Department of Electrical and Computer Systems Engineering

Technical Report MECSE-8-2006

Data-flow: An Old/New Approach

B.N. Dickson and G.K. Egan

MONASH UNIVERSITY

Data-flow: An Old/New Approach.

B. N. Dickson and G. K. Egan
Department of Electrical & Computer Systems Engineering
Monash University 3800
Melbourne, Australia

***Abstract*— Computer architectures have until recently been dominated by the von Neumann style architectures. The improvement in re-configurable hardware with the development of larger Field Programmable Gate Arrays (FPGAs) has allowed other styles of architectures to be implemented. One of these styles of architecture is data-flow. The data-flow architecture implemented on an FPGA, as described here, is a significant subset of the CSIRAC II data-flow architecture. The performance of this architecture was compared with a von Neumann style control-flow architecture, and comparable results were obtained with noticeable performance benefits with some programs.**

***Index Terms*— Computer architecture, data flow computing, field programmable gate arrays**

I. Introduction

THE study of computer architectures has been largely dominated by the von Neumann style architectures, which even von Neumann considered as being interim pending more advanced implementation technologies. However at the time it was proposed in the 1940's [1] the von Neumann architecture was the best solution for the available hardware. Despite the improvements in technology allowing other styles of architecture to be implemented, industry's conservative approach has meant that very few of these have been implemented in practice. One of these styles of architectures developed by research organisations was the data-flow architecture. Some of these designs were Monsoon, Manchester Data-flow Machine, and CSIRAC II [2]. With the recent advances in re-configurable hardware, caused by the development of large Field Programmable Gate Arrays (FPGAs), the implementation of these more advanced architectures has become viable.

I. BACKGROUND

A. How do data-flow architectures work

Data-flow architectures only execute instructions when all of the required data is available. This is in contrast to control-flow architectures, based on the von Neumann design, where instructions are executed independent of whether the data is available. The advantage with data-flow is that all of the dependencies commonly existent in control-flow architectures are avoided as only instructions containing all their data are executed.

B. N. Dickson is a graduate of the Department of Electrical and Computer Systems Engineering, Monash University, Clayton, Victoria 3800 Australia.

G.K. Egan is Professor of Telecommunications and Information Engineering in the Department of Electrical & Computer Systems, Monash University, Melbourne, Australia, (greg.egan@eng.monash.edu.au).

The programs executed by the data-flow machine are 'directed graph(s) consisting of named *nodes*, which represent instructions, and *arcs*, which represent data dependencies among nodes. Operands are propagated along the arcs in the form of data packets, called *tokens*.' [2] These directed graphs are normally written in a higher level language and compiled into a set of node descriptions and input tokens to be read into the processor. When the processor is executing the program, the tokens are fired into the processor and then executed by the Execution Unit. This directed graph approach allows the nodes to be executed in any order, thus improving performance and in some cases simplifying programs that may be extremely complicated to represent as a list of instructions.

A. Different types of data-flow architectures

Data-flow architecture can be split into a few groups: Static (Dennis), Static Queued, Dynamic, and Hybrid. Static architectures are the most restrictive as 'An enabled node is fired if there is no token on any of its output arcs' [1]. This is a problem as more than one token could appear on an arc, thus control tokens are used to block the execution until the output arc is available. Static Queued architecture permit tokens to be queued on arcs, thus eliminating the requirement of an interconnecting network for the control tokens. Dynamic architectures also avoid the problem by altering the enabling and firing rule to be 'A node is enabled and fired as soon as tokens with identical tags are present on all input arcs' [1]. These tags contain information about the destinations and a colour field to separate tokens on the same arc. In a normal application this could cause the number of tags to increase rapidly, thus a hybrid of Static Queued and Dynamic was proposed. Hybrid architecture uses static queuing for the inner loops while the dynamic loop unrolling is used for the outer loops, thus obtaining the benefits of a dynamic architecture while reducing the number of different tags [2].

