Abstract

Pattern discovery in genetic sequences is one of the major problems in contemporary biological research. Many algorithms exist for locating such patterns, and most of these are based on a small set of building blocks for testing calculated patterns. One of these building blocks is the Position Weight Matrix (PWM), which represents the problem as a matrix of scores that is compared to a series of sequence windows to produce a set of PWM scores.

The PWM calculation is the bottleneck in most of these algorithms. This paper therefore explores the viability of making an implementation of a PWM on a Field-Programmable Gate Array (FPGA) in order to exploit the large inherent parallelism in the PWM algorithm. Several different solutions are explored in the overall design as well as in the various modules, in order to balance factors such as throughput, accuracy and bit resolution, dataset storage capacity, substring length, alphabet size and chip utilization. The speedups attained are then compared to pure software solutions.

Contents

1 Introduction **7**

1.1 The Cray XD1 Supercomputer 8
1.2 Field-Programmable Gate Arrays 10
1.3 The Significance of In Silico Motif Discovery 12
1.4 The PWM Algorithm 13
1.5 Earlier work in this area 14

2 The FPWM Prototype **17**

2.1 The Hardware Implementation 17
2.1.1 The Control Module 18
2.1.2 The Memory Module 21
2.1.3 The PWM Module 22
2.1.4 The Adder Module 23
2.1.5 The Result Module 24
2.2 The Hardware-Software Interface 26
2.3 Implementation Problems 28
2.4 Summary 31

3 The FPWM Prototype Simulator **33**

3.1 The FPWMSim Implementation 33
3.2 FPWMSim Data Files 34
3.3 Using FPWMSim 35
3.4 Summary 37

4 Results **39**

4.1 Performance Measurements 39
4.1.1 Compared to general-purpose CPUs 39
4.1.2 Compared to Interagon's Pattern Matching Chip 41
4.2 Resource Measurements 42
4.2.1 Bit Resolution 42
4.2.2 PWM length 43
4.2.3 Alphabet size 43
4.2.4 Sequence Length 44
4.3 Summary 45

5 Future Work **47**
5.1 Module Variations 47
5.1.1 Changing the bit resolution 47
5.1.2 Changing the PWM length 48
5.1.3 Changing the alphabet size 49
5.1.4 Result Module: Simple filtered output 49
5.2 Local Parallelized FPWM Cores 50
5.3 Multi-Node FPWM Implementations 54
5.3.1 Parallelizing Work 55
5.3.2 Parallel Computation 55
5.4 Summary 57

6 Conclusions **59**

List of Figures

1 The Cray XD1 System. 9
2 The FPGA's interconnection facilities. 10
3 Two steps of the PWM algorithm. 15
4 An overview illustration of the entire system. 19
5 An overview illustration of the entire FPGA, including the various modules of the FPWM, and the FPGA's connections to the outside world. 21
6 An 8/4 PWM Module 23
7 A full eight-element Adder Module. 24
8 A Storage Element. 26
9 A simplified four-element Result Module. 27
10 FPWMSim during operation. 36

List of Tables

1 The resource consumptions of each module on the FPWM with the default configuration. 42
2 The results of varying the bit resolution on the FPWM prototype. 43
3 The results of varying the PWM length on the FPWM prototype. 43
4 The results of varying the alphabet size on the FPWM prototype. 44

1 Introduction

Position Weight Matrices (PWMs) are often used to represent patterns in biological sequences. A PWM is based around a matrix of element scores, holding one score value for each particular element for each column position in the matrix. A set of scores for a string of elements is computed by sequentially looking at fixed-length *windows*, or substrings, and looking up the value for each element of the substring. In the case of *log-likelihood PWMs*, which are examined here, these values are summed to produce a set of score sums, called the string's *profile*.

The overall goal of this paper was to explore the viability of using a Field-Programmable Gate Array (FPGA) implementation of a PWM in order to speed up this computation without severely compromising accuracy or flexibility, since the PWM calculation is a bottleneck in several popular algorithms used to discover biological motifs. Using an FPGA enables a large amount of parallel computation to be performed, provided the algorithm in question is highly parallelizable, as is the case with PWMs.

This paper presents an FPGA implementation of a PWM, where each particular part of the PWM process is implemented as an individual module, to easily allow replacement of alternative implementations. The implementation was done on a Cray XD-1 Supercomputer, which has a base configuration of six dual-CPU SMP nodes, each having an FPGA directly connected to the SMP through a HyperTransport bus. Several different solutions were explored in the overall design as well as in the various modules, in order to balance factors such as throughput, accuracy and bit resolution, dataset storage capacity, substring length, alphabet size and chip utilization.

This implementation was to be used as a black box PWM solver for Gibbs Sampling and MEME pattern discovery implementations ([16], [17]), with performance measurements of these implementations being compared to naive and optimized software solutions, as well as an earlier solution running on Interagon's Pattern Matching Chip. Tests were also to be run to both calculate the speedup of the PWM solver by itself, using synthetic data, as well as the overall speedup of the algorithms using the PWM, using real data sets. Unfortunately, this goal was never fully reached due to problems with the implementation, described in detail in Section 2.3. However, due to the predictable nature of the implementation, some performance estimations are given where empirical performance tests of PWM software implementations are compared to the realistic theoretical performance of the hardware

implementation.

1.1 The Cray XD1 Supercomputer

The implementation in this project was done on a Cray XD1 Supercomputer, which is largely a general-purpose computer, except for certain special hardware used for reconfigurable computing. As the underlying hardware of this computer was described in detail in an earlier project ([7]), this paper will only give a short overview of its technical capabilities in lieu of a full treatise.

The Cray XD1 installation at NTNU consists of a single Cray XD1 chassis that contains a total of six nodes. Each node is physically seated on its own *compute blade*, and consists of two single-core 64-bit AMD Opteron 250 Series general-purpose CPUs and 4 GB RAM shared between the two CPUs in a SMP configuration, as well as a Xilinx Virtex-II Pro XC2VP50-7 FPGA[1] connected to the SMP using Cray's proprietary RapidArray transport. The compute nodes are connected through the so-called RapidArray fabric on the main board, which also has facilities for connecting several chassis together. This configuration is shown in Figure 1 [2].

On the software side, each node runs a separate instance of the SuSE GNU/Linux operating system, independent of the other nodes. There is no memory sharing between the nodes, and all communication is done through the aforementioned RapidArray fabric. Several libraries optimized for inter-node parallelization on the Cray XD1 are provided, of which the most important one is Message Passing Interface (MPI). These libraries take advantage of many interesting features of the RapidArray transport, such as the possibility to directly write to an SMP's RAM, bypassing the OS kernel.

The most interesting feature of the Cray XD1, and the one which is the focus of this paper, is the possibility of using the attached FPGAs for application acceleration. A more general description of the technology behind FPGAs is described below, in Section 1.2. Using these, it is possible to create a design specialized for highly parallelized pattern discovery, which is then uploaded to the FPGA and executed there as a circuit. This design, in conjunction with a controlling C application, would be able to exploit the lack of data dependency inherent in such applications, significantly increasing the performance compared to similar

[1] More recent releases of this hardware platform used the larger Virtex-4 family.

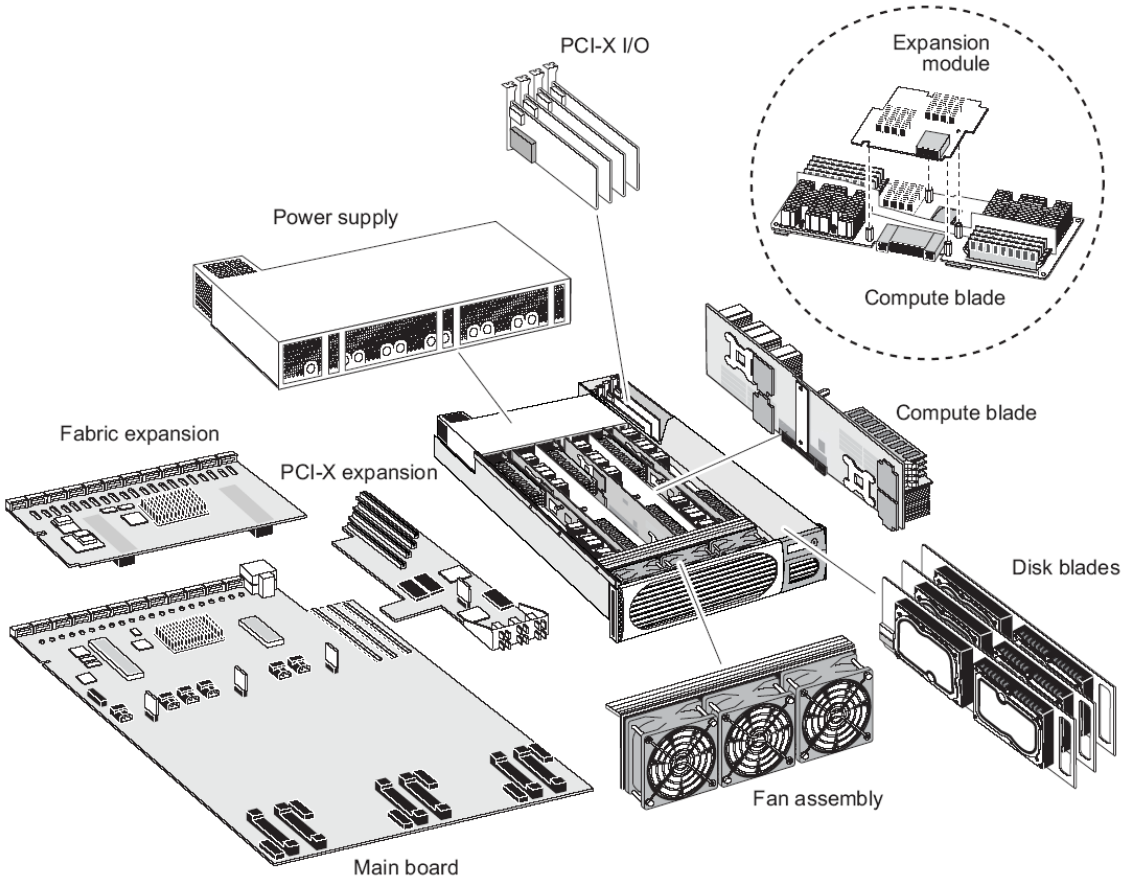

Figure 1: The Cray XD1 System.

implementations running on general-purpose sequential processors. Figure 2 shows the FPGA together with its various communication facilities [2].

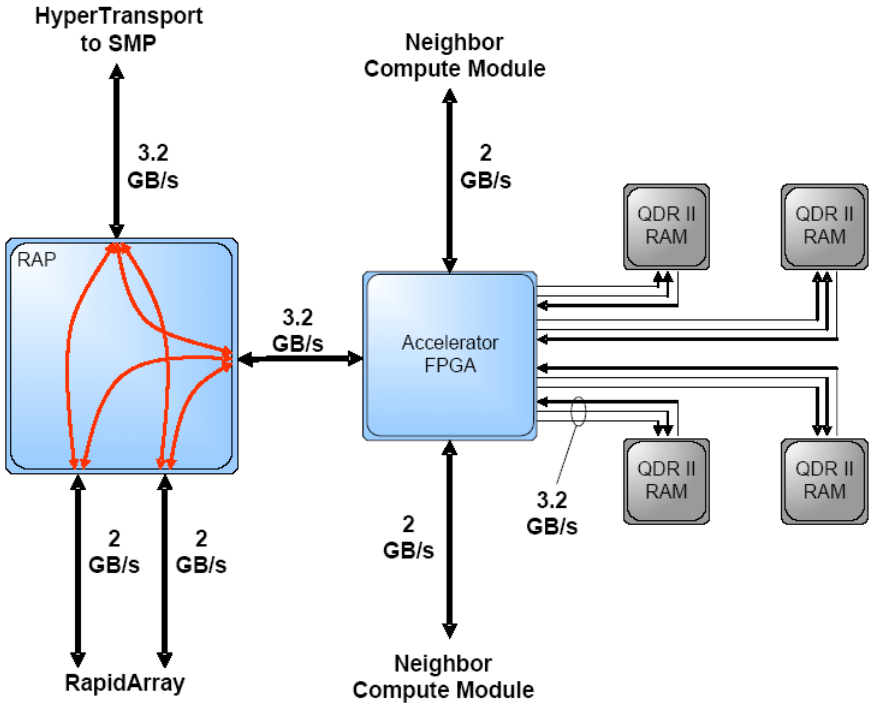

Figure 2: The FPGA's interconnection facilities.

