an address expression, the `TREE_CODE` of the `rhs1` node is of type `ADDR_EXPR`. These checks and simple examples of the two types of setting the pointer address are given in the code fragment of figure 4.7.

If the value where the pointer points to is accessed, then the node type is of type `INDIRECT_REF`. Trivially, if the `lhs` node is of type `INDIRECT_REF`, then the value where

```
if (TREE_CODE(TREE_TYPE(gimple_assign_lhs(stmt))) == POINTER_TYPE)
{ //Set address of pointer
  if (TREE_CODE(gimple_assign_rhs1(stmt)) == ADDR_EXPR)
    //address from address expression (&example)
  else if (TREE_CODE(TREE_TYPE(gimple_assign_rhs1(stmt))) ==
     POINTER_TYPE)
    //address from other pointer
  else
    //unexpected address assignment of pointer
}
```

Figure 4.7: Determine how the address of a pointer was set

the pointer points to was written. When the `rhs1` node is of type `INDIRECT_REF`, the value where the pointer points to was read.

Special Case: Array Access through Pointer

As mentioned before, an array can be provided in the argument list of the called function. By providing the array, behind the scenes the address to the first element of this array is provided. Accesses to the array from within the called function would therefore not appear as array accesses, but as pointer accesses. The address of the first element of the array is the same as the in the function argument provided address to the whole array. Accesses to this first element does therefore appear as accesses to the provided function argument. Accesses to other elements are more complex. First, the corresponding address to such an element is calculated. This is done by multiplying the element index with the size of the data type of the array. Then, the resulting value is added to the address of the first element. This will be written to a new pointer. To access the element in question, the newly created pointer is accessed.

Since we are highly interested in array accesses, it is necessary to detect the procedure above, in order to make it possible to reconstruct the accesses to arrays when analyzing the arrays later on. Chosen was to keep the PlugIn as simple as possible and do all intensive work in the analyzer (or parser). Hence, no conversion from pointer access to array access is done by the PlugIn. Only the creation of a new pointer which is used to access an array element is detected and logged. These pointers can be found by checking the statement for three conditions:

1. The `TREE_TYPE` of the `rhs1` node is a `POINTER_TYPE`. This is the pointer which points to the first element of the array.
2. The `TREE_CODE` of the `rhs1` node is of type `PARM_DECL`. This stands for parameter declaration and is the address to the array which was provided by the function call.
3. The Operator type is `pointer-plus`.

The name of the array is the name of the `rhs1` node. The index of the accessed element is the value of the `rhs2` node. Keep in mind that this value has been multiplied by the size of the data type. The resulting address is also logged. This enables the analyzer to determine which pointer accesses actually belong to array accesses.

Save Node Access

After all checks above have been completed, it is known what kind of access has to be logged. The actual logging in most situations is done by one function. This is the `saveNodeAccess()` function. This function checks what kind of node has to be logged (i.e. the `TREE_CODE` of the node) and inserts the appropriate function call into the AST. The following `TREE_CODE`s are implemented:

- `PARM_DECL` : A variable provided as function parameter.
- `VAR_DECL` : A normal variable.
- `INTEGER_CST` : An integer constant.
- `ARRAY_REF` : An array reference
- `COMPONENT_REF` : A struct or class element.
- `INDIRECT_REF` : A pointer access.

If another `TREE_CODE` is found, a warning is printed to the user screen. In case the found type is `ARRAY_REF`, the index of the array is also saved. If the array is a multi-dimensional array, all indices are logged.

If a `COMPONENT_REF` is found, the recursive function `logComponentRef()` is called. A node of type `COMPONENT_REF` can be a `struct` or a `class`. The function `logComponentRef()` tries to find all parent variables, until it reaches the top most level. In *C* coding style, these elements are separated by a period '.' (or arrow '->' in case of a pointer). This is a safe solution, because periods are not allowed within variable names. However, GCC may use periods within variable names. This can occur if a temporary variable had to be inserted, which is related to a real variable. Normally, temporary variables do not get any names, but in these cases it might. The name provided is the original variable name, a period and a unique number (e.g. 'name.1'). This is the reason that in a log file, the period character may be used for variable names. The character chosen to separate the different components of a `struct` or `class` in the log file is the hash character '#'. More on detecting structs from source code can be found in Section 4.4.3.

Data Types

When an access to a node is logged, the data type and if possible the value of the variable is also logged. The data type is found by the function `getTreeTypeInfo()`. This function uses the `TREE_TYPE` of the node in question to determine its data type. The following `TREE_TYPE`s are supported:

- `INTEGER_TYPE` : All variants of integers, like int, char, unsigned, etc.
- `BOOLEAN_TYPE` : Can either be `true` of `false`.

- `RECORD_TYPE` : A struct or class.
- `REFERENCE_TYPE` or `POINTER_TYPE` : Pointer accesses.

If the type of the node is not supported, a warning is printed on the screen and the returned type is `UNKNOWN`. The access will still be logged, but no data type is provided. The most important data type not supported is the floating point data type (`REAL_TYPE`). In most HDL solutions, this data type is not desirable. To support this data type, new log functions have to be added to the `analyze.cpp` file. This makes logging more complex. Therefore, the support for floating point data types has been left as future work.

Besides the name of the data type, a special value is determined and returned to the caller of the `getTreeTypeInfo()` function. This special value is called the `valueType` and is of type `int`. The eight least significant bits present the number of precision bits of the data type found. In the current supported `TREE_TYPE`s, only `INTEGER_TYPE` has a precision. The precision can be found by using the GCC macro `TYPE_PRECISION()`. The other bits are used as flags. Table 4.1 shows these flags and their meaning.

Hex value of flag	**Meaning**	**Description**
0x100	Signed	Set if data type is signed
0x200	Boolean	Set if data type is boolean
0x400	No value	Set if data type has no integer representable value

Table 4.1: Flags of valueType

Whether a data type is signed can be determined by the GCC macro `TYPE_UNSIGNED()`.

4.3.5.2 Logging Call Statements

In the GCC PlugIn, call statements are handled by the function `processGimpleCall()`. Besides the name of the called function, the parameters and the return value are logged. Parameters can be a normal variable, an integer value, an address expression or a parameter of the calling function. Logging the parameters works much like logging normal accesses. Using a loop, all parameters are logged. The number of parameters can be found using the GCC function `gimple_call_num_args()`. If the called function returned a value and this value was saved in a variable, than this is logged as well. The return node can be found using the GCC function `gimple_assign_lhs()`.

4.3.5.3 Logging Condition Statements

A condition statement is always the last statement of a Basic Block. It determines what the next Basic Block will be to execute. There are basically six different conditions, namely:

- `LT_EXPR` : Less than
- `LE_EXPR` : Less than or equal
- `GT_EXPR` : Greater than
- `GE_EXPR` : Greater than or equal
- `EQ_EXPR` : Equal
- `NE_EXPR` : Not equal

If the condition has to be more complex, the result will first be calculated with normal operations and saved into a variable. This variable is then used within the condition statement.

A condition statement contains two nodes, the `lhs` and `rhs` nodes. They are available to the PlugIn by calling GCC functions `gimple_cond_lhs()` and `gimple_cond_rhs()`. The type of condition can be found by `gimple_cond_code()`.

The PlugIn function `processGimpleCond()` searches for these data and inserts the corresponding log functions into the AST.

4.3.6 PlugIn Arguments

GCC can pass arguments to the PlugIn. To be able to distinguish arguments for GCC and arguments for the PlugIn, all PlugIn arguments use the prefix `-fplugin-arg-plugin-`. In this prefix, the last part (in this case `plugin`) is the name of the plugin used. As can be seen, the name of our PlugIn is `plugin`.

The PlugIn supports only four arguments. These arguments are explained below.

-fplugin-arg-plugin-**help** is the commonly used help argument which informs the user about the other arguments.

-fplugin-arg-plugin-**verbose** is used to print a short description of the kind of statement or node being processed. In the code fragment below, a short example with the resulting output is provided.

```
               Analyzing function 'main'
int a, b;      CONST : 3
               VAR : a
a = 3;         VAR : a
b = a + 5;     CONST : 5
               VAR : b
```

Figure 4.8: Simple example code (left) and the corresponding verbose output (right)

-fplugin-arg-plugin-**debug** is used to print a longer description of how the AST is being analyzed. This argument can be combined with the verbose argument. Most of this debugging info contains the functions which are called. This argument is in particular handy when new features are added to the PlugIn. The resulting output with both debug argument and verbose argument enabled and using the same example as above, is given in figure 4.9.

```
Analyzing function 'main'
==DEBUG==   Logging BasicBlock
==DEBUG==   Logging BasicBlock -> DONE! (id=0x40fe8b80 action=s)
==DEBUG==   Logging Access
==DEBUG==   Logging Node Access
CONST : 3
==DEBUG==   Logging Node Access -> Done! (name=const)
==DEBUG==   Logging Access -> DONE! (access=r)
==DEBUG==   Logging Access
==DEBUG==   Logging Node Access
VAR : a
==DEBUG==   Logging Node Access -> Done! (name=a)
==DEBUG==   Logging Access -> DONE! (access=w)
==DEBUG==   Logging Access
==DEBUG==   Logging Node Access
VAR : a
==DEBUG==   Logging Node Access -> Done! (name=a)
==DEBUG==   Logging Access -> DONE! (access=r)
==DEBUG==   Logging Access
==DEBUG==   Logging Node Access
CONST : 5
==DEBUG==   Logging Node Access -> Done! (name=const)
==DEBUG==   Logging Access -> DONE! (access=r)
==DEBUG==   Logging Operation
==DEBUG==   Logging Operation -> DONE! (operation=plus type=2)
==DEBUG==   Logging Access
==DEBUG==   Logging Node Access
VAR : b
==DEBUG==   Logging Node Access -> Done! (name=b)
==DEBUG==   Logging Access -> DONE! (access=w)
==DEBUG==   Logging BasicBlock
==DEBUG==   Logging BasicBlock -> DONE! (id=0x40fe8b80 action=e)
```

Figure 4.9: PlugIn output if both verbose and debug options are enabled

-fplugin-arg-plugin-**functions** is used to provide the functions to analyze to the PlugIn. If no functions are provided, a warning message is put on the screen and compiling continues without analyzing any functions. If more functions have to be included into the log file, the functions can be put after each other in one string. Thus, double quotes are needed. If the functions `foo()` and `bar()` are to be analyzed, the complete argument would be: `-fplugin-arg-plugin-functions="foo bar"`.

4.4 Parser / Analyzer

The second tool needed for analyzing functions is the Parser / Analyzer. This tool reads the input file(s) and builds a tree from it. Later on, this tree is used to generate the desired reports. Figure 4.10 shows this process in a schematic way. This section describes the working of the Parser / Analyzer.

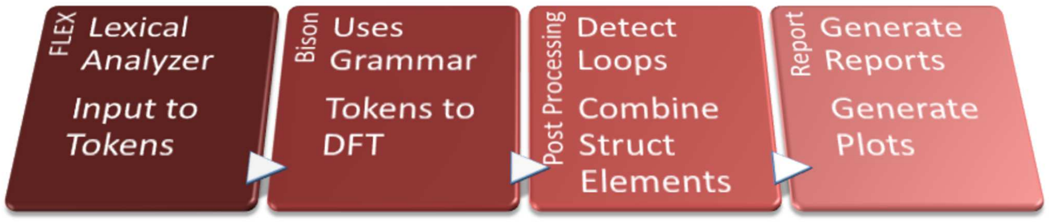

Figure 4.10: Parser / Analyzer overview

4.4.1 Parsing Log Files

The first task of the Parser / Analyzer is parsing the log file or log files (depending on the size of the data to log). The code to parse these log files was generated by Flex [36] and Bison [37]. Flex is a lexical analyzer. It uses regular expressions to recognize all parts of the input stream and convert these matches into tokens. These tokens are used by Bison. Bison knows the grammar of the log file. It uses the tokens to match the grammar and with that, it 'understands' what the log file has to say. The input data to Flex is defined in file `lex.l`. This file describes what regular expressions are to convert to which tokens. Bison uses the `grammar.y` file to retrieve the grammar of the log files. This `grammar.y` file also describes how the DFT is build. The first log file to parse is `temp_0.log`. In the user terminal, the progress of parsing is shown. If the file was completely parsed, the next file is opened. This file has file name `temp_1.log`. If the file exists, it is parsed, if the file does not exist, the parser moves on to its next step. This process repeats itself until all log files are parsed.

4.4.1.1 DFT - Data Flow Tree

The Data Flow Tree (DFT) is a tree containing many lists and other elements. Figure 4.11 presents this graphically.

The DFT itself is a globally defined list. This list contains all functions analyzed. The definition of the `Function` data type and that of all other DFT related data types can be found in the `dft.h` file. Almost all elements of the DFT are saved with location information, i.e. the origination source file and source line.

A `Function` contains one `BasicBlocks` (`BasicBlocks` is a data type). In this one `BasicBlocks`, all individual `BasicBlock`-s are saved and in addition all memory ac-

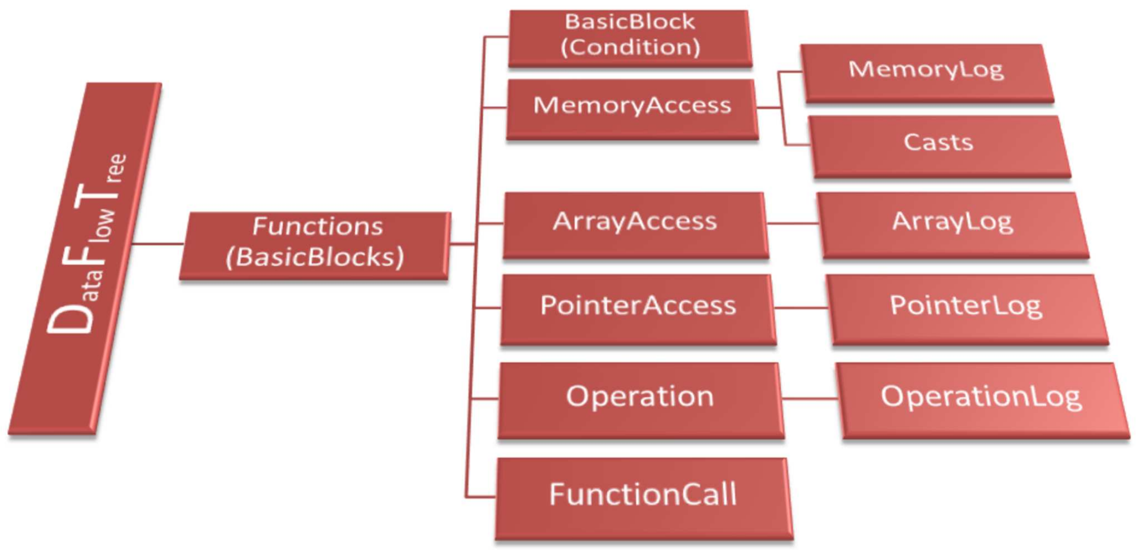

Figure 4.11: DFT - Data Flow Tree

cesses, array accesses, pointer accesses, operations and function calls are saved here. [1] These are all provided to the `BasicBlocks` by the `BasicBlock`-s. After a `BasicBlock` has been added to `BasicBlocks`, all memory accesses, array accesses, pointer accesses, operations and function calls are merged into `BasicBlocks` and deleted from `BasicBlock` to minimize memory usage. It is very important to merge this data, because the order in which `BasicBlock`-s are executed in, is not known yet. To keep track of the order of accesses to memory for example, the memory accesses are flattened. While merging the accesses from the newly parsed `BasicBlock` into `BasicBlocks`, the `BasicBlock` ID is saved to all these accesses, so it can be determined later to which `BasicBlock` a certain access belongs to.

One `BasicBlock` contains zero or more statements and if applicable the exit condition. `statements` is behind the scenes also of type `BasicBlock`. The difference in practice is, that a `BasicBlock` contains the already parsed statements and adds its unique identifier and if applicable the exit condition to it, while `statements` is used to add individual statements to the `statements` list.

`statements` contains all individual statements. If for example a memory access is found, it is added to this `statements` list.

The adding and merging in all steps above are in many cases quite complex operations. The reason for this is because sometimes accesses or operations need to be merged into existing accesses or operations, while in other cases they must be added. A lot of links exist between different parts of the tree. These links have to be updated accordingly.

[1] Please note the difference in used font. There exists a struct named '`BasicBlocks`' and a struct named '`BasicBlock`'. If '`BasicBlock`' is used plural, this is represented as '`BasicBlock`-s'.

4.4.1.2 Building the DFT

In the previous section a basic overview of the DFT was given. This section explains all steps in great detail. Some steps are trivial, while others are very complex. Many steps have a couple of properties in common. This will be discussed first.

Common Properties

All parts of the DFT actually related to real statements in the original source code, consist of two parts. The first is a description of the type of access or operation. Information like the variable name, data type and some counters are saved here. The second part is the actual access or operation. Here the location information is saved and if applicable, the value which was read or written. In case of an operation, the operands and the resulting access is saved via pointers. If more accesses or operations of the same type exist within a Basic Block, only the second part is saved, because the first past would hold the same information for all accesses or operations of the same type. The first part contains a list called `access` in which all instances of the second part are saved.

Another important common property is how strings are saved. Almost all elements of the DFT have to save a at least one string. Examples are the name of the access, the data type, the type of operation and the source file name. Since it is likely for strings in the DFT to occur multiple times, they are not saved on an individual basis, to minimize memory usage. The function `getStringPointer()` is used when a string is to be saved. This function makes use of a global list. The list contains all strings used. If a new string has to be added, the list is searched for the existence of this string. If the string already exists, the new string is freed and the pointer to the string in the list is returned. If the string does not already exist, it is added to the list and its pointer is returned.

Struct elements are parsed centrally for all access types (i.e. memory, array and pointer accesses). This results in a `StructElement` containing a name and a pointer to the next `StructElement`, if applicable. If the access does not consists of a struct element, the `StructElement` points to `NULL` and its name is the name of the access. If the access is a struct element, for example `a.b`, then the first `StructElement` contains the name `a` and points to the second `StructElement`. The second `StructElement` points to `NULL`, indicating no deeper levels were used in this access. The second `StructElement` also contains the name `b`. The function `getNameFromStructElement()` finds the name of the last `StructElement` and removes the last `StructElement` from the chain. The remaining `StructElement` chain and the resolved name is returned to the caller. This works this way because the parser parses the name of the variable always as a linked list of `StructElements`.

`MemoryAccess`

In the Parser, `MemoryAccess`es are all variable accesses which are not pointers or arrays. When a new memory access is parsed, a new `MemoryAccess` element is created and returned. This is done in the `newMemoryAccess()` function. First the name and correct chain of `StructElements` is found and saved. Also the read and write counters are set to zero and new lists for casts and accesses are declared. An access in which the source file, source line, direction (*r* (read) and *w* (write)) and the value if applicable is created. This access (or `MemoryLog` as it is called in the parser) is saved to the `access` list.

The newly created `MemoryAccess` is saved into the `statements` list. If the `MemoryAccess` is the first of the `statements` list, it is saved and the read or write counter is set to 1 accordingly. If the `MemoryAccess` was not the first to be saved, the list of already saved `MemoryAccess`es is searched for a `MemoryAccess` with the same name. If such an element is found, the `MemoryLog` of the new `MemoryAccess` is saved into the found `MemoryAccess`. All other information is discarded. The read and write counters of the found `MemoryAccess` will be updated accordingly. If no `MemoryAccess` with the same name was found, the new `MemoryAccess` is added to the list as it was if it was the first element to be added.

`ArrayAccess`

Array accesses are saved in the same way as memory accesses are, but with one addition: the indices of the array are saved in the `ArrayLog`. An `ArrayIndex` contains only two elements, namely the name of the variable used to provide the index value and the index value itself. If a constant value was used, the name will be `const`. In `ArrayLog` the `ArrayIndex` is saved in a list. If a multi-dimensional array was accessed, all instances of `ArrayIndex` are saved in this list. The function `newArrayAccess()` carries out this job.

`PointerAccess`

Pointer accesses are also saved much like memory accesses are. The difference here is that there are two additional counters beside the read and write counters of the other access types. `PointerAccess` has also a set and get counter. These counters keep track of the amount of times the address of the pointer was changed or retrieved, respectively. The direction saved in `PointerLog` can have more different values than in the other access types. Beside the *r* (read) and *w* (write) values, *g* (get), *s* (set), *p* (set from other pointer), *e* (set from address expression) and *c* (pointer cast) can be used. Also the actual address at the time of the access is saved. This is done by the `newPointerAccess()` function. When the `PointerAccess` is saved into the `statements` list, an important additional check is done. This check is part of the array access through pointer detection system. The working of this system is further explained in Section 4.4.1.3

`Cast`

Casts are copies of variables essentially holding the same data, but in a different format. In the DFT this is saved by making a connection between both `MemoryAccess`es. In the Parser, the connection itself is the `Cast`. If a cast is parsed, a new `Cast` is created and both `MemoryAccess`es (the source and the destination) are created. As with the different accesses above, a `Cast` contains also two parts. The first part saves a pointer to both `MemoryAccess`es and a counter which keeps track of the number of casts from the source `MemoryAccess` to the destination `MemoryAccess`. An access list is saved containing all cast instances, complete with source file, source line and if applicable the value. This is the second part.

When the Cast is saved into the `statements` list, both source and destination `MemoryAccess`es are searched for in the `statements` list. If they exist, the pointers in the `Cast` will be updated. If they do not exist already, they will be added to the `statements` list. After this, checked is if the `Cast` has occurred before. If not, the `Cast` is added to the `casts_to` list of the source `MemoryAccess`. If the `Cast` did occur before, the `CastLog` is saved to the existing `Cast`. After this all, the cast instance counter is updated.

`Operation`

The function `newOperation()` creates the new `Operation` instance. This function saves the name of the operation (e.g. `plus`, `multiply`, etc.) and the type of operation. This can be either Unary or Binary. Also the `OperationLog` is created and the source file and source line are saved in it.

When the newly created `Operation` is added to the `statements` list, the list is searched for operations with the same name. If found, the new `OperationLog` is added to the existing `Operation`. If not found, the new `Operation` is added to the `statements` list. `OperationLog` also saves the input variable(s) and the output variable. The input variable(s) are parsed just before the new `Operation`. In the `statements` list, two pointers are used to point to the last two parsed variable accesses. These pointers are now used to set the input variables of `OperationLog`. Naturally, if the type of operation was Binary, the last two accesses are saved to `OperationLog`, while if the type was Unary, only the last access is saved to `OperationLog`. The output variable is not parsed yet, but will be directly after the operation is handled. To be able to save this output variable to `OperationLog`, a pointer to `OperationLog` is saved along with a flag indicating that an output variable is to be saved to a `OperationLog`. On all kind of accesses parsed, this flag is checked. If the flag was found to be set, the output variable of `OperationLog` is saved and the flag is cleared.

`FunctionCall`

The last type of element saved to a `BasicBlock` are `FuncionCalls`. There exist a number of variations within function calls. Some function calls have a return value and the number of parameters can differ. The function called may also be a function to analyze. If this is the case, the new function will be analyzed before the previous function was finished. The previous function will continue after the called function returns. This implies that new functions can be added to the DFT from two places, either from the top level, or from within another function. If the function already exists in the DFT, the new instance of this function is saved in the `other_instances` list of that function, but only if the *save_duplicate* argument was used. More on arguments can be found in Section 4.4.5.

The number of parameters can be different for each function. This is why they are saved in a list. If a `FunctionCall` does not contain any parameters, the parameter list will be empty.

The new `FunctionCall` is saved in the `statements` list. If more function calls to the same function exist, they are all saved in the statements list separately. If the function call contained a return value, the corresponding `MemoryAccess` is saved as it would be in case of a stand-alone `MemoryAccess`. For now, returning pointers is not supported.

From `statements` to `BasicBlock`

The formed collection of `statements` is already of type `BasicBlock`. Only the identifier of the `BasicBlock` and the condition have to be combined to finish the construction of the `BasicBlock`. This is done by the function `newBasicBlock()`. If the `BasicBlock` does not contain a condition, the condition pointer will be set to `NULL`. Both tokens indicating the start and end of a `BasicBlock` contain the identifier. If the identifiers are not equal, something went wrong during parsing. An error message is provided to the user in such cases.

While parsing the log files, the user will be kept informed about the progress. This is done by displaying the total file size to parse and the already parsed file size, along with the percentage expressing both. This data must be updated regularly, but not too often, since that will slow down the parsing process. Chosen was to do these updates in the `newBasicBlock()` function. `BasicBlock`-s are loaded frequently, but of course not as frequent as a statement occurs. To minimize the update process even further, an update is only being written to the screen if the parsed file size in MB actually changed.

`Condition`

There exists a great difference in conditions and other elements as discussed above. A `BasicBlock` can contain many `MemoryAccess`es, `ArrayAccess`es, `PointerAccess`es, `Operation`s and `FunctionCall`s, but it can contain only one `Condition`. This eliminates the need for a `ConditionLog` type. The condition itself contains both `MemoryAccess`es which are to be compared, the name of the condition and the source file and line.

From `BasicBlock` to `Function`

Each function has an `BasicBlocks` element. This `BasicBlocks` contains a list with all `MemoryAccess`es, a list with all `ArrayAccess`es, a list with all `PointerAccess`es, a list with all `Operation`s, a list with all `FunctionCall`s and a list with all `BasicBlock`-s. All access types, operations and calls have to be copied from all `BasicBlock`-s to the `BasicBlocks`. By doing this as soon as a complete `BasicBlock` has been parsed, the correct access order will be kept in `BasicBlocks`. The accesses, operations and calls will be removed from `BasicBlock`, to safe memory.

While the `MemoryAccess`es are copied to `BasicBlocks`, the `casts_from` lists are filled. After this, reports are able to find where an access casts to or where it was casted from.

The first `BasicBlock` can easilly be copied into `BasicBlocks`. When other `BasicBlock`-s need to be added, saving the elements is more complex. In all cases elements are merged whenever possible. If the new `BasicBlock` contains a variable that already exists in the `BasicBlocks`, the access logs are merged and the counters are updated. As soon as an element is no longer needed (because of a merge), that element is freed from memory. These merges are done on all elements, including

FunctionCalls. Merging PointerAccesses and ArrayAccesses are complex processes and are further explained in Section 4.4.1.3.

The new BasicBlock itself is also saved to BasicBlocks. If the block was added before, the passes counter is incremented. To test if a BasicBlock was added before, the identifier is used. After a whole function was parsed successfully, the passes counters of each BasicBlock contains the number of times the BasicBlock was executed. Each BasicBlock has a prevBlocks and a nextBlocks list. In these lists the identifiers of the BasicBlocks executed before and after the BasicBlock are saved. This information is needed to be able to detect loops from the execution order and number of passes. To keep track of the last parsed BasicBlock, the identifier of that block is saved in a stack. If another BasicBlock is parsed, the value saved in the stack is updated. When a new function is parsed while another function was still in the process of being parsed (i.e. a function call to another function being analyzed), a new layer is added to the stack. After the new function has been processed, this stack layer is removed.

4.4.1.3 Detecting ArrayAccess from PointerAccess

Probably the most complex operation done during parsing is differentiating accesses to arrays from pointer accesses. As was explained halfway Section 4.3.5.1, when passing an array as an argument to a function, the address of the first element of the array is passed. Accesses to that array are now complex pointer accesses. If the first element of the array is accessed, a PointerAccess to that address is logged by the PlugIn. Accesses to any other element are also logged as PointerAccesses, but this time to new pointer locations. These locations are calculated by the address offset of the element in question to the address of the first element of the array. This is detected by the PlugIn and an ArrayPointer element is logged. This ArrayPointer contains information about the name of the array, the index used, the name of the pointer created and the address of this pointer. This ArrayPointer is saved in a list. The list is saved in a stack, for the same reason as the BasicBlock identifier of the previous section was saved in a stack.

With the information described above being available during parsing, PointerAccesses to addresses that exist in the list with ArrayPointers can easily be detected as ArrayAccesses. The name and index used for that ArrayAccess are saved in the ArrayPointer and can be copied from there. But there is more to do. If a PointerAccess is recognized as an ArrayAccess and there already exist PointerAccesses with the same name as the ArrayAccess being detected, then these PointerAccesses are actually ArrayAccesses to index 0. Because the address of the first element (with index 0) is always known, it is not calculated by GCC and therefore not detected as ArrayPointer by the PlugIn. And if a BasicBlock is merged to BasicBlocks, it can occur that within BasicBlocks PointerAccesses exist with the same name as the ArrayAccesses detected in the newly to merge BasicBlock. In all cases the PointerAccess is converted to an ArrayAccess to index 0. If a PointerAccess is converted to an ArrayAccess, and the PointerAccess was used with an Operation, the link to the Operation needs to be updated as well.