A. The architecture being implemented

The CSIRAC II data-flow architecture was a Hybrid created in 1978 by Egan and was later implemented into hardware in the early 80's [1]. Some of the CSIRAC II architecture features include: a self loading Node Store, buffers on both the Matching Store and Execution Unit, most instructions found in a conventional processor and some extras, external network, 128-bit tokens, colours and ordered execution [2]. CSIRAC II allows the nodes to be loaded through the Input List and also allows them to be changed while the processor is still executing. The external network allows the architecture to operate with multiple processors while also allowing input and output to peripheral devices to be separate from the processors executing the program. The tokens used in this architecture are 128-bit long. These tokens store information about

which processor the token is going to be executed on, the ALU to be used on that processor, a colour field allowing loop unrolling and recursion, the type of node the token is for, and input side of the node, data type and the data required for that node.

Data-flow programs have two types of nodes, monadic which have one input and dyadic which have two. These types of tokens are not evenly fired into the processor, thus there has to be some elasticity in the processor to allow for these runs of monadic and failed dyadic tokens. CSIRAC II accommodates this by having buffers before the Matching Store and Execution Unit. Some other architectures like the Manchester Data-flow Machine created by researches led by Watson and Gurd at Manchester University neglected this elasticity causing their processor to continually stall while the Matching Store collected tokens from the linked list [3].

I. DESIGN

The data-flow architecture being implemented for this project was a subset of the CSIRAC II data-flow processor. The processor was designed only to execute programs with graphs of less than 125 nodes with only integer and bit operations being available. The reason for this was to reduce the amount of hardware required to implement it, thus enabling it to fit onto the Altera Cyclone FPGA being used. The basic structure of the processor can be seen in Figure 1.

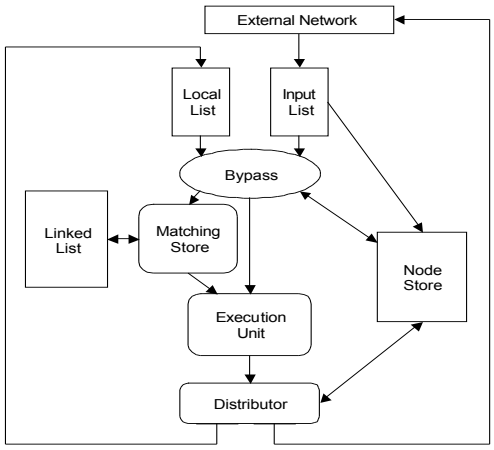

Fig 1. Block diagram of the CSIRAC II processor architecture.

The processor is split into six distinct sections: the Lists, Bypass, Matching Store, Node Store, Execution Unit, and Distributor. Because of this separation, each section was implemented as stand-alone function blocks with communication channels between them. The details of each section are as explained in the following subsections.

A. Lists

The lists hold the node descriptions and tokens before they enter the processing sections of the processor. There are two lists: one is a Local List, and the other is the Input List. The Local List is filled from the Distributor on that processor whereas the Input List is filled from an external source, which is normally another processor's Distributor, thus allowing the program and data to be entered into the processor.

B. Bypass

The Bypass selects between the two lists with a bias to the Local List, as filling this list will cause the processor to stall. The Bypass then adds the function and literal from the Node Store to the token, if required, before distributing the token to either the Matching Store or the Execution Unit. This causes dyadic tokens to pass through the Bypass, which is not necessary, however it does simplify the logic required in the Node Store to collect the function and literal.

C. Matching Store

The Matching Store is where tokens required for nodes with multiple inputs are matched together. This is accomplished by storing the tokens in an array indexed by the element number. When a token with the same element number is found it is sent to the Execution Unit Buffer with the data from the matching token if it is the opposite input, otherwise it is added to a Linked List connected to that element number. As the new token is added to the end of the Linked List, the order is preserved thus ensuring that the correct set of input data for each node is found.

D. Node Store

The Node Store is where the information for each node is recorded. The Node Store was split into two storage sections, one containing the function and literal and the other containing where the data is sent after executing the node. The reason for splitting the Node Store is that obtaining the function and literal when the tokens are being manipulated, to insert them into Execution Unit Buffer, reduces the delay before the Execution Unit can begin execution while not affecting the size of the buffers required. At the beginning of execution the Node Store contains no information about the program being executed, thus all of the node descriptions are entered through the Input List. Thus the operations of the Node Store consist of three sections: one for each of the storage sections to collect the information and one to insert the node descriptions into the Node Store.