1.2 Field-Programmable Gate Arrays

A *Field-Programmable Gate Array* (FPGA) is a reconfigurable circuit that can be loaded with a design specification, which is form of user-specified program, to perform a given task. The design specification is typically given in the form of a *bit file*, which contains a representation of each individual programmable bit in the FPGA. When the bit file is loaded to the FPGA, this configures the various *Configurable Logic Blocks* (CLBs) in the FPGA, as well as the routing matrix that connects the CLBs, to perform the functionality specified by the design.

The design also describes the way in which the CLBs are connected to the *Input/Output Blocks* (IOBs) located at the FPGA's boundary, which are hooked up to the physical pin-outs on the FPGA's exterior, allowing it to be interconnected with other circuitry.

This circuitry could for instance be other digital or digital/analogue hardware components controlled by the FPGA, other FPGAs that cooperate in solving a task, or in the case of the Cray XD1, a RapidArray processor that connects it to the host node.

Traditionally, FPGAs have been popular as a use for a prototyping tool for *Application Specific Integrated Circuits* (ASICs). Generally, many of the same design rules apply for these two technologies. However, the major difference is that moving an ASIC design to silicon is a very expensive process in low quantities, while FPGAs, due to their reprogrammability, can in many cases easily be used in place of an ASIC. If put in a circuit by means of a socket, replacing and reprogramming the FPGA is effortless, and offers large cost savings over using ASICs for prototyping directly.

However, the focus of this paper lies in the use of FPGAs in conjunction with traditional general-purpose CPUs for increasing the performance of scientific applications. It is not strictly a recent idea, and papers describing ways of combining general-purpose processors and programmable logic systems appeared as early as 1963 ([8], [9], [10]). However, the necessary hardware for doing this efficiently on a large scale has only appeared in the last few years.

Most of the early platforms consisted of an extension card that was hosted by a standard PC, and connected to its host computer through a traditional PC interconnect such as the PCI bus. The drawbacks of this approach should be obvious. Firstly, the PCI bus has a relatively high latency. Secondly, the bandwidth is often limited, with conventional PCI being limited to 133 MB/s. Thirdly, the bus is shared with other devices, meaning that contention can cause a much lower rate than the theoretical maximum. The last point is particularly important if there is a requirement to host several PCI-based FPGA cards in the same computer.

A more viable alternative to using the PCI bus for communication between the FPGA and its host node is to use a dedicated bus, which in latency and bandwidth terms is closer to the CPU than devices connected through PCI. The solution chosen on the Cray XD1, where the FPGA is situated directly on the compute blade, has the advantage of being connected directly to the CPUs through the HyperTransport bus, in the form of the RapidArray transport. This makes for a low latency and high bandwidth interconnect, enabling the FPGA to be used both for applications that require fine-grained parallelism with regard to the SMP node, and for applications that have high bandwidth demands.

[11] lists a number of performance advantages of applying reconfigurable computing to scientific applications. The most important and also most obvious of these is that the resources of the hardware can be applied where it is most needed, instead of using the static arrangement found in general-purpose CPUs. Components and datapaths can be defined, created and arranged to exploit task-specific parallelism in ways impossible to attain otherwise. The main reason for this is that general-purpose CPUs are created to be as fast as possible over a large number of applications, which means that much of the CPU is idle during any given operation, while a reconfigurable platform can be made with a very high resource utilization for one particular application. Another point to make is that a large part of a CPU's silicon is used by control logic, such as branch predicion and systems required to deal efficiently with virtual memory, as well as other units used to boost general performance such as memory caches. By having a core specially designed for a particular application, all unnecessary logic can be thrown out, making the actual work-performing logic a much larger percentage than on a general-purpose CPU.

There are of course drawbacks. The complexity of creating a design for a reconfigurable platform is close or equal to that of any other hardware design, and is generally done using relatively low level languages such as VHDL and Verilog. The time required for creating a design of any complexity is therefore very high. The high development cost is the main reason reconfigurable computing is not more wide-spread than it is, but certain signs, such as hardware support from major supercomputing companies such as Cray and SGI, indicate that it could become more common. Certain developments in using software development tools for hardware development, such as the C-based SystemC and Impulse C, should also make reconfigurable computing more available for software programmers. (These tools are not covered further in this paper, which is based on VHDL for the hardware development.)

A more comprehensive treatise on FPGAs, including an overview of some projects that have successfully used FPGAs for application acceleration, can be found in [7].

1.3 The Significance of In Silico Motif Discovery

Motif discovery is an important part of what is considered bioinformatics, and is an active field of research. Its main use is to discover interactions between transcription factors in genetic sequences in order to pave the way for the discovery of so-called promoter regions, or in other words, locate the parts of a genome that are likely to control a particular aspect

of a living entity. This is done in order to chart factors that affect everything from the color of a person's hair to diseases leading from genetic defects.

In the earlier days of biological research, most of the work was manual. However, as the data available on the various genomes increased, manual comparisons of different data sets became excessively hard to perform. Therefore, scientists started work on using computers for pattern discovery in the late 70s and early 80s. [3] gives an overview of the work that has been done in this field in the past three decades. Today, the vast amount of data gathered from genomes of various species are far beyond what can be efficiently examined by non-computational means. The computational complexity of many of the problems are in fact so vast that they in their full scale only be solved in a reasonable amount of time by means of supercomputers or massively distributed efforts [2].

[4] mentions three overlapping categories of transcription: "identification of properties associated with regulatory sequences, construction and analysis of quantitative models for the binding to DNA of individual [transcription factors], and the identification of combinations of transcription factor binding sites likely to be associated with regulatory processes." The focus here is on the second of these categories, which is considered the most basic level. The discovery of individual transcription factor bindings is a necessary building block for the other two, and used together these methods can discover exactly what combinations of which transcription factors cause a particular attribute of an organism.

The solutions in this paper are not however particular to bioinformatics, and the development and analysis of the solutions are done on a lower level, corresponding to a form of string matching. The biological background is therefore not very important in this context, and will not be discussed outside this introduction.

1.4 The PWM Algorithm

The introduction of Position Weight Matrices is usually credited to a paper by Rodger Staden in 1983 [5], as an in silico method for locating signals in nucleic acid sequences, according to the article including "ribosome binding sites, promoter sequences and splice junctions". Since its introduction, it has become an essential part of a large number of

[2] An somewhat related example of using distributed computing in bioinformatics is Rosetta@home, which harnesses the power of tens of thousands of personal computers to predict the folded three-dimensional shape of proteins [1].

motif discovery methods. [6] lists 119 publicized motif discovery methods, of which 59 use a PWM as the cornerstone of the method.

In its most common form, a PWM consists of a matrix of position weights of log-likelihoods, representing the value (likelihood in log form) of each element (nucleotide) for a given position in the window. The matrix has a length equal to the length of the motif it is to be compared against, and a height equal to the number of distinct elements in the data sequence (typically 4, for A T G C). A PWM score is defined as $\sum_{j=1}^{N} m_{i(j),j}$ where $i(j)$ is the score for a given element j and N is the number of columns in the PWM. That is, for a given window in the data sequence, the PWM score is calculated by looking up the score for each element in the window, and adding the individual element scores together. The resulting set of indexed scores, usually called the profile, is the final product of the algorithm [5].

Figure 3 illustrates two consequtive steps of the PWM algorithm[3]. In the first step, the window shown at top contains the elements in the sequence stream. The scores of these elements are looked up in the PWM, and each individual score is added together to provide the window score, shown at the bottom. In the second step, the window has shifted one step to the right, discarding the 'A' corresponding to the lowest index presently in the window, and bringing in a new 'G'. This process is repeated until one window score has been calculated for each possible window.

Variations of the basic PWM do exist, and the algorithm has been extended to fit several more specialized applications. However, as we here only deal with the basic log-likelihood version, this is outside the scope of this paper.

1.5 Earlier work in this area

Using FPGAs in bioinformatics is not entirely new, and several papers have been submitted in the area over the last few years. For example, [13] describes a method where FPGAs are used to compute sequence alignments by means of a Smith-Waterman dynamic programming matrix, where a single FPGA achieves a speedup of between 45 and 50 on a Xilinx Virtex-II XC2V6000 FPGA, compared to running it on a Pentium IV 3GHz CPU.

[3]Log-likelihood PWM scores, where each value is the logarithm of a value between 0 and 1 inclusively, are actually negative, but this is not significant to the algorithm, so the negative signs are dropped throughout this paper.

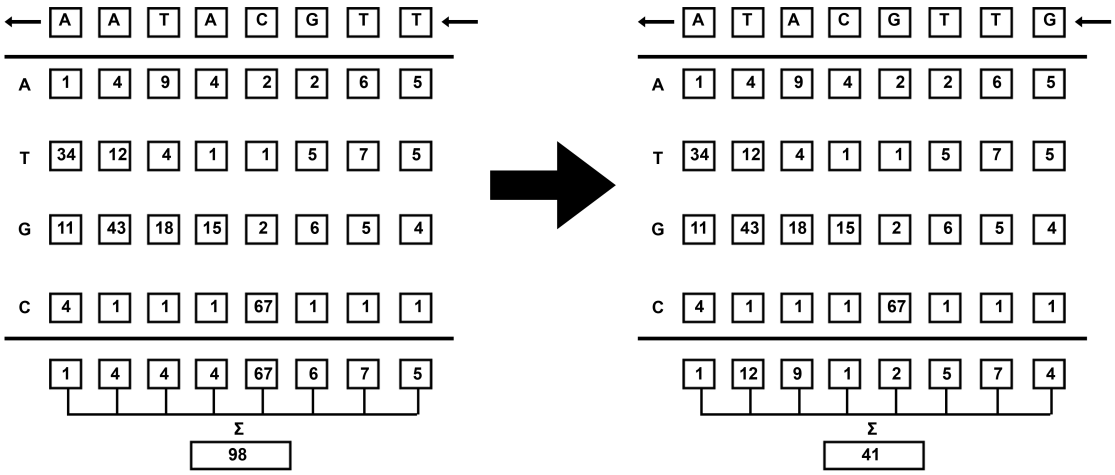

Figure 3: Two steps of the PWM algorithm.

[12] describes an implementation for discovering similarities within proteomes, called PROSIDIS (PROtein SImilarity DIScovery), where a speedup of 5 using a Xilinx Virtex XV1000 FPGA was attained, compared to a Pentium III 1GHz CPU.

However, no mentions on the specific task of using FPGAs for PWM matching have been found during the preliminary studies of this project, so it is assumed that this work is either novel, or that any papers mentioning the subject have either not been submitted, or have not been indexed by relevant keywords in any of the major article databases used in the search[4].

[4]This include all Computer Science and Medical databases available though UBiT.

2 The FPWM Prototype

An FPGA-based PWM implementation prototype dubbed the *FPWM* was developed as a way to gain real-life intimacy with the FPGA platform and get a base for future work. The implementation accepts an element string from the SMP node and stores it in the local SRAM of the FPGA, as well as a score matrix which is stored in the FPGA itself. The elements are then read from the SRAM into a window buffer in the FPGA, after which the scores are looked up in the score matrix, and added together by a parallelized and pipelined adder. The results from the adder are then stored in a queue that saves the eight highest-scoring results, which are written back to the SMP when the operation is completed.

2.1 The Hardware Implementation

The hardware implementation of the FPWM is relatively simple compared to more general processor cores. The implementation is modular, to make it possible to, for instance, replace the current result queue with an implementation using a larger queue and/or an alternative algorithm without altering the other modules, or increase the width of the PWM without affecting the memory and result queue modules. The modules are organized in a pipeline structure, where each module pass a self-containing package of data, usually consisting of one or more data elements as well as an index, to the module below it in the hierarchy. Several of the modules also employ a pipeline structure internally, to enable the FPWM to run at the maximum speed of the FPGAs installed on the Cray XD1. The different modules are described in greater detail below.