In the two sections below, the places where PointerAccesses may be detected as

`ArrayAccess`es are pointed out.

Save `PointerAccess` into `BasicBlock`

The first step is to determine if the address of the pointer exists in the list of `ArrayPointers`. If it does, that would indicate that the `PointerAccess` is really an `ArrayAccess`. Now an `ArrayAccess` with the same name is searched. If not found, a new `ArrayAccess` is added, otherwise the existing `ArrayAccess` is used. After this, the list containing the `PointerAccess`es is searched for pointers with the same name. If they do exist, they are transformed into `ArrayAccess`es.

If no `ArrayPointer` was found with the same address as the new `PointerAccess`, the pointer list is searched for a `PointerAccess` with the same name. If that exists, the new access is merged with the existing one. If it does not exist, the list with `ArrayAccess`es is searched for an array with the same name. If such an access has been found, the `PointerAccess` is merged into the found `ArrayAccess`. If none of these was found, the `PointerAccess` is added to the list of `PointerAccess`es.

If an existing `PointerAccess` had to be transformed to an `ArrayAccess`, the `Operation`s using the old `PointerAccess` are updated to use the new `ArrayAccess`. This is done by function `updateOperationLinks()`.

Merging Accesses from `BasicBlock-s`

When a `BasicBlock` is saved into `BasicBlocks`, all accesses, operations and function calls are processed one by one. This prevents multiple accesses to be saved into `BasicBlocks` with the same name. When an `ArrayAccess` is being processed, the list of existing pointers is searched for `PointerAccess`es with the same name. If found, these `PointerAccess`es are transformed into `ArrayAccess`es to index 0. Of course, operation links are updated if applicable.

When a `PointerAccess` if processed, the list of existing `ArrayAccess`es is searched for an `ArrayAccess` with the same name. If found, the `PointerAccess`es are saved as `ArrayAccess`es to index 0.

4.4.2 Detecting Loops

One of the most important properties of the presented analyzer is the ability to keep track of loops. Almost all reports that can be generated can provide loop specific information. This comes in very handy when optimizing an algorithm for use with HLS tools, like *Catapult C*. Since hardware can do tasks concurrent, loops may be pipelined to improve throughput. Different examples of this are given in Chapter 5. If memory accesses occur within a loop, these are likely to be a bottleneck. Besides this, some operations are expensive. If these operations are used a lot within a loop, they can form a bottleneck as well. This is why being able to do analysis with loops is very important for this project.

Unfortunately, loops are not provided to the GCC PlugIn, they have to be detected by the PlugIn or the Parser. Since the goal was to keep the PlugIn as simple as possible, most detection is being done by the Parser. As already mentioned before, the Basic

Blocks are of help. The function `findLoops()` detects all loops. This section explains how it does that.

4.4.2.1 Basic Block Observations

Basic Blocks are discussed already a few times. The most important property of a Basic Block is that it is a collection of statements which are all executed if the Basic Block is executed. No conditional statements or jumps can occur within a Basic Block. If a conditional statement exists, it is always the last statement of that Basic Block. The conditional statement determines what Basic Block is to be executed next.

In practice, this means that al conditional statements, like `if .. then .. else` or loops with a conditional exit, will result in multiple Basic Blocks. If a Basic Block is executed multiple times, that would indicate that the Basic Block is part of a loop. Unfortunately, loops exist in a lot of varieties. A for-loop and a while-loop have their conditional check in the beginning, while a do-loop does the check at the end. Additionally, loops can contain `continue` or `break` statements, which would result in additional jumps. Also conditional executions (e.g. if-statements) may exist within a loop, generating more Basic Blocks.

To give a better overview of how Basic Blocks are created within loops or conditions, some examples are provided below.

Figure 4.12 shows a simple conditional statement with the Basic Block identifiers. On the far right side, the execution order is given. The debug option of the GCC PlugIn was used to find the Basic Block identifiers, while the generated log file was used to get the execution order.

```
Example code            BasicBlock ID        || Execution order
int main (void) {                            ||
    int a, b=5;         // id=0x40fe6940     ||
                                             || 0x40fe6940
    if (b < 3)                               || 0x40fe69c0
        a = 8;          // id=0x40fe6980     || 0x40fe6a00
    else                                     ||
        a = 5;          // id=0x40fe69c0     ||
}                       // id=0x40fe6a00     ||
```

Figure 4.12: Basic Blocks created with if-statement

More interestingly are loops. The next example is a simple for-loop in which a simple calculation is done. The initial value of the iterator is set in the BasicBlock before the loop starts. As expected, the BasicBlock containing the condition is executed four times, while the body of the loop was executed three times. This is shown in figure 4.13.

```
Example code          BasicBlock ID        ‖ Execution order
int main (void) {                          ‖
    int a = 8, i;     // id=0x40fe6940     ‖ 0x40fe6940
                                           ‖ 0x40fe69c0
    for (i=1;                              ‖ 0x40fe6980
         i<4;         // id=0x40fe69c0     ‖ 0x40fe69c0
         i++)         // id=0x40fe6980     ‖ 0x40fe6980
    {                                      ‖ 0x40fe69c0
        a *= i;                            ‖ 0x40fe6980
    }                                      ‖ 0x40fe69c0
}                     // id=0x40fe6a00     ‖ 0x40fe6a00
```

Figure 4.13: Basic Blocks created with simple for-loop

The structure of a while-loop is much like that of a for-loop. In both cases the initial value of the iterator is set before the loop starts and before each iteration the condition is checked. In the example of figure 4.14 only the type of loop has been changed for comparison. As can be seen, the execution order of both for-loop and while-loop is the same.

```
Example code          BasicBlock ID        ‖ Execution order
int main (void) {                          ‖
    int a = 8, i;     // id=0x40fe6900     ‖ 0x40fe6900
                                           ‖ 0x40fe6980
    i = 1;                                 ‖ 0x40fe6940
    while (i < 4)     // id=0x40fe6980     ‖ 0x40fe6980
    {                                      ‖ 0x40fe6940
        a *= i;       // id=0x40fe6940     ‖ 0x40fe6980
        i++;                               ‖ 0x40fe6940
    }                                      ‖ 0x40fe6980
}                     // id=0x40fe69c0     ‖ 0x40fe69c0
```

Figure 4.14: Basic Blocks created with simple while-loop

Do-loops check the condition at the end of each iteration. This changes how the statements are assigned to the Basic Blocks and also changes the execution order of the Basic Blocks. An example is given in figure 4.15.

In the example of figure 4.16, a more complex loop structure is given. This example contains a nested loop and uses a conditional `break` and conditional `continue` statement.

As expected, the code has been split up into Basic Blocks. Every time a conditional statement is found, a new Basic Block is started. Also the statements where the conditional statements can jump to, do always initiate a new Basic Block.

```
Example code                  BasicBlock ID     ||  Execution order
int main (void) {
    int a = 8, i;             // id=0x40fe6900  ||
                                                ||  0x40fe6900
    i = 1;                                      ||  0x40fe6940
    do                        // id=0x40fe6940  ||  0x40fe6940
    {                                           ||  0x40fe6940
        a *= i;                                 ||  0x40fe6940
        i++;                                    ||  0x40fe6980
    } while (i <= 4);                           ||
}                             // id=0x40fe6980  ||
```

Figure 4.15: Basic Blocks created with simple do-loop

```
Example code                          BasicBlock ID     ||  Execution order
                                                        ||  0x40fe6a80   0x40fe6ac0
                                                        ||  0x40fe6d00   0x40fe6c80
int main (void) {                                       ||  0x40fe6ac0   0x40fe6b00
    int a = 8, i, j;                  // id=0x40fe6a80  ||  0x40fe6c80   0x40fe6b80
                                                        ||  0x40fe6b00   0x40fe6c00
    for (   i=0;                                        ||  0x40fe6b80   0x40fe6c40
        i<2;                          // id=0x40fe6d00  ||  0x40fe6c00   0x40fe6c80
        i++)                          // id=0x40fe6cc0  ||  0x40fe6c40   0x40fe6b00
    {                                                   ||  0x40fe6c80   0x40fe6b40
        for (   j=0;                  // id=0x40fe6ac0  ||  0x40fe6b00   0x40fe6c40
            j<5;                      // id=0x40fe6c80  ||  0x40fe6b40   0x40fe6c80
            j++)                      // id=0x40fe6c40  ||  0x40fe6c40   0x40fe6b00
        {                                               ||  0x40fe6c80   0x40fe6b80
            if (j == 1)               // id=0x40fe6b00  ||  0x40fe6b00   0x40fe6c00
                continue;             // id=0x40fe6b40  ||  0x40fe6b80   0x40fe6c40
            if (j == 3)               // id=0x40fe6b80  ||  0x40fe6c00   0x40fe6c80
                break;                // id=0x40fe6bc0  ||  0x40fe6c40   0x40fe6b00
            a += (i*5) + j;           // id=0x40fe6c00  ||  0x40fe6c80   0x40fe6b80
        }                                               ||  0x40fe6b00   0x40fe6bc0
    }                                                   ||  0x40fe6b80   0x40fe6cc0
}                                     // id=0x40fe6d40  ||  0x40fe6bc0   0x40fe6d00
                                                        ||  0x40fe6cc0   0x40fe6d40
                                                        ||  0x40fe6d00
```

Figure 4.16: Basic Blocks created with more complex nested loops. (The execution order is presented from top to bottom and from left to right.)

4.4.2.2 Using Basic Block Information to Retrieve Loops

Each loop contains a number of Basic Blocks. The only question is, how to find out which Basic Block does belong to what loop and how to find the number of loops. During parsing, the number of passes of each Basic Block is saved. Also the Basic Blocks before and after a Basic Block are saved. For each Basic Block it is known which

Blocks may be executed before this Basic Block and which Blocks can be executed after this Basic Block. This information is used to detect loops. All Basic Blocks and their information of the example from figure 4.16 is shown in table 4.2.

Basic Block	**Passes**	**Previous BB**	**Next BB**
0x40fe6**a80**	1	NULL	0x40fe6**d00**
0x40fe6**d00**	3	0x40fe6**a80** 0x40fe6**cc0**	0x40fe6**ac0** 0x40fe6**d40**
0x40fe6**ac0**	2	0x40fe6**d00**	0x40fe6**c80**
0x40fe6**c80**	8	0x40fe6**ac0** 0x40fe6**c40**	0x40fe6**b00**
0x40fe6**b00**	8	0x40fe6**c80**	0x40fe6**b80** 0x40fe6**b40**
0x40fe6**b80**	6	0x40fe6**b00**	0x40fe6**c00** 0x40fe6**bc0**
0x40fe6**c00**	4	0x40fe6**b80**	0x40fe6**c40**
0x40fe6**c40**	6	0x40fe6**c00** 0x40fe6**b40**	0x40fe6**c80**
0x40fe6**b40**	2	0x40fe6**b00**	0x40fe6**c40**
0x40fe6**bc0**	2	0x40fe6**b80**	0x40fe6**cc0**
0x40fe6**cc0**	2	0x40fe6**bc0**	0x40fe6**d00**
0x40fe6**d40**	1	0x40fe6**d00**	NULL

Table 4.2: All Basic Blocks with their previous and next Basic Blocks and the number of passes from the example of figure 4.16

A number of conclusions can be drawn from the example of figure 4.16 and table 4.2 and with the help of other code fragments where was looked at in the same way:

- The first Basic Block which has more than one pass is part of a loop.
- If a Basic Block is part of a loop, then all its next Basic Blocks are also part of that loop if:
    1. The number of passes of the next Basic Block is not more than the number of passes of the original Basic Block.
    2. The number of passes of the next Basic Block is more than the number of passes of the level below the original Basic Block.
- If a Basic Block is part of a loop, then a next Basic Block is part of a nested loop if its number of passes is greater than the number of passes of the original loop.
- If a Basic Block is part of a Loop, then all its previous Basic Blocks are part of the same loop, if they are not already assigned to another loop.

Let's take a closer look to these conclusions. The first Basic Block which has more than one pass is a loop. This is trivial, since a Basic Block is executed only once if it is not part of a loop. If a Basic Block has more passes than its current level, then the Basic Block is a new loop and the number of passes of this Basic Block is the same as the number of iterations of the new found loop. This always holds, because the order in which the Basic Blocks occur in the list of Basic Blocks is the same as the order in which the Blocks were parsed. And that is the same order in which the Blocks were executed. The first executed Block of a loop is the conditional check (with for-loops and while-loops) or the top of the main body of the loop (do-loop). In both cases there exist no other Basic Blocks within the same loop with more iterations.

Important to notice from the example of figure 4.16 is that Basic Blocks within a loop can sometimes have very few iterations compared to the loop they exist in. Two examples of this are the `continue` and `break` statement in the inner loop. The Basic Blocks where these statement belong to are executed only twice, but their loop is executed eight times. Hence, if the next Basic Block of another Basic Block which is part of a loop has a number of passes less than or equal to the number of passes of the level above the current level, it cannot be determined yet to what level this next Basic Block belongs to. But if the number of passes was greater than the number of passes of the level above the current level, it is sure that this next Basic Block belongs to the current level and loop.

If a Basic Block belongs to a loop, then all its previous Basic Blocks belong also to that level, or they belong to the level above but are used to enter the current level. In the last case, the previous Basic Block is already assigned to a level (or loop), because it was this Basic Block which actually found the current Basic Block to be part of a new (nested) loop. Therefore, all previous Basic Blocks which are not yet assigned to another loop, are part of the current loop. This is very important, since it enables us to assign Basic Blocks to a loop which have a fewer of equal number of passes as the level above the current level.

The conclusions result in the following algorithm:

1. Walk over the Basic Blocks. If a Basic Block has more than one pass and is not already assigned to a loop, create a new loop for it and go to step 2. If all Basic Blocks have been seen, the loops are detected.

2. Walk recursively over the next Basic Blocks of the Basic Blocks already assigned to a loop. If a next Basic Block has more passes than the current Basic Block, create a new loop. If a next Basic Block has less or equal passes than the current Basic Block and it has more passes than the level above it, append the Basic Block to the current loop. If all already assigned Basic Blocks have been checked, continue to step 3.

3. Walk over all Basic Blocks which are already assigned to a loop. All previous Basic Blocks of these Blocks not already assigned to a loop belong to the current loop. Go to step 4.

4. If a Basic Block was assigned to a loop in step 2 or 3, then go to step 2 again. If no Blocks were appended to any loop, continue with step 1.

In table 4.3 the loops of the example of figure 4.16 are detected. Step 2 and 3 of the algorithm above are executed three times. The first two times, Basic Blocks are assigned to loops. This is presented in step 1 and 2 for the first iteration and in step 3 and 4 for the second iteration.

Loop	**Loop Passes**	**Previous Level Passes**	**Step 1: next Blocks**	**Step 2: previous Blocks**	**Step 3: next Blocks**	**Step 4: previous Blocks**
Root	1	N/A	0x40fe6**a80**			
Loop 1	3	1	0x40fe6**d00** 0x40fe6**ac0**	0x40fe6**cc0**		0x40fe6**bc0**
Loop 1_1	8	3	0x40fe6**c80** 0x40fe6**b00** 0x40fe6**b80** 0x40fe6**c00** 0x40fe6**c40**	0x40fe6**b40**		

Table 4.3: Detecting loops from Basic Blocks information from the example of figure 4.16

After all loops have been found, the start line and end line of the source code is set for the loops. This is done by finding the minimal and maximal source line number of the Basic Blocks of the loop.

One last property of the loop find algorithm to notice is that the number of iterations of a loop is set to the maximum number of passes within the loop. This is not entirely correct. For-loops and while-loops do check for a condition before starting an iteration. If such a loop is not quit using a break statement, the number of conditional checks is always one more than the number of iterations. This causes the presented framework to report one iteration to many in these cases.

The function `findLoops()` starts the algorithm to find all loops from the DFT. The function `findNewLoop()` is called when a new loop has been found. The function `walkBBforward()` is recursively called on all next Basic Blocks of Basic Blocks already assigned to a loop. This is done in step 2 of the algorithm and when a new loop has been detected. Function `walkBBreverse()` implements step 3 of the algorithm and tries to add previous Basic Blocks to loops.

4.4.3 Combining Struct Elements

After the parsing process has finished, all accesses (i.e. `MemoryAccess`, `ArrayAccess` and `PointerAccess`) have a saved name and a saved `StructElement`. A `StructElement` can have its own `StructElement`. This is saved in some kind of linked listed. If an access is not a struct, the `StructElement` setting is set to `NULL`.

The function `findStructElements()` tries to combine different accesses with the same root level `StructElement`(s). Each access contains three lists not mentioned before, namely: `element_memory`, `element_array` and `element_pointer`. These lists are to be filled by the function `findStructElements()`. If a struct named a has two elements $a.b$ and $a.c$, the saved structure before and after combining these accesses is as shown in figure 4.17.

```
a->name = "a";                          a->name = "a";
a->struct_element = NULL;               a->struct_element = NULL;
                                        a->element_memory[0] = b;
b->name = "b";                          a->element_memory[1] = c;
b->struct_element->name = "a";
b->struct_element->next = NULL;         b->name = "b";
                                        b->struct_element = NULL;
c->name = "c";
c->struct_element->name = "a";          c->name = "c";
c->struct_element->next = NULL;         c->struct_element = NULL;
```

Figure 4.17: Fragment of the DFT before (left) and after (right) combining struct elements

The first step is to try to find all accesses which do have a common root `StructElement`. In the previous example this would mean that for accesses $a.b$ and $a.c$, access a is found and that both elements $a.b$ and $a.c$ are saved into the root access a. This process is done by function `findStructElementsFunction()`.

The second step is to recursively find nested struct elements. This is done by function `findStructTree()`. An example of a nested struct element can be $a.b.c$ and $a.b.d$. In this example, the first step combines the common root a, while the second step combines the common parent $a.b$.

The difference in handling between `MemoryAccesses`, `ArrayAccesses` and `PointerAccesses` is minimal during these two steps. To reduce code replication, a fourth access type has been defined: `UniformAccess`. `UniformAccess` contains all common elements of the original accesses in the same order as the original accesses. This makes it possible to use a pointer cast to access all types of accesses with the same code.

4.4.4 Generating Reports

After the DFT has been completely formed and all post-processing has finished, reports can be generated. All reports come directly from the data within the DFT. Adding a new report is therefore very easy, as long as the data required by the report is available in the DFT.

4.4.4.1 Prompt

When the parser is ready for generating reports, a prompt is presented to the user. By entering commands to this prompt, the parser is controlled. Besides generating reports, some commands are available for selecting a function, loops or just to quit the parser. These commands are described below.

help presents basic help about the available commands. The output of the `help` command is shown below.

```
command> help
== HELP ==
show                         Shows report on screen.
save filename                Saves report to given filename.
save-h filename              Same as save, but hides the markup.
                               Data is seperated by '\t'.
show-plot name               Opens a graph showing the accesses of the
                               array named 'name'.
save-plot file-name.png name Saves a graph showing the accesses of the
                               array named 'name' in file file-name.png.
select function              Selects the function named 'function' to use
select-loop numbers          Selects the loop indicated by 'numbers'.
                               To select 'Loop_1_2', enter command
                                 'select-loop 1 2'.
                               To deselect a loop, enter 'select-loop 0'.
exit | bye | quit | q        Closes this application.

To get more information about the options of the show,
save and save-h command, type : help show.
```

exit,**bye**, **quit** and **q** will terminate the parser.

select is used to select a function. Except for one report, all reports show properties about a function. In order to generate these reports, a function has to be selected first.

select-loop selects a loop within a function. Some reports can output information about one loop. An example of such a report is the operation report. When a loop is selected, only the operations occurring in the selected loop are used to generate the report. To deselect a loop, enter '`select-loop 0`'.

4.4.4.2 Reports

All reports can either be shown on screen or be appended to a text file. To show the report on screen, the command '`show`' is used. To append a report to a file, use the '`save`' command and provide the file name with it. The special command '`save-h`' also appends the report output to a file, but does not use any markup. Data is separated by the tab character ('\t').

Different reports can be generated by providing multiple report commands behind the `show`, `save` or `save-h` command. The command `show memory operation` will show all memory accesses and after that all operations.

In this section, all available reports are presented and explained. The examples are all products of the function of figure 4.18. This function comes from the Lucas [38] Optical Flow algorithm. More about optical flow algorithms can be found in Appendix A. In this example, `PIC_Y` is defined as `120` and `PIC_X` as `80`.

```
void compute_ders_3x3(
int Ix[PIC_Y*PIC_X],
int Iy[PIC_Y*PIC_X],
int It[PIC_Y*PIC_X],
int pic0[PIC_Y*PIC_X],
int pic1[PIC_Y*PIC_X],
int pic2[PIC_Y*PIC_X])
{
    int x, y, n;
    int p,pp,tmp,tmp2;
    n = 7;

    for (x = 7; x < PIC_X-n; x++)
    {
        for (y = 7; y < PIC_Y-n; y++)
        {
            It[PIC_Y*x+y] = (pic2[PIC_Y*x+y] - pic0[PIC_Y*x+y]);
            Ix[PIC_Y*x+y] = (pic1[PIC_Y*(x+1)+y] - pic1[PIC_Y*(x-1)+y]);
            Iy[PIC_Y*x+y] = (pic1[PIC_Y*x+y+1] - pic1[PIC_Y*x+y-1]);
        }
    }
}
```

Figure 4.18: Example algorithm (part of Lucas [38])

`show functions` is the only report which does not need a function to be selected first. The report contains all by the GCC PlugIn logged functions. The source file and the source line where the function begins, is also reported. Passes indicates the number of times the function was executed. If another value than 1 is reported, all other reports contain only the data of the first execution.

```
command> show functions

                             Functions |      Source file |     Line |    Passes
================================================================================
                      compute_ders_3x3 |         top2.cpp |      747 |         1
```

`show memory` shows all accesses to variables other then arrays or pointers within the selected function (or loop). The name, type, number of read accesses from the variable and the number of write accesses to the variable are reported. If a variable is a cast, then this is indicated in front of the name of the variable. Also the number of times the variable was cast is presented. In some occasions, a variable is casted from two or more different variables. If this happens, an asterisk is shown. The cast variable is shown below each original variable. The presented data type is the name of the struct or class used, or *bool* for boolean variables or ***sint_x*** for integers, where *s* is 'u' in case of an unsigned integer and *x* is the number of bits used to represent the value. The data type *uint_8* is an unsigned integer data type using 8 bits. In *C* this would probably be the `unsigned char` data type. The example below is the shortened output of this report. This example shows a lot of temporary variables indicated with the *tmp*{...} prefix. This is because of the many multi operation statements.

```
command_(compute_ders_3x3)> show memory

  Memory access (excl. arrays) | DataType |    Reads |   Writes
 ---------------------------------------------------------------
                             x |   int_32 |    63097 |       67
                             n |   int_32 |     7129 |        1
                 tmp{106152} |   int_32 |       67 |       67
                 tmp{106153} |     bool |        0 |       67
                             y |   int_32 |    77022 |     7062
                 tmp{106158} |   int_32 |     7062 |     7062
                 tmp{106159} |     bool |        0 |     7062
                 tmp{106161} |   int_32 |     6996 |     6996
                 tmp{106162} |   int_32 |        0 |     6996
 |->CAST (6996x)   tmp{106163} |  uint_32 |     6996 |        0
(...)
* = This variable has more than one cast source. If nested casts exist,
these are also printed at each instance of this variable.
```

`show array` shows all array accesses of the selected function (or loop). This report has the same appearance as the memory report.

```
command_(compute_ders_3x3)> show array

                  Array access | DataType |    Reads |   Writes
 ---------------------------------------------------------------
                            Iy |   int_32 |        0 |     6996
                            Ix |   int_32 |        0 |     6996
                          pic1 |   int_32 |    27984 |        0
                            It |   int_32 |        0 |     6996
                          pic0 |   int_32 |     6996 |        0
                          pic2 |   int_32 |     6996 |        0
```

`show pointer` shows all pointer accesses of the selected function (or loop). This report has two columns more compared to the memory and array reports. These are the *Set Addr* and *Get Addr* columns, representing the number of times the address where the pointer points to was set or read, respectively. Because the example code does not contain any pointers, the report below is from another function. (In fact, there are many pointer accesses in this function. This is because the arrays are provided to the function using arguments. The parser has successfully translated all pointer accesses to array accesses. How this works is further explained in Section 4.4.1.3.)

```
command_(main)> show pointer

                Pointer access | DataType |     Reads |    Writes | Set Addr | Get Addr
 ------------------------------------------------------------------------------------
                             d | *  int_8 |         0 |         1 |        1 |        0
```

show memory-access *x* shows all accesses to the memory named *x* within the selected function (or loop). The report presents the value which was read or written, whether the access was a read (indicated with *R*) or a write (indicated with *W*). Also the source file and line number of the access are reported. The example below is truncated.

```
command_(compute_ders_3x3)> show memory-access n
  ==> Memory n
            Value | R/W |      Source File |  Line No
-------------------------------------------------------
                7 |  W  |         top2.cpp |      768
                7 |  R  |         top2.cpp |      780
                7 |  R  |         top2.cpp |      782
                7 |  R  |         top2.cpp |      782
(...)
```

show array-access *x* shows all accesses to the array named *x* within the selected function (or loop). The report also shows the indices of the array. If a multi-dimensional array is reported, all indices (including their values) are shown. The example below is again truncated.

```
command_(compute_ders_3x3)> show array-access Ix
  ==> Array Ix
           Index Variable |           Index Value | R/W |      Source File |  Line No
 ------------------------------------------------------------------------------------
 [tmp{106182}]            | [3388]                |  W  |         top2.cpp |      794
 [tmp{106182}]            | [3392]                |  W  |         top2.cpp |      794
 [tmp{106182}]            | [3396]                |  W  |         top2.cpp |      794
 [tmp{106182}]            | [3400]                |  W  |         top2.cpp |      794
(...)
```

show pointer-access *x* shows all accesses to the pointer named *x* within the selected function (or loop). The access type is indicated with *R* (read), *W* (write), *S* (set address) or *G* (get address). The address and value are shown if known. The example below is of the same function as where the pointer report was generated from.

```
command_(main)> show pointer-access d
  ==> Pointer d
            Value |    Address | R/W/S/G |      Source File |  Line No
-----------------------------------------------------------------------
                0 | 0xbf8d6b9d |    S    |         main.cpp |      374
                9 | 0xbf8d6b9d |    W    |         main.cpp |      375
```

show memory-min-max *x* shows the minimal and maximal value of the variable named *x* within the selected function (or loop). Only boolean and integer type values are supported.

```
command_(compute_ders_3x3)> show memory-min-max x
  ==> Memory x
MAX value = 73
MIN value = 7
```

show operation shows all types of operations used within the selected function (or loop). The number of times the operation was used is given as well. In the example below, the number of multiplications is twice as many as one would probably expect. This is because the address of the pointer needed to access the array elements is calculated. This calculation uses a multiplication to get the offset from the first element. This is further explained in section 4.4.1.3.

```
command_(compute_ders_3x3)> show operation

                    Operations |                               Count
 ---------------------------------------------------------------
                  greater-than |                                7129
                         minus |                               28117
                          plus |                               98010
                          mult |                              125928
```

show operator-access *x* shows all instances in which the operation named *x* is used within the selected function (or loop). The operands and the result variable are reported, along the source file and line. The example below shows the *greater-than* operator. This operator is used in the conditional check of both for-loops. In this specific case, the results are saved in temporary variables of type `bool`. These variables are used in the conditional statements of the for-loop. The example below is truncated.

```
  ==> Operation greater-than
    Operand 1 |    Operand 2 |       Result |      Source File |  Line No
---------------------------------------------------------------------------
  tmp{106152} |            x |  tmp{106153} |         top2.cpp |      780
  tmp{106158} |            y |  tmp{106159} |         top2.cpp |      782
  tmp{106158} |            y |  tmp{106159} |         top2.cpp |      782
  tmp{106158} |            y |  tmp{106159} |         top2.cpp |      782
  tmp{106158} |            y |  tmp{106159} |         top2.cpp |      782
(...)
```

show condition shows all conditional statements used within the selected function (or loop). The report looks the same as the *operation* report.

```
command_(compute_ders_3x3)> show condition

          Conditional Branches |                               Count
 ---------------------------------------------------------------
                     not_equal |                                   2
```

show loop shows all loops within the selected function. Each loop has a unique sequence of numbers. All loops existing in the top most level are numbered as `loop_x`, where `x` is a unique sequential number. If a loop contains a nested loop, the name of the nested loop is the name of its parent loop with as suffix `_x`, where x is again a unique sequential number. Loop `loop_1_1` is the first nested loop of the first loop. Besides the loop name, the source file and the line numbers at which the loop starts and ends are given. These line numbers are found by looking at the Basic Block line numbers, which are found by looking at the statement line numbers. Therefore, the presented line numbers are the line numbers of the first and last statement of a loop, and not the line numbers where the developer placed the curly brackets for example. The number of nested loops and passes is also provided. As explained in section 4.4.2, the number of passes (or iterations) can be of by one, because of the additional conditional check of most loop types. The number of passes of nested loops is the total amount of passes. If no `break` statements were used in these loops, the actual number of iterations can be found by dividing the presented amount of passes by the number of passes of its parent loop. In the example below, the number of iterations of the nested loop per iteration of its parent loop is $\frac{7062}{67-1} - 1 = 106$.

```
command_(compute_ders_3x3)> show loop

          Loops |        Src File | Start Ln |   End Ln |   Nested |   Passes
 ---------------------------------------------------------------------------
         loop_1 |        top2.cpp |      780 |      800 |        1 |       67
       loop_1_1 |        top2.cpp |      782 |      800 |        0 |     7062
```