Together, the various modules perform a large number of elementary operations in parallel every cycle when the pipeline has been filled, enabling the FPWM to process one element of the input string each cycle. Due to memory bandwidth limitations on the Cray XD-1, an ideal implementation could process up to 32 elements each cycle using 8-bit elements, or 64 elements using 4-bit elements, but size constraints in contemporary FPGAs would likely prevent an implementation on the hardware used in this project to exceed roughly 8 elements/cycle with an implementation similar to the one used here. These numbers exclude pipeline warm-up effects, which add insignificant delays for reasonably large sequence lengths (see Section 4.1).

The pipelines in the FPWM do not suffer many of the problems pipelines in contemporary general CPUs do [5]. The main reason for this is the lack of hazards inherent in the algorithm. A computation is never dependent on the results of any other computation except for those directly leading up to it in the pipeline, meaning that no data hazards ever occur. Similarly, there is no significant dynamic control in the pipeline, nor any possibility or need to change the execution flow, so there are no branch hazards. For these reasons, the pipeline will never have to be stalled or flushed, and with the exception of the pipeline warm-up at the beginning of a computation sequence and the pipeline cooldown at its end, there is no penalty involved in using a pipeline here except for the use of some additional hardware resources and a comparatively higher implementation complexity.

Figure 4 shows an overview of the entire system, and indicates the interconnections between the various devices. The SMP node and the FPGA are displayed connected through the RapidArray fabric, and we can see the User Application part (here dubbed the *FPWM Core*) being connected to the rest of the system through Cray's proprietary RT and SRAM cores.

2.1.1 The Control Module

The Control Module is responsible for coordinating the operation of the other elements. As much of the coordination happens directly between the other modules themselves, it is not very involved in the actual computation. What it does do is receive and interpret the commands from the SMP node (through the RT Core), and pass them along to other modules as needed, such as when the element sequence is loaded to the SRAM and when the PWM is initialized. It is also responsible for starting the FPWM's computation sequence, maintaining the element index used by the memory module, and telling the result queue to start writing the result data back to the SMP node when the computation sequence has completed.

The implementation of the Control Module is based around a simple five-state FSM: Idle, Active, Cooldown, Output and Reset. These is also an independent system for commu-

[5] As a case in point, the Netburst architecture used in the Intel Pentium IV Prescott core used no less than 31 pipeline stages, making for an extreme branch misprediction penalty and intolerable power requirements. The core was deemed a failure that never fulfilled its high expectations, as it was supposed to breach the 10GHz barrier but never made it past 4GHz, and further development on the core was abandoned.

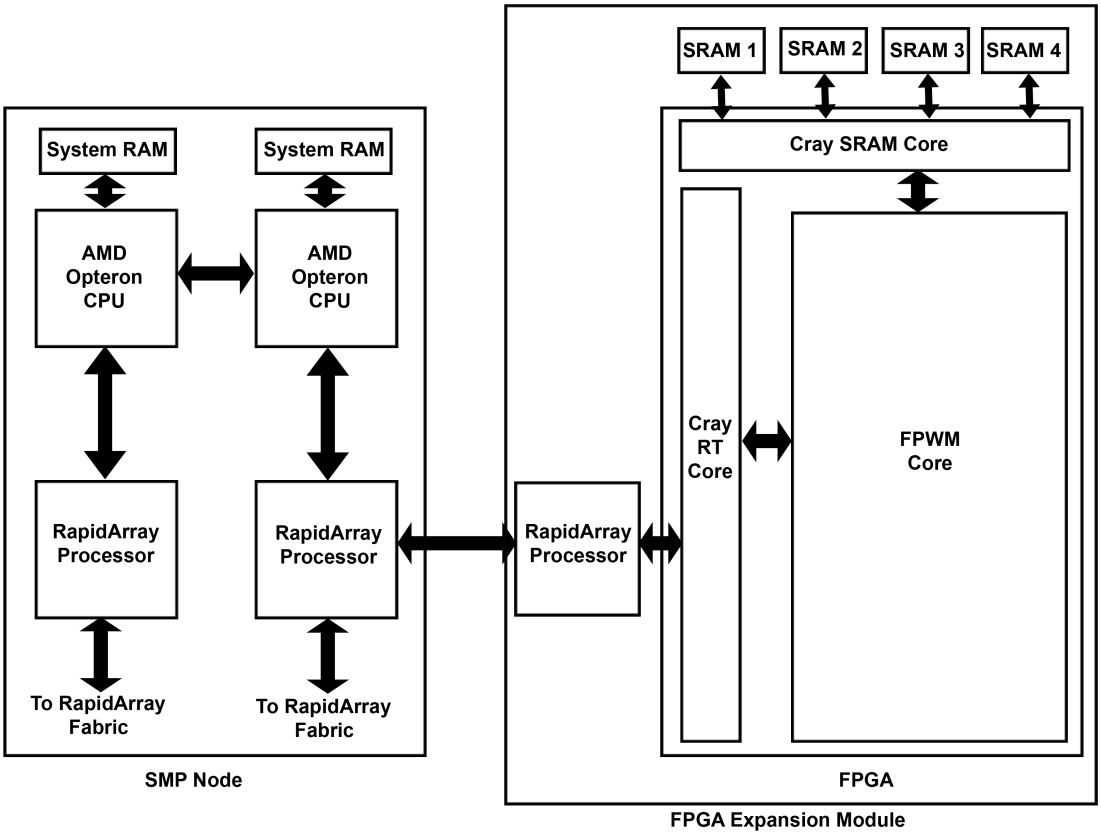

Figure 4: An overview illustration of the entire system.

nication with the host SMP node, so it is possible to write data to the FPWM while computations take place. In short, control signals for the pipeline and execution flow is provided by the first system, while control signals for writing to the SRAM and PWM is provided by the second system.

While in the Idle state, the Control Module does not read from memory, the index counter is not incremented, and the Output Module is signaled to not output any data. While in this state, the SMP node is expected to set the element stream and PWM values, the start and end offset of the element sequence in the SRAM on which the computation is to be performed, as well as a return pointer to a location in the host SMP node's memory space, to which the FPWM can write the results when the computation sequence is complete. As soon as the module receives a particular signal from the host SMP node, namely a write of 0x01 to 0x78 in the FPGA's internal address space, it switches from the Idle state to the Active state.

In the Active state, the index counter, which is initialized by the host SMP node during the Idle state as the pointer to the memory start offset, is incremented by one each cycle, and used as an address pointer for the Memory Module. The read signal to the Memory Module is also asserted, and after a memory fetch delay of eight cycles the Memory Module starts outputting the element stream in its output bus, one at a time. Note that no other control signals are asserted during the Active stage, as the PWM Module and Adder Module are controlled only by the data and index they receive, while the Result Module does not need any commands at this time.

When the index counter reaches the memory end offset set earlier by the SMP node, the FSM enters the Cooldown stage, and the read signal to the Memory Module is deasserted. The Control Module then waits for the last elements in the pipeline to clear. The wait period is equal to the number of addition steps, plus the memory fetch delay, plus three cycles for PWM lookup and result sort delays.

After the Cooldown stage is complete, the FSM enters the Output stage. In this stage, the Result Module is signaled to start outputting the signal. The Control Module then enters the Reset stage on the next cycle while the Result Module finishes the output sequence, where certain signals are reset to prepare for a new compute cycle. After this stage, the module finally returns back to the Idle stage, and can again accept work from the host SMP node.

Figure 5 shows the FPWM core with its various modules, and indicates the amount and direction of data flowing between the various modules.

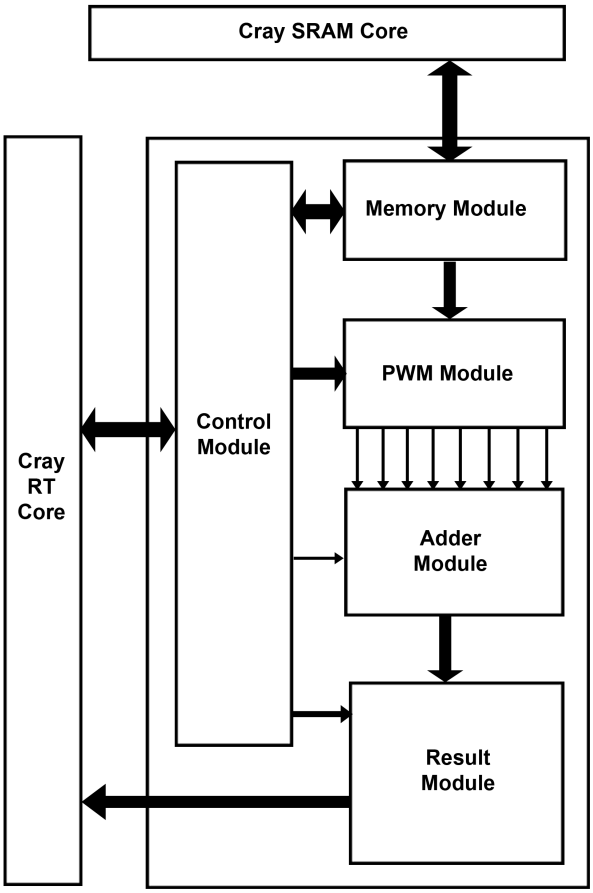

Figure 5: An overview illustration of the entire FPGA, including the various modules of the FPWM, and the FPGA's connections to the outside world.

2.1.2 The Memory Module

The Memory Module is responsible for handling the communication between the FPWM and the FPGA's off-chip SRAM. It has two major functions: writing the element sequence from the Control Module to the SRAM during initialization, and reading elements back from the SRAM and feeding them as a stream to the PWM, together with the element indexes, during computation. These two processes are functionally separate to allow interleaving, and it is up to the Control Module to tell the Memory Module what action(s) it should perform when.

The memory write functionality of the Memory Module is designed to accept data from the SMP node via the Control Module, and write them to the FPGA's off-chip SRAM. This functionality is rather simple, as the Control Module simply forwards the local address part of the memory address together with the data received, and asserts a write signal to the Memory Module. The Memory Module then uses the SRAM Core provided by Cray to write this data to the given position in the external SRAM.

The memory read function's main responsibility is matching up the indexes it is given by the Control Module with the data elements it fetches from the SRAM. It does this by asserting the address bus with the index when it's retrieved from the Control Module, while at the same time inserting the index into an eight-step shift register to match the eight-cycle memory fetch delay from the SRAM. After this delay, the index/element pair is passed on to the PWM Module.

2.1.3 The PWM Module

The PWM Module, together with the Adder Module, is the actual implementation of the PWM algorithm. Its primary function is taking a stream of elements from the Memory Module, and looking up the scores in the score matrix for every element in each window. The set of scores for each window, the sum of which is hereby referred to as the *window score*, is passed to the Adder Module along with the smallest index of the elements in the window (i.e., the left-most index), hereby referred to as the *window index*.

The PWM itself can consist of a virtually arbitrary number of rows and columns (subject to the restrictions placed by the total resources available on the FPGA), but is fixed to eight rows and four columns in the prototype. What this means is that it can look for sequences with a length of up to eight elements, using an alphabet size of four. Each matrix element itself is an integer, set to 32 bits in the prototype, denoting the value of a particular element in a particular position.

After the pipeline warm-up, when the element window buffer in the PWM is filled with a number of elements equal to the length of the PWM, the PWM Module simultaneously looks up the integer score for each of the elements in the window, thus processing a full window of elements in just a single cycle. The resulting window score is then passed on to the Adder Module, along with the window index.

Figure 6 shows the current PWM module with a window size of eight and alphabet size of four. Note that the selection lines are only displayed for the first column.