`show call` shows all function calls within the selected function. The report contains the function name of the call, the number of parameters, whether the function returned a value and the number of calls to these functions. Operations performed on classes are function calls to their definitions (e.g. `MyClass::operator+` is the function which defines the `+` operation of class `MyClass`). The example below is from a different part of the Lucas algorithm (the `eigensolve()` function), because the example of figure 4.18 does not contain any function calls.

```
command_(eigensolve)> show call

                     Function Calls | # Params |   RetVal |    Calls
 -----------------------------------------------------------------
    ::ops_with_other_types::operator== |        2 |      YES |        1
               ::ac_fixed::__comp_ctor |        2 |       NO |       33
     ::ops_with_other_types::operator/ |        2 |      YES |        2
                 ::ac_fixed::operator- |        2 |      YES |        8
                 ::ac_fixed::operator+ |        2 |      YES |       10
                                ::sqrt |        2 |       NO |        3
                 ::ac_fixed::operator* |        2 |      YES |       21
     ::ops_with_other_types::operator* |        2 |      YES |        1
                 ::ac_fixed::operator> |        2 |      YES |        2
                                 ::div |        3 |       NO |        2
```

`show dependency` shows all dependent accesses within the selected loop. In column *MAP* a single character shows the type of dependency. The values can be *M* (memory), *A* (array) or *P* (pointer). The *Depend.* column shows the number of times a read access occurred after a write access within the selected loop. If the variable was written and then read, the dependency also increases. The *Writes* column shows the number of times a variable was written. If the variable is of type array, the presented values are valid for the most dependent element of the array. If the variable is pointer type, getting and setting the address can also influence the dependency. Dependencies are interesting when considered is to pipeline a loop. Indices and iterators are typically shown in dependency lists. These are most likely not problematic when using pipelining. If actual variables or memory places (array elements) are dependent, pipelining can become a problem. Sometimes a dependency shows optimization opportunities. If an array element is written multiple times, sometimes the value written can be temporarily saved in a register. Only the last write is actually needed then. The example below shows a fraction of the dependency report of the example of figure 4.18.

```
command_(compute_ders_3x3->Loop_1_1)> show dependency

  MAP |                  Variable Name |  Depend. |   Writes
 -----------------------------------------------------------
   M  |                              y |     6997 |     6996
   M  |                    tmp{106158} |     7062 |     7062
   M  |                    tmp{106159} |        0 |     7062
   M  |                    tmp{106161} |     6996 |     6996
(...)
```

4.4.4.3 Plots

One of the most powerful features of the presented framework is generating plots. At the current version, plots of the index order of arrays can be generated. A nice addition for the future can be generating plots of the values of memories or arrays. Plots are generated using gnuplot [39]. The command **`show-plot`** ***`x`*** and **`save-plot`** ***`filename.png x`*** are used to show or save a plot, respectively. Here x denotes the array to plot. If the plot is to be saved, the PNG file format is used. In the example plot of figure 4.19, array `Ix` of the example from figure 4.18 is shown. The y-axis represents the index of the array access. The x-axis represents the access number. Reading the plot from left to right gives an overview of the order in which the indices were called. In the shown example, it can clearly be seen that the first access to the array started around index 4000. Later accesses use higher indices, since the graph is increasing. By zooming in using the user interface of gnuplot, individual accesses can be seen. Different examples of this are presented in Chapter 5. If the index used in an access was 0, a value of -1 is used in the plot. This is done because a value of 0 cannot be seen in a graph like this. Each access is represented by a vertical bar. An access to index 5 will result in a bar with a height of 5. If index 0 was accessed, the height of the bar would be 0 (and therefore not visible). By setting the height to -1 in this case, the bar becomes visible abain.

4.4.5 Parser / Analyser Arguments

As with most applications, the Parser / Analyzer can handle arguments. This section describes these arguments.

`--help` prints the common help page on the screen. A short description of the available arguments is given.

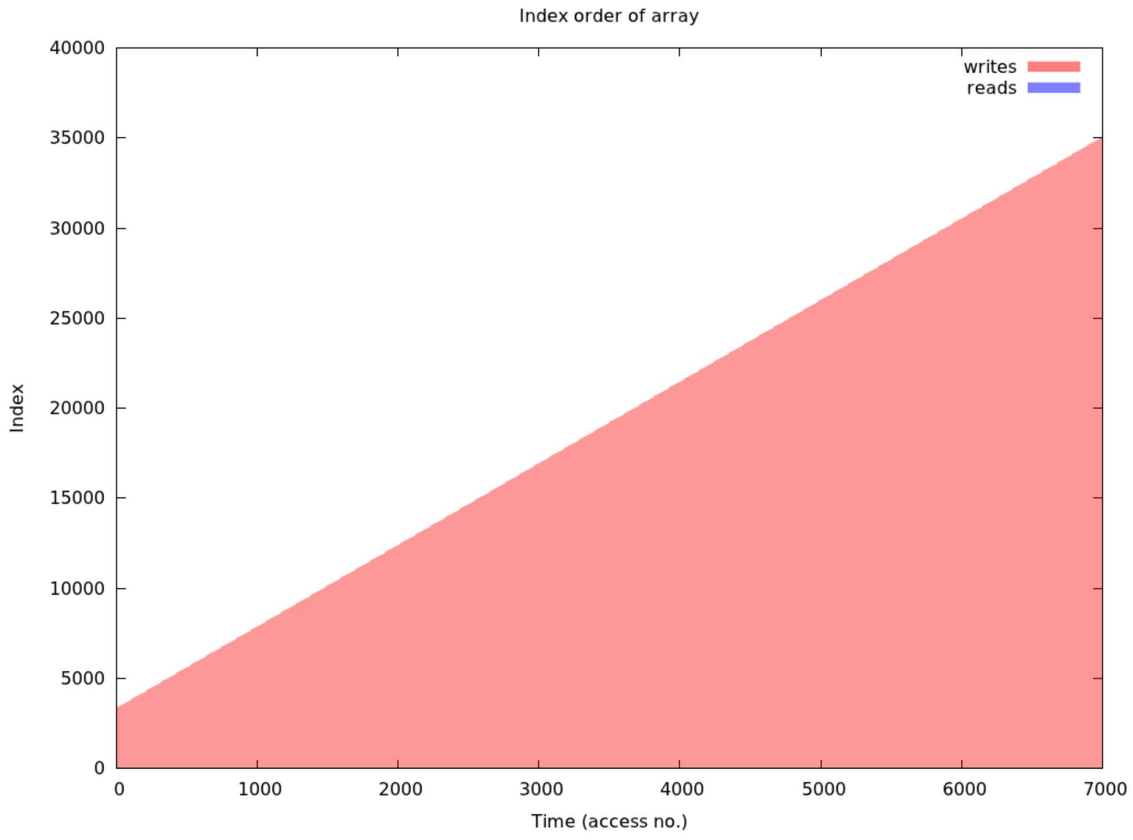

Figure 4.19: Plot of the access pattern of array `Ix`

`--command` *`commands`* is used to run commands right after the parsing process is completed. The prompt is given after the commands provided here are finished. Multiple commands can be separated using a semicolon. This argument can be very helpful when using the parser in an automated environment.

`--command-file` *`filename`* is used for executing commands from a file. Multiple commands can be separated using a semicolon or a new line ('\n'). The prompt is given after all commands are executed.

`-h` can be used to hide the title which is normally shown when the Parser / Analyzer is loading.

`-s` skips the prompt. This option can be used in combination with the `--`*`command`* or `--`*`command-file`* arguments to automatically close the Parser / Analyzer if all commands are executed.

`-t` shows the detected tokens. This option is useful when debugging the grammar code.

`-d` saves duplicate functions. If a function is executed multiple times, only information about the first execution is saved. When this option is used, all passes of the function are saved. At this moment, no reports are making use of the additional information being saved.

4.5 Conclusion

The presented framework consists of two parts. The first part is the GCC PlugIn which inserts function calls to analyze functions into the AST of GCC. By compiling an algorithm with GCC and the presented GCC PlugIn, the resulting executable file will generate a log file (or log files, depending on the resulting log file size) when executed. This log file is parsed by the second part of the presented framework: the Parser / Analyzer. This part of the framework generates a Data Flow Tree from the log files and allows the user to generate reports from this DFT interactively.

Chapter 5 uses the Lucas & Kanade [38] Optical Flow algorithm as a case study to present the effectiveness and capabilities of the presented framework.

Results: Optimizing the Lucas Algorithm 5

The Presented framework is put to the test on a real algorithm, namely the Lucas & Kanade [38] Optical Flow algorithm. This chapter describes the very basics of this algorithm and how it was analyzed using the framework. The performed optimizations are discussed and their results are presented. At the end of this chapter, it will be clear that a dramatic speedup has been achieved compared to a previous implementation.

5.1 The Lucas & Kanade Algorithm

This section describes the basic working of the Lucas & Kanade algorithm and how it has been implemented before in hardware, and how it can be analyzed by the framework.

5.1.1 Basic Working

The basic working of the Lucas & Kanade algorithm from a designer's point of view is presented here. More information about Optical Flow algorithms in general can be found in Appendix A. More information about the mathematical working of the Lucas & Kanade algorithm can be found in Section A.2.1.2 and in the official papers of Lucas [38] and Lucas & Kanade [5].

The Lucas & Kanade implementation used in this chapter is the one presented by Ren [40], who adopted the implementation of Hurkmans [41].

As shown in figure 5.1, the algorithm contains three basic steps. The first step is to blur the input images. The second step is to calculate the spatial and temporal derivatives from the blurred input images. The third step is to calculate the velocities from the spatial and temporal derivatives.

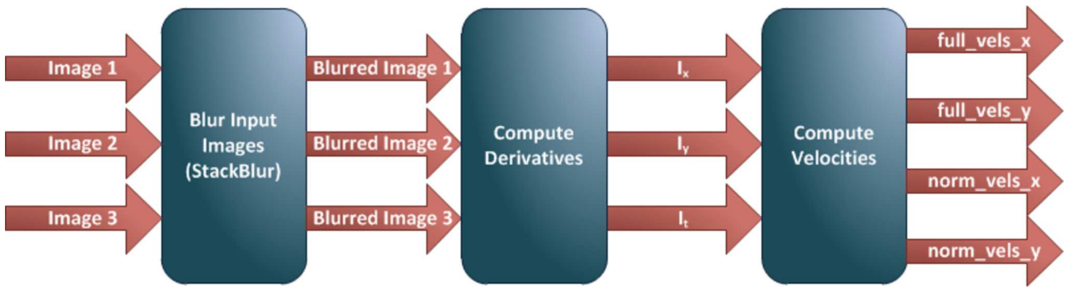

Figure 5.1: Basic system overview Lucas algorithm

5.1.1.1 Step 1: Blur the Input

In the used implementation, three input images are needed at a time to calculate the optical flow. All these input images need to be blurred. Different blurring algorithms have been used over time. Examples are a Gaussian filter or the StackBlur [42] filter. In the used implementation, the StackBlur algorithm is used because of the lower computational intensity. StackBlur blurs the image in horizontal and vertical direction separately. Each pixel is given the weighted average of the n pixels around it. The closer a pixel lays to the pixel being blurred, the higher the weight of that pixel. The value of n has to be an odd number, so the pixel being blurred has as many pixels at one side of itself as it has on the other side. Higher values for n will result in a more blurred image. Lower values for n will result in a less blurred image. The beauty of the StackBlur algorithm is where it got its name from: it uses a stack. The stack is used to minimize the amount of calculations by keeping tracks of the values of the pixels used to blur. More on this is explained in the thesis reports of Hurkmans [41] and Ren [40].

5.1.1.2 Step 2: Compute Derivatives

There are a number of ways to calculate the derivatives. Hurkmans [41] did test a couple of options and found his three-point central differencing filter implementation to be the best tradeoff between accuracy and computational effort. This filter does a subtraction of the two adjacent pixel values in x, y or t direction to find the derivatives. To find all three derivatives (I_x, I_y and I_t), the values of six pixels are loaded and subtracted.

5.1.1.3 Step 3: Compute Velocities

The velocities are calculated by the Least-squares method. Again, Hurkmans [41] found a tradeoff. This time it is to use a 3x3 neighborhood size for the LS computation. Nine values of I_x, I_y and I_t are loaded and different calculations are performed on them.

5.1.1.4 Connecting all Computing Blocks

The input of the algorithm exists of three memories containing the three input images. In the original source code these memories are called `f1i`, `f2i` and `f3i` for the first, second and third image of a sequence, respectively. The input images are connected to the input of the StackBlur computing block.

The connection between the StackBlur and the Derivatives computing block is done using the three memories `f1o`, `f2o` and `f3o` for the blurred versions of the first, second and third input image, respectively.

The connection from the Derivative and the Velocity computation block is done by the three memories `Ix`, `Iy` and `It` for the spatial derivative in the x direction, the spatial derivative in the y direction and the temporal derivative (i.e. in t direction), respectively.

The output of the Velocity computation block is also the output of the algorithm. The output memories are `full_vels_x`, `full_vels_y`, `norm_vels1_x` and `norm_vels1_y` for the full velocities in x and y direction and the normal velocities in x and y direction, respectively. More information about and the definitions of full and normal velocities can be found in Section A.2.1.

5.1.2 Previous Hardware Implementations

Within the Circuits and Systems group where this thesis work has been performed, the Lucas algorithm has been implemented in hardware at least two times before. The first time, Hurkmans [41] manually transformed the algorithm into a *VHDL* description and was able to successfully run simulations. More recently, Ren [40] made adjustments to the existing *C* code, to make it compatible with the *Catapult C* HLS tool. The *C* code of Ren was used in this chapter to find optimization opportunities.

5.1.3 Prepare for Analysis

A couple of small changes were done on the given source code before the presented framework was put to the test. The first change was adding a header file to the code. The original source code contained two source files, namely a file containing the Lucas algorithm and a file containing the test bench. By adding a header file for the original Lucas source file, a single file is used to refer to from both source files. This comes in handy when the include to the analyze library is done as discussed in Section 4.3. The code fragment where the include is done, is shown in figure 5.2.

The second change was to place the definition of the image size at a central location. In the original code, each function had its own size definition. Although the sizes throughout the whole algorithm were defined the same, it was not possible to adjust the size efficiently. The original image size is `316 x 252`. When analyzing the algorithm, this size can proof to be very big. Chosen was to lower the image size if the algorithm is analyzed and use the original size in any other situation. The lower resolution used when analyzing the algorithm will improve the analyze speed. The lower size is defined to be `120 x 80`. The code fragment of figure 5.2 shows how this is done. If `ANALYZE` is defined, the analyze library is loaded and the image size is reduced.

```
#ifdef ANALYZE
#include <analyze.h>
#define PIC_X 80   //smaller image to test with
#define PIC_Y 120
#else
#define PIC_X 252
#define PIC_Y 316
#endif
```

Figure 5.2: Include `analyze.h` and defining the image size (code fragment from: `top2.h`)

5.2 Optimizing Hardware Implementation

In the following sections, the original algorithm as provided by Ren [40] is optimized. Different reports can be generated to find optimization opportunities. Only reports and plots leading to an optimization are showed below.

First, all three parts of the algorithm are optimized. After this, the pipelines are adjusted to better match the optimized algorithm. During the optimization process, the same goal as Ren had was used for fair comparison. This goal is speed, regardless of resource usage. Although in some situation it occurred that the increase in resources was considered completely off balance and therefore not implemented. In cases where the design decision was made in favor of resource usage over speed, this is explicitly mentioned.

5.2.1 Optimizing Step 1: StackBlur

The StackBlur computational block reads three images, performs some calculations on them and writes the results to three memories. A good start can be opening the array accesses report. The output is shown below.

```
command_(stackblur)> show array

                 Array access | DataType |    Reads |   Writes
 --------------------------------------------------------------
                      datain3 |  uint_8  |    19200 |     9600
                      datain2 |  uint_8  |    19200 |     9600
                      datain1 |  uint_8  |    19200 |     9600
                      stackx3 |  int_32  |    28800 |    10440
                      stackx2 |  int_32  |    28800 |    10440
                       stackx |  int_32  |    28800 |    10440
                      stacky3 |  int_32  |    28800 |    10160
                      stacky2 |  int_32  |    28800 |    10160
                       stacky |  int_32  |    28800 |    10160
                     dataout3 |  int_32  |        0 |     9600
                     dataout2 |  int_32  |        0 |     9600
                     dataout1 |  int_32  |        0 |     9600
```

The `datain`*x* arrays are the image input memories and the `dataout`*x* arrays are the blurred image output memories. All stack arrays are small arrays and are implemented in registers by *Catapult C*.

A few things stand out. Since the image size was defined to be 120 x 80, the number of accesses to the input and output memories are expected to be 9600. But, the input memories are read twice as often and are also written. Another thing to notice is that the data type of the output memory is wider than one would expect. The StackBlur algorithm calculates a new weighted average for each pixel. The new value of a pixel lays therefore always somewhere between the maximal and minimal value of the pixels where the average was taken about. If all input pixels could be represented in a data type of 8 bits, the output pixels should be representable in the same 8 bit data type. Therefore, the output data type was changed to an 8 bit representation, minimizing memory usage. At this moment, no reports exist viewing the minimal or maximal value of an array, but there is a report showing the minimal and maximal value of a variable. Since the values to write to the output array are first saved in a variable, the min-max report of this variable was generated and shown below. As can be seen, the values stay well between the bounds of the 8 bit unsigned data type (0 - 255).

```
command_(stackblur)> show memory-min-max value
  ==> Memory value
MAX value = 154
MIN value = 52
```

Let's take a look at the many accesses to the input memories. The StackBlur algorithm has two major loops. In one loop the rows are blurred, while in the other loop the columns are blurred. The dependency report of the first loop is given below.

```
command_(stackblur->Loop_1)> show dependency

  MAP |                 Variable Name |  Depend. |   Writes
 ----------------------------------------------------------
(...)
   A  |                       datain3 |        1 |        1
   A  |                       datain2 |        1 |        1
   A  |                       datain1 |        1 |        1
   A  |                       stackx3 |     1440 |     1560
   A  |                       stackx2 |     1440 |     1560
   A  |                        stackx |     1440 |     1560
```

All normal variables have been removed from this report to save space. These variables are mostly temporal variables and the loop iterators. The shown stack arrays are used as variables as well. More interestingly are the input memories. There is a dependency in the first loop in the input memories. The report shown below presents the array dependencies of the second loop.

```
command_(stackblur->Loop_2)> show dependency

  MAP |                 Variable Name |  Depend. |   Writes
 ----------------------------------------------------------
(...)
   A  |                       stacky3 |     1440 |     1520
   A  |                       stacky2 |     1440 |     1520
   A  |                        stacky |     1440 |     1520
```

Here, the input memories do not occur in the dependency list. Hence, the unexpected write accesses occur in the first loop. Figure 5.3 shows the access pattern plot of `datain1`. It can clearly be seen that the first 67% of the accesses are reads and writes and the last 33% contains reads only. The second part comes from the second loop, while the first part comes from the first loop. Two magnifications are shown in the figure. Clear is that all elements are first read and a few accesses later been written.

After locating the accesses of the `datain1` array in the source code, it becomes clear that this memory is used as input memory and as buffer between the two stages of the StackBlur algorithm (i.e. the two loops). Using the input memories to write to is not

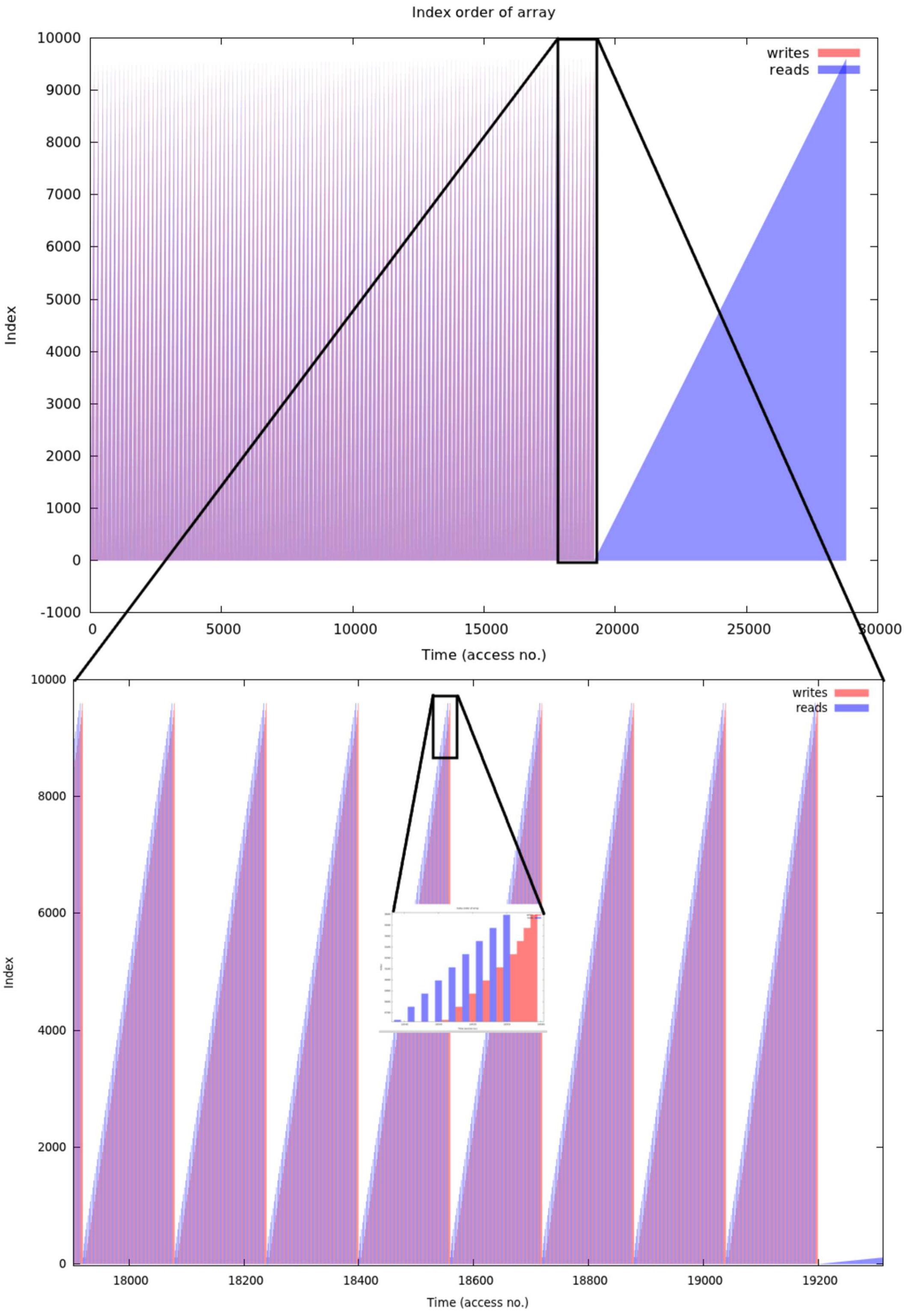

Figure 5.3: Array access pattern of `datain1` before optimizations are done

the best looking solution here. Besides that, next iterations are again reading three images. Two of these images are read before by the iteration before. By reading a partly blurred image as if it is the original image will of course cause over blurred images. A good optimization might be reusing the two already blurred images, although this is not implemented for now. More on this is explained in Section 5.4.1.

Using another memory to buffer the images between the two stages is desirable. A few options exists. The first option is using the StackBlur output memories. This option has the disadvantage of causing a new dependency in the second loop, because the output memories are first read and then written from within the same loop. Another option is to introduce three new memories to buffer the data. This will increase the total memory usage dramatically and is not considered to be desirable. The third option is to use the memories that exist between the Derivative and Velocity computational blocks (`Ix`, `Iy` and `It`). These memories are not used yet and can safely be used here. The disadvantage here is that if the design must be pipelined over the three main stages, the memories are used by two blocks at the same time, which will cause failure. Since pipelining the whole design is not done for now, this third option was chosen.

After this optimization was implemented, the source code was manually analyzed further. A couple more optimizations where found during this quick overview. The first is the assignment of the `addr` variable. This variable is used to point to the pixel of interest in both input and output memories. The value of `addr` is in almost all cases trivial: one more than the previous iteration. Although the address could be calculated with a simple addition, it was calculated with a multiplication and an addition. This was changed in the code. An example of this can be found in figure 5.4. A second optimization was performed in one of the nested loops. The loop is shown in figure 5.4.

The value of `rxy` is defined as 3. If the resulting values of `sum`x and `sum_in`x are taken into account, it can be seen that this loop can be reformed, eliminating all multipliers while the final result is not affected.

Another optimization done to the StackBlur computational block is shown in figure 5.5. Here the values of `sum`x and `sum_out`x are calculated by a lot of operations. This can easily be simplified as shown.

```
for(i = 1; i <= rxy; i++)
{
  if(i <= wm)
    src_pix_index_x =
src_pix_index_x + 1;

  addr = PIC_Y * (src_pix_index_x) +
src_pix_index_y;

  pix = datain1[addr];
  pix2 = datain2[addr];
  pix3 = datain3[addr];

  stackx[i + rxy] = pix;
  stackx2[i + rxy] = pix2;
  stackx3[i + rxy] = pix3;
  sum = sum + pix * (rxy + 1 - i);
  sum2 = sum2 + pix2 * (rxy + 1 - i);
  sum3 = sum3 + pix3 * (rxy + 1 - i);
  sum_in = sum_in + pix;
  sum_in2 = sum_in2 + pix2;
  sum_in3 = sum_in3 + pix3;
}
```

```
for(i = 1; i <= rxy; i++)
{
  int irxy = i + rxy;

  addr = addr + PIC_Y;

  pix1 = datain1[addr];
  pix2 = datain2[addr];
  pix3 = datain3[addr];

  stackx1[irxy] = pix1;
  stackx2[irxy] = pix2;
  stackx3[irxy] = pix3;
  sum_in1 = sum_in1 + pix1;
  sum_in2 = sum_in2 + pix2;
  sum_in3 = sum_in3 + pix3;
  sum1 = sum1 + sum_in1;
  sum2 = sum2 + sum_in2;
  sum3 = sum3 + sum_in3;
}
```

Figure 5.4: Inner loop of StackBlur before optimization (left) and after optimization (right)

```
pix = datain1[addr];
pix2 = datain2[addr];
pix3 = datain3[addr];

for(i = 0; i <= rxy; i++)
{
  stackx[i] = pix;
  stackx2[i] = pix2;
  stackx3[i] = pix3;
  sum = sum + pix * (i + 1);
  sum2 = sum2 + pix2 * (i + 1);
  sum3 = sum3 + pix3 * (i + 1);
  sum_out = sum_out + pix;
  sum_out2 = sum_out2 + pix2;
  sum_out3 = sum_out3 + pix3;
}
```

```
pix1 = datain1[addr];
pix2 = datain2[addr];
pix3 = datain3[addr];

sum_out1 = (unsigned)pix1 << 2;
sum_out2 = (unsigned)pix2 << 2;
sum_out3 = (unsigned)pix3 << 2;
sum1 = sum1 + pix1 * 10;
sum2 = sum2 + pix2 * 10;
sum3 = sum3 + pix3 * 10;

for(i = 0; i <= rxy; i++)
{
  stackx1[i] = pix1;
  stackx2[i] = pix2;
  stackx3[i] = pix3;
}
```

Figure 5.5: Inner loop of StackBlur before optimization (left) and after optimization (right)

One last optimization done on the StackBlur computational block is a strange looking operation which is located in the inner loop of this block. This is shown in figure 5.6. First, a value is multiplied by 512 and then shifted 13 positions to the right. Multiplying by 512 is the same as shifting 9 positions to the left. And shifting 9 positions to the left and then shifting 13 positions to the right is the same as shifting 4 positions to the right. A very simple, though effective optimization.

The operation report of the original StackBlur implementation is shown below.

```
command_(stackblur)> show operation

                  Operations |                           Count
 --------------------------------------------------------------
          less-than-or-equal |                           21402
                        plus |                          303800
                        mult |                          129000
                       minus |                          204960
                     convert |                           28800
                      rshift |                           57600
```

```
mul_sum_x = mul_sum_y = 512;
shr_sum_x = shr_sum_y = 13;                 value1 = sum1 >> 4;
value = (sum * mul_sum_x) >> shr_sum_x;     value2 = sum2 >> 4;
value2 = (sum2 * mul_sum_x) >> shr_sum_x;   value3 = sum3 >> 4;
value3 = (sum3 * mul_sum_x) >> shr_sum_x;
```

Figure 5.6: Source code fragment of strange operations before optimization (left) and after optimization (right)

5.2.1.1 New Report Results

After the optimizations are implemented, the results shown by the used reports are changed. This section shows the same reports and (basic) plots as before, but now the optimized algorithm is used as input.

First the new array access report is shown. It can be seen that the accesses to both input and output memories are as expected 9600. The memory accesses to the buffer memories (`Ix`, `Iy` and `It`) are 9600 writes and 9600 reads. The plot (without the magnifications) of array `datain1` is shown in figure 5.7.

```
command_(stackblur)> show array

                Array access | DataType |    Reads |   Writes
 --------------------------------------------------------------
                     stackx3 |   int_32 |    28800 |    10440
                     stackx2 |   int_32 |    28800 |    10440
                     stackx1 |   int_32 |    28800 |    10440
                     datain3 |   uint_8 |     9600 |        0
                     datain2 |   uint_8 |     9600 |        0
                     datain1 |   uint_8 |     9600 |        0
                          It |   int_32 |     9600 |     9600
                          Iy |   int_32 |     9600 |     9600
                          Ix |   int_32 |     9600 |     9600
                     stacky3 |   int_32 |    28800 |    10160
                     stacky2 |   int_32 |    28800 |    10160
                     stacky1 |   int_32 |    28800 |    10160
```

```
        dataout3 |    uint_8 |          0 |      9600
        dataout2 |    uint_8 |          0 |      9600
        dataout1 |    uint_8 |          0 |      9600
```

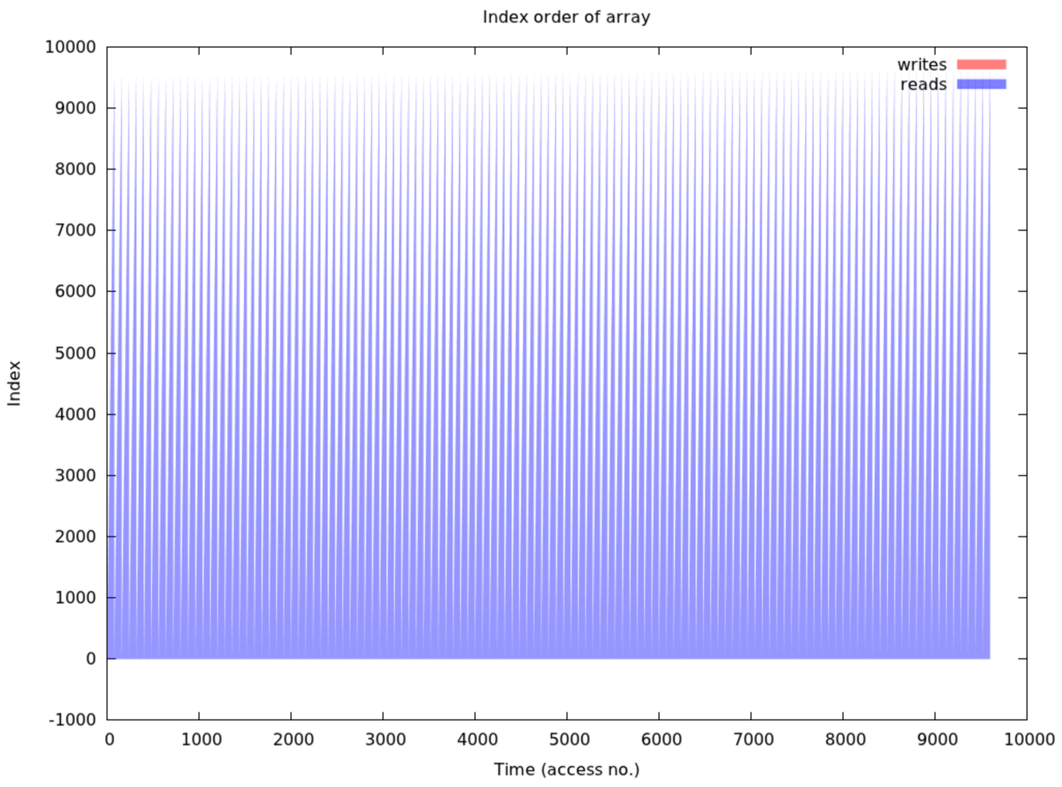

Figure 5.7: Array access pattern of `datain1` after optimizations are done

The new operation report shows a great decrease of multiplications. A new operation is added to the list. This is the *lshift* operation used in figure 5.5.

```
command_(stackblur)> show operation

                      Operations |                             Count
 ---------------------------------------------------------------
              less-than-or-equal |                             21402
                            plus |                            293360
                            mult |                             58320
                          lshift |                               600
                           minus |                            203160
                         convert |                             86400
                          rshift |                             57600
```

5.2.2 Optimizing Step 2: Compute Derivatives

The Derivative computational block again reads three input memories, does some calculations, and writes the result to three output memories. The input memories contain the blurred images produced by the StackBlur computational block. These memories are called `pic0`, `pic1` and `pic2`. The output memories are the three derivatives and are called `Ix`, `Iy` and `It`. The first step would be to generate the array access report.

```
command_(compute_ders_3x3)> show array

               Array access | DataType |    Reads |   Writes
 --------------------------------------------------------------
                         Iy |   int_32 |        0 |     6996
                         Ix |   int_32 |        0 |     6996
                       pic1 |   int_32 |    27984 |        0
                         It |   int_32 |        0 |     6996
                       pic0 |   int_32 |     6996 |        0
                       pic2 |   int_32 |     6996 |        0
```

The array report shows that all outputs and two of the inputs are accessed 6996 times, which is the number of pixels of the images minus an offset. Only the second input memory (`pic1`) is accessed more often. Figure 5.8 shows the access profile of this array. Some magnifications are shown to give a better idea of how the array was accessed.

The sudden changes in index are caused by the change of one image line to the next. At the beginning and the end of an image line, an offset exists, which causes the sudden change in index. Concluded from the two magnifications in the bottom right corner of the figure can be that there are four sequential accesses repeating themselves. The repetitions are the iterations of a loop. One access can be found in the lower half of the figure, two in the middle and one in the upper half of the figure. Each index is accessed four times, of which two accesses exists only two iterations from each other. A trivial optimization is buffering some values of the `pic1` image to reduce the number of memory accesses to this memory. Ultimately, the number of accesses to the same index can be reduced to one, but that would require to buffer two complete image lines. These image lines need to be saved in two additional memories. Chosen was to add only three additional registers and combine the two accesses in the middle. This solution involves minimal resource increase, while reducing the original number of accesses to `pic1` by 25%. The source code before and after this optimization are shown in figure 5.9.

Another optimization that was done in the code fragment shown in figure 5.9 was the reduction of multiplications and additions. By caching the basic multiplication (i.e. `PIC_Y * x`), the total number of operations decreases. The operation report of the original implementation is shown below.

```
command_(compute_ders_3x3->Loop_1)> show operation

                 Operations |                              Count
 --------------------------------------------------------------
               greater-than |                               7129
                      minus |                              28117
                       plus |                              98010
                       mult |                             125928
```

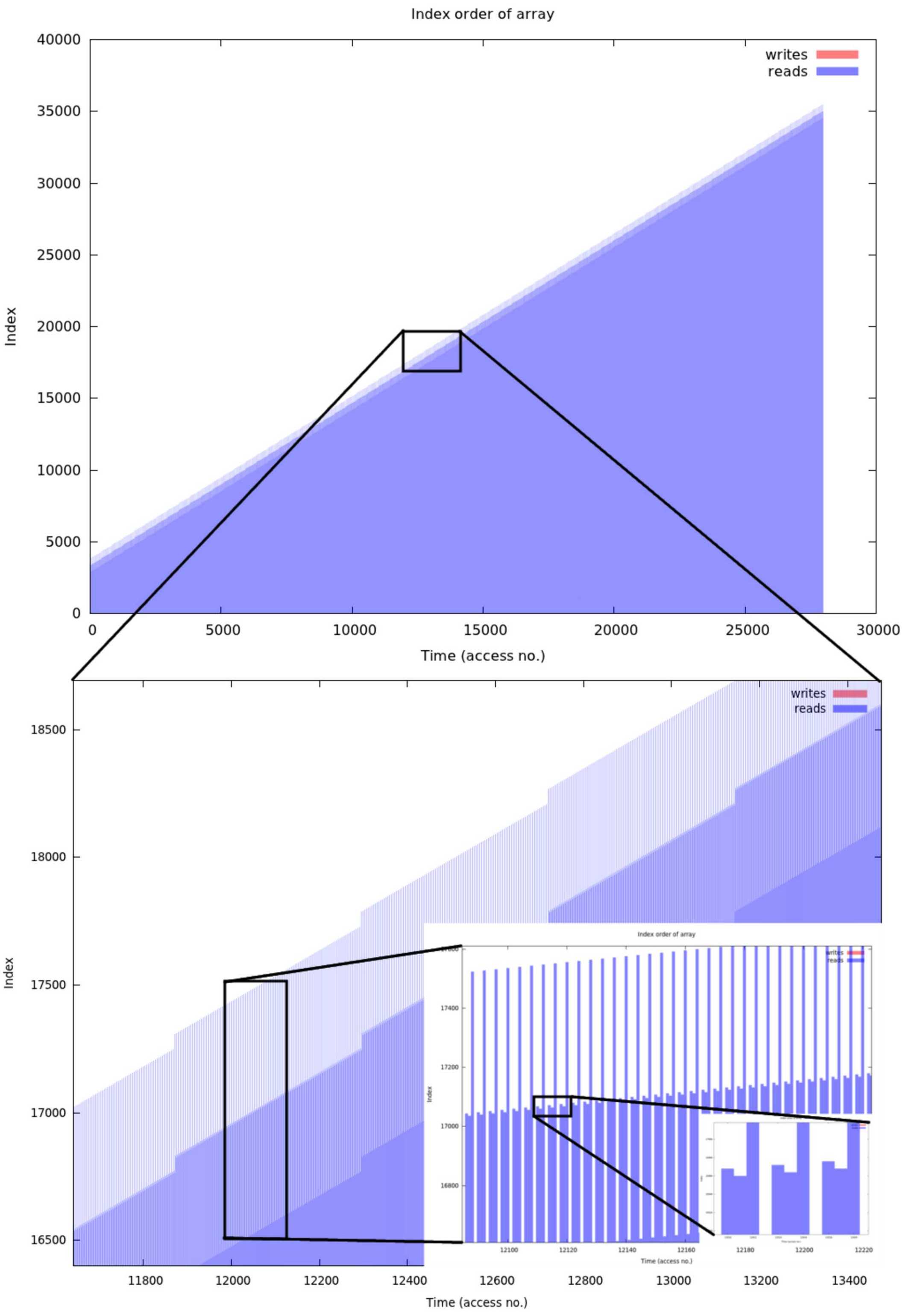

Figure 5.8: Array access pattern of `pic1` before optimizations are done

```
for (x = 7; x < PIC_X-n; x++)
{
  for (y = 7; y < PIC_Y-n; y++)
  {
    It[PIC_Y * x + y] = (pic2[PIC_Y * x + y] - pic0[PIC_Y * x + y]);
    Ix[PIC_Y * x + y] = (pic1[PIC_Y*(x+1) + y] - pic1[PIC_Y*(x-1) + y]);
    Iy[PIC_Y * x + y] = (pic1[PIC_Y * x + y+1] - pic1[PIC_Y * x + y-1]);
  }
}

pic_y_x = PIC_Y * (n-1);
for(x = n; x < pic_x_n; x++)
{
  pic_y_x += PIC_Y;
  pic1_1 = pic1[pic_y_x + n];
  pic1_2 = pic1[pic_y_x + n - 1];

  for(y = n; y < pic_y_n; y++)
  {
    pic_y_x_y = pic_y_x + y;
    It[pic_y_x_y] = (pic2[pic_y_x_y] - pic0[pic_y_x_y]);
    Ix[pic_y_x_y] = (pic1[pic_y_x_y + PIC_Y] - pic1[pic_y_x_y - PIC_Y]);
    pic1_3 = pic1_2;
    pic1_2 = pic1_1;
    pic1_1 = pic1[pic_y_x_y + 1];
    Iy[pic_y_x_y] = (pic1_1 - pic1_3);
  }
}
```

Figure 5.9: Source code of the derivative block before optimization (top) and after optimization (bottom)

5.2.2.1 New Report Results

After the optimizations are done, the new reports show the improvements. The array access report below shows a clear reduction of accesses to array `pic1`. As expected, the array access pattern of array `pic1` shown in figure 5.10, shows three accesses per iteration, instead of the four it used to be.

```
command_(compute_ders_3x3)> show array

                Array access | DataType |    Reads |   Writes
 ---------------------------------------------------------------
                        pic1 |   uint_8 |    21120 |        0
                          Iy |   int_32 |        0 |     6996
                          Ix |   int_32 |        0 |     6996
                          It |   int_32 |        0 |     6996
                        pic0 |   uint_8 |     6996 |        0
                        pic2 |   uint_8 |     6996 |        0
```

Also the number of operations has decreased considerably. The reported number of multiplications has been reduced by 83%. The multiplications reported are (except for

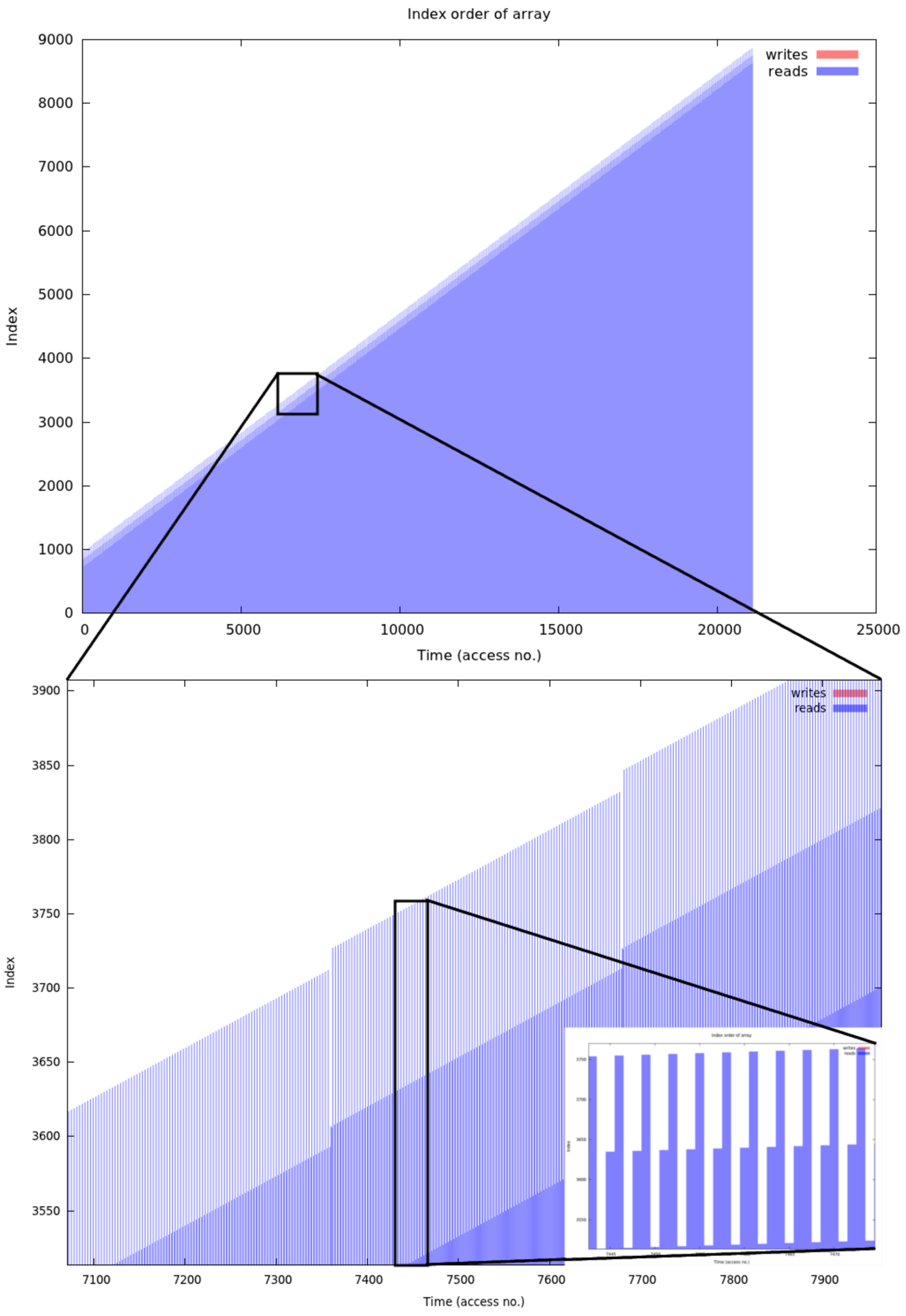

Figure 5.10: Array access pattern of `pic1` after optimizations are done

one) all the result of the calculations done by GCC to find the addresses of the array elements.

```
command_(compute_ders_3x3)> show operation

                Operations |                          Count
 --------------------------------------------------------------
                      mult |                          20989
                      plus |                          35311
                     minus |                          20990
                 less-than |                           7129
```

5.2.3 Optimizing Step 3: Compute Velocities

The largest (and most complicated) computational block is the Velocity block. This block uses the `Ix`, `Iy` and `It` memories written by the Derivatives block as inputs. The resulting outputs are the full and normal velocities. These velocities also form the output of the entire system.

Again, the first steps are to generate the array access report and the operation report. Both are shown below.

```
command_(compute_vels)> show array

              Array access | DataType |    Reads |   Writes
 --------------------------------------------------------------
              norm_vels1_x | ac_fixed |        0 |    13357
              norm_vels1_y | ac_fixed |     9600 |    13357
               full_vels_x | ac_fixed |        0 |    10618
               full_vels_y | ac_fixed |     9600 |    10618
                        MI | ac_fixed |     4784 |     4784
                        It |   int_32 |    43056 |        0
                        Iy |   int_32 |    43056 |        0
                        Ix |   int_32 |    43056 |        0
                        B2 | ac_fixed |        0 |     9568
                        M2 | ac_fixed |     4784 |    23920
                         B | ac_fixed |        0 |    19136
                         M | ac_fixed |     4784 |    38272
                         v | ac_fixed |        0 |     7514
```

```
command_(compute_vels)> show operation

                Operations |                          Count
 --------------------------------------------------------------
                 not-equal |                             32
                     minus |                           4913
              pointer-plus |                             24
        less-than-or-equal |                           9601
                      plus |                         172290
                      mult |                         669916
              greater-than |                           4889
```

The M, MI, M2, B, B2 and v arrays are small arrays mapped to registers by *Catapult C*. These arrays are all instances of the *C++* class ac_fixed. Because of the partial support for *C++* classes, not all read accesses are shown in the reports.

The total number of output values per output array is 9600 (i.e. 120 x 80 pixels). As can be seen in the array report, all output arrays are written more times than there are elements. This might be an optimization opportunity. Also two of the outputs are read 9600 times. This is not desirable. The input memories (Ix, Iy and It) are read almost 4.5 times more often than the number of elements they have. Another remarkable result from the operation reports is the enormous amount of multiplications done. This is also worth checking out.

As always, let's start with the memory accesses. The first array to generate the access pattern plots for is the Ix array. The plot is shown in figure 5.11.

Plots for the other inputs look similar. As can be seen, three sequential accesses are done in the lower index numbers. After that, three sequential accesses are done with some higher indices. Finally, three sequential accesses are done at higher indices again. This process of nine accesses repeats itself and forms the iterations of the inner loop of the Velocity computational block. The three jumps in index numbers are jumps to the next line within the image. When considering the offset of 14 used to access the input memories, a total of $(120 - (2 * 14)) * (80 - (2 * 14)) = 4784$ elements need to be read from all input memories. From figure 5.11 follows that nine accesses are done each iteration. And $9 * 4784 = 43056$, which is the reported amount of memory accesses. The conclusion is that each iteration accesses nine elements from each input memory, of which eight are read before and only one element has never been read before. When considering the working of the Velocity computational block, this makes perfect sense, since this block uses a 3x3 window of input derivative values. When walking from the top left corner of the image to the bottom right, only the bottom right value of the window is completely new. As done with the Derivative computational block, some registers can be used to buffer the last three accesses. By doing this three times (for each level of indices once), the number of accesses per iteration can be reduced from 9 to 3. But when looking at the source code, an interesting observation can be done. A fragment of the source code is shown in figure 5.12.

This code fragment clears up some question marks about the enormous amount of multiplications discussed earlier. Keep in mind that this code fragment exists in the inner loop, hence, it is executed very often. The three columns of code are the three levels of indices accessed and therefore are the three lines of the image being read. A very important observation can be made when studying the code: three accesses to all three input memories are done, and at each group of accesses, the same operations are done. These operations are Ex*Ex, Ey*Ey, Ex*Ey, Ex*Et and Ey*Et. Also the outcome of these multiplications for all three pixels per group are added.

There are a number of possibilities to optimize the code. The option which uses the least amount of additional area, is probably adding a couple of registers, to buffer the values of the input memories which are also used in the next two iterations. Or even better, save the results of the calculations performed on these input memories. This also minimizes the amount of multipliers and adders to be used. When doing this, the total

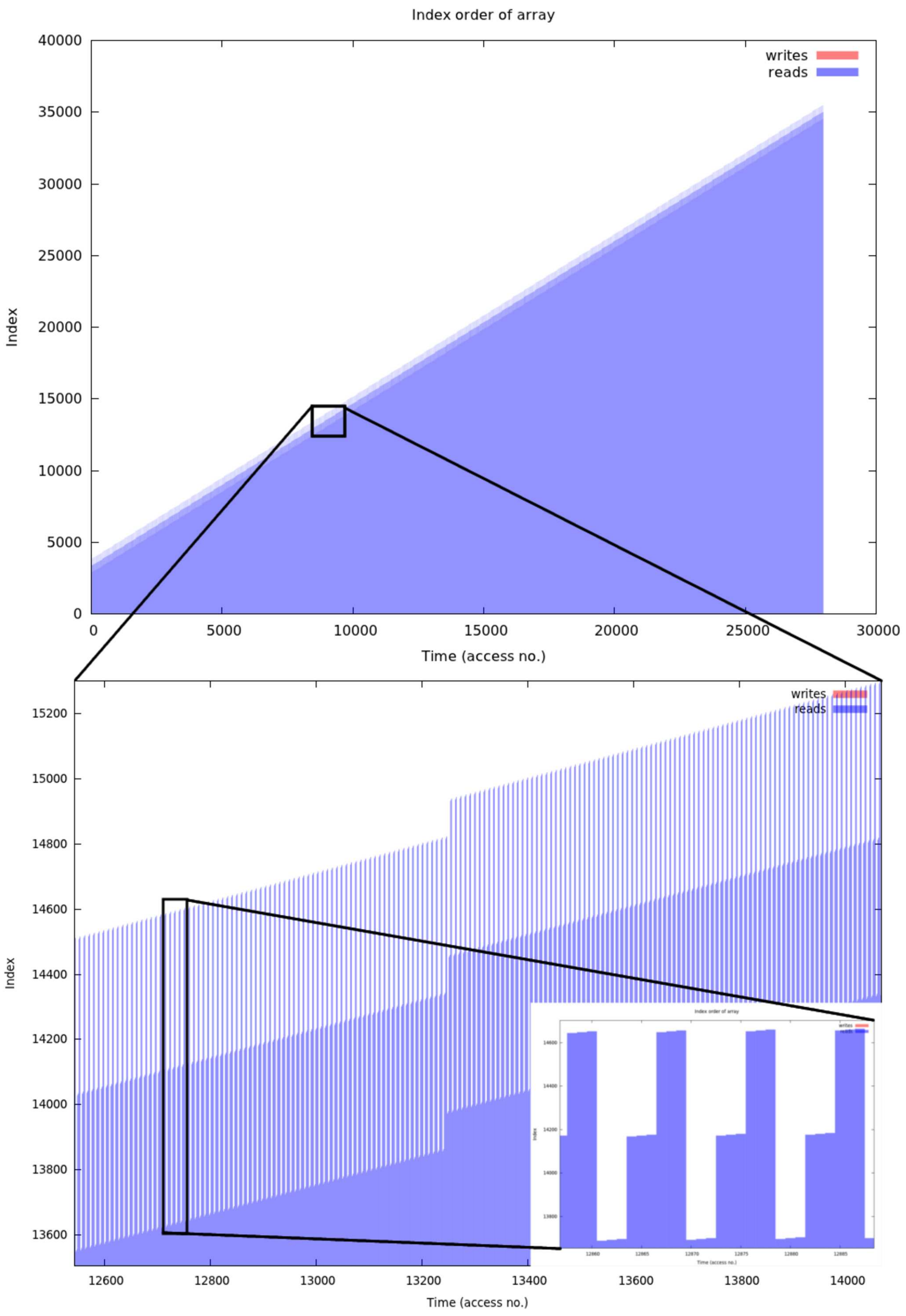

Figure 5.11: Array access pattern of `Ix` before optimizations are done

```
l = i - 1;                 l = i;                     l = i + 1;
Ex = Ix[PIC_Y*l+m1];       Ex = Ix[PIC_Y*l+m1];       Ex = Ix[PIC_Y*l+m1];
Ey = Iy[PIC_Y*l+m1];       Ey = Iy[PIC_Y*l+m1];       Ey = Iy[PIC_Y*l+m1];
Et = It[PIC_Y*l+m1];       Et = It[PIC_Y*l+m1];       Et = It[PIC_Y*l+m1];

Ex1 = Ix[PIC_Y*l+j];       Ex1 = Ix[PIC_Y*l+j];       Ex1 = Ix[PIC_Y*l+j];
Ey1 = Iy[PIC_Y*l+j];       Ey1 = Iy[PIC_Y*l+j];       Ey1 = Iy[PIC_Y*l+j];
Et1 = It[PIC_Y*l+j];       Et1 = It[PIC_Y*l+j];       Et1 = It[PIC_Y*l+j];

Ex2 = Ix[PIC_Y*l+m2];      Ex2 = Ix[PIC_Y*l+m2];      Ex2 = Ix[PIC_Y*l+m2];
Ey2 = Iy[PIC_Y*l+m2];      Ey2 = Iy[PIC_Y*l+m2];      Ey2 = Iy[PIC_Y*l+m2];
Et2 = It[PIC_Y*l+m2];      Et2 = It[PIC_Y*l+m2];      Et2 = It[PIC_Y*l+m2];

M2[0] += (Ex * Ex);        M2[0] += (Ex * Ex) *2;     M2[0] += (Ex * Ex);
M2[3] += (Ey * Ey);        M2[3] += (Ey * Ey) *2;     M2[3] += (Ey * Ey);
M2[1] += (Ex * Ey);        M2[1] += (Ex * Ey) *2;     M2[1] += (Ex * Ey);

B2[0] += (Ex * Et);        B2[0] += (Ex * Et) *2;     B2[0] += (Ex * Et);
B2[1] += (Ey * Et);        B2[1] += (Ey * Et) *2;     B2[1] += (Ey * Et);

M2[0] += (Ex1 * Ex1) *2;   M2[0] += (Ex1 * Ex1) *4;   M2[0] += (Ex1 * Ex1) *2;
M2[3] += (Ey1 * Ey1) *2;   M2[3] += (Ey1 * Ey1) *4;   M2[3] += (Ey1 * Ey1) *2;
M2[1] += (Ex1 * Ey1) *2;   M2[1] += (Ex1 * Ey1) *4;   M2[1] += (Ex1 * Ey1) *2;

B2[0] += (Ex1 * Et1) *2;   B2[0] += (Ex1 * Et1) *4;   B2[0] += (Ex1 * Et1) *2;
B2[1] += (Ey1 * Et1) *2;   B2[1] += (Ey1 * Et1) *4;   B2[1] += (Ey1 * Et1) *2;

M2[0] += (Ex2 * Ex2);      M2[0] += (Ex2 * Ex2) *2;   M2[0] += (Ex2 * Ex2);
M2[3] += (Ey2 * Ey2);      M2[3] += (Ey2 * Ey2) *2;   M2[3] += (Ey2 * Ey2);
M2[1] += (Ex2 * Ey2);      M2[1] += (Ex2 * Ey2) *2;   M2[1] += (Ex2 * Ey2);

B2[0] += (Ex2 * Et2);      B2[0] += (Ex2 * Et2) *2;   B2[0] += (Ex2 * Et2);
B2[1] += (Ey2 * Et2);      B2[1] += (Ey2 * Et2) *2;   B2[1] += (Ey2 * Et2);
```

Figure 5.12: Source code fragment of the Velocity block before optimization

area needed might be even less. Another option is to add some additional memories to save all eight values read from each input memory which are used in upcoming iterations. Or even better, save the outcome of the calculations. This will reduce the number of multipliers and adders immensely. Of course, this last option needs the most memory to buffer all data that will be reused. Because of the significant decrease of memory accesses and multiplications, this last option was considered to be the best.