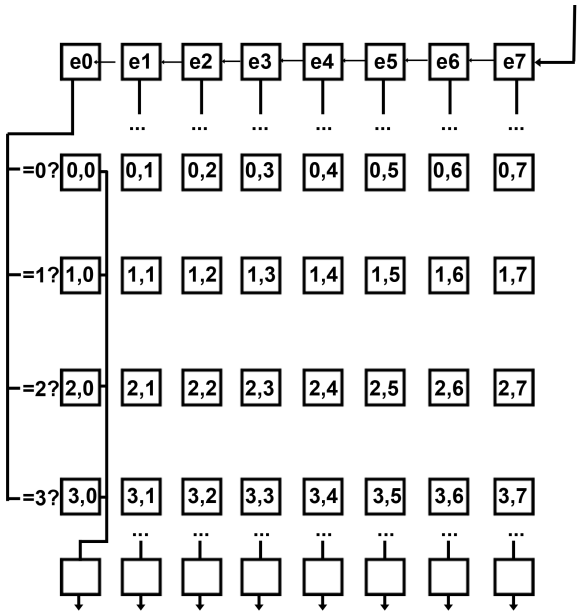

Figure 6: An 8/4 PWM Module

2.1.4 The Adder Module

The Adder Module, shown in Figure 7, is the final stage of the actual PWM algorithm, and is responsible for finalizing the window score by adding together the set of element scores produced by the PWM Module, before providing the sum to the Result Module along with the window index.

The prototype implementation does this using a set of 38-bit[6] adders in a pipelined tree configuration, capable of creating one window score each cycle with a delay of $log_2 pwm\_length$ cycles. Note that there is a pipeline stage built into each level of the adder tree, which is the reason for the delay. The initial output from the PWM Module is first summed together in pairs using $pwm\_length/2$ adders, producing $pwm\_length/2$ partial sums. These sums and the window index are stored in a pipeline memory element. On the next cycle, these are passed on to the next $pwm\_length/4$ adders to create $pwm\_length/4$ partial sums, and

[6]While each adder is defined as 38 bits, more than what is needed for all but the last level, the excessive bits on each stage are automatically removed by the synthesizer's optimization process.

so on until a single sum has been computed.

When the result score has been reduced to a single number, the actual PWM algorithm is complete for that window, and the window score together with its index is passed to the Result Module to determine whether that particular score will be stored or discarded.

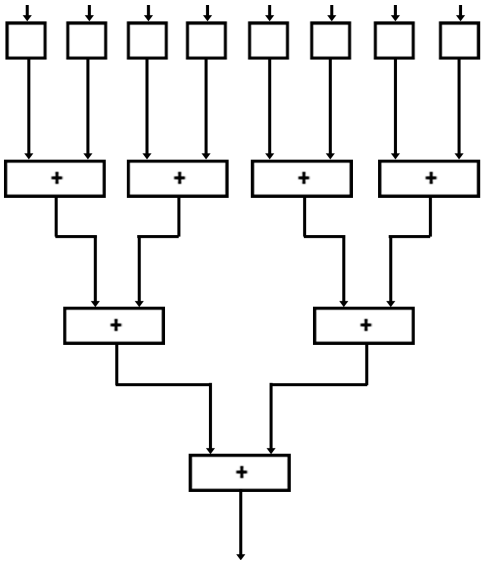

Figure 7: A full eight-element Adder Module.

2.1.5 The Result Module

The Result Module is responsible for returning the results of the computation to the SMP node. Various approaches can be used in this module, both in the number of results returned and in the different levels of post-processing that can be performed. The prototype implementation uses a sorting mechanism that maintains a sorted array of result-index pairs, capable of accepting one new result from the Adder each cycle. When the computation is complete, the Control Module sends it a certain signal that initiates a write of the content of the result queue to a predetermined memory location on the SMP node itself.

The prototype Result Module is based around a number of fairly complex Storage Elements, each of which can store a single result-index pair. The prototype implementation has eight of these Storage Elements, interconnected in a chain. The Storage Elements work by simultaneously comparing the result value it has stored with the result value provided by the Adder Module each cycle, and combining it with data from the neighboring Storage

Elements to do one of three actions: (1) Take the result-index pair from the next-higher Storage Element as its own, (2) Insert the new result-index pair as its own, or (3) Keep its current value.

The actual algorithm is relatively simple, and can be expressed as the following:

```
(1) IF new_result > old_result AND next Storage Element in chain reports
new_result > old_result THEN
Replace current result-index pair with result-index pair from next Element

(2) ELSIF new_value > old_value AND next Storage Element in chain reports
new_result <= old_result THEN
Replace current result-index pair with the newly arrived result-index pair

(3) ELSE
Do nothing
```

The Result Queue as a whole can accept and sort one result per cycle, but sorting each element must be performed over two cycles, as the Storage Elements must exchange information to determine the correct course of action for each given result. In the first cycle, each Storage Element determines whether it will have its contents replaced during the NEXT cycle, that is, if the new result is larger than the currently stored result OR larger than the result being written to it THIS cycle, if any. This comparison is then propagated from each Storage Element to the previous Storage Element in the chain. In cycle two, the Storage Elements will then perform action (1) if the next Storage Element in the chain is to be replaced, (2) if the next Storage Element in the chain is not replaced but the new result is found to be larger than the current result stored in this Storage Element, and (3) otherwise.

Note that these two sort cycles happen concurrently, and in all of the Storage Elements simultaneously. Which action is taken on an element on a given cycle is determined by the data generated from the next Storage Element in the chain in the previous cycle, thus creating a rather large and complex unit.

Special attention also has to be given to the first and last Storage Element in the chain. For the last Storage Element, it is a given that if it is replaced, its result-index pair is

not forwarded anywhere, and is thus discarded. For the first Storage Element, there is no higher-order Storage Element to communicate with, so this Element can be somewhat simplified, as the only actions that can occur are (2) and (3).

Figure 8 shows the internals of the Storage Element, while a simplified, four-element Result Module is illustrated in Figure 9. Note that certain control lines are not shown on the latter, to avoid excessive clutter.

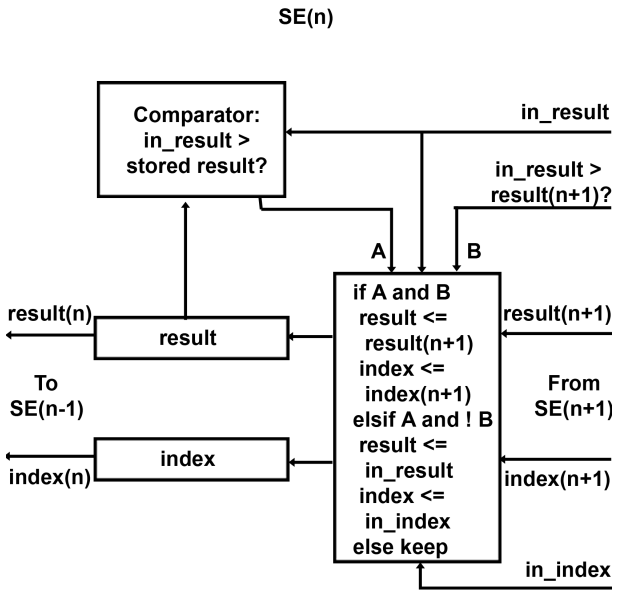

Figure 8: A Storage Element.

2.2 The Hardware-Software Interface

An intermediate C interface for use by client applications was developed in order to simplify control of the FPWM. Calling the functions in the interface indirectly invokes communication with the FPWM through the FPGA API provided by Cray, which transfers the actual data between the SMP node and the RT interface hosted on the FPGA itself, from where it can be accessed by the user design.

The interface consists of three major functions:

`int fpwm_init()` prepares the FPGA for operation, loads the FPWM design to the FPGA, and returns the needed file descriptor to the user application, which should be used to

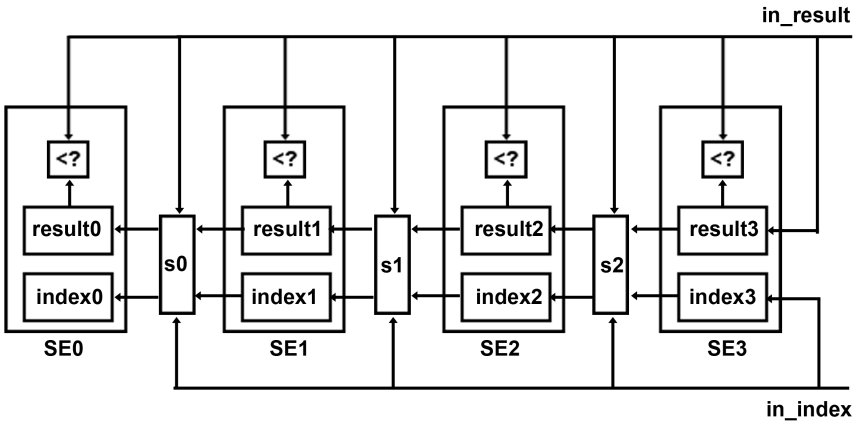

Figure 9: A simplified four-element Result Module.

reference the FPGA on future calls to the library.

`int* fpwm_load_dataset(int handle, int* mempointer, char* dataset, unsigned long offset, unsigned long length)` loads the FPGA's SRAM with the dataset that is to be used by the FPWM. Due to a limitation in the current design, the function will create a memory area used for writing to the FPWM the first time it is called (with mempointer set to NULL), which is returned to the user application as a pointer. The interface expects to be given this pointer as an argument on future calls to this function. Note that char* dataset should be a pointer to the first byte of the sequence to be loaded.

`offset` and `length` define which parts of the char array will be written to the FPGA, and where in the FPGA's SRAM it will be written. For example, if offset is given as 32 and length as 16, 16 bytes will be transferred starting from offset 32 in the char array to offset 32 in the FPGA's SRAM. It is the user application's responsibility to make sure these values are valid (not out-of-bounds), and that they do not overwrite already written data unless the overwrite is intentional.

(Note: It is technically possible to use Cray's API to write directly to the memory area given by the returned memory pointer to alter data on the FPGA, but this is not recommended as the underlying functionality of the interface may change in the future.)

`long* fpwm_compute_pwm(int handle, long* pwm, unsigned long range_start, unsigned long range_length)` is used to start the computation itself. The `pwm` argument

is a pointer to the first score value of the PWM, which must be the same size or be padded to equal the size `FPWM_PWM_MAX_ROWS * FPWM_PWM_MAX_COLS`, defined by the header file. The PWM should be stored in the standard C row-major order, and if called externally from a programming language using column-major order (such as FORTRAN) it must be translated by the user application, or simply stored as a flat array. The `range_start` and `range_length` arguments define the memory range in the FPGA's SRAM (that is, the element sequence) on which the computation is to take place.

It should be noted that since the prototype FPWM did not reach a fully working condition within the duration of the project, the interface remains largely untested.

At the current time, the interface is relatively low-level, and leaves much of the book keeping to the user application. Much of this is because the interface currently does not have an internal state, which means information must be stored by the user application instead of within the interface itself. This could be regarded as a weakness, as it exposes parts of the internals to the user application, and makes for a weaker encapsulation than what is usually desired. It also requires the user application to operate at a byte level when referring to the element sequence memory (datasets), while in many cases it could be desired to refer to it as named sets or similar instead.

The solution to these problems that has the highest degree of encapsulation would be to contain the interface in a C++ class, and store the data required for this functionality in this class instead of within the user application itself. For C++ applications, this would be the best solution, and one that should be explored in order to ease the implementation of the user application. Of course, a C++-contained class would be hard to use from a plain C application, so such an implementation might not always be desirable. In C, a struct could fill these demands in place of a class, and while such a solution still requires that the user application keeps track of a pointer to the struct, and the user application could still access and modify its data, it is a better solution than the simple approach used today as it offers a higher level of containment.

2.3 Implementation Problems

Due to the complexity of writing hardware specifications with VHDL, combined with the relative inexperience with such programming on behalf of the author and the unpolished nature of the hardware platform and tools used during the development, several major

problems occurred during development, greatly slowing down development and forcing some less than optimal solutions on the code side.

The one factor that caused the largest amount of problems and frustrations was the primary implementation tool, Xilinx ISE Foundation 7.1i. For one, the tool is very choosy about what language constructs it is willing to synthesize. Many of the array constructs that were attempted in the project were either not possible to synthesize, not possible to simulate, or both. This included several built-in standard functions in the IEEE 1164 library standard, of which the lack support made the code much more complex than what should have been necessary.

Worse still, the tool has a tendency to terminate during the synthesizing process for no apparent reason. Error messages such as

```
ERROR:DeviceResourceModel:1071 - NP_NODE::getwirewitharc failed.
FATAL_ERROR:Par:Portability/export/Port_Main.h:127:1.12.12.6 -
This application has discovered an exceptional condition from which it
cannot recover. Process will terminate. To resolve this error, please
consult the Answers Database and other online resources at
http://support.xilinx.com. If you need further assistance, please open
a Webcase by clicking on the "WebCase" link at http://support.xilinx.com
ERROR: XST failed
Process "Synthesize" did not complete.
```

and

```
Building and optimizing final netlist ...
FATAL_ERROR:Xst:Portability/export/Port_Main.h:127:1.13.276.1 -
This application has discovered an exceptional condition from which it
cannot recover. Process will terminate. To resolve this error, please
consult the Answers Database and other online resources at
http://support.xilinx.com. If you need further assistance, please open
a Webcase by clicking on the "WebCase" link at http://support.xilinx.com
ERROR: XST failed
Process "Synthesize" did not complete.
```

were all too common, and the provided resources generally provided very little information on how to correct them. Considering that these errors usually occurred during the *Place and Route* phase, roughly 30-40 minutes into the synthesizing process, and that they occasionally made the host computer lock up completely, much time was wasted dealing with them. To make matters even worse, in many cases the errors did not seem to be caused by any particular code lines. Often the error would go away after changing the order in which certain lines of recently added code was written (which, due to the parallel nature of VHDL generally does not have any syntactic or semantic significance except for within defined constructs), or simply by cleaning the project files (i.e., deleting all the files generated by ISE) and re-running the synthesizing process.

Another problem that slowed down the process was the lack of documentation available for the Cray XD1 platform. Certain subjects, such as the differing memory access schemes internally and externally of the FPGA (or more precisely, the important difference between byte addressing on a quad-word boundary and quad-word addressing), were very poorly described. Other functionality mentioned in the documentation, such as the use of the Xilinx RocketI/O interface for direct communication between FPGAs on neighboring nodes were not explained at all, despite being illustrated on diagrams such as Figure 2 [7].

Together, these problems and the large amount of debugging that had to be performed partially because of them slowed down development to such an extent that, while developing a codebase that simulated correctly by itself was done within the first month of the project, it was followed by a three-month debugging and troubleshooting process. Therefore, a working (but so far largely untested) version of the FPWM was only available four days before the project's deadline.

In summary, the goal of creating an FPGA-based PWM matcher was only partially attained, primarily due to the lack of time and hardware availability to test and benchmark the final version of the implementation. The prototype was at the end of the project working perfectly in simulations, but as the Cray XD1 platform was unavailable at the last stages of the project due to user account problems, there was no chance to test the final product. It follows that the desired empirical measurements of the performance of the FPWM, both by itself and in conjugation with the two other projects, were not obtained.

[7] Note that release 1.2 of the node software, which was used on the current Cray XD1 until the start of 2006, also had extreme stability problems, where the entire node would suddenly lock up and reboot when the FPGA was used, even using the reference designs from Cray itself [7]. These problems did however seem to have been corrected in the current 1.3.1 release.

Estimated values for these measurements are however provided below, in Section 4.1.

2.4 Summary

This section has presented the FPWM prototype along with its various modules, and described the inner workings of its implementation. A C interface for communicating from a C application running on the SMP node has also been discussed. Finally, several major problems encountered during the implementation process have been listed and described.

While the implementation at its current stage is not fully tested, and cannot be said to be operative, the main codebase for the prototype has been developed and simulated correctly[8]. Work still remains on testing and improving the interface, as well as making it more encapsulated and thus easier to work with for application programmers. Work will continue on the FPWM prototype even after the end of this phase of the project, and will hopefully end up as a useful tool for bioinformatics.

[8]Current versions of the simulations can be found in the accompanying datafiles as test bench waveforms, as most are too large and complex to be rendered properly within this document. The largest test bench tracks 65 signals over 250 time steps, and would require a decent number of A3 pages laid side-to-side to make it legible in a paper format.

3 The FPWM Prototype Simulator

Because of the earlier mentioned problems with getting the prototype up and running on the provided hardware, a simulator dubbed FPWMSim was implemented in Java to more easily test the prototype implementation, and give a framework that could be used to measure the potential performance of module variations. The simulator also gives a complete visualization of the contents of all registers in the pipeline, along with additional information about which actions the virtual FPWM is currently performing.

3.1 The FPWMSim Implementation

FPWMSim consists of a number of base classes that provide the framework for the functionality:

- FPWMSim.java provides simulation control and the main drawing facilities, as well as the threading and control functionality. Its main responsibilities are initializing the other classes, as well as managing the order in which the modules fire, and the passing of communication packages between them.

- PrefManager.java is responsible for parsing input data from a plain text file containing configuration data, as well as the PWM scores and the element stream. The other classes have a reference to this class, and can request the various data as needed.

- FPWM_DataPackage.java is an abstract class representing the data exchanged between two modules. A derived class must be provided for each pair of modules that require intercommunication, containing the data fields required by the transaction.

- FPWM_Module.java is an abstract class representing a Module. All classes that are to act as modules in the simulator must be derived from this class.

One of the main goals of FPWMSim was to mimic the exact workings of the real FPWM. In order to do this, the virtual FPWM is implemented as a number of Modules that exchange data via a simple interface, just as in the FPGA-based FPWM implementation. In each cycle, an FPWM_Module is passed an FPWM_DataPackage from the module above it in the hierarchy, acts on the content of this data as well as its internal state, and forwards a new data package to the module below it.

The modules of the implementation are 1:1 equivalent to the modules in the FPWM, and consist of FPWM_Module_Adder.java, FPWM_Module_Mem.java, FPWM_Module_PWM.java and FPWM_Module_Result.java. FPWMSim.java takes the role of the Control Module as the actual setup process of the memory and PWM are not simulated (but the time needed for this setup is estimated). An additional module, FPWM_Module_Output.java, has also been added to display various statistics, control information, and the current output of the virtual FPWM.

Since the platforms used to implement the FPWM and its simulator are fundamentally different, the modules do not compute their state in the exact same way. However, the internal state of each register after a given cycle, including the precision of the processing, should still remain identical, except for the Result Module, which was somewhat simplified in the simulator due to time constraints. For all cases, the output still remains identical, and the simulation output properly indicates the most important functionality of the modules.

3.2 FPWMSim Data Files

FPWMSim operates on two data files, an input file defaulting to data.in and an output file defaulting to data.out. The data.in file is expected to contain two constructs, SEQ and PWM, describing the sequence and PWM, respectively. No more than one sequence of each type should be present in the file, but if duplicate constructs are found, the last construct of that type will be used. A typical file would look like this:

```
SEQ,32
1,0,2,3,1,2,3,3,2,1,0,0,0,0,3,1,2,3,1,2,2,2,1,2,1,1,0,3,3,3,2,1
/SEQ

PWM,8,4
32, 12,  4,  1, 24,  3, 89,  4,
 5, 43,  1, 23,  3, 12, 94,  4,
32, 12,  4,  1, 24,  3, 89,  7,
 5, 43,  1, 23,  3, 12, 94,  7
/PWM
```

`SEQ,32` denotes the start of a sequence construct with 32 elements. The elements are expected to start on the next line, and are a comma-separated list of alphabet indexes. Line breaks in this sequence is allowed, and can be used to improve readability if the file is manually created. `/SEQ` on a line by itself denotes the end of the sequence construct.

`PWM,8,4` denotes the start of a PWM construct with 8 columns and 4 rows (i.e., a PWM of length 8 with an alphabet size of 4). The elements are expected to start on the next line, as a comma-separated list of PWM score values given in a row-major order (C-style, which is stored in a row-by-row basis, as opposed to column-major order as is used in FORTRAN). Line breaks can be used to improve readability as in the example, but they are not required. `/PWM` on a line by itself denotes the end of the PWM construct.

A line can be commented out by starting it with either of the following characters:

`# ! // -`

In-line comments are not recognized and must be avoided.

The output is given as a list of Index-Result pairs, with one pair on each line, separated by a comma.

3.3 Using FPWMSim

Invoking FPWMSim with the default settings is done through the command `javaw FPWMSim` . It defaults to reading data from the file data.in and writing output to data.out, but this can be changed with the `-indata` *`filename`* and `-outdata` *`filename`* directives on launch.

When the program has loaded, it will start in the PAUSED state. To control the simulation, use the arrow keys on the keyboard in the following fashion:

Arrow Key Left: Switch between the PAUSED and RUNNING simulation states.

Arrow Key Right: Skip one cycle while in the PAUSED state. This button has no effect in the RUNNING state.

Arrow Key Up/Down: Increase and decrease the simulation speed used in the RUNNING state.

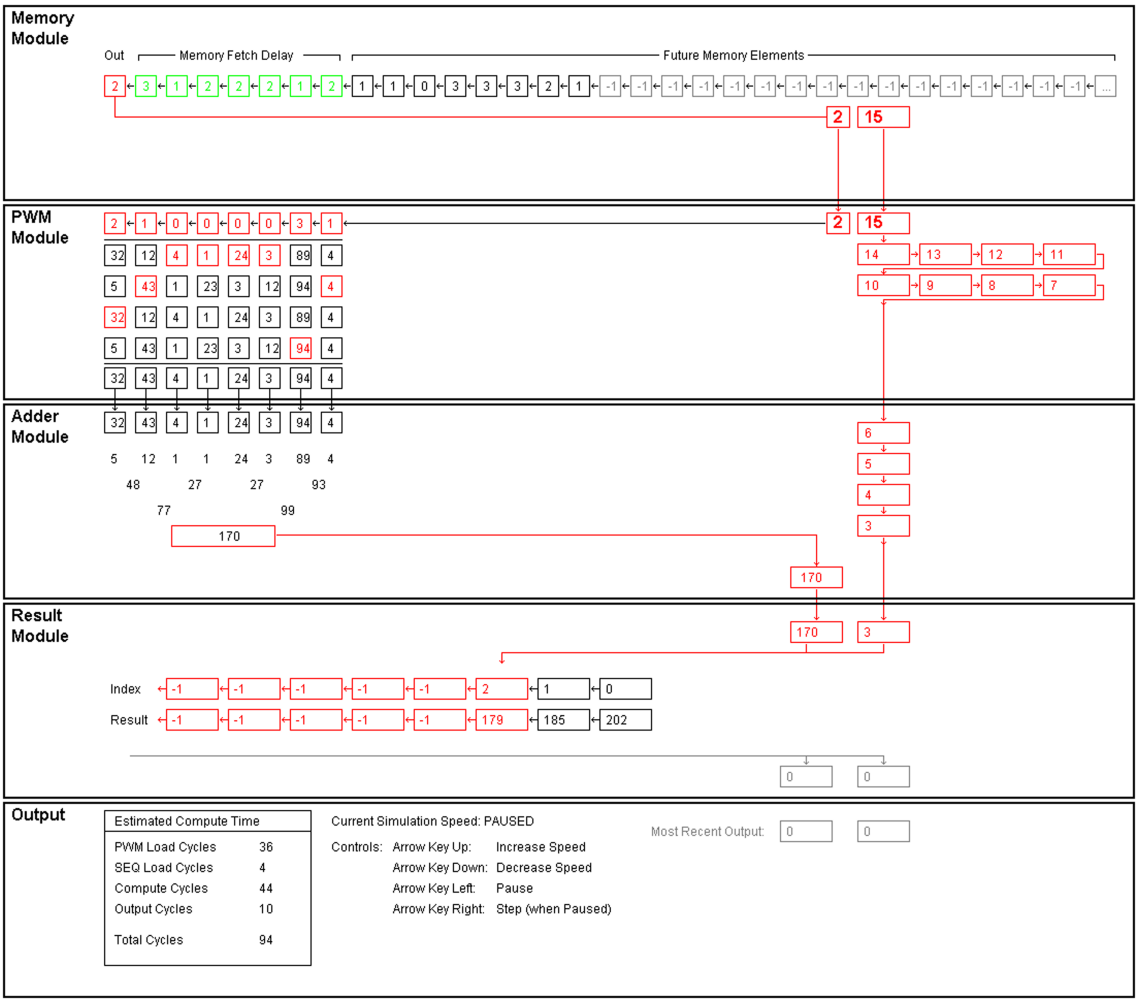

Figure 10: FPWMSim during operation.