Each of the values has a factor where they will be multiplied with before being added to one of the elements of M2 or B2. The middle pixel has a factor of 4. The pixels right next to it have a factor of 2. The other pixels (i.e. the four pixels in the corners) have a factor of 1. Hence, both in row and column direction, the middle value of the three has twice the weight of its two neighbors. This property is being used while optimizing the algorithm. To clarify this, the equation below expresses the formula for

`M2[0]` in a modular way.

$$\begin{aligned}
sum_1 &= 1*(Ex_{11}^2) + 2*(Ex_{12}^2) + 1*(Ex_{13}^2) \\
sum_2 &= 1*(Ex_{21}^2) + 2*(Ex_{22}^2) + 1*(Ex_{23}^2) \\
sum_3 &= 1*(Ex_{31}^2) + 2*(Ex_{32}^2) + 1*(Ex_{33}^2) \\
M2[0] &= 1*sum_1 + 2*sum_2 + 1*sum_3
\end{aligned} \tag{5.1}$$

In the iteration one image line below, the values of sum_2 and sum_3 can be reused as values of sum_1 and sum_2, respectively. Two memories are needed to save these values. The number of elements within these memories need to be the same as there are pixels in an image line. In each iteration, the value of sum_3 needs to be calculated. This is done by buffering the values of Ex_{32}^2 and Ex_{33}^2, since they will be Ex_{31}^2 and Ex_{32}^2 in the next iteration. After the calculation is done, the value of sum_3 is saved in one of the memories.

The total amount of memories needed for this system is ten. That is two for each of the five sub values.

A code fragment of the solution is given in figure 5.13.

The output memories are a point of interest as well. The plot of the access pattern is provided in figure 5.15. It can be seen that first all indices are written and read from the beginning to the end. Later on, some values are written again. The first writes and reads can be found at the top of the Velocity block source code. Here a default value to all elements is written. This is done as shown in figure 5.14. The reads come from the double assignment used. (First the value is written to `norm_vels1_y`, then the value of `norm_vels1_y` is read and written into `norm_vels1_x`.) This can be solved by simply assigning the variables one-by-one.

More interestingly is the fact that all elements are written with a default value, regardless of the calculations done by the algorithm. This keeps the algorithm easy to understand, but not very efficient. The default values are needed in case no value could be calculated for the element. Also the offset is filled with default values, since they will never get assigned otherwise.

The optimization performed is to only write the values within the offset area at the beginning of the algorithm. The middle values are assigned when the calculations of an element are finished. If no value could be found, the default value will be written. This eliminates the need of writing data multiple times to the same element. This holds for all four output memories. The new code fragment that fills only the values covered by the offset is shown in figure 5.16.

```
index0 = PIC_Y * (i+1) + j + 1;
int jmask0 = j & 0x1;
int jmask1 = (j - 1) & 0x1;

Ex3 = Ix[index0];
Ey3 = Iy[index0];
Et3 = It[index0];

Exx = Ex3 * Ex3;
sum_xx = sum_tmp_xx[jmask1] << 1;
sum_xx += Exx;
sum_xx += sum_tmp_xx[jmask0];
sum_tmp_xx[jmask0] = Exx;

Eyy = Ey3 * Ey3;
sum_yy = sum_tmp_yy[jmask1] << 1;
sum_yy += Eyy;
sum_yy += sum_tmp_yy[jmask0];
sum_tmp_yy[jmask0] = Eyy;

Exy = Ex3 * Ey3;
sum_xy = sum_tmp_xy[jmask1] << 1;
sum_xy += Exy;
sum_xy += sum_tmp_xy[jmask0];
sum_tmp_xy[jmask0] = Exy;

Ext = Ex3 * Et3;
sum_xt = sum_tmp_xt[jmask1] << 1;
sum_xt += Ext;
sum_xt += sum_tmp_xt[jmask0];
sum_tmp_xt[jmask0] = Ext;

Eyt = Ey3 * Et3;
sum_yt = sum_tmp_yt[jmask1] << 1;
sum_yt += Eyt;
sum_yt += sum_tmp_yt[jmask0];
sum_tmp_yt[jmask0] = Eyt;
```

```
if ((i & 0x1) == 0x1)
{
  M2[0] = sum_xx + sum_buf1_xx[j] +
    (sum_buf2_xx[j] << 1);
  M2[3] = sum_yy + sum_buf1_yy[j] +
    (sum_buf2_yy[j] << 1);
  M2[1] = sum_xy + sum_buf1_xy[j] +
    (sum_buf2_xy[j] << 1);
  B2[0] = sum_xt + sum_buf1_xt[j] +
    (sum_buf2_xt[j] << 1);
  B2[1] = sum_yt + sum_buf1_yt[j] +
    (sum_buf2_yt[j] << 1);
  sum_buf1_xx[j] = sum_xx;
  sum_buf1_yy[j] = sum_yy;
  sum_buf1_xy[j] = sum_xy;
  sum_buf1_xt[j] = sum_xt;
  sum_buf1_yt[j] = sum_yt;
}
else
{
  M2[0] = sum_xx + sum_buf2_xx[j] +
    (sum_buf1_xx[j] << 1);
  M2[3] = sum_yy + sum_buf2_yy[j] +
    (sum_buf1_yy[j] << 1);
  M2[1] = sum_xy + sum_buf2_xy[j] +
    (sum_buf1_xy[j] << 1);
  B2[0] = sum_xt + sum_buf2_xt[j] +
    (sum_buf1_xt[j] << 1);
  B2[1] = sum_yt + sum_buf2_yt[j] +
    (sum_buf1_yt[j] << 1);
  sum_buf2_xx[j] = sum_xx;
  sum_buf2_yy[j] = sum_yy;
  sum_buf2_xy[j] = sum_xy;
  sum_buf2_xt[j] = sum_xt;
  sum_buf2_yt[j] = sum_yt;
}
```

Figure 5.13: Source code fragment of the Velocity block after optimization

```
for(iv1=0;iv1<PIC_Y*PIC_X;iv1++)
{
  full_vels_x [iv1] = full_vels_y [iv1] = 100.0;
  norm_vels1_x[iv1] = norm_vels1_y[iv1] = 100.0;
}
```

Figure 5.14: Source code fragment of the initialization phase of the output values before optimization

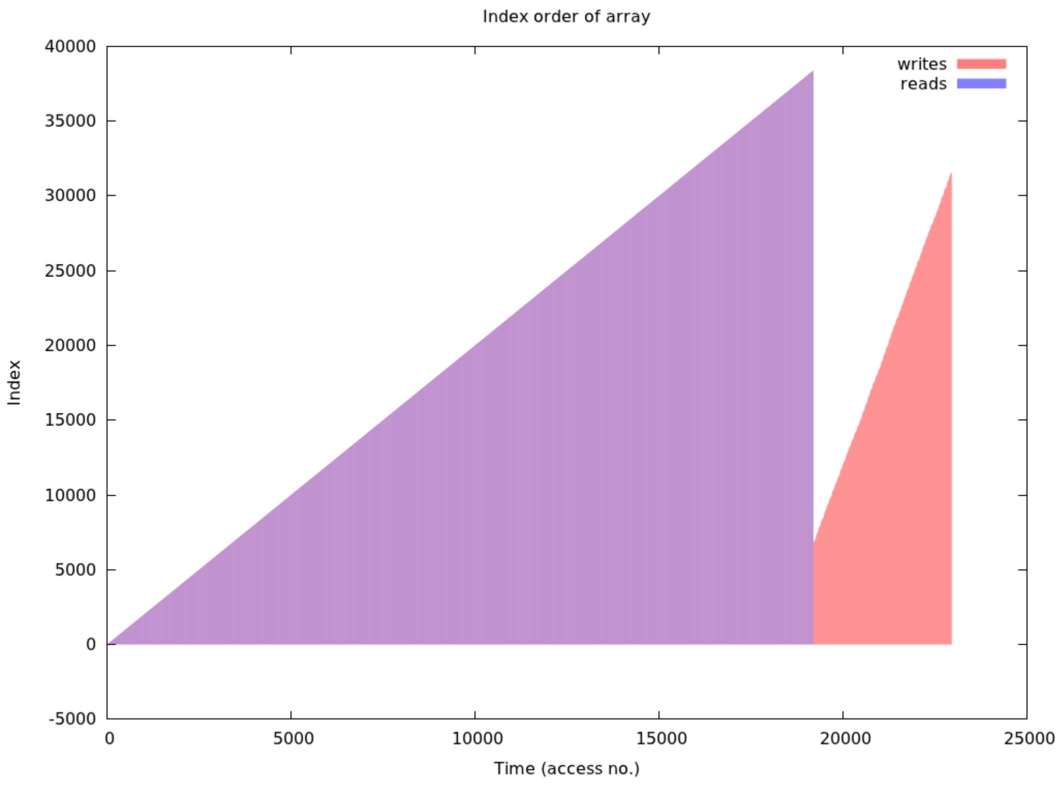

Figure 5.15: Array access pattern of `norm_vels1_y` before optimizations are done

```
index = 0;
for(i = 0; i < n; i++)
{
  for(j = 0;j < PIC_Y; j++)
  {
    int index1 = index + j;
    full_vels_x[index1]  = 100.0;
    full_vels_y[index1]  = 100.0;
    norm_vels1_x[index1] = 100.0;
    norm_vels1_y[index1] = 100.0;
  }
  index += PIC_Y;
}

index = PIC_Y * (PIC_X - n);
for(i = PIC_X - n; i < PIC_X; i++)
{
  for(j = 0;j < PIC_Y; j++)
  {
    int index1 = index + j;
    full_vels_x[index1]  = 100.0;
    full_vels_y[index1]  = 100.0;
    norm_vels1_x[index1] = 100.0;
    norm_vels1_y[index1] = 100.0;
  }
  index += PIC_Y;
}
```

```
index = PIC_Y * n;
for(i = n; i < (PIC_X - n); i++)
{
  for(j = 0; j < n; j++)
  {
    int index1 = index + j;
    full_vels_x[index1]  = 100.0;
    full_vels_y[index1]  = 100.0;
    norm_vels1_x[index1] = 100.0;
    norm_vels1_y[index1] = 100.0;
  }
  for(j = PIC_Y - n; j < PIC_Y; j++)
  {
    int index1 = index + j;
    full_vels_x[index1]  = 100.0;
    full_vels_y[index1]  = 100.0;
    norm_vels1_x[index1] = 100.0;
    norm_vels1_y[index1] = 100.0;
  }
  index += PIC_Y;
}
```

Figure 5.16: Source code fragment of the initialization phase of the output values after optimization

5.2.3.1 New Report Results

To compare the results before and after the optimizations described above, the generated reports which were shown before are regenerated on the new code shown in this section. The first report to look at is the array access report.

```
command_(compute_vels)> show array

                 Array access | DataType |    Reads |   Writes
 ------------------------------------------------------------
                 norm_vels1_y | ac_fixed |        0 |     9600
                 norm_vels1_x | ac_fixed |        0 |     9600
                  full_vels_y | ac_fixed |        0 |     9600
                  full_vels_x | ac_fixed |        0 |     9600
                   sum_tmp_yt |   int_32 |     9936 |     5076
                   sum_tmp_xt |   int_32 |     9936 |     5076
                   sum_tmp_xy |   int_32 |     9936 |     5076
                   sum_tmp_yy |   int_32 |     9936 |     5076
                   sum_tmp_xx |   int_32 |     9936 |     5076
                           It |   int_32 |     5076 |        0
                           Iy |   int_32 |     5076 |        0
                           Ix |   int_32 |     5076 |        0
                  sum_buf2_yt |   int_32 |     4784 |     2484
                  sum_buf2_xt |   int_32 |     4784 |     2484
                  sum_buf2_xy |   int_32 |     4784 |     2484
                  sum_buf2_yy |   int_32 |     4784 |     2484
                  sum_buf2_xx |   int_32 |     4784 |     2484
                  sum_buf1_yt |   int_32 |     4784 |     2484
                  sum_buf1_xt |   int_32 |     4784 |     2484
                  sum_buf1_xy |   int_32 |     4784 |     2484
                  sum_buf1_yy |   int_32 |     4784 |     2484
                  sum_buf1_xx |   int_32 |     4784 |     2484
                           B2 | ac_fixed |        0 |     9568
                           M2 | ac_fixed |     4784 |    19136
                           MI | ac_fixed |     4784 |     4784
                            B | ac_fixed |        0 |     9568
                            M | ac_fixed |     4784 |    19136
                            v | ac_fixed |        0 |     7440
```

As can be seen, a lot of buffer arrays are added to the design. Also the output arrays are all written 9600 times and never read. No redundant work is done there anymore. Also the amount of read accesses to the inputs has decreased considerable. In the figures 5.17 and 5.18, the access patterns of both input and output are presented. The input `Ix` is as expected a nice increasing line. No more redundant accesses are done to the input memories. The output memory accesses result in a complex looking plot. In the first half of the plot, the default values for offset locations are written. The output field is surrounded by an offset field. First the pixels above the real output field are written, then the pixels below the real output field and finally the pixels left and right from the real output field. The last half of the plot are the writes to the real output field. If no value could be calculated, the default value is written here as well. This is why this part of the plot increases more evenly than the original plot of figure

5.15. The magnification shows how the indices are written while the system outputs the default values. No index is written more than once.

The operation report is shown below. Especially the number of multiplications has decreased enormously. The original implementation used as many as 669916 multiplication operations, while the optimized version only uses 88816 multiplications. This is a reduction of 87%. Most multiplications shown are now used to calculate the addresses of the array elements.

```
command_(compute_vels)> show operation

                   Operations |                              Count
 --------------------------------------------------------------
                    less-than |                                798
           less-than-or-equal |                               4183
                         mult |                              88816
                         plus |                             137154
                        minus |                             120051
                 greater-than |                               5128
                      bit-and |                              15012
                       lshift |                              48760
                    not-equal |                             153088
                 pointer-plus |                             114816
```

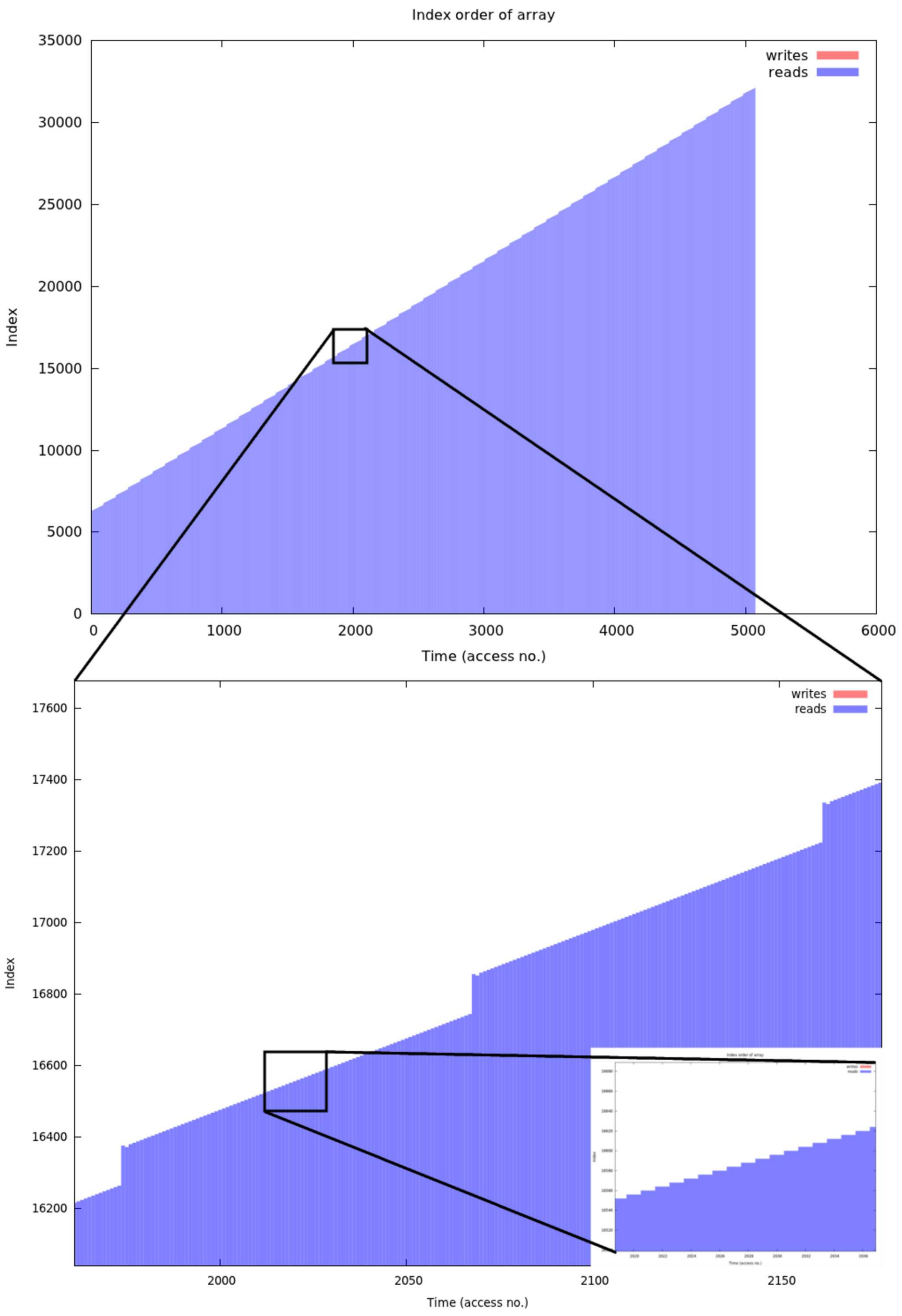

Figure 5.17: Array access pattern of Ix after optimizations are done

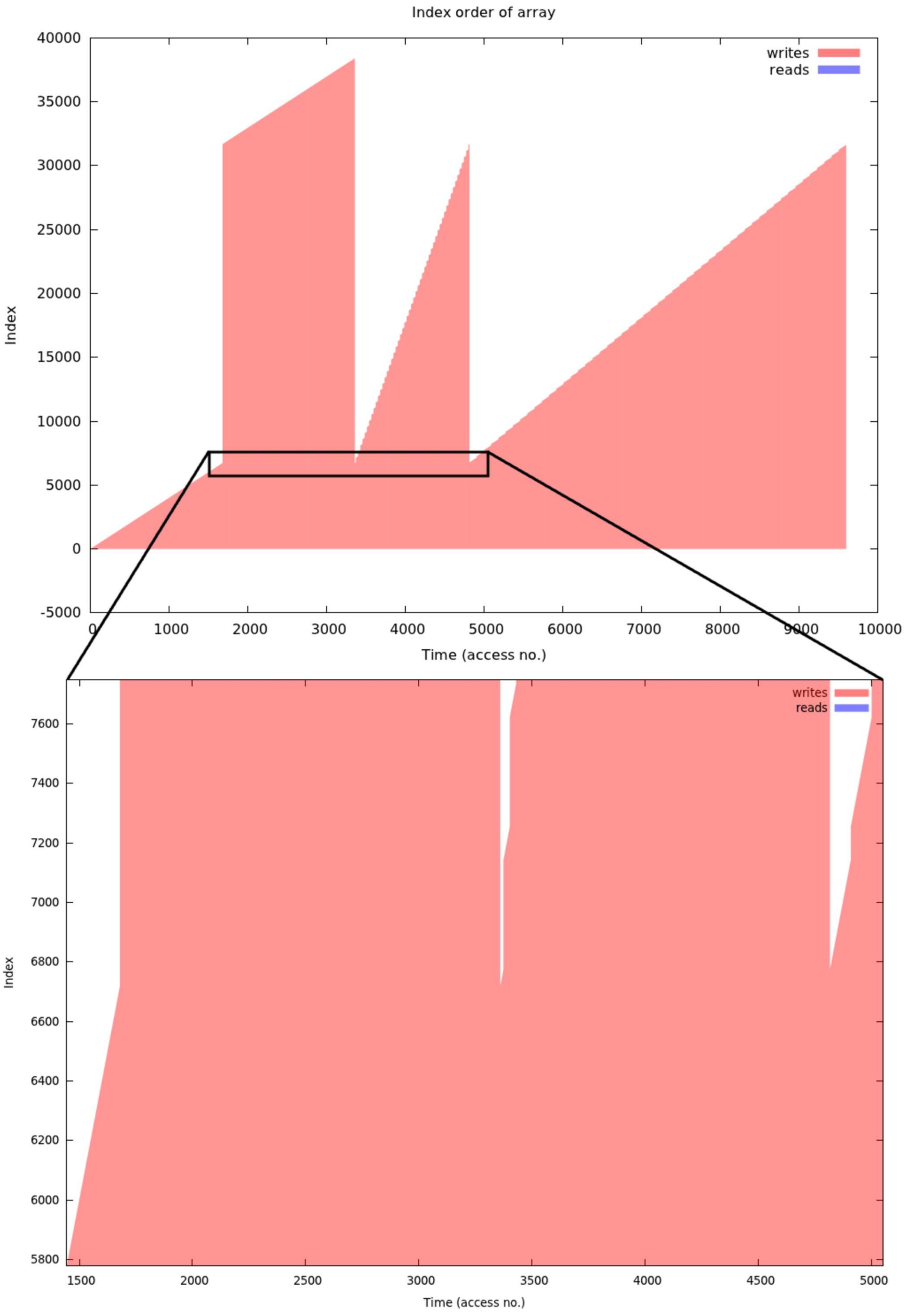

Figure 5.18: Array access pattern of `norm_vels1_y` after optimizations are done

5.3 Pipelining the Design

When a system is pipelined, different parts of the system run simultaneously. In case of the algorithm discussed in this chapter, there are two ways of pipelining the algorithm. The first way is letting *Catapult C* pipeline loops of the algorithm. The second way is pipelining the different stages of the algorithm. Both are discussed in this section.

5.3.1 Pipelining Loops

Loops can be pipelined very easily by providing an Initiation Interval (or II) to *Catapult C* as described in Section 2.3. The original version of the source code written by Ren [40] also used pipelining of loops to increase the speed of computational parts of the algorithm. In many cases, memory accesses form a bottleneck for further increasing the number of instances of the loop running simultaneously. Because the number of memory accesses has been decreased by optimizing the source code, more room for pipelining can be expected. The same three blocks of the algorithm as were optimized in the previous section, are pipelined in this section. Results and settings of the implementation by Ren and the optimized implementation are compared to each other. To retrieve all data from the implementation from Ren, his source code was used with the settings provided in the report [40]. In the Velocity computational block, some values are different from the report by Ren. The cause of this is unknown. All reported area values are taken from reports of *Catapult C* and are included for comparison purposes.

5.3.1.1 Pipelining Loops Step 1: StackBlur

Ren had pipelined both inner loops of the StackBlur computational block with an II value of 3. In the original implementation, the first inner loop contained one read and one write to the input memories. Because all memories are single port memories, only one memory access can be executed per clock cycle. The maximum number of II for the first inner loop is 2 in the original implementation. The second inner loop does only access all memories once per iteration. This means that no memory access will cause problems when pipelining this loop to the maximum, which is an II of 1.

In the new design, both loops do not access any memory more than once. Therefore, both inner loops can be pipelined with an II of 1. In table 5.1 the settings for the StackBlur computational block and the throughput time and area usage of the whole system are given.

5.3.1.2 Pipelining Loops Step 2: Compute Derivatives

Ren had pipelined the inner loops of the Derivative computational block with an II value of 4. His implementation contains four accesses to memory `pic1`. Therefore, the lowest II value possible was 4. All other memory accesses were read only once. The memory access to `pic1` forms the bottleneck of this entire block.

	Implementation By Ren [40]	**Optimized Implementation**
Loops		
ROW	Not pipelined or unrolled	Not pipelined or unrolled
ROW:for	Fully unrolled	Fully unrolled
ROW:for#1	Fully unrolled	Fully unrolled
ROW:for#2	Pipelined II=3	Pipelined II=1
COLUMN	Not pipelined or unrolled	Not pipelined or unrolled
COLUMN:for	Fully unrolled	Fully unrolled
COLUMN:for#1	Fully unrolled	Fully unrolled
COLUMN:for#2	Pipelined II=3	Pipelined II=1
Results		
ROW: Throughput time [us]	4820	1640
COLUMN: Throughput time [us]	4810	1630
Overall used area	46432.79	46646.24
Overall throughput time [ms]	112.07	105.71
Overall FPS	8.92	9.46
Overall efficiency (area / FPS)	5203.72	4930.97

Table 5.1: *Catapult C* optimization settings and results (StackBlur)

In the optimized solution, the number of accesses to the `pic1` memory has decreased to three. Hence, the minimal number for II will also be 3, which has also been chosen as the new value. As can be seen, the `pic1` memory still forms the bottleneck for this part of the system. In table 5.2 the settings for the Derivative computational block and the throughput time and area usage of the whole system are given.

	Implementation By Ren [40]	**Optimized Implementation**
Loops		
compute_ders_3x3:for	Not pipelined or unrolled	Not pipelined or unrolled
compute_ders_3x3:for:for	Pipelined II=4	Pipelined II=3
Results		
Throughput time [us]	5750	4340
Overall used area	46432.79	46238.51
Overall throughput time [ms]	112.07	110.66
Overall FPS	8.92	9.04
Overall efficiency (area / FPS)	5203.72	5116.8

Table 5.2: *Catapult C* optimization settings and results (Compute Derivatives)

5.3.1.3 Pipelining Loops Step 3: Compute Velocities

In the original implementation by Ren, the Velocity computational block contained two main loops. The first loop writes the default value to all output elements, while the second loop is a big loop containing all velocity calculations. This loop calls a number of different functions, for example a division function, square root function and the `eigensolve()` function which solves eigenvalues and eigenvectors.

Only the first (small) loop was pipelined by Ren with an II value of 4. In the optimized design, this one loop has changed to three loops. The first and second loop set the default value for the top and bottom elements. Both loops contain one nested loop. The third loop sets the default values for the left and right elements. Two nested loops are used for this (one for the left side and one for the right). All nested loops never accesses a memory more than once per iteration. The smallest II value possible is therefore 1. This value has been used for all four inner loops. The outer loops were not pipelined.

All called functions from the second loop were fully unrolled by Ren, but no pipeline was used. As will be discussed later, this loop is the most time consuming part of the whole algorithm. Pipelining this loop will probably increase the overall throughput and used area dramatically. Chosen was to determine the II value of this loop by calculating the results for some values of II and choose the most appropriate value tested. In the optimized implementation, two buffers have to be filled before the second loop can be executed. The filling of the buffer is done in a new nested loop. The inner loop was pipelined with an II value of 1, since no memory is read more than once per iteration.

To improve readability, all loops in this block are named in the optimized implementation. The *DEF_VAL* loops correspond to the first loop of the implementation by Ren, which sets the default values. The *CALC* loops corresponds to the second loop of the implementation by Ren, which calculates the actual velocities.

In table 5.3, the used settings and the results of both implementation by Ren and the optimized implementation are provided. The second loop is not pipelined for now.

5.3.2 Pipelining the Overall System

A trivial optimization is pipelining the three stages of the algorithm. This is one of the future work notes of Ren [40]. At this time, only one of the three main computational blocks is active at a time. This means that if one block is active, the other two are idle, which is not very efficient. Although pipelining the three main blocks may be a great opportunity to improve the throughput, it does not come for free. When no pipeline is used, *Catapult C* may schedule resources in such a way that it reuses them in multiple blocks. When the design is fully pipelined, the resource sharing possibilities are less. Hence, the number of resources necessary will probably increase. Also buffer memories are needed. If for example the Derivative computational block calculates the derivative values of the next iteration, the Velocity computational block still needs the derivative values of the iteration before. Thus, a buffer is needed. The Derivative memories are now also used as buffer in the StackBlur block. This can no longer be done when

	Implementation By Ren [40]	**Optimized Implementation**
Loops		
compute_vels:for	Pipelined II=4	N/A
compute_vels:DEF_VAL_1	N/A	Not pipelined or unrolled
compute_vels:DEF_VAL_1:for	N/A	Pipelined II=1
compute_vels:DEF_VAL_2	N/A	Not pipelined or unrolled
compute_vels:DEF_VAL_2:for	N/A	Pipelined II=1
compute_vels:DEF_VAL_3	N/A	Not pipelined or unrolled
compute_vels:DEF_VAL_3:for	N/A	Pipelined II=1
compute_vels:DEF_VAL_3:for#1	N/A	Pipelined II=1
compute_vels:for#1	Not pipelined or unrolled	N/A
compute_vels:for#1:for	Not pipelined or unrolled	N/A
CALC	N/A	Not pipelined or unrolled
CALC:for	N/A	Not pipelined or unrolled
Results		
DEF_VAL: Throughput time [us]	6370	1638
CALC: Throughput time [us]	114840	69690
Overall used area	46432.79	50042.87
Overall throughput time [ms]	112.07	86.70
Overall FPS	8.92	11.53
Overall efficiency (area / FPS)	5203.72	4338.72

Table 5.3: *Catapult C* optimization settings and results (Compute Velocities). *DEF_VAL* are the loops which are setting the default values and *CALC* are the loops calculating the velocities.

the design is fully pipelined. Also some loops are fully pipelined, as described in the previous sections. Memories may be accessed every clock cycle during execution of a block. If another block also needs to access these same memories, there is no time for these accesses as long as the first block is not finished. Again, memory will probably be the bottleneck.

Another important note would be that the execution time of a pipeline stage will be as fast as the slowest block in the pipeline. It is therefore desirable that all blocks consume about the same amount of time, to prevent the faster blocks being idle for too long. In the implementation by Ren, a huge difference in execution time exists between the blocks. This is shown in table 5.4. Given these numbers, pipelining the whole system would not improve the overall speed too much. Although the optimized implementation has much better comparable execution times, no pipeline is added yet. The execution times of the optimized implementation presented in table 5.4 are of the final implementation of Section 5.4.4.

Main Block	**Execution Time [ms] By Ren [40]**	**Execution Time [ms] Opt. Impl.**
StackBlur	9.63	1.64
Derivative	5.75	2.17
Velocity	121.21	6.19

Table 5.4: Execution time of the three main blocks in the implementation by Ren [40] and the optimized implementation

5.4 Results

This section shows the overall results of the optimized implementation. First a word on calculating the number of Frames Per Second (FPS) is given.

5.4.1 Calculating Frames Per Second

The number of frames per second can be calculated by dividing the number of frames calculated per iteration by the total throughput time of that iteration. The throughput time can be read from the *Catapult C* results (i.e. the cycle report). The number of frames per iteration is one.

In the report of Ren [40], the number of frames per iteration was set to three. This is incorrect. To complete the calculation of one frame, three input images (or input frames) are needed. All three images must be blurred, before they can be used. To calculate the next frame, again three images are needed. All three images must be blurred. Two of the three images were read and blurred in the iteration before. Only one image is new and must be blurred. Hurkmans [41] used a simple pipeline to do this. Each system iteration, only one image is read and blurred. The other two blurred images are used from the previous iteration. Ren [40] did not use such a pipeline. He decided to read and blur all three images at once. Therefore, each system iteration, three images are read and blurred, but still only one output frame is calculated. The next iteration, two of the already blurred images must be read and blurred again. The correct number of frames calculated per iteration is therefore one. The optimized implementation did not implement the reuse of the already blurred images. This is left as future work.

Ren reported to be able to process 22.47 FPS. This number is incorrect and should be 7.49 FPS according to the data provided in the report of Ren [40]. When running *Catapult C* with the source code and settings provided by Ren, the number of FPS is a little higher: 8.92 FPS.

5.4.2 Run-time on a PC

Although this work is not about improving the sequential execution time, it did. This is because many calculations (especially multiplications) were optimized away. Changes in memory organization are probably not of any effect in sequential execution. The reported times below are found by measuring the execution time on a machine with an Intel Centrino Duo T7200 processor and a sufficient amount of memory (2 GB). No additional compiler optimizations were selected. Table 5.5 shows the results.

Main Block	**Execution time [ms] Implementation By Ren [40]**	**Execution time [ms] Optimized Implementation**
StackBlur	13.071	11.032
Derivative	2.183	1.291
Velocity	17920.256	17656.626
Overall System	**17935.510**	**17668.949**

Table 5.5: Run-time on a PC with an Intel Centrino Duo T7200 processor and 2 GB of RAM

5.4.3 Determine Clock Frequency and Velocity Pipeline

A number of clock frequencies have been tested. Figure 5.19 shows a graph representing the outcome of these tests. The test values are also presented in table 5.6. All tests are run with an Assignment Overhead of 20% (one of the contraints within *Catapult C*). This forces *Catapult C* to safe some space per clock cycle, which is necessary to provide headroom for the extraction process of *Catapult C*, so negative slack is omitted. The Velocity calculation block is not yet pipelined.

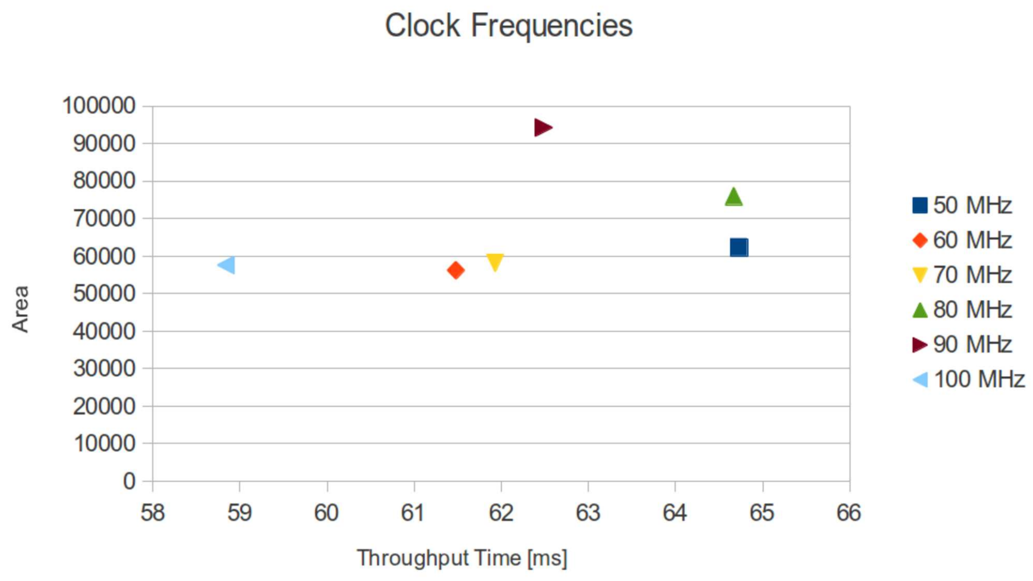

Figure 5.19: Results for different frequencies

As can be seen, 50, 70, 80 and 90 MHz are no winners. They use all more area for less performance, compared to 100 MHz. The least area is used by the 60 MHz solution, while the best performance can be expected with the 100 MHz solution. Because the

Frequency	Throughput Time [ms]	FPS	Used Area	Slack
50 MHz	64.73	15.45	62326.02	+2.18
60 MHz	61.48	16.27	56201.67	+1.86
70 MHz	61.93	16.15	58159.65	+0.91
80 MHz	64.67	15.46	75975.88	+0.63
90 MHz	62.49	16.00	94238.79	+1.37
100 MHz	58.84	17.00	57581.75	+0.70

Table 5.6: Results for different frequencies

increase in area between these two solutions is not significant compared to the gained speed, the 100 MHz solution was selected to be used.

The same tests were done on the algorithm with different II values for the velocity calculation loop. The used clock frequency is set to the same initial frequency as it was for the tests above, i.e. 50 MHz. Figure 5.20 shows the test outcome in a graphical way, while table 5.7 show the list of results.

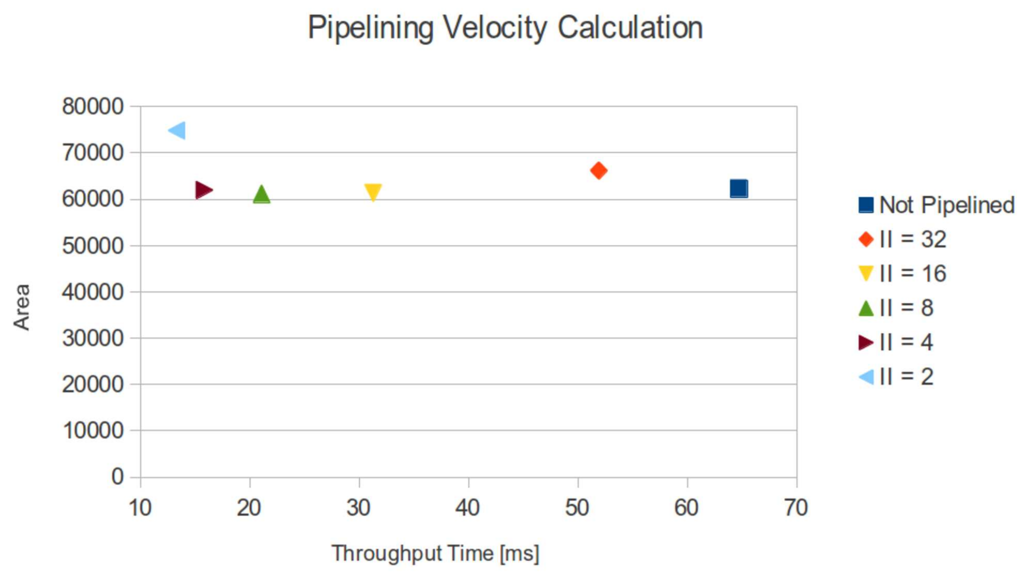

Figure 5.20: Results for different II values of the velocity calculation loop

As can be seen, the performance increases significantly when the most computational intensive part of the algorithm is pipelined. The II values of 32 and 16 are not the best to choose, because they use more area for less FPS. The II value of 4 results in negative slack and is therefore rejected. An II value of 2 gives a very fast solution, but needs a lot more area to realize that. An II value of 8 seems to be the best compromise and is therefore chosen.

Now the chosen II value and frequency are combined. Also an II value of 6 was tested as well with the chosen frequency. This value may result in a faster solution,

II value	Execution Time CALC loop [ms]	Throughput Time (Overall) [ms]	FPS	Used Area	Slack
Not Pipelined	54.21	64.73	15.45	62326.02	+2.18
32	41.40	51.93	19.26	66177.08	+2.87
16	20.79	31.31	31.94	61351.18	+1.97
8	10.58	21.11	47.37	61187.26	+2.71
4	5.35	15.87	63.01	61977.11	**-0.34**
2	2.81	13.33	75.02	74825.90	+1.41

Table 5.7: Results for different II values of the velocity calculation loop

which does not need too much additional area and still has a positive slack. *Catapult C* can be set to put *high effort* in optimizing the area. This is also tested with a frequency of 100 MHz and an II value of 8. All results are shown in table 5.8.

II value	Throughput Time [ms]	FPS	Used Area	Slack	Time needed by *Catapult C* [h:m:s]
8 (normal effort)	10.00	100.0	64477.6	+0.70	3:50:18
6 (normal effort)	8.71	114.8	69708.6	+0.77	4:42:16
8 (high effort)	10.00	100.0	64475.2	+0.70	5:39:15

Table 5.8: Final performance tests at 100 MHz

An II value of 6 produces a very fast solution: 114.8 FPS. But, it also needs a considerable amount of additional area. Therefore, the solution running at 100 MHz and using an II value of 8 for the velocity calculation loop is chosen. The *high effort* setting of *Catapult C* needed almost 2 more hours to come up with the results, but was not able to decrease the amount of area noticeable. These results were created on a machine with an Intel i7 920 processor and 6 GB of memory.

5.4.4 Final Results

The existing HLS implementation of the Lucas & Kanade [38] Optical Flow algorithm implemented by Ren [40] has been optimized using the presented framework. The presented new implementation of the Lucas & Kanade algorithm is capable of processing 100 frames per second at a relative small increase of used resources (area). This solution is generated with *Catapult C* and has a positive slack.

The optimized and heavily pipelined solution of this chapter can be compared with the previous implementations using *Catapult C* by Ren [40] and the hand written *VHDL* solution by Hurkmans [41]. All solutions are designed for use with a Xilinx Virtex 5 FPGA and an input image size of 316 * 252 pixels. The results are shown in table 5.9.

If the new solution is compared with the solution by Ren with a positive slack (i.e. the 50 MHz implementation), concluded can be that the performance has increased by a factor 13.7, while the area has increased by a factor of 1.47. The increase in performance is found by comparing the number of frames per second both implementations can process. These values are indicated with a grey cell color.

	Hand Written By Hurkmans [41]	***Catapult C*** **50 MHz By Ren** [40]	***Catapult C*** **100 MHz By Ren** [40]	***Catapult C*** **New Solution**
Clock Frequency	30.884 MHz	50 MHz	100 MHz	100 MHz
FPS	5.33	7.32	7.49	100.0
Reports by *Catapult C*				
Area	N/A	43759.89	42855.66	64477.6
Slack	N/A	+0.82	-0.24	+0.70

Table 5.9: Comparison between the presented implementation and the different previous implementations

Conclusion and Future Work 6

Now that the framework is finished and its working has been demonstrated with a case study, conclusions about both the framework and the used case study can be made.

This chapter first gives a short summary of the presented framework, then the conclusion concerning the framework is presented and some future work notes are given. This chapter ends with some future notes on the case study.

6.1 Summary

The presented framework is capable of analyzing memory accesses and operation usage on a per function and per loop basis. Providing analysis on a per loop basis is of great help to an High Level Synthesis designer, since loops are very important structures in HLS development.

Investigated was which optimizations can be set in the HLS tool (in this case *Catapult C*) and how optimizations in the software source code are of great importance to the performance of the hardware implementation. The conclusion from this research is that HLS tools use loops from the original software description to set loop unrolling and pipelining in the resulting hardware. This emphasizes the need of being able to analyze an algorithm on a per loop basis.

Different options for implementing such an analysis tool were investigated. The chosen solution was to use a relatively new feature of the GNU Compiler Collection, namely the GCC PlugIn feature.

The presented framework exists of two main parts. The first part is the GCC PlugIn and the second part is the Parser / Analyzer. The PlugIn is able to analyze the Abstract Syntax Tree of each function and inserts function calls into this AST to log functions in the developed analyze library. By executing the resulting executable file, a log file is created containing many log lines. Each log line describes a memory access, operator use, function call, etc. The Parser / Analyzer parses the generated log file. A Data Flow Tree is generated in memory by parsing the log file. After the DFT is generated, post processing does some modifications to this DFT and detects the loops. Now all information is available, reports can interactively be generated by the user. Included are reports that provide information about memory accesses and operator usage. Other reports present the loop structure of the algorithm and shows dependencies within these loops. Detailed information about each and every access to a variable can be reported. Array accesses are presented with the used indices, even when the array was passed as an argument to a function. Finally, plots can be generated presenting information about the access order of the elements within an array. These plots can be used to find optimization opportunities and to provide an instant understanding about the access pattern of an array, without having to fully review the source code first.

Since the complete framework consist of three different stages (i.e. generate the log file, convert the log file into the DFT and generate reports from the DFT), adding new reports to the framework is quite easy. In many cases, the information that must be provided in the new report is already available in the DFT. In these cases, only a new report has to be added in the third stage. If information is needed which is not yet logged into the log file, then the GCC PlugIn has to be extended. Since the PlugIn is designed hierarchically, existing *GIMPLE* node processing functions can easily be located and extended or new functions can be added to the PlugIn.

6.2 Conclusion

The main goal of the framework is to provide algorithm specific information about memory accesses and operator usage. Both memory accesses and used operators have to be reported on a per loop basis and a per function basis. This is necessary because HLS tools provide a loop unroll and loop pipelining feature. In order to set the settings in HLS tools effectively, information about the loops to unroll or pipeline is essential.

The presented framework is capable of analyzing both memory accesses and operator usage. Furthermore, the framework provides information about function calls, loop structures and the value bounds of variables.

Designers can use their normal design flow. By enabling the presented PlugIn by default, only the functions to analyze have to be provided as argument to the GCC compiler. This involves minimal changes to the command which starts the compilation process. Examples of this are provided in the source code package of the presented framework. In order to be able to analyze an algorithm, minimal changes are needed to the original source code. In many cases, just a single `#include` statement has to be added.

Arrays are often mapped to memories and are therefore of great importance to an HLS engineer. Arrays are detected and accesses to them are logged, regardless of how the array was declared. Either a local array or an array that was passed as parameter to a function will be analyzed.

The design of the framework is modular. New report types can be added efficiently to the Parser / Analyzer. In many cases, the information needed for a report is already provided by the log file and therefore available in the Data Flow Tree. In case the data needed for the report is not already available in the DTF, the GCC PlugIn has to be extended. The PlugIn itself has a hierarchical design to improve code readability and to make maintenance efficient.

Once the DFT has been build, reports can interactively be generated by the user. Commands can also be provided as argument to the Parser / Analyzer, which allows a designer to generate reports in an automated way.

The presented case study used the presented framework to optimize an existing implementation of the Lucas & Kanande Optical Flow algorithm. Many reports were generated to provide a good insight to the memory accesses and operator usage of each part of the system. The ability to generate plots concerning array access patterns proved to be one of the most powerful features of the framework. These plots can be generated very quickly and provide all access information about an array in great detail at both loop and function level.

By using all information provided by the framework, the designer was able to optimize the entire design. Many pipeline opportunities can now be used more efficiently, increasing the overall system performance tremendously. An increase in speed by a factor of 13.7 was achieved, while the used area increased only by a factor of 1.47. This improves the overall efficiency enormously.

6.3 Future Work

This section describes some suggested future work notes, which will further improve the compatibility and capability of the framework.

- Fully support 64 bit systems. Systems which use the 64 bit architecture also use 64 bit addresses instead of 32 bit addresses. This is not fully supported by the framework at this moment.
- Improve struct handling. When memory accesses are added to the list of accesses, they are merged to already saved accesses with the same name. In the search for other accesses with the same name, the stuct element value is not taken into account yet.
- Extend the operation-access report to show the values of the used memories. This would require a link between the different memory access logs and the operation access logs.
- Provide support for multi-dimensional arrays when generating an array access pattern plot. At this moment, multi-dimensional arrays are fully supported, except when generating a plot.
- Provide the option to combine all accesses of a multiple called function into one report (i.e. If a function is called multiple times, only the first call is fully logged and will be included into the reports. In some cases it can be of interest what other accesses were involved in other instances of the called function.).
- Provide logging of the values of array accesses (if data type is `int` or `boolean`). Normally, a node representing a value can be passed as argument to the logging function. This is unfortunately not possible with an array node. Therefore, the values of the array node are not logged at this moment.

- Extend the support for *C++* language. By fully supporting *C++*, algorithms which are making use of *Catapult C* classes like `ac_int<>` and `ac_fixed<>` can be analyzed more precisely. This would also enable *SystemC* source code to be analyzed by the framework, which opens a whole lot of possibilities.
- Replace some of the used lists in the DFT by hash tables. This would improve the parsing speed. Many lists (e.g. all access lists) are regularly searched for the names of the accesses it's holding. If these lists are converted to hash tables with a hash function on the name of the access, these searches would be faster.
- Extend the use of the information logged about function arguments. If a variable (e.g. an array) is passed by reference to a function, a link to the memory accesses within this function can be made. This would provide a more global representation of the memory accesses to that variable. The framework already supports narrowing down the search space for memory accesses and operator usage by using loop information, but this suggested feature would also enable the framework to widen the search space of memory accesses to a global level.
- Implement support for floating point values.

6.4 Future Work on Case Study: The Lucas Algorithm

In a case study, the Lucas Optical Flow algorithm was used. This is covered in Chapter 5. An existing HLS implementation of the Lucas algorithm was used. Using the presented framework, optimization opportunities were found and implemented. After the optimizations were performed, *Catapult C* was used to generate the *VHDL* code. The resulting output uses 1.47 times more hardware than the original implementation, but the gained speedup has a factor of 13.7.

Besides this great result, a couple of future work notes on the Lucas HLS implementation can be made:

- Since the execution time of all blocks now lay very close to each other compared to the original HLS implementation, pipelining the three basic computational blocks of the algorithm will probably improve the performance and efficiency of the used resources.
- Reuse the blurred images which are already blurred in the previous two iterations. This minimizes the resource usage of the StackBlur computational block considerable.

Optical Flow Background A

Optical flow applications are applications which calculate the direction and speed of movement in a sequence of images. This sequence is captured by a camera in most cases, but other techniques are possible as well. This chapter gives the constraints to any optical flow algorithm and will discuss the working of the optical flow algorithms used (or referred to) the most in literature.

A.1 Limitations and Constraints of Optical Flow

In order to retrieve an image sequence from real world velocities, the real world 3D velocities are projected to a 2D motion field. This implies immediately the first limitation. A fast moving object far away and a slow moving object nearby might cause the same motion velocities to be calculated by the optical flow algorithm. Another limitation caused by the lack of the third dimension is the detection of an object moving towards the camera. From the 2D motion field, the object seems to grow, while in reality the object has a certain velocity in the direction to the camera. These limitations can be overcome by introducing a 3D capture device (e.g. multiple cameras to detect depth or a special camera with build in depth measurements).

Another limitation is the well known aperture problem. The image is taken from a window or aperture of the real world. Objects moving outside the window will not be visible on the image. But more interesting are objects partly outside the window. Figure A.1 illustrates this. The dark blue bar represents an object on the first image of a sequence. The light blue bar illustrates the same object on the second image of the same sequence. As can be seen, the bar has been moved. On the left a small aperture is given in which not all edges of the bar can be detected. On the right image, the same bar is shown, but now the aperture is larger and all edges of the bar can be seen. In the left image, the motion is ambiguous. It looks like the motion goes to the right. Because all edges are known at the right image, the exact motion of the bar can be determined, which is to the bottom-right. The fact that the motion (and with that the velocity) of the bar cannot be determined in the left image, is known as the aperture problem.

Motion between different images are often detected by intensity changes of the pixels within the image. To detect the correct motion, it is important that the intensity of all objects in the image and of the image background is constant. Otherwise, a pixel in image n cannot be matched to the corresponding pixel in image $n + 1$. An obvious problem concerning this is noise. Noise is causing the intensity to change randomly.

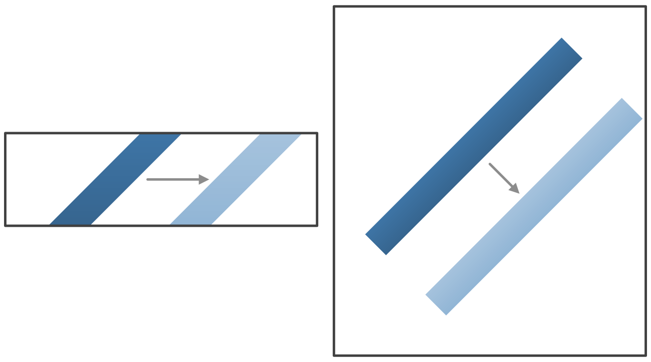

Figure A.1: Aperture problem

This causes errors in the calculated motions. The constant intensity constraint is often referred to as the *gradient constraint*.

A.2 Optical Flow Techniques

Different optical flow techniques are reported over the years. In this section the main classes of these techniques are discussed and the different algorithms proposed using the discussed techniques are given. One of the most referred works on the optical flow research topic is the paper of Barron et al. [43] and its renewed and extended version [1]. This paper gives a very nice overview of the commonly used techniques and algorithms. Besides that, all discussed algorithms (nine in total) are tested on accuracy. It is highly recommended to read the work of Barron et al. [1] for readers without any specific experience on optical flow. This section used a lot of information from this paper.

In general, four different techniques have been used with optical flow algorithms:

- Differential technique
- Region-based technique
- Energy-based technique
- Phase-based techniques

This chapter only discusses the first two techniques, because the last two are considered too computational intensive for fast practical use [43].

A.2.1 Differential Techniques

Optical flow algorithms using the differential technique compute velocity from spatiotemporal derivatives of (filtered versions of) the image. Often the filter used is some kind of blurring or smoothing filter to minimize the influence of noise and other intensity changes. Horn & Schunck [8] were the first to use this technique. They used first-order derivatives based on image translations:

$$I(x', y', t) = I(x - ut, y - vt, 0) \tag{A.1}$$

This equation is for 2D images. Intensity $I(x', y', t)$ is the intensity of pixel (x', y') at time t and got there through a displacement (u, v) of pixel (x, y) at time 0. In literature the generalized form of this equation is used a lot. The spatial information (in this case (x, y)) is placed in vector $\vec{x}$ and the displacement information (in this case (u, v)) is placed in vector $\vec{v}$. The generalized formula then becomes:

$$I(\vec{x'}, t) = I(\vec{x} - \vec{v}t, 0) \tag{A.2}$$

Equation A.1 can be rewritten to:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{A.3}$$

When expending this formula with Taylor series and neglecting higher order terms, this will result in:

$$I(x, y, t) = I(x, y, t) + \frac{\delta I}{\delta x}\delta x + \frac{\delta I}{\delta y}\delta y + \frac{\delta I}{\delta t}\delta t \tag{A.4}$$

This formula can be rewritten using the chain rule. $I(x, y, t)$ is subtracted from both sides.

$$\begin{aligned} 0 &= \frac{\delta I}{\delta x}\frac{\delta x}{\delta t} + \frac{\delta I}{\delta y}\frac{\delta y}{\delta t} + \frac{\delta I}{\delta t} \\ 0 &= I_x u + I_y v + I_t \\ 0 &= \nabla I(\vec{x}, t) \cdot \vec{v} + I_t(\vec{x}, t) \end{aligned} \tag{A.5}$$

where I_x denotes the partial derivative of I over x, I_y the partial derivative of I over y, I_t and $I_t(\vec{x}, t)$ the partial derivative of I over t and $\nabla I(\vec{x}, t) = (I_x, I_y)^T$. All three forms are used in literature and are called the *gradient constraint equation*. A motion constraint line can be drawn from this equation:

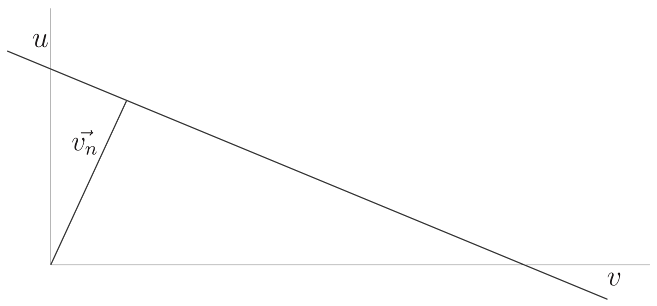

Figure A.2: The Motion Constraint Line

The correct velocity (full velocity) is a point on the constraint line. The minimal velocity is called the normal velocity ($\vec{v_n}$). The gradient constraint equation can provide the normal speed s and normal direction $\vec{n}$:

$$\begin{aligned} \vec{v_n} &= s\vec{n} \\ s &= \frac{-I_t(\vec{x},t)}{||\nabla I(\vec{x},t)||} \\ \vec{n} &= \frac{\nabla I(\vec{x},t)}{||\nabla I(\vec{x},t)||} \end{aligned} \tag{A.6}$$

The gradient constraint equation (A.5) contains two unknown components of $\vec{v}$, constrained by only one linear equation, as graphically presented in figure A.2. An additional constraint is therefore necessary to be able to solve both components of $\vec{v}$.