While in the PAUSED state, the simulation progress will not proceed automatically, and is controlled by the user with the right arrow key. While in the RUNNING state, the simulation will proceed at a rate specified by the user using the up and down arrow keys. Figure 10 shows FPWMSim after completing 25 steps of the default test run.

Note that the simulator is intended as a tool to experiment with various FPWM configurations, and to visualize its operations, and therefore does not currently have a non-interactive mode. It is not optimized as a PWM solver, and would be grossly inefficient for this usage, so this functionality will likely not be required, as much more powerful software-based PWM solvers exist.

3.4 Summary

This section has presented a simulator framework for the FPWM that enables a developer to visualize the inner workings of the chip, and to a certain extent measure the effect any changes have on the output (although this instrumentation remains somewhat sketchy). While it still has some drawbacks, such as the lack of general drawing facilities (i.e., each module is responsible for drawing itself from scratch), testing a module written in Java on the simulator before doing a full-blown VHDL implementation should prove an efficient way to avoid wasting time on inferior solutions, provided the solution cannot be modelled exactly. This is mainly because Java is a much more comfortable language to work in than VHDL, but also because the changes resulting from the implementation become readily visualized, making it easier to gain an overview of the current state of the implementation.

4 Results

While a working FPGA-based PWM matcher was created within the time available for this project, some of the desired measurements were not ready in time to be included in this paper, and most of the actual data from the different implementation techniques that were to be explored has not been obtained. However, as earlier mentioned, due the nature of the FPWM prototype it is easy to measure its performance theoretically. These figures are provided below, in Section 4.1. Section 4.2 provides the data obtained as well as some estimated values for resource consumption given different configurations, while Section 5 outlines the work required to get the rest of these data, as well as the required work needed to increase the performance by moving from a single-core, single-node implementation to a multi-core, multi-node implementation.

4.1 Performance Measurements

Due to the predictable nature of the prototype FPWM, it is effortless to accurately estimate the required processing time for a given work load. The total computation time can be split into *Sequence Load Time*, *PWM Load Time*, *Compute Time* and *Output Time*. For a result given in Cycles, using the constraints of the prototype, these can be computed as such:

$$
\begin{aligned}
C_{seqload} &= sequence\_length * (1/8) * (9/8) \\
C_{pwmload} &= pwm\_columns * pwm\_rows * (9/8) \\
C_{compute} &= sequence\_length + memory\_fetch\_delay + log_2(pwm\_columns) + 1 \\
C_{output} &= result\_queue\_length * (9/8) + 1
\end{aligned}
$$

For all steps involved in data transfer between the SMP and the FPWM, there is a factor (9/8) applied due to the burst mechanism in the RT interface. A total of eight quad-words can be transferred in one burst of eight cycles, followed by a one-cycle cooldown.

4.1.1 Compared to general-purpose CPUs

A standard PWM matching algorithm with a PWM length of 20 and alphabet size of 4 run on a Pentium M 1.8GHz computer, matching a 4MB dataset 100 times, was clocked to about 30 seconds, while matching a 0.5MB dataset 100 times was clocked to about 4

seconds. Matching speed therefore varied from about 12.5 MB/s to 13.5 MB/s on this particular platform[9]. Running the 4MB example on a theoretical FPWM would yield the following results:

$$
\begin{aligned}
C_{seqload} &= 4 * 2^{20} * (1/8) * (9/8) = 589'824\ cycles \\
C_{pwmload} &= 20 * 4 * (9/8) = 72\ cycles \\
C_{compute} &= 4 * 2^{20} + 8 + log_2 20 + 1 = 4'194'311\ cycles \\
C_{output} &= 8 * (9/8) + 1 = 9\ cycles \\
C_{total} &= 1 * C_{seqload} + 100 * (C_{pwmload} + C_{compute} + C_{output}) = 420'029'024\ cycles
\end{aligned}
$$

Running at 200MHz, this gives a total time of 2.1 seconds, or roughly 15x speedup, with a matching speed of 200 MB/s. Note that the sequence load time is only included once, since the same sequence is used in all 100 runs. Also, the time used for any computations outside the PWM matching itself, which is assumed to be very low if at all measurable even for the CPU example, is not included in this figure.

Even though a theoretical result such as this should be taken with a grain of salt, the FPWM is highly predictable in nature, so the only questionable parts of this estimation are the terms involving communication with the SMP. Since the communication between the SMP and the FPGA are somewhat abstracted in a memory mapping from the SMP's memory to the FPGA, and this is subject to factors such as the load on the memory bus, this figure cannot be established with 100% certainty.

As can be seen from the above equation, for a relatively large number of runs, the $C_{seqload}$ factor becomes negligible, accounting for about 0.1% of the total time in this example. Similarly, $C_{pwmload}$ and C_{output} are also negligible for runs on any reasonably large datasets. Under these assumptions, a reasonable estimate can therefore be calculated using $C_{compute}$ by itself for the current prototype. In fact, seeing as $C_{compute}$ itself only has one dominating term, using the length of the element sequence alone gives a decent estimate for the required match time.

In theory, the PWM matcher should be capable of a matching speed as high as 6.4 GB/s, which is the limit of the memory bandwidth on the Cray XD1. This speed, which could be attained using 32 parallel cores in a configuration such as the one described below in Section 5.2, would reduce the time needed for $C_{compute}$ to about 13'125'000 cycles (about 70 milliseconds), and increase the speedup to about 480x compared to the attained perfor-

[9]The reason the tests were not run on the AMD Opteron CPUs, which most likely would have performed somewhat better, was that the test tool for unknown reasons failed to compile on this platform.

mance on the CPU used in this test. However, due to the limitations on today's FPGAs, it is unlikely that one could fit that many cores on one single FPGA without making significant compromises regarding resolution and PWM length, so this number can currently only be considered a theoretical limit on the potential for this method.

However, it is important to keep Amdahl's law[10] in mind when making such statements. Numbers quoted to the author say that about 95% to 99% of the time spend in most of these algorithms is the PWM matching, while the rest is mostly spent on building new PWMs, meaning that the bottleneck at these speedups would move from the PWM matching to the PWM building, and prevent any total speedup for the algorithm as a whole to exceed 20x (at 95%) to 83x (at 99%). Therefore, as the speedup for the PWM matching itself increases, it becomes much more important that the user application itself is optimized in order to keep up with the increase in performance. Attempts to move parts or the whole of the remaining algorithm to the FPGA, provided it can be properly parallelized, should also be considered in this case.

4.1.2 Compared to Interagon's Pattern Matching Chip

The Pattern Matching Chip (PMC) is a special ASIC-based chip developed by Interagon, and is geared towards a more general usage in pattern-matching. Earlier empirical runs on the PMC using MEME on a half-megabyte dataset measured it at a maximum speedup of about 9x compared to a pure CPU when five PMC chips were used in parallel, while further increasing the number of parallel matchers created a bottleneck in the CPU, and caused so much overhead that the total time required for the run increased [15]. Unfortunately, as there was no time to do empirical comparisons head-to-head using real data, no hard conclusions can be made regarding the difference in performance between this chip and the FPWM, but judging from the numbers presented earlier in Section 4.1.1, even the current prototype should provide a higher level of performance, even discounting the additional processing involved in the MEME algorithm. This performance advantage is largely rooted in the specialized nature of the FPWM, as it is designed to do one single function very fast, while the PMC has a more general-purpose design.

[10] Amdahl's law provides a limit on the overall speedup attainable for an algorithm when one particular part of the algorithm is optimized. It is often formulated as $S = \frac{1}{(1-p)+s/p}$, where S is the total speedup, s is the speedup for the optimized part of the algorithm, and p is the proportion of the time originally spent in that part of the algorithm [14].

4.2 Resource Measurements

One of the goals of this paper was to chart the resource consumptions and performance implications of the prototype given changes to certain factors, such as input- and output resolution, PWM length, alphabet size and sequence length. This is given in the tables below. However, with the current state of the prototype, not all values could be easily determined, so some are extrapolated from current data, by increasing or decreasing the relative resource consumption with respect to the known changes that occur in each module. Extrapolated and estimated values are denoted by being enclosed in parentheses.

The prototype's resource consumption in slices for each module by itself is given in Table 1. Note that the total resource consumption is not a simple sum of the resource consumption of each module, as a certain amount of optimization and slice sharing takes place between the various modules [11]. The Total figure also includes the RT and SRAM cores. The maximum number of slices available for this particular FPGA model is 23'616.

Module	**Slices**
Control	239 slices
Memory	64 slices
PWM	1033 slices
Adder	273 slices
Result	653 slices
Total	1943 slices

Table 1: The resource consumptions of each module on the FPWM with the default configuration.

4.2.1 Bit Resolution

The input and output resolution for PWM values and output results can currently be changed easily simply by altering a constant in the package file (see Section 5.1.1). The resolution used has a relatively major impact on the resource consumption on the chip, but only a minor effect on the maximum speed attainable, which either way is above the

[11] A "slice" on the Virtex-II Pro XC2VP50-7 FPGA consists of two 4-input LUTs (Look-Up Tables) and two Slice Flip Flops. In many cases, a particular module will consume more LUTs than flip flops, or vice versa. The synthesizer will attempt to organize this in a fashion so that as much as possible of a particular slice is used, and functionality from LUT-heavy modules will therefore often partially share slices with flip flop-heavy modules.

maximum speed allowed by the underlying hardware (200 MHz). The output resolution has to be chosen so that the largest possible result can be stored, but superfluous bits will generally be optimized away by the synthesizing software, and is chosen to accommodate a maximum window size (PWM length) of 64 (6 adder levels). Results for changing the bit resolution while keeping all other factors static are displayed below in Table 2.

Bit Resolution	**Total Slices**	**Max Speed**
4/10	734 slices	220.495 MHz
8/14	905 slices	220.495 MHz
16/22	1223 slices	220.495 MHz
32/38	1943 slices	217.125 MHz

Table 2: The results of varying the bit resolution on the FPWM prototype.

4.2.2 PWM length

The PWM length is the largest factor in deciding the total resource consumption of the FPWM, as it not only will linearly grow the largest module on the chip (the PWM Module), but also linearly increase the number of required adders in the Adder Module. While seven adders are required for eight elements, sixteen elements require fifteen adders and thirty-two elements require thirty-one adders. It does not affect the maximum speed however, as these modules are built to scale effortlessly to an arbitrary number of elements. The changes in resource consumption for altering these values are shown below in Table 3.

PWM length	**Total Slices**
8	1943 slices
16	(3800 slices)
32	(5600 slices)

Table 3: The results of varying the PWM length on the FPWM prototype.

4.2.3 Alphabet size

The current prototype supports an alphabet size of up to 128 without making any major changes to the implementation. Increasing it will cause the size of the PWM Module to increase linearly, but unlike changing the PWM length, it does not require any changes to

the Adder Module. While it could decrease the maximum speed, as the required multiplexers for the PWM lookups will be larger, with higher latency, the current PWM can run at a contained speed of 311.444 MHz, so there is plenty of room for such an increase. Since these localized changes in speed did not affect the overall speed of the implementation, this has therefore not been included. The effect on resource consumption of altering the alphabet size is shown in Table 4.

The reason for the large increases in resource consumption when the score matrix is increased is that it at the current default level of 8/4/32 bit consumes about half the total resources of the FPWM, since it is implemented as registers instead of normal memory. The reason for this is to allow a large amount of parallel lookups, something that would be much harder to do with the normal integrated memory modules provided by the FPGA. Since the alphabet size, and therefore the matrix, is doubled at each step, the total resource consumption of the PWM Module is doubled, and since it at an alphabet size of 128 consumes about 31700 of the 32687 slices (about 50% more than can actually fit on the FPGA), it approximates the FPWM itself doubling at each step.

Alphabet Size	**Total Slices**
4	1943 slices
8	2914 slices
16	4901 slices
32	8848 slices
64	16719 slices
128	32687 slices

Table 4: The results of varying the alphabet size on the FPWM prototype.

4.2.4 Sequence Length

In the current version, sequence length is fixed to a maximum of 16M elements. Decreasing this limit will not lead to any major savings in resource consumption, while increasing it is hard to do on the current implementation, as it is limited by the available memory in the attached SRAM. Running sequences larger than 16M either requires a redesign to use the SMP's memory to store the sequence, or using a parallelized method such as the one described below in Section 5.3. No data is therefore given for this factor.

4.3 Summary

This section has provided some empirical and estimated performance data for the FPWM, as well as tables showing the resource consumption of the various modules given certain changes in the FPWM's configuration. With an estimated speedup of 15x compared to a contemporary general-purpose CPU, using about 8% of the FPGA's total resources, the true potential of this method has not yet been unlocked. However, unless there are any significant flaws with the estimation process used here, the results indicate that the method presented indeed holds a large potential, in particularly if one is able to exploit the entire memory bandwidth through parallel cores, as is described in Section 5.2.

5 Future Work

One of the goals of the project was to examine variations of the modules and overall design of the base FPWM prototype, to realize one or more high-performance implementations optimized on one or more of the following metrics: bit resolution and accuracy, PWM length, alphabet size, and throughput. A multi-node implementation using several FPGAs in parallel was also planned as an extension of the base project if there was any remaining time after the work on the prototype was finished. Unfortunately, due to the aforementioned problems with getting an implementation up and running on the target hardware, there was not a chance to do much of this work.

However, several variations and modifications were outlined during the project, and these will be described here. In most cases, the concepts described are sound, and would be relatively easy to implement on a working base prototype. These variations are mostly confined to Section 5.1. Some less-explored variations for increasing performance by using stream parallelization are presented in Section 5.2 and Section 5.3.

5.1 Module Variations

The simplest variations that can be done on the FPWM are changing the bit resolution (accuracy), PWM length, and alphabet size. These minor alterations all keep the base functionality of their modules. Of the more drastic alterations that does not involve additional parallelization, the module with the highest potential for change is the Result Module, of which one alternative implementation is described here.

5.1.1 Changing the bit resolution

The FPWM uses two different resolution parameters, an *input resolution* and an *output resolution*. The former denotes the number of bits in the PWM score values, while the latter denotes the number of bits in the window score. These resolution parameters can be adjusted independently, simply by changing the `RESOLUTION_INPUT` and `RESOLUTION_OUTPUT` variables in `user_pkg.vhd`. Since the addition of any two numbers can at most produce a number that uses one more bit than the largest of the numbers, the output resolution should generally be equal to the input resolution plus the number of levels in the Adder,

to not risk an overflow in the calculation, but it could be set lower if the score matrix is known to have characteristics that avoid such overflow in all cases.

The prototype has set the input resolution to 32 bits and the output resolution to 38 bits. These numbers are chosen to fit the current Result Module, which outputs the results as a result-index pair packed into a single long, consisting of a 26-bit index and a 38-bit result. If a larger resolution is required, the Result Module needs to be modified to output one result as a set of longs. If a smaller resolution is required, no changes need to be made to this module.

5.1.2 Changing the PWM length

Increasing the PWM length is somewhat more complicated, as some parts of the VHDL code currently do not scale automatically, due to the complexity involved and due to problems with creating scalable code which was also synthesizable [12]. Firstly, the `PWM_ROWS` and `PWM_ROW_BITS` variables need to be changed to the new size. Secondly, the number of columns in the memory matrix of the PWM Module needs to be changed to the new module. Thirdly, the number of data connections between the PWM Module and the Adder Module need to be changed to match the new length. Finally, the Adder Module must be altered to cope with the additional input lines. This will usually involve adding or removing whole pipeline levels on the addition tree, but is a relatively simple operation to perform.

Except for the changes in the PWM- and Adder Modules, no further functional changes need to be made to the other modules to cope with the additional length. However, the Control Module must be updated with the new Cooldown Delay of the circuit as a whole (i.e., how long it takes after the last memory element is sent from the Memory Module to the last potential result arrives at the Result Module). This change is equal to the number of elements added to or removed from the Adder Module.

A finalized design should be able to perform many of these operations automatically, simply by altering a constant. There was however no time to do this before the end of the project.

[12] Unlike most software programming languages, VHDL does not necessarily guarantee that code which is both syntactically and semantically correct, and simulates correctly, is possible to synthesize into a working chip. An example of an element that often is impossible to synthesize is a memory structure with a dimension higher than two.

5.1.3 Changing the alphabet size

The alphabet size can be changed by increasing the `PWM_COLS` and `PWM_COL_BITS` variables in user_pkg.vhd, as well as increasing the size of the memory matrix in the PWM module. In the prototype, each element is encoded using three of the eight bits available in the byte, with the actual alphabet being bits 1:0, and bit 2 being a `NULL` element marker [13].

Using the current system, the alphabet size can be increased to 128 without exceeding the byte boundary. Larger alphabet sizes could be used, but this would require using a multi-byte encoding, which would increase the memory bandwidth required. In most cases, this would not be a problem as there is still much spare bandwidth available, but it would be problematic with a parallel solution using more than 16 cores (see Section 5.2). Of course, seeing as the current FPGA according to the numbers presented in Section 4.2.3 cannot accommodate an alphabet size larger than 64 with a PWM length of 8, the concerns brought up by this point are largely moot.

5.1.4 Result Module: Simple filtered output

The earliest design draft used a Result Module that simply took a threshold value, and transferred any results that passed this threshold immediately to the SMP node. The advantage this design had was its simplicity, and that it would allow an arbitrary number of results to be obtained from one run. As the design was discarded for the sorting Result Module early in the process, the final workings of the module were not finalized, but it was suggested that it could be developed later, as an alternative for problems that required a larger result set.

While simple, there is one problem that has to be solved to allow this variation of the module to work. Since the RT Interface has a one-cycle cooldown after a write, any consecutive series of results, caused either by a low threshold or certain repeating patterns in the data, could normally not be written fast enough. The simplest solution to this problem is to stall the pipeline if two results clear the threshold on consecutive cycles. A more advanced solution could employ buffers to lower the number of stalls required, and

[13] The reason a bit reserved as a `NULL` marker instead of using an encoding scheme where, say, an element of all 1s is `NULL`, is that this simplifies the logic required to recognize the `NULL` elements. As there is plenty of memory bandwidth available, this was deemed more important than saving one bit for each element in the element stream.

combine it with burst writes to increase bandwidth if there are more than one result present in the buffers.

Note that in the worst-case event that all results are larger than the threshold, the system would be unable to process the writes fast enough even with taking full advantage of the burst write facilities, as over each nine-cycle period, nine results would be generated while eight would be written to the SMP. A system for stalling the pipeline would therefore be needed in any case, unless it is acceptable to discard results when the buffers are full.

5.2 Local Parallelized FPWM Cores

At the start of the project, a huge parallelized solution that would utilize the full 6.4GBps of memory bandwidth provided by the Cray XD1 was envisioned, but as the project took form and the components were finalized it became obvious that the FPGA installed on the Cray XD1 was not large enough to hold the required number of processing cores, at least not with the implementation developed in this project. A certain amount of parallelization is still realistic, as the FPWM prototype consumes about 8% of the total resources on the chip. Accounting for the overhead and additional circuitry needed to control additional cores, a realistic expectation would be that eight cores is a reasonable goal using the configuration of the current FPWM, while sixteen could be possible if some time was spent optimizing the design. If a larger FPGA could be obtained, more cores would of course be realistic.

To parallelize the PWM algorithm over several cores, the only observation needed is that any window score can be computed independently of all other window scores. That is, to compute window score i in a problem that normally would yield j window scores, none of the other $j - 1$ window scores need to be computed. Using this observation, an implementation with a number of separate and independent cores that each produce the window score for a different index each cycle can be envisioned.

The implementation suggested here assumes that there is one memory controller shared between all the cores. This memory controller feeds a larger shared element window buffer of size $pwm\_length + num\_cores$, to which the PWM module of each core is wired in such a way that the first core receives elements 1 through pwm_length, the second core receives element 2 through $pwm\_length + 1$, and so on, until the last core which receives element num_cores through $pwm\_length + num\_cores$. For each cycle, the window element buffer

is shifted by *num_cores* elements (discarding elements that are shifted out), with the same number of elements being loaded into the element window buffer by the memory module.

Using this scheme, on the first cycle the first core will start computation of the window score of the first index, the second core will start computation of the window score of the second index, and so on. The computation cycles will thus simply be cut to $1/num\_cores$, as there is no additional overheard in the memory module or shared memory element buffer.

The problems start when the set of scores have been calculated, and are to be stored by the Result Module. The implementation of the Result Module used in the prototype FPWM, while powerful, is unable to accept more than one element each cycle. Having one shared Result Module with the current implementation would therefore not be possible. Two different solutions to this problem will be discussed next, one using a large shared enhanced Result Module, modified to handle more than one element at a time, and one where each core has its own Result Module followed by a shared Result Aggregation Module that processes data from the individual Result Modules and outputs the highest-scoring of the aggregated results.

Shared Result Module

To use one shared Result Module that hooks directly into the Adder Module of each of the cores, the Result Module implementation must be enhanced to cope with a number of results equal to the number of cores each cycle. This doesn't necessarily mean that it must be able to sort this many elements each cycle, but that it must deal with the results in a way that has an acceptable performance according to some standard, namely that it on average does not have to stall the pipeline cores more frequently than a set value.

Since it would be hard (but most likely not impossible) to make an implementation that actually sorts more than one result per cycle, the suggested solution reuses the core Storage Elements from the old Result Module, but appends pruners to accept the elements while they are being sorted. There is one pruner for each core input, with the task of discarding any results that are less than the current minimum result stored in the Storage Elements (that is, the left-most result). If the result is larger, it is given to the Result Queue as usual. If more than one pruner has a larger result, the pipeline cores are stalled to give the Result Module time to catch up, and the results are given to the Result Module one at a time.

This solution could be improved by combining each pruner with a small buffer in order to

decrease the frequency of which the pipeline cores have to be stalled. Even if the result in the buffer is found to be stale in the sense that it is no longer a candidate for insertion by the time it has cleared the buffer, this would be handled by the Result Module in the final stage of the sorting. Whether the additional cycles required to sort false positives is acceptable compared to the compute cycles lost due to pipeline stalls would be dependent on the particular problem, and simulations on several different problems should be run before making a decision one way or another.

Another improvement would be to set a minimum value to decrease the stalls that would otherwise be experienced while the Result Module fills up during the start of a computation. When the computation first starts, the Result Module is empty, and every result from the pipeline cores would be accepted by the pruners as a potential result. This would lead to several stalls, which could possibly be prevented by setting such a value. Determining a good minimum value could however be problematic, and is not examined further here. The time lost to these stalls would also be relatively negligible for large runs, but could become a large factor for very small problem sizes.

Separate Result Modules combined with Result Aggregation Module

An alternative to the solution examined above is to employ one separate and independent Result Module per processing core. The Result Modules here can reuse the design in the prototype FPWM without significant alterations, and since there is no communication between the Result Modules, and they each receive exactly one result each cycle, there is no need to stall the pipelines in this scenario.

After the compute sequence has completed, the Result Module in each of the core will now hold a sorted array of the results computed by this particular core. To combine these results into one result array that can be returned to the SMP, another unit dubbed the Result Aggregation Module is needed. This module takes care of post-computation sorting of the result sequences from the individual Result Modules, an operation which is significantly simplified by the fact that each individual result sequence already is sorted.

The Result Aggregation Module simply needs to implement a variant of the last stage of the standard merge sort algorithm. A number of pointers are initialized for each of the Result Module result arrays, pointing to the largest (right-most) element of each array. For each cycle of the sort algorithm, the largest number pointed to is stored in the Result Aggregation Module together with its index, and the pointer for that particular result array

is decremented to the next-lower result. This is repeated a number of cycles equal to the number of required results.

Another advantage of this implementation is that it is possible to return more results than are kept in each individual result buffer. However, in this case it cannot be guaranteed that all the largest results are returned. To show this, consider the extreme case where each result module can hold eight results, and the nine top results are generated by the same core. Only eight of these results will be returned, together with lesser-fitting results from other cores if the number of requested results is higher than eight. However, since the distribution of data between the cores is done by interleaving, it can be assumed that the results in most cases will be relatively evenly distributed between the cores, but a user application should not rely on this.