In image sequences, not every pixel will move independently. All pixels representing a moving object are moving at similar velocities. By making use of this property, an additional constraint is found to be able to solve the equation. This new constraint is called the *smoothness constraint*. Discontinuities in flow can be expected if one object moves over another. This implies that algorithms which are making use of the smoothness constraint are likely to generate poor results at the edges of moving objects. Basically two implementations of this constraint are proposed in literature. The first implementation is called the *global smoothness constraint*. This technique assumes smoothness of the flow over the whole image. Algorithms using this constraint prefer solutions which show more smoothness. The other implementation of the smoothness constraint is called the *local smoothness constraint*. This constraint uses pixels in the neighborhood to estimate the flow of the current pixel.

Another way to constrain equation A.5 further is by using second-order derivatives:

$$\begin{bmatrix} I_{xx}(\vec{x},t) & I_{xy}(\vec{x},t) \\ I_{yx}(\vec{x},t) & I_{yy}(\vec{x},t) \end{bmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} + \begin{pmatrix} I_{tx}(\vec{x},t) \\ I_{ty}(\vec{x},t) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{A.7}$$

Because this second-order derivative implies that the first-order partial derivatives must be conserved ($\frac{d\nabla I(\vec{x},t)}{dt} = 0$), an additional constraint was introduced. First order deformations like rotation and dilation should not be present in the image [1]. This limits the use of algorithms which are making use of second-order derivatives. To measure image velocities A.7 can be combined with A.5 to yield an over-determined system of linear equations. If the aperture problem prevails in a local neighborhood, then the second-order derivatives can usually not be measured accurately enough to determine the tangential component of $\vec{v}$, because of the sensitivity of numerical differentiation [1]. This is why second-order differential methods are often assumed to be sparser and less accurate than estimates from first-order differential methods.

Another constraint to differential techniques is that $I(\vec{x},t)$ must be differentiable. This implies that aliasing has to be avoided. Smoothing filters at the input are used to do this. Another thing to keep in mind is that movements can only be detected if the movement between two successive frames is not too much, i.e. it has to stay within the search window. If aliasing cannot be avoided, then using differential techniques in a coarse-fine manner can be used. These techniques first estimate the coarse scales form where aliasing is less severe. The calculated estimates are later on used as initial guesses for the finer scales. Among others, Johannesson and Gökstorp [44] have used this technique successfully.

A.2.1.1 Horn & Schunck (Global first order)

Horn & Schunck [8] combined the gradient constraint A.5 with a global smoothness term as a constraint for the velocity flow field:

$$\int\int (\nabla I \cdot \vec{v} + I_t)^2 + \alpha^2(||\nabla u||^2 + ||\nabla v||^2)dxdy \tag{A.8}$$

where α is the regulation factor which determines the amount of smoothness in the image. By minimizing the formula above, the values of u and v can be estimated. To do this, iterative equations are used:

$$\begin{aligned} u^{n+1} &= \bar{u}^n - \frac{I_x[I_x\bar{u}^n + I_y\bar{v}^n + I_t]}{\alpha^2 + I_x^2 + I_y^2} \\ v^{n+1} &= \bar{v}^n - \frac{I_y[I_x\bar{u}^n + I_y\bar{v}^n + I_t]}{\alpha^2 + I_x^2 + I_y^2} \end{aligned} \tag{A.9}$$

where n is the iteration number, u^0 and v^0 are the initial velocity estimates which are set to zero and $\bar{u}^n$ and $\bar{v}^n$ are the averages of the neighborhood of u^n and v^n.

The original paper used first-order differences to estimate intensity derivatives:

$$\begin{aligned} I_x &= \frac{1}{4}\{I_{x,y+1,t} - I_{x,y,t} + I_{x+1,y+1,t} - I_{x+1,y,t} + I_{x,y+1,t+1} - I_{x,y,t+1} + I_{x+1,y+1,t+1} - I_{x+1,y,t+1}\} \\ I_y &= \frac{1}{4}\{I_{x+1,y,t} - I_{x,y,t} + I_{x+1,y+1,t} - I_{x,y+1,t} + I_{x+1,y,t+1} - I_{x,y,t+1} + I_{x+1,y+1,t+1} - I_{x,y+1,t+1}\} \\ I_t &= \frac{1}{4}\{I_{x,y,t+1} - I_{x,y,t} + I_{x+1,y,t+1} - I_{x+1,y,t} + I_{x,y+1,t+1} - I_{x,y+1,t} + I_{x+1,y+1,t+1} - I_{x+1,y+1,t}\} \end{aligned} \tag{A.10}$$

Barron et al. [1] suggested to use 4-point central differences to estimate the derivatives, because they find it relatively crude to use the original method. The mask coefficients used for the 4-point central differences are $\frac{1}{12}(-1, 8, 0, -8, 1)$. This results in:

$$\begin{aligned} I_x &= -\frac{1}{12}I_{x-2,y,t} + \frac{8}{12}I_{x-1,y,t} - \frac{8}{12}I_{x+1,y,t} + \frac{1}{12}I_{x+2,y,t} \\ I_y &= -\frac{1}{12}I_{x,y-2,t} + \frac{8}{12}I_{x,y-1,t} - \frac{8}{12}I_{x,y+1,t} + \frac{1}{12}I_{x,y+2,t} \\ I_t &= -\frac{1}{12}I_{x,y,t-2} + \frac{8}{12}I_{x,y,t-1} - \frac{8}{12}I_{x,y,t+1} + \frac{1}{12}I_{x,y,t+2} \end{aligned} \tag{A.11}$$

The local averages $\bar{u}^n$ and $\bar{v}^n$ are found by calculating a weighted sum of the adjacent pixels:

$$
\begin{aligned}
\bar{u}_{x,y,t} =& \frac{1}{6}\{u_{x-1,y,t} + u_{x1,y+1,t} + u_{x+1,y,t} + u_{x,y-1,t}\} \\
&+ \frac{1}{12}\{u_{x-1,y-1,t} + u_{x-1,y+1,t} + u_{x+1,y+,t} + u_{x+1,y-1,t}\} \\
\bar{v}_{x,y,t} =& \frac{1}{6}\{v_{x-1,y,t} + v_{x1,y+1,t} + v_{x+1,y,t} + v_{x,y-1,t}\} \\
&+ \frac{1}{12}\{v_{x-1,y-1,t} + v_{x-1,y+1,t} + v_{x+1,y+,t} + v_{x+1,y-1,t}\}
\end{aligned}
\tag{A.12}
$$

If the brightness gradient is zero in parts of the image, the assigned velocity to that part will be the average of the neighboring velocity estimates. This way, the velocity field has a density of 100%. This is a typical property of optical flow techniques using the global smoothness constraint. However, the main problem of these global smoothness constraint is noise. When the brightness gradient becomes less, the influence of the smoothness constraint becomes more. Thus if the brightness gradient is large, the smoothness constraint has less influence. Because noise causes larger brightness gradients, the influence of the smoothing is limited. The resulting flow field therefore still contains noise.

The number of iterations n has to be at least the size of the largest area to be filled in. Barron et al. [1] have used values up to 100. Except for the derivative estimates, Barron at el. changed the smoothness filter as well. As can be seen in the formulas above, Horn & Schunck used only a spatial smoothing filter. The smoothing regularization factor α of formulas A.8 and A.9 has an original suggested factor of 100. Barron et al. [1] added a spatiotemporal Gaussian pre-filter and lowered the value of α to 0.5. The spatiotemporal pre-filter is defined with a standard deviation of 1.5 pixels in space and 1.5 frames in time (1.5 pixels-frames). The advantage of pre-filtering the input is that noise is reduced before starting the actual optical flow algorithm.

A.2.1.2 Lucas & Kanade (Local first order)

Lucas & Kanade [38, 5] have proposed probably the most famous optical flow algorithm. A lot of optical flow applications are making use of the Lucas & Kanade algorithm itself or a variation of it. Lucas & Kanade have assumed that the velocity of neighboring pixels is constant. This idea is based on the fact that the pixels representing an object have similar velocities. A trivial problem exist on pixels representing the edge of moving objects. This assumption is used together with the gradient constraint (formula A.5). By taking at least two neighboring pixels, the normal velocities of these pixels yields the full velocity. When using more than two pixels, the system is over determined. The full velocity can be found using the (weighted) least squares approach.

When taking n pixels from neighborhood Ω, using gradient constraint A.5 the linear system of this neighborhood becomes:

$$\begin{bmatrix} I_x(\vec{x_1}) & I_y(\vec{x_1}) \\ I_x(\vec{x_2}) & I_y(\vec{x_2}) \\ \vdots & \vdots \\ I_x(\vec{x_N}) & I_y(\vec{x_n}) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\vec{x_1}) \\ I_t(\vec{x_2}) \\ \vdots \\ I_t(\vec{x_n}) \end{bmatrix} \tag{A.13}$$

where I_x, I_y and I_t are the partial derivatives as before and u and v form the displacement vector. The solution to u and v can be found by minimizing the error defined by the sum:

$$S = \sum_{\vec{x} \in \Omega} W^2(\vec{x})[\nabla I(\vec{x}, t) \cdot \vec{v} + I_t(\vec{x}, t)]^2 \tag{A.14}$$

where $W(\vec{x})$ denotes the weight of the selected pixel in order to give center pixels of the neighborhood more influence. The solution to this function can be found by:

$$A^T W^2 A \vec{v} = A^T W^2 \vec{b} \tag{A.15}$$

where for n pixels $\vec{x_i} \in \Omega$,

$$\begin{aligned} A &= [\nabla I(\vec{x_1}, t), \ldots, \nabla I(\vec{x_n}, t)]^T, \\ W &= diag[W(\vec{x_1}), \ldots, W(\vec{x_n})], \\ b &= -[I_t(\vec{x_1}), \ldots, I_t(\vec{x_n})]^T \end{aligned}$$

The solution to (A.15) is $\vec{v} = [A^T W^2 A]^{-1} A^T W^2 \vec{b}$. When writing $A^T W^2 A$ and $A^T W^2 \vec{b}$ further out, the following solutions to these are found:

$$\begin{aligned} A^T W^2 A &= \begin{bmatrix} \sum W^2(\vec{x}) I_x^2(\vec{x}) & \sum W^2(\vec{x}) I_x(\vec{x}) I_y(\vec{x}) \\ \sum W^2(\vec{x}) I_y(\vec{x}) I_x(\vec{x}) & \sum W^2(\vec{x}) I_y^2(\vec{x}) \end{bmatrix} \\ A^T W^2 \vec{b} &= \begin{bmatrix} \sum W^2(\vec{x}) I_x(\vec{x}) I_t(\vec{x}) \\ \sum W^2(\vec{x}) I_y(\vec{x}) I_t(\vec{x}) \end{bmatrix} \end{aligned} \tag{A.16}$$

where all sums are taken over pixels in the neighborhood Ω. Note that there exist only a solution to $\vec{v}$ if $A^T W^2 A$ is invertible.

Barron et al. [1] published a measuring method to determine the probability of the correctness of the velocities. They used the eigenvalues of $A^T W^2 A$, $\lambda_1 \geq \lambda_2$, and a predefined threshold value τ. If both eigenvalues λ_1 and λ_2 are greater than threshold τ, then the full velocity estimate of formula (A.15) is used to calculate $\vec{v}$. If $\lambda_1 \geq \tau$ and $\lambda_2 < \tau$ then the normal velocity estimate is computed. If both eigenvalues λ_1 and λ_2 are less than threshold τ, then no velocity is calculated at all. The by Barron et al. [1] used value for τ is 1.

As with the Horn & Schunck algorithm, Barron et al. [1] used a spatiotemporal Gaussian filter as pre-smoothing filter. The standard deviation used is 1.5 pixels-frames. Also the same derivative method was used, which is the 4-point central differences method with coefficients $\frac{1}{12}(-1, 8, 0, -8, 1)$. The spatial neighborhoods are 5 x 5 pixels and the window function W^2 is isotropic with the effective 1D weights $(0.0625, 0.25, 0.375, 0.25, 0.0625)$.

The advantage of this method over Horn & Schunck is that this method is relatively insensitive to noise, because of the pre-smoothing filter and the local approach. The disadvantage is that the generated flow field is not very dense. This is a side effect of the local approach. If a large uniform part of the image is moving, no motion can be detected inside this part of the image. Because these pixels have the same intensity, no displacement can be detected.

A.2.1.3 Bruhn, Weickert and Schnörr (Hybrid first order)

As shown above, the main advantage of using the global smoothness constraint is that the resulting flow field has a density of 100%, but its disadvantage is that these algorithms are relatively sensitive to noise. Algorithms based on the local smoothness constraint are not very sensitive to noise and therefore produce better results. Unfortunately, these algorithms are not very dense.

Bruhn et al. [45] have observed this same difference in smoothness constraints as well. In order to combine the best of both worlds, they proposed a method of combining the Lucas & Kanade algorithm with the Horn & Schunck algorithm. Now a relatively noise insensitive solution was found giving good optical flow results and has a density of 100%.

Both Lucas & Kanade and Horn & Schunck find the solution to $\vec{v}$ by minimizing a formula (A.8 and A.14). When these formula's are compared, some similarities can be found:

$$\begin{aligned}
\text{HS}: & \int\int (\nabla I \cdot \vec{v} + I_t)^2 + \alpha^2(||\nabla u||^2 + ||\nabla v||^2)dxdy \\
\text{LK}: & \sum_{\vec{x}\in\Omega} W^2(\vec{x})[\nabla I \cdot \vec{v} + I_t]^2
\end{aligned}$$

The basic idea is to merge both formulas into one new formula which has to be minimized to find the values for $\vec{v}$:

$$\int\int W^2(\vec{x})[\nabla I \cdot \vec{v} + I_t]^2 + \alpha^2(||\nabla u||^2 + ||\nabla v||^2)dxdy \tag{A.17}$$

A.2.1.4 Nagel (Global second order)

Nagel [46, 47] has used second order derivatives to measure optical flow and used a global smoothness constraint like Horn & Schunck [8]. Besides that, Nagel has worked on a solution to the poor quality velocity estimates generated by Horn & Schunck on edges of objects. As mentioned earlier, Horn & Schunck used a global smoothness constraint, which results in a smooth velocity flow across the image. Naturally, the velocity flow is not smooth at the edge of a moving object. Nagel has proposed a new constraint: the *oriented-smoothness constraint*. This constraint does not impose smoothness at steep intensity gradients (edges). The resulting solution can be found by minimizing the functional:

$$\int\int (\nabla I^T \vec{v} + I_t)^2 + \frac{\alpha^2}{||\nabla I||_2^2 + 2\delta}\left[(u_x I_y - u_y I_x)^2 + (v_x I_y - v_y I_x)^2 + \delta(u_x^2 + u_y^2 + v_x^2 + v_y^2)\right] dxdy \tag{A.18}$$

The solution of the functional above can be found in a similar way as the solution of Horn & Schunck can be found. This solution is presented below and taken from Barron et al. [1].

By making use of Gauss-Seidel iterations, the solution can be found by:

$$\begin{aligned} u^{n+1} &= \xi(u^n) - \frac{I_x(I_x\xi(u_n) + I_y\xi(v^n) + I_t)}{I_x^2 + I_y^2 + \alpha^2} \\ v^{n+1} &= \xi(v^n) - \frac{I_y(I_x\xi(u_n) + I_y\xi(v^n) + I_t)}{I_x^2 + I_y^2 + \alpha^2} \end{aligned} \tag{A.19}$$

where n is the iteration number and $\xi(u^n)$ and $\xi(v^n)$ are given by

$$\begin{aligned} \xi(u^n) &= \bar{u}^n - 2I_x I_y u_{xy} - \vec{q}^T(\nabla u^n) \\ \xi(v^n) &= \bar{v}^n - 2I_x I_y v_{xy} - \vec{q}^T(\nabla v^n) \end{aligned} \tag{A.20}$$

where u_{xy} and v_{xy} are the estimated partial derivatives of $\vec{v}$, $\bar{u}^n$ and $\bar{v}^n$ are the neighborhood averages of u^n and v^n and $\vec{q}$ is defined as

$$\vec{q} = \frac{1}{I_x^2 + I_y^2 + d\delta}\nabla I^T\left[\begin{pmatrix} I_{yy} & -I_{xy} \\ -I_{xy} & I_{xx} \end{pmatrix} + 2\begin{pmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{pmatrix} W\right] \tag{A.21}$$

where W is defined as

$$\vec{q} = \frac{1}{I_x^2 + I_y^2 + d\delta}\begin{pmatrix} I_y^2 + \delta & -I_x I_y \\ -I_x I_y & I_x^2 + \delta \end{pmatrix} \tag{A.22}$$

Barron et al. [1] suggest to use the same pre-smoothing filter as for Horn & Schunck and for Lucas & Kanade. The intensity derivatives are also estimated using 4-point central differences and are cascaded in different directions to obtain the second order derivatives. The velocity derivatives are found by using 2-point central-difference kernels, $\frac{1}{2}(1, 0, -1)$. The second order velocity derivatives were computed as cascades of the first order derivatives. Barron et al. used 100 iterations ($n = 100$).

A.2.1.5 Uras, Girosi, Verri and Torre (Local second order)

Uras et al. [48] used the second order derivatives formula (A.7) to solve the optical flow field. Formula A.7 can be rewritten as:

$$H\vec{v} = -\nabla I_t \tag{A.23}$$

where H is the Hessian of the image brightness pattern (with respect to the spatial components) and $\nabla I_t = (I_{tx}, I_{ty})^T$ is the temporal derivative.

Using this formula, the initial estimates for $\vec{v}$ can be found as long as the determinant of the Hessian H is not zero. After this, the velocity field is divided into square parts of size n x n. The best n velocity estimates are then taken from this square. To determine the quality of the velocity estimates, the following formula is used:

$$\Delta = \frac{||M^T \nabla I||}{||\nabla I_t||} \tag{A.24}$$

where

$$M = \begin{bmatrix} u_x & u_y \\ v_x & v_y \end{bmatrix}$$

When writing this further down, Δ can be found by:

$$\Delta = \frac{\sqrt{(I_x u_x + I_y v_x)^2 + (I_x u_y + I_y v_y)^2}}{\sqrt{I_{tx}^2 + I_{ty}^2}} \tag{A.25}$$

The lower the value of Δ, the better the velocity estimate. From the n resulting estimates, the best one is selected to represent the velocity of the entire n x n block. This is done by choosing the velocity estimate with the smallest conditional number $\kappa(H)$ of Hessian H.

Theoretically, if no velocity estimate can be found (i.e. if all $det(H) = 0$) within a certain block, that block will not get a velocity assigned. According to Uras et al. [48], this is not likely to happen in real applications.

Before the velocity estimates are calculated, the image is smoothed by using a Gaussian filter. Barron et al. [1] have investigated this algorithm as well and proposed a standard deviation of 3 pixels in space and 1.5 frames in time, where the original paper [48] used 5 pixels in space and 1 frame in time. Barron et al. calculated all derivatives using 4-point central differences. Second order derivatives are estimated using a cascaded version of this.

Uras et al. used $\kappa(H)$ as confidence measure for the velocity estimates. Barron et al. [1] suggest that using $det(H)$ is more reliable. This confidence measure can also be used to threshold the velocity field. When doing this, the accuracy increases, but the density decreases. A threshold of $det(H) \geq 1.0$ can result in good quality velocity estimates [1].

Uras et al. have proposed another method of processing the initial velocity estimates (i.e. the velocities resulting from (A.23)). This second method is by smoothing the velocity field by using a Gaussian filter with a standard deviation of 8 pixels.

A.2.2 Region-Based Techniques

Another technique to determine optical flow is region-based. Here the velocity $\vec{v}$ is defined as the displacement $\vec{d}(d_x, d_y)$ that matches the transition of a region between two images the best. A region is a square selection of pixels. Region-based techniques tries to match an entire region of pixels of one image to a region of pixels on a second image. To determine which match is the best, the similarity between the two regions has to be determined. The best similarity can be found using the following sum-of-squared-differences (SSD):

$$SSD_{1,2}(\vec{x};\vec{d}) = \sum_{j=-n}^{n}\sum_{i=-n}^{n} W(i,j)[I_1(\vec{x}+(i,j)) - I_2(\vec{x}+\vec{d}+(i,j))]^2 \tag{A.26}$$

where W is a discrete 2D window function and displacement $\vec{d}$ is integer only.

Region-based techniques produce one displacement vector per region. This will result in a sparse flow field. Some algorithms based on this technique use this in combination with a pyramid coarse-to-fine manner (e.g. Anandan, as described in next section). A great advantage of region-based techniques is that the regions are normally not overlapping. This makes this kind of algorithms good candidates for large parallelization as all regions undergoes the same treatment.

A.2.2.1 Anandan

Anandan [49, 50] proposed an algorithm which uses this region-based technique to estimate optical flow. Anandan observed the fact that large scale displacements can be detected using low resolution versions of the image. Small displacements can only be detected using full (or at least higher) resolutions. The resulting approach was to first make an estimate of displacement at low resolution, resulting in a coarse estimate of the displacement. The next step is increasing the resolution and with that the accuracy. The displacement estimates of the previous level are used to estimate the displacement of the current level. This is repeated until the full resolution displacement vector is found. This method is also known as the pyramid method.

A confidence measure is needed to determine the certainty of the displacement estimate. If for example the displacement was estimated of a homogeneous area of the image, no component of displacement can be reliably determined.

To end up with a dense flow field, Anandan assumed the flow field to be smooth. This assumption was also made by Horn & Schunck [8], as discussed earlier. The confidence measure is used at each level to determine the amount of filling in necessary.

Obviously, the first step is to build the pyramid. This is done by applying a 5 x 5 Gaussian convolution to the current layer and then sub-sampling that layer into the new layer. Every layer is smoothed this way before the next layer is formed. At the coarsest level, the maximum displacement is 1 pixel. Hence, the search area in the second image is a 3 x 3 pixel area centered around the corresponding pixel in the first image. The next level inherits the initial displacement of the coarser level. The search area here is a 3 x 3 pixel area in the second image centered around this. The best match can be found by minimizing the SSD measure over the search area.

The displacement estimated at a coarse level is projected at the four pixels below (at one level finer). However, if the estimated displacement is incorrect, the search space of the four pixels below does not contain its correct counterpart. If such incorrect displacements have been made at a very coarse level, a lot of pixels are infected later on. To help reduce this from happening, Anandan used the *overlapped pyramid projection scheme*. This scheme projects the displacement result to a 4 x 4 pixel area as illustrated in figure A.3. This way, each pixel at a finer level receives four estimates from the coarser level. The search space for the finer level now becomes 4 x 3 x 3 pixels.

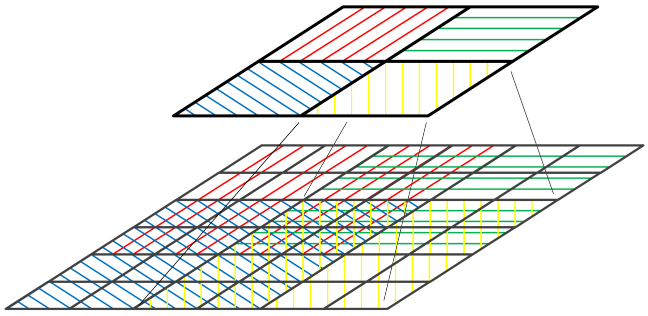

Figure A.3: Overlapped Pyramid Projection Scheme

Since the best match is determined by the minimal SSD, some observations can be done about the confidence of the measure. If a lot of candidates produce small SSD's, all these candidates are good. It is not sure which one reflects the actual displacement. Another problem may arise if none of the candidates result in small SSD's. No good match could be found. Anandan introduced confidence measures c_{min} and c_{max} and derives these from the principle curvatures C_{min} and C_{max} of the SSD surface at the minimum:

$$c_{max} = \frac{C_{max}}{k_1 + k_2 S_{min} + k_3 C_{max}}, c_{min} = \frac{C_{min}}{k_1 + k_2 S_{min} + k_3 C_{min}} \quad \text{(A.27)}$$

where k_1, k_2 and k_3 are normalization parameters and S_{min} is the SSD value corresponding to the best match. Anandan uses the following values [49]: $k_1 = 150$, $k_2 = 1$ and $k_3 = 0$.

Anandan implemented the smoothness constraint of the velocity estimates by using the confidence measures:

$$\int\int (u_x^2 + u_y^2 + v_x^2 + v_y^2) + c_{max}(\vec{v} \cdot \vec{e}_{max} - \vec{v}_0 \cdot \vec{e}_{max})^2 + c_{min}(\vec{v} \cdot \vec{e}_{min} - \vec{v}_0 \cdot \vec{e}_{min})^2 \quad \text{(A.28)}$$

where $\vec{e}_{max}$ and $\vec{e}_{min}$ are the directions of maximum and minimum curvature of the SSD surface (at the minimum) and $\vec{v_0}$ propagates the displacement from the higher level. As did Horn & Schunck [8], Anandan used Gauss-Seidel iterations to find the solution:

$$\vec{v}^{n+1} = \bar{v}^k + \frac{c_{max}}{c_{max}+1}[(\vec{v}_0 - \bar{v}^k) \cdot \vec{e}_{max}]\vec{e}_{max} + \frac{c_{min}}{c_{min}+1}[(\vec{v}_0 - \bar{v}^k) \cdot \vec{e}_{min}]\vec{e}_{min} \quad \text{(A.29)}$$

where $\bar{v}^k$ is the average of the neighborhood of $\vec{v}^k$, computed using the mask:

$$\frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

A.2.2.2 Singh

Singh [51] has used the region-based technique as well. The method of Singh consists of two steps. The first step is called the conservation information step. In this step the initial displacement estimates are computed using the assumption that some image-property does not change over time. Singh used the Laplacian of intensity. The second step is called the neighborhood information step. This step uses knowledge of the velocity distribution in a small spatial neighborhood. At the end, the results from both steps are combined to end up with the final estimate.

The conservation information step uses three filtered images of the sequences, I_{-1}, I_0 and I_{+1}. The filter is a band-pass filter with impulse response $\delta(\vec{x}) - G(\vec{x})$, where $\delta(\vec{x})$ is a Dirca delta function and $G(\vec{x})$ is an isotropic Gaussian with standard deviation 1.0 [1]. The three filtered images are now used to compute SSD values:

$$SSD_0(\vec{x}, \vec{d}) = SSD_{0,1}(\vec{x}, \vec{d}) + SSD_{0,-1}(\vec{x}, -\vec{d}) \tag{A.30}$$

where $SSD_{i,j}$ is formula A.26. Singh now uses these SSD values to obtain the *response-distribution*:

$$R_c(\vec{d}) = e^{-kSSD_0} \tag{A.31}$$

where $k = -ln(0.95)/min(SSD_0)$. If the minimal value of SSD_0 is zero, the smallest non-zero value of SSD_0 is used. An estimate of the sub-pixel true-velocity $\vec{v}_c = (u_c, v_c)$ can be found by:

$$u_c = \frac{\sum R_c(\vec{d}) d_x}{\sum R_c(\vec{d})}, v_c = \frac{\sum R_c(\vec{d}) d_y}{\sum R_c(\vec{d})} \tag{A.32}$$

Singh suggests to use the eigenvalues of the inverse covariance matrix as measures of confidence. The covariance matrix is given by:

$$S_c = \frac{1}{\sum R_c(\vec{d})} \begin{bmatrix} \sum R_c(\vec{d})(d_x - u_c)^2 & \sum R_c(\vec{d})(d_x - u_c)(d_y - v_c) \\ \sum R_c(\vec{d})(d_x - u_c)(d_y - v_c) & \sum R_c(\vec{d})(d_y - v_c)^2 \end{bmatrix} \tag{A.33}$$

All summations in the formulas above are carried out over $-N \leq d_x,\ d_y \leq +N$, where N is the maximum displacement which can be detected by the algorithm. Values of N up to 4 are used by Barron et al. [1]. To be able to detect large velocities (i.e. large displacements between images), Singh suggests to use a pyramid approach, as Anandan [49, 50] did.