Solution Comparison

The two solutions presented differ both in their impact on the overall performance, their implementation complexity, and the amount of chip resources they consume. It is hard to make accurate estimations without doing a test implementation, but generally it can be assumed that the Shared Result Module will have a higher implementation complexity, as it has to manage sorting from several cores and deal with stalling as well as buffers. On the other hand, it will likely have lower chip resource demands than the solution with a Result Aggregation Module, as it avoids storing results that will not be used.

On the performance side, the obvious drawback of the Shared Result Module is the need for pipeline stalls. However, the impact of this is highly problem dependent, and should in many cases be negligible, but in the worst-case situation, where each result is larger than any results before it, the performance would degrade to that of a single core. The alternative also has its drawbacks however, as an additional sorting stage is introduced following the computation stage. Nevertheless, since the time required for this stage is linear to the number of elements that are to be returned, the time required here will always be predictable and negligible for any realistic problem sizes.

Using the prototype implementation as a starting point, it would be less complex to move to the solution with the Result Aggregation Module, as this simply involves duplicating the current core and creating the new module, which operation is relatively simple.

5.3 Multi-Node FPWM Implementations

In the case of the Cray XD1, there are a total of six nodes available per chassis, with each node holding two AMD Opteron general-purpose CPUs as well as one FPGA-based application acceleration processor. The current prototype implementation is only designed to use one single node, with application logic being responsible for any multi-node parallelism. This solution does have some advantages, as the application logic is free to do problem-dependent optimizations with the data distribution, as well as utilize the additional CPUs as it sees fit. It does however have the distinct disadvantage of having to reinvent the wheel with each new application.

For many algorithms that use PWMs, the PWM calculation itself greatly dominates the computation time required. In this case, a simple library that utilizes all the available nodes, each running a small piece of control code on their host CPUs, would likely be sufficient. That is, the user application itself is contained on one single node, which exploits the computation capacity of the other nodes for the PWM calculation alone. If this assumption can be made, it is relatively simple to create a library that can be called from this node without making any changes to the current interface. This library would be responsible for initializing the other nodes to hold the necessary control code, and divide the work load between the various nodes.

If the assumption does not hold, and the computations performed by the application logic itself is sufficiently time-consuming that it is desirable to utilize the computation power of several CPUs to run it, a more complex solution may be required. Again, what would be considered the best implementation is highly dependent on the nature of the user application. If each node produces work independently, or in such a way that each node can be assumed to be able to keep its own FPGA busy, today's system could be applied directly, where each FPGA only receives and processes work from a single node. If the application nodes cooperate in some way in building one or more PWM work units that cannot be directly mapped 1:1 between a node and an FPGA in a way that keeps all the FPGAs busy, it could however be better to utilize one node as a control node that accepts these work units and distributes them through a library such as the one mentioned in the previous paragraph.

As the problem where each of the nodes can be directly mapped to a single FPGA does not require any special treatment, the focus should be on how to make it possible to utilize

a set of nodes, each with one instance of an FPWM implementation running on its FPGA, as if it were one single FPWM.

5.3.1 Parallelizing Work

As mentioned earlier in Section 5.2, there is very little data dependency involved in computing the window scores in the PWM algorithm. However, there are two reasons not to use the same attack vector for multi-node parallelizing as was used in the multi-core parallelizing. Firstly, transferring data between distinct nodes is generally more efficient for contiguous data than for interleaved data, as memory accesses on contemporary computers are typically done in 64- or 128-bit wide chunks, while each data element is eight bits. Secondly, the granularity on inter-node communication cannot be as fine-grained as in the internals of the FPWM. To avoid spending precious compute cycles on wastefully repacking data, it is therefore preferred to divide the workload into a number of contiguous element streams equal to the number of FPWMs available.

However, it is important to note that a clean split would ignore any results occurring on the element stream near the split. That is, if the FPWM uses a PWM length of eight, and an element stream of length 64 were split into two sequences, one ranging from 1-32 and the other from 33-64, window scores with indexes ranging from 26 to 32 would not be computed, as parts of the required data would not be present on either of the computing nodes. An overlap of a number of elements equal to the PWM length is therefore required. Using the aforementioned example, the stream could for instance be split into 1-35 on node 1 and 29-64 on node 2.

5.3.2 Parallel Computation

When the element streams have been distributed to their respective nodes together with the score matrix itself through a distributed-memory parallel system such as MPI, no special considerations need to be taken in order to compute the scores for each individual stream. However, as each partial element stream would most likely be stored beginning at index 0 of the FPGA's SRAM, the order of the streams must be preserved, in order to determine the correct global index for each individual result when it has been computed. The rationale for doing an index reordering instead of using the global index exclusively is that in addition to simplifying certain parts of the FPWM's operation, avoiding the need to

change it to support multi-node parallelization, it allows a multi-node implementation to use larger element streams than the single-node implementation supports, without altering the FPWM to use the SMP's memory.

As was discussed earlier, the overhead of using a multi-core solution is negligible for reasonably large sequences, and will accelerate the speed roughly linearly to the number of cores in the implementation. The multi-node implementation on the other hand does have some overhead. This overhead can mostly be isolated to three factors: the time required to scatter the element stream, the time required to broadcast the PWM score matrix, and the time required to gather the results back to the primary node. However, the actual time required for these three factors depends highly on the distribution mechanism used. Using MPI as an example, many of the implementations differ in how they perform the elementary operations such as broadcast and scatter. The reasons for this mostly lie in the different abilities of the target hardware, such as if a broadcast mechanism is available or if the system must depend on point-to-point communication alone.

Determining where the major overhead of a given multi-node implementation is depends both on said differences in functionality and performance of the underlying distribution mechanism, and the nature of the problem (or rather, set of problems) given by the user application. If only a single run is to be performed, or the element stream changes between the runs, it can be assumed that the scattering of the element stream to the various nodes will be a potential bottleneck. A typical stream ranges from 0.5 to 8 MB, the largest of which would take 4 milliseconds to scatter on a Cray XD1 (assuming the full bandwidth of 2 GB/s per node is achieved), while the computation process itself would take about 40 milliseconds on the current single-core implementation (using the estimates given in Section 4.1). On a multi-core implementation, this number will decrease roughly linearly with the number of cores used, making the time required for the scatter the dominant term when the number of cores hits 5 or 6, due to the additional overhead of transferring the sequence data to the FPGA itself.

The PWM score matrix, while generally much smaller (128 bytes on the prototype, 1024 bytes on a 32-bit 64x4 implementation), will usually be replaced between each run, requiring a broadcast for each run. The gather of the results at the end, which also happens on each run, is 64 bytes per node with the current 8x 64-bit results. However, getting an accurate estimate of the real impact of this overhead is hard without doing simulations directly on the target hardware, as communication latency and the load of the network, in addition to

pure bandwidth, greatly affects the results.

Generally, applying a multi-node solution on problems that require a scatter of the element stream for each run will most likely not be fruitful, but this limitation applies on the basic FPWM too, where the load time would dominate the total runtime with implementations that have 8 or more cores. The time taken for the PWM broadcast and the result gather would on the other hand most likely be negligible for large runs, but could have a significant effect if the user application often requests runs on relatively short element sequences, or the latency in the underlying interconnect is high.

Note that this multi-node implementation can be used together with the multi-core solution described in Section 5.2 without making additional modifications.

5.4 Summary

This section has presented a number of options that can be explored to more fully assess the potentials of using reconfigurable computing for pattern discovery in biological sequences, while restricting itself to the implementation of the PWM part of the implementation. The possibility of moving more of the application logic to the FPGA itself to further speed up the process is of course an interesting one, but as it would make the solution less general, and requires expert knowledge outside the scope of this paper, this has been deferred.

Due to the problems encountered while attempting to implement a working PWM on the Cray XD1's FPGAs, the true potential of this application of reconfigurable computing have still not been adequately explored. While the theoretical calculations show that there is indeed a large amount of potential in this approach, it is difficult to make hard statements without having empirical measurements of the performance on real problems. Furthermore, as has been examined in this section, the level of additional performance that could be harvested from parallelizing the work over multiple cores and nodes is particularly hard to estimate, so empirical data would be required to make a solid argument that the performance is indeed much higher than can be achieved using traditional general-purpose processors. Nevertheless, the estimates presented here do indicate that the increase in performance would be significant, so it is therefore the opinion of the author that exploring these options would be a viable strategy.

6 Conclusions

This paper has presented a solution to an FPGA-based PWM matcher in the form of the so-called *FPWM Prototype*, using the hardware facilities on the Cray XD1 Supercomputer. The prototype implementation currently runs as a single core on a single node of the Cray, and provides a theoretical PWM matching capability roughly 15 times greater than a contemporary Pentium M general-purpose CPU. Theoretical and empirical data regarding performance and resource consumption for this implementation have been provided.

A method for increasing the speedup to a theoretical maximum of 480x has also been described, using a multi-core implementation on a single chip. This theoretical limit could potentially be attained with today's hardware, but would require certain compromises with regard to bit resolution and PWM length in order to fit on the FPGA. A full-scale implementation providing the capabilities required by many of today's algorithms would most likely not reach this speed, but as the FPGA currently installed on the Cray is also available in a larger variant (the Virtex-4 family), it is reasonable to assume that such an implementation could indeed be feasible on contemporary hardware.

A method for using several nodes on the Cray XD1 transparently for the user application, in order to further increase the performance, has also been described. However, as theoretical performance estimation on such hardware is a highly inexact science, and empirical measurements could not be performed at this time due to the state of the prototype, no estimates have been provided for this method.

While some of the original goals were attained, other parts of the project could be considered a failure. Due to a number of implementation problems, a working FPWM was not available in time for use with the two other projects mentioned in the introduction, involving hardware acceleration of the Gibbs Sampling and MEME algorithms. The main problem with the cooperation between these projects was that it relied on the FPWM being in a finished and working condition before the work involving it could begin, which turned out to be much harder and take much longer time than what was first envisioned. The planned empirical measurements of the performance boost for these algorithms are therefore not yet available.

References

[1] David Baker, "Rosetta@home Website," http://boinc.bakerlab.org/rosetta/; accessed July 3., 2006.

[2] Cray XD1 Documentation, "Cray XD1 Programming (S-2433-131)," Cray Private, 2005.

[3] Gary D. Stormo, "DNA binding sites: representation and discovery," Oxford University Press, 2000.

[4] Wyeth W. Wasserman & William Krivan, "In silico identification of metazoan transcriptional regulatory regions," Springer-Verlag, 2003.

[5] Rodger Staden, "Computer methods to locate signals in nucleic acid sequences," IRL Press Limited, 1983.

[6] Geir Kjetil Sandve & Finn Drabløs, "A survey of motif discovery methods in an integrated framework," NTNU, Unpublished, 2005.

[7] Lars Krutådal, "Introduksjon til bruk av FPGA i vitenskaplige beregninger," NTNU, Unpublished, 2005.

[8] Gerald Estrin, "FIXED + VARIABLE Computer," IEEE Trans. on Electronic Computers, V12, pp.747-754,755-773, 1963.

[9] Gerald Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer," Annals of the History of Computing, IEEE, Volume 24, Issue 4, Oct.-Dec. 2002 Page(s):3 - 9, 2002.

[10] Mario Schaffner, "A computer architecture and its programming language," IEEE Trans. on Computers, V27, N12, pp.1015-1028, 1978.

[11] Andre DeHon, "Notes on Coupling Processors with Reconfigurable Logic," MIT Transit Project, 1995.

[12] Alessandro Marongiu, et.al, "Designing hardware for protein sequence analysis," Bioinformatics, Vol.19, no. 14 2003, pp. 1739-1740, 2003.

[13] Tim Oliver, et.al, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," Bioinformatics, Vol.21, no. 16 2005, pp. 3431-3432, 2005.

[14] Peter S. Pacheco, "Parallel Programming with MPI," Morgan Kaufmann Publishers, Inc., pp. 259-260, 1997.

[15] Øyvind Bø Syrstad & Lars Andreas Eidsheim, "Akselerering av MEME-algoritmen ved hjelp av PMC," NTNU, Unpublished, 2005.

[16] Øyvind Bø Syrstad, "Metoder for akselerering av MEME," NTNU, Unpublished, 2006.

[17] Lars Andreas Eidsheim, "Parallell hardwareakselerert Gibbssampler," NTNU, Unpublished, 2006.