At the second step (neighborhood information) the objective is to propagate velocity by neighborhood information. It is assumed that the velocity of neighboring pixels of a certain pixel gives information about the velocity of the pixel under consideration. Singh determines this velocity $\vec{v}_n = (u_n, v_n)$ by:

$$u_n = \frac{\sum_i R_n(\vec{v}_i) u_i}{\sum_i R_n(\vec{v}_i)}, v_n = \frac{\sum_i R_n(\vec{v}_i) v_i}{\sum_i R_n(\vec{v}_i)} \tag{A.34}$$

where $\vec{v}_i = (u_i, v_i)$ are the velocities of the neighborhood pixels and $R_n(\vec{v}_i)$ is a Gaussian function of the distance between the neighborhood pixel and the central pixel. The neighborhood has a size of $(2w+1)$ x $(2w+1)$. Singh used $w=1$, while Barron et al. [1] found $w=2$ to produce better estimates. The corresponding covariance matrix is:

$$S_n = \frac{1}{\sum_i R_n(\vec{v}_i)} \begin{bmatrix} \sum_i R_n(\vec{v}_i)(u_i-u_n)^2 & \sum_i R_n(\vec{v}_i)(u_i-u_n)(v_i-v_n) \\ \sum_i R_n(\vec{v}_i)(u_i-u_n)(v_i-v_n) & \sum_i R_n(\vec{v}_i)(v_i-v_n)^2 \end{bmatrix} \tag{A.35}$$

Both estimates can be fused together by minimizing the errors of both. This can be expressed in the following formula:

$$\int\int (\vec{v}-\vec{v}_n)^T S_n^{-1}(\vec{v}-\vec{v}_n) + (\vec{v}-\vec{v}_c)^T S_c^{-1}(\vec{v}-\vec{v}_c)dxdy \tag{A.36}$$

Because $\vec{v}_n$ and S_n require to know the velocity of the neighboring pixels, Singh derives iterative equations to find the estimates (Gauss-Seidel relaxation):

$$\begin{aligned} \vec{v}_n^0 &= \vec{v}_c \\ \vec{v}_n^{k+1} &= \left[S_c^{-1} + (S_n^k)^{-1}\right]^{-1} \left[S_c^{-1}\vec{v}_c + (S_n^k)^{-1}\vec{v}_n^k\right] \end{aligned} \tag{A.37}$$

where k is the iteration number. Barron et al. [1] used at most 25 iterations.

The covariant matrices S_c and S_n are combined to obtain a new covariant matrix of the entire system. This matrix is defined as: $[S_c^{-1} + S_n^{-1}]^{-1}$. The eigenvalues of this matrix λ_1 and λ_2 function as the confidence measure of the final estimates. A threshold τ can be used to ignore estimates with low confidence measures.

Log File Format Description B

This appendix describes the format of the log file generated by executing an application which was compiled with GCC and the presented PlugIn.

A log file consists of log lines. Each line refers to a statement or an event in the original source code. Each line is ended with a semicolon and a new line character, i.e. '`;\n`'. Different elements within a line are separated by a comma.

B.1 Log Function

A function is logged whenever a new function which was selected to get analyzed is started. The first statement of the function will be the statement responsible for creating this log line to the log file.

Example function log:

```
F,foo,main.cpp,5;
```

Element	**Description**
`F`	Indicates a function log line
`foo`	The name of the function recorded
`main.cpp`	The source file where the function exists
`5`	The line within the source file where the first statement exists

Table B.1: Element description of function log line

B.2 Log Basic Block

A basic block log line is written to the log file whenever a basic block is entered or left. The statements for logging this are inserted as first and last statement of the basic block for the start and end log line respectively. If the last original statement of the basic block is a `RETURN` or `CONDITIONAL` statement, the statement recording the end of the basic block is inserted as the second to last statement in that basic block.

Example basic block log:
`B,s,0x12345678,main.cpp,5;`

Element	**Description**
`B`	Indicates a basic block log line
`s`	Indicates whether the basic block did start ('s') or end ('e')
`0x12345678`	The unique basic block identifier
`main.cpp`	The source file where the basic block exists
`5`	The line within the source file where the first statement exists if the block was just started or the last statement exists if the block has ended

Table B.2: Element description of basic block log line

B.3 Log Memory Access

A memory access is logged whenever an access to a variable other than array or pointer is recorded. Constants are also logged here. When the recorded access involved a read from memory, the statement recording the log is inserted above the statement it indicates. If a write was recorded, the statement recording the access is inserted below the actual memory access.

Example memory access log:
`M,r,name,uint_32,35,main.cpp,5;`

Element	**Description**
`M`	Indicates a memory access log line
`r`	Indicates whether the access was a read ('r') or write ('w')
`name`	The name of the variable
`uint_32`	The type of the variable
`35`	The value of the variable if type is integer or boolean ('false' and 'true')
`main.cpp`	The source file where the memory access exists
`5`	The line within the source file where the access exists

Table B.3: Element description of memory access log line

B.4 Log Array Access

An array access is logged whenever the access is recorded. When the recorded access involved a read from memory, the statement recording the log is inserted above the statement it indicates. If a write was recorded, the statement recording the access is inserted below the actual array access.

Example array access log:
`A,r,name[index1=0][index2=2],uint_32,35,main.cpp,5;`

Element	Description
`A`	Indicates an array access log line
`r`	Indicates whether the access was a read ('r') or write ('w')
`name`	The name of the array
`[index1=0]`	The index and its value that was used to point to the array element (if a multi-dimensional array was accessed, all indices are located after each other)
`uint_32`	The type of the array
`35`	The value of the array if type is integer or boolean ('false' and 'true') (This field is not yet implemented with array accesses)
`main.cpp`	The source file where the access exists
`5`	The line within the source file where the access exists

Table B.4: Element description of array log line

B.5 Log Pointer Access

A pointer access is logged whenever the access is recorded. When the recorded access involved a read from memory or the pointer address was read, the statement recording the log is inserted above the statement it indicates. If a write or address set was recorded, the statement recording the access is inserted below the actual pointer access.

Example pointer access log:
`P,r,name,0x12345678,*uint_32,35,main.cpp,5;`

Element	Description
`P`	Indicates a pointer access log line
`r`	Indicates whether the access was a read ('r'), write ('w'), get address ('g') or set address ('s')
`name`	The name of the pointer
`0x12345678`	The address of the pointer
`*uint_32`	The type of the pointer
`35`	The value of the pointer if type is integer or boolean ('false' and 'true')
`main.cpp`	The source file where the access exists
`5`	The line within the source file where the access exists

Table B.5: Element description of pointer log line

B.6 Log Pointer Set Address from Expression

A pointer set address from expression is logged whenever the access occurs. The statement logging the access is inserted below the actual pointer set address from expression statement.

Example pointer set address from expression log:
`PSE,dest,src,0x12345678,main.cpp,5;`

Element	Description
`PSE`	Indicates a pointer set address from expression log line
`dest`	The name of the pointer for which the address was set
`src`	The name of the expression holding the address
`0x12345678`	The new address of the pointer
`main.cpp`	The source file where the address was set
`5`	The line within the source file where the address was set

Table B.6: Element description of pointer set address from expression log line

B.7 Log Pointer Set Address from Pointer

A pointer set address from pointer is logged whenever the access occurs. The statement logging the access is inserted below the actual pointer set address from pointer statement.

Example pointer set address from pointer log:
`PSP,dest,src,0x12345678,main.cpp,5;`

Element	**Description**
`PSP`	Indicates a pointer set address from pointer log line
`dest`	The name of the pointer for which the address was set
`src`	The name of the pointer from which the address was used
`0x12345678`	The new address of the pointer
`main.cpp`	The source file where the address was set
`5`	The line within the source file where the address was set

Table B.7: Element description of pointer set address from pointer log line

B.8 Log Cast

A cast is logged whenever it occurs. The statement logging the cast is inserted below the actual cast statement.

Example cast log:
`C,srcName,int_32,destName,int_8,35,main.cpp,5;`

Element	**Description**
`C`	Indicates a cast log line
`srcName`	The name of the source variable
`int_32`	The data type of the source variable
`destName`	The name of the destination variable
`int_8`	The data type of the destination variable
`35`	The value of the cast if type is integer or boolean ('false' and 'true')
`main.cpp`	The source file where the cast exists
`5`	The line within the source file where the cast exists

Table B.8: Element description of cast log line

B.9 Log Pointer Cast

A pointer cast is logged whenever it occurs. The statement logging the pointer cast is inserted below the actual pointer cast statement.

Example pointer cast log:
`CP,src,dest,0x12345678,main.cpp,5;`

Element	Description
`CP`	Indicates a pointer cast log line
`src`	The name of the source pointer
`dest`	The name of the destination pointer
`0x12345678`	The address which was used in the pointer cast
`main.cpp`	The source file where the pointer cast exists
`5`	The line within the source file where the pointer cast exists

Table B.9: Element description of pointer cast log line

B.10 Log Operation

An operation is logged whenever it occurs. The statement logging the operation is inserted below the actual operation statement.

Example operation log:
`O2,plus,main.cpp,5;`

Element	Description
`O2`	Indicates an operation log line (binary operations are indicated with 'O2'; unary operations are indicated with 'O1')
`plus`	The description of the operation
`main.cpp`	The source file where the operation exists
`5`	The line within the source file where the operation exists

Table B.10: Element description of operation log line

B.11 Log Condition

A condition is logged whenever it occurs. The statement logging the condition is inserted above the actual condition statement.

Example condition log:
`CO,var1Name,int_32,35,equal,var2Name,int_8,45,main.cpp,5;`

Element	Description
`CO`	Indicates a condition log line
`var1Name`	The name of the first variable which is used to check the condition
`int_32`	The type of the first variable
`35`	The value of the first variable if type is integer or boolean ('false' and 'true')
`equal`	The type of condition check
`var2Name`	The name of the second variable which is used to check the condition
`int_8`	The type of the second variable
`45`	The value of the second variable if type is integer or boolean ('false' and 'true')
`main.cpp`	The source file where the condition exists
`5`	The line within the source file where the condition exists

Table B.11: Element description of condition log line

B.12 Log Array Set Pointer

An array set pointer is logged whenever the creation of a pointer to an element in an array is detected. The statement logging this event is inserted below the statement creating the pointer. This log line begins with an 'A', as an array access does. The difference is the second character, which is always an 's' here.

Example array access log:
`A,s,arrayName[index=0],0x12345678,ptrName;`

Element	Description
`A`	Indicates an array access log line
`s`	Indicates that a pointer was set to an array element
`arrayName`	The name of the array
`[index=0]`	The index and its value that was used to point to the array element
`0x12345678`	The address of the created pointer
`ptrName`	The name of the created pointer

Table B.12: Element description of array set pointer log line

B.13 Log Function Call

A function call is logged whenever it occurs. A function log contains two log lines. The first indicates the start of the function call, the second indicates the end. The start of the function call is inserted before the call itself. The end of the function call is inserted after the call itself. The actual function call exists between the two log lines. If the function to be called is also analyzed, all log lines of that function are placed between the start and end log lines.

Example function call log:

```
CAS,function(parm1=35,parm2=0),main.cpp,5;
CAE,retName,int_32,48,main.cpp,5;
```

Element	**Description**
`CAS`	Indicates the start of a function call
`function`	The name of the function to be called
`parm1`	The name of the first parameter (if any)
`35`	The value of the first parameter (if any parameter and type is integer or boolean)
`parm2`	The name of the second parameter (if any; all parameters are logged, separated by commas)
`0`	The value of the second parameter (if any parameter and type is integer or boolean)
`main.cpp`	The source file where the function call exists
`5`	The line within the source file where the function call exists
`CAE`	Indicates the end of a function call
`retName`	The name of the variable in which the return value of the called function is saved (if no value is returned, this field is empty)
`int_32`	The data type of the return value of the function to be called (if no value is returned, this field is empty)
`48`	The value of the return value of the function to be called (if type is integer or boolean) (if no value is returned, this field is empty)
`main.cpp`	The source file where the function call exists
`5`	The line within the source file where the function call exists

Table B.13: Element description of function call log line

C Analyze Library Description

The presented PlugIn inserts function calls into the AST which will generate the log file as described in Appendix B. The inserted function calls are defined in the analyze library (`analyze.cpp`). The linker stage of the compilation process combines the library object file with the object files of the algorithm which is to be analyzed. The result is one executable file.

This appendix describes the functions defined in the analyze library.

C.1 castInt

This function returns the integer value based on the value passed as argument and the valueType as described in Section 4.3.5.1.

Name	**Type**	**Description**
	Return	
	`int`	The value corresponding to the input arguments
	Arguments	
`value`	`int`	The value of the access to log
`valueType`	`int`	The valueType of the access to log

Table C.1: Return value and arguments of `castInt` function

C.2 print

Prints a string to the screen when running the executable.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`text`	`const char*`	The string to print

Table C.2: Return value and arguments of `print` function

C.3 writeToFile

Writes a string to the log file.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`text`	`const char*`	The string to print to the file

Table C.3: Return value and arguments of `writeToFile` function

C.4 logFunction

Logs the start of a function.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`function`	`const char*`	The function name
`sourcefile`	`const char*`	The source file of the function
`sourceline`	`const int`	The source line where the function starts

Table C.4: Return value and arguments of `logFunction` function

C.5 logVarAccessInt

Logs all memory accesses to variables other than array or pointer.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`access`	`const char`	Indicates a read ('`r`') or write ('`w`')
`variable`	`const char*`	The name of the variable
`type`	`const char*`	The data type of the variable
`value`	`const int`	The value of the variable (if type is integer or boolean)
`valueType`	`const int`	The valueType of the variable
`sourcefile`	`const char*`	The source file of the variable
`sourceline`	`const int`	The source line of the variable

Table C.5: Return value and arguments of `logVarAccessInt` function

C.6 logPtrAccessInt

Logs all pointer accesses.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`access`	`const char`	Indicates a read ('`r`'), write ('`w`'), get address ('`g`') or set address ('`s`')
`pointer`	`const char*`	The name of the pointer
`type`	`const char*`	The data type of the pointer
`value`	`const int*`	The address of the pointer
`valueType`	`const int`	The valueType of the pointer
`sourcefile`	`const char*`	The source file of the pointer
`sourceline`	`const int`	The source line of the pointer

Table C.6: Return value and arguments of `logPtrAccessInt` function

C.7 logPtrSetExpr

Logs pointer set address from expression.

Name	Type	Description
Return		
N/A		
Arguments		
`nameDest`	`const char*`	The name of the pointer
`nameSrc`	`const char*`	The name of the expression
`pointer`	`const int*`	The address of the pointer
`sourcefile`	`const char*`	The source file of the pointer
`sourceline`	`const int`	The source line of the pointer

Table C.7: Return value and arguments of `logPtrSetExpr` function

C.8 logPtrSetPtr

Logs pointer set address from other pointer.

Name	Type	Description
Return		
N/A		
Arguments		
`nameDest`	`const char*`	The name of the pointer of which the address was set
`nameSrc`	`const char*`	The name of the pointer of which the address was read
`pointer`	`const int*`	The address of the pointer
`sourcefile`	`const char*`	The source file of the pointer
`sourceline`	`const int`	The source line of the pointer

Table C.8: Return value and arguments of `logPtrSetPtr` function

C.9 logOperation

Logs all operations.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`operation`	`const char*`	The name of the operation
`sourcefile`	`const char*`	The source file of the operation
`sourceline`	`const int`	The source line of the operation
`type`	`const char`	The type of operation: unary ('`1`') or binary ('`2`')

Table C.9: Return value and arguments of `logOperation` function

C.10 logCastInt

Logs all casts between variable, except for pointer casts.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`srcVar`	`const char*`	The name of the source variable
`srcType`	`const char*`	The data type of the source variable
`destVar`	`const char*`	The name of the destination variable
`destType`	`const char*`	The data type of the destination variable
`value`	`const int`	The value of the variable (if type is integer or boolean)
`valueType`	`const int`	The valueType of the variable
`sourcefile`	`const char*`	The source file of the cast
`sourceline`	`const int`	The source line of the cast

Table C.10: Return value and arguments of `logCastInt` function

C.11 logPtrCast

Logs all pointer casts.

Name	Type	Description
Return		
N/A		
Arguments		
`srcVar`	`const char*`	The name of the source pointer
`destVar`	`const char*`	The name of the destination pointer
`value`	`const int*`	The address of the pointer
`sourcefile`	`const char*`	The source file of the pointer cast
`sourceline`	`const int`	The source line of the pointer cast

Table C.11: Return value and arguments of `logPtrCast` function

C.12 logCond

Logs all conditional branches.

Name	Type	Description
Return		
N/A		
Arguments		
`var1Name`	`const char*`	The name of the first variable
`var1Type`	`const char*`	The data type of the first variable
`var1Value`	`const int`	The value of the first variable (if type is integer or boolean)
`var1ValueType`	`const int`	The valueType of the first variable
`name`	`const char*`	The type of condition
`var2Name`	`const char*`	The name of the second variable
`var2Type`	`const char*`	The data type of the second variable
`var2Value`	`const int`	The value of the second variable (if type is integer or boolean)
`var2ValueType`	`const int`	The valueType of the second variable
`sourcefile`	`const char*`	The source file of the condition
`sourceline`	`const int`	The source line of the condition

Table C.12: Return value and arguments of `logCond` function

C.13 logBasicBlock

Logs all starts and ends of basic blocks. This function also keeps track of the file size of the log file. If the size exceeds 1 GB, a new log file is started.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`action`	`const char`	Indicates a start ('`s`') or end ('`e`')
`bb`	`const long`	The unique identifier of the basic block
`sourcefile`	`const char*`	The source file of the basic block
`sourceline`	`const int`	The source line of the where the basic block starts or ends

Table C.13: Return value and arguments of `logBasicBlock` function

C.14 logArrayIndex

Logs a single array index.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`name`	`const char*`	The name of the variable used to index the array
`value`	`const int`	The value of the index
`valueType`	`const int`	The valueType of the index

Table C.14: Return value and arguments of `logArrayIndex` function

C.15 logArrayAccessIntStart

Logs every character of an array access log before the array index (or indices) is printed. This function is also used if a pointer is found which points to an array element.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`access`	`const char`	Indicates a read ('`r`'), write ('`w`') or set array pointer ('`s`')
`variable`	`const char*`	The name of the array

Table C.15: Return value and arguments of `logArrayAccessIntStart` function

C.16 logArrayAccessIntEnd

Logs every character of an array access log after the array index (or indices) is printed.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`type`	`const char*`	The data type of the array
`value`	`const int`	The value of the array element (not yet implemented)
`valueType`	`const int`	The valueType of the array element
`sourcefile`	`const char*`	The source file of the array access
`sourceline`	`const int`	The source line of the array index

Table C.16: Return value and arguments of `logArrayAccessIntEnd` function

C.17 logArrayAccessIntSetPtr

Logs every character of a pointer detected to point to an array element log after the array index (or indices) is printed.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`value`	`const void*`	The address of the pointer (and array element)
`newVarName`	`const char*`	The name of the pointer pointing to the array element

Table C.17: Return value and arguments of `logArrayAccessIntSetPtr` function

C.18 logCallStart

Logs every character of a function call log before the parameter(s) are printed.

Name	**Type**	**Description**
	Return	
N/A		
	Arguments	
`function`	`const char*`	The name of the called function

Table C.18: Return value and arguments of `logCallStart` function

C.19 logCallPar

Logs a single function parameter.

Name	Type	Description
Return		
N/A		
Arguments		
`parName`	`const char*`	The name of the variable used as parameter
`parValue`	`const int`	The value of parameter (if type is integer or boolean)
`parValueType`	`const int`	The valueType of the parameter
`last`	`int`	Indicates if this was the last parameter to log (this determines if a comma has to be printed or not)

Table C.19: Return value and arguments of `logCallPar` function

C.20 logCallEnd

Logs every character of a function call log after the parameter(s) are printed.

Name	Type	Description
Return		
N/A		
Arguments		
`sourcefile`	`const char*`	The source file of the function call
`sourceline`	`const int`	The source line of the function call

Table C.20: Return value and arguments of `logCallEnd` function

C.21 logCallReturn

Logs the return of a function call. The values for retName and retType may be empty (i.e. `""`) if no return value was saved (or exists).

Name	Type	Description
Return		
N/A		
Arguments		
`retName`	`const char*`	The name of the variable in which the return value was saved
`retType`	`const char*`	The type of return value
`retValue`	`const int`	The value of the return value (if type is integer or boolean)
`retValueType`	`const int`	The valueType of the return value
`sourcefile`	`const char*`	The source file of the function call
`sourceline`	`const int`	The source line of the function call

Table C.21: Return value and arguments of `logCallReturn` function

C.22 openLogFile

Opens a new log file. A function call to this function is inserted as first statement in the main function of the algorithm to analyze.

Name	Type	Description
Return		
N/A		
Arguments		
`filename`	`const char*`	The filename of the file to create (without the _0.log extension)

Table C.22: Return value and arguments of `openLogFile` function

C.23 closeLogFile

Closes the new log file. A function call to this function is inserted as last statement in the main function of the algorithm to analyze.

Name	**Type**	**Description**
Return		
N/A		
Arguments		
N/A		

Table C.23: Return value and arguments of `closeLogFile` function

C.24 createNewLogFile

Creates a new log file. The index number of the log file is incremented here. This function is called by the `logBasicBlock()` function if the current log file exceeds 1 GB in file size.

Name	**Type**	**Description**
Return		
N/A		
Arguments		
N/A		

Table C.24: Return value and arguments of `closeLogFile` function

Bibliography

[1] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," vol. 12, pp. 43 – 77, Feb. 1994.

[2] K.-T. Song and J.-H. Huang, "Fast optical flow estimation and its application to real-time obstacle avoidance," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, pp. 2891 – 2896, 2001.

[3] A. Gern, R. Moebus, and U. Franke, "Vision-based lane recognition under adverse weather conditions using optical flow," in *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 2, pp. 652 – 657, Jun. 2002.

[4] G. Monteiro, M. Ribeiro, J. Marcos, and J. Batista, "Wrongway drivers detection based on optical flow," in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 5, pp. 141 – 144, Oct. 2007.

[5] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," *PhD Dissertation, Dept. of Computer Science, Carnegie-Mellon University*, 1981.

[6] Y. Kong, X. Zhang, Q. Wei, W. Hu, and Y. Jia, "Group action recognition in soccer videos," in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pp. 1 – 4, Dec. 2008.

[7] G.-J. Kim, K.-Y. Eom, M.-H. Kim, J.-Y. Jung, and T.-K. Ahn, "Automated measurement of crowd density based on edge detection and optical flow," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, pp. 553 – 556, May 2010.

[8] B. K. P. Horn and B. G. Schunck, "Determining optical flow," 1981.

[9] J. M. Cardoso and P. C. Diniz, "Compilation techniques for reconfigurable architectures," Apr. 2008.

[10] O. S. Dragomir, E. Moscu-Panainte, K. Bertels, and S. Wong, "Optimal unroll factor for reconfigurable architectures," 2008.

[11] O. Dragomir, T. Stefanov, and K. Bertels, "Loop unrolling and shifting for reconfigurable architectures," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 167 – 172, Sept. 2008.

[12] M. Fingeroff, "High-level synthesis blue book," Jan. 2010.

[13] Y. Dong, J. Zhou, Y. Dou, L. Deng, and J. Zhao, "Impact of loop unrolling on area, throughput and clock frequency for window operations based on a data schedule method," in *Image and Signal Processing, 2008. CISP '08. Congress on*, vol. 1, pp. 641 – 645, May 2008.

[14] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The illiac iv computer," *Computers, IEEE Transactions on*, vol. C-17, pp. 746 – 757, Aug. 1968.

[15] Control Data Corporation, "Control data 6400/6500/6600 computer systems reference manual," 1969.

[16] T. VanCourt and M. Herbordt, "Application-specific memory interleaving enables high performance in fpga-based grid computations," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pp. 305 – 306, Apr. 2006.

[17] T. VanCourt and M. Herbordt, "Application-specific memory interleaving for fpga-based grid computations: A general design technique," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1 – 7, Aug. 2006.

[18] J. S. Liptay, "Structural aspects of the system/360 model 85, ii: The cache," *IBM Systems Journal*, vol. 7, no. 1, pp. 15 – 21, 1968.

[19] D. A. Patterson and J. L. Hennessy, "Computer organisation and design," 2007.

[20] A. Gil, J. Benitez, M. Calvi ando, and E. Go andmez, "Reconfigurable cache implemented on an fpga," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pp. 250 – 255, Dec. 2010.

[21] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A sw performance estimation framework for early system-level-design using ne-grained instrumentation," 2006.

[22] R. Yan and S. C. Goldstein, "Mobile memory: Improving memory locality in very large recongurable fabrics," Apr. 2002.

[23] T. Harmon, "Volta," 2007. Online available: http://volta.sourceforge.net/.

[24] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution proler," Apr. 2004.

[25] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: Analyzing memory system bottlenecks in programs," in *In Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 1 – 12, 1992.

[26] S. A. Ostadzadeh, R. J. Meeuws, C. Galuzzi, and K. Bertels, "QUAD - a memory access pattern analyser," Mar. 2010.

[27] Valrind Developers, "Valgrind documentation," 2011. Online available: http://valgrind.org/docs/manual/index.html.

[28] "Valgrind tools," 2011. Online available: http://valgrind.org/info/tools.html.

[29] Free Software Foundation, "GDB: The GNU project debugger," Jul. 2011. Online available: http://www.gnu.org/s/gdb/.

[30] IBM, "Debugging with the eclipse platform," May 2007. Online available: http://www.ibm.com/developerworks/library/os-ecbug.

[31] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *In PLDI 05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190 – 200, ACM Press, 2005.

[32] Free Software Foundation, "GCC, the GNU compiler collection," Aug. 2011. Online available: http://gcc.gnu.org/.

[33] Free Software Foundation, "Documentation of the internals of the GNU compilers," 2010. Online available: http://gcc.gnu.org/onlinedocs/gccint/index.html.

[34] J. Merrill, "GENERIC and GIMPLE: A new tree representation for entire functions."

[35] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan, "Designing the mccat compiler based on a family of structured intermediate representations," 1992.

[36] The Flex Project, "Flex: The fast lexical analyzer," Feb. 2008. Online available: http://flex.sourceforge.net/.

[37] Free Software Foundation, "Bison: GNU parser generator," May 2011. Online available: http://www.gnu.org/software/bison/.

[38] B. D. Lucas, "Generalized image matching by the method of differences," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pp. 280 – 287, 1984.

[39] T. Williams and C. Kelley, "Gnuplot," Mar. 2011. Online available: http://www.gnuplot.info/.

[40] X. Ren, "Rtl implementation of an optical flow algorithm (lucas) using the catapult c high-level synthesis tool," Aug. 2011.

[41] T. Hurkmans, "System performance analysis and fixed-point architecture of a gradient-based optical flow algorithm," Dec. 2009.

[42] M. Klingemann, "StackBlur." Online available: http://incubator.quasimondo.com/processing/fast_blur_deluxe.php.

[43] J. Barron, D. Fleet, S. Beauchemin, and T. Burkitt, "Performance of optical flow techniques," in *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR '92., 1992 IEEE Computer Society Conference on*, pp. 236 – 242, Jun. 1992.

[44] M. Johannesson and M. Gökstorp, “Video-rate pyramid optical flow computation on the linear simd array ivip,” in *Computer Architectures for Machine Perception, 1995. Proceedings. CAMP ’95*, pp. 280 – 287, Sept. 1995.

[45] A. Bruhn, J. Weickert, and C. Schnrr, “Lucas/kanade meets horn/schunck: Combining local and global optic flow methods,” *International Journal of Computer Vision*, vol. 61, pp. 211 – 231, 2005.

[46] H. H. Nagel, “Displacement vectors from second-order intensity variations in image sequences,” *CGIP 21*, pp. 85 – 117, Mar. 1982.

[47] H.-H. Nagel and W. Enkelmann, “An investigation of smoothness constraints for the estimation of displacement vector fields from image sequences,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-8, pp. 565 – 593, Sept. 1986.

[48] S. Uras, F. Girosi, A. Verri, and V. Torre, “A computational approach to motion perception,” *Biological Cybernetics*, vol. 60, pp. 79 – 87, 1988.

[49] P. Anandan, “Measuring visual motion from image sequences,” *PhD thesis, COINS. Dept., Univ. Massachusetts, Amhers*, 1987.

[50] P. Anandan, “A computational framework and an algorithm for the measurement of visual motion,” *International Journal of Computer Vision*, vol. 2, pp. 283 – 310, 1989.

[51] A. Singh, “An estimation-theoretic framework for image-flow computation,” in *Computer Vision, 1990. Proceedings, Third International Conference on*, pp. 168 – 177, Dec. 1990.