E. Execution Unit

The Execution Unit uses an arithmetic logic unit (ALU) that is the same as in common architectures with the only difference being the reduced instruction set. The instruction set has been reduced as the processor was designed to only implement integers and a selection of the possible instructions specified for the architecture in [4].

F. Distributor

The Distributor sends the newly created tokens to either another processor, or to the Local List depending on the distribution system being implemented.

G. The features of the subset being implemented

The subset of the CSIRAC II data-flow architecture implemented has the following instructions: Add, Subtract, Multiply, Shift Up, Shift Down, Equal to, Not Equal to, Greater or Equal to, Pass if True, Pass if False, Switch, Nop, and Replicate. These instructions can be either Bypass or Normal nodes, where the type of nodes indicates the number of inputs: Bypass has one input and Normal has two. The data types available on the processor are 8-bit, 16-bit and 32-bit integers, single bit, and node descriptions. All of these data types are contained in single word tokens except the node descriptions which in this implementation were restricted to the Input List and Node Store. The other restrictions are that nodes can have a maximum of two destinations, only Bypass nodes can have literals, and there is a maximum of 125 nodes in the directed graphs being implemented on the architecture. The processor it self is a single processor which accepts 128-bit input tokens and outputs 32-bit values if the destination is an output node.

H. Design considerations

The buffer and list sizes had to be carefully considered as filling one of these could cause the processor to deadlock. In this implementation the FPGA resources determined the buffer and list sizes. For the buffers and lists using RAM, the size was dependent on the smallest number of RAM blocks required as there was a limitation on these blocks and once a block was required there was no reason not fully utilizing it. On the other hand the lists using an array were restricted to the absolute minimum size to enable the processor to fit on the FPGA. Even though the FPGA resources dictated the size of the buffers and lists, the comparative size required had to be considered. The Local List was made the largest, as filling this list will cause the processor to deadlock. Similarly the Execution Unit Buffer size was obtained by the maximum influx possible in the processor. The smallest in this implementation was the Input List, as there was no external network this list only contained triggering tokens, and a stall would not affect the rest of the processor.

As the processor was the only implemented part of the architecture, some of the data fields could be reduced while others could be excluded. The excluded fields are colour, process, and processor since the colours were not being implemented, there was only one ALU and only one processor. The data field was reduced to 32-bits since this was the largest data type the processor was capable of handling. The data type field was reduced to 4-bits so these two fields fit neatly into the 36-bit words of the RAM blocks, thus reducing the number of RAM blocks required for the Linked List while still being large enough to hold all of the data types. The node id and addresses to the Linked List were reduced to their minimum size.

As there are two lists feeding the Bypass a selection has to be made as to which list the next token will be taken from. The main requirement is that the Local List never becomes full, as this will deadlock the processor. Due to there being no external network, the only tokens in the Input List will be priming tokens thus the Local List will normally be empty when the Input List contains tokens. Therefore tokens were taken from the Local List unless the list was empty as there was no benefit in complicating the design to ensure the Input List never became full.

I. Implementation of the Architecture

Handel-C was used to implement this architecture as it is a higher-level language than VHDL, thus simplifying the code. The architecture is split into five sections as described. In addition to these sections there is a function to retrieve the program, which is stored in external RAM, and load it into the Input List, and a function to operate the display.

During the execution of this program, values can be viewed on the LEDs located on the development board by using the appropriate instruction. A delay section of code was added to the design to stall the processor while the value was being displayed.

Another feature of the processor is a memory led, which indicates when one of the memory sections has become full which will probably cause the processor to deadlock. The cause of this in most cases is the Local List becoming full due to excessive branching in the program, however allowing un-matched tokens to build up in the Matching Store has the same effect as the memory allocated for the Linked List becomes full. These problems can be avoided by writing the program so it does not excessively diverge before converging and ensure that no one side of a dyadic node enters the Matching Store excessively more often at any particular point in the execution.

A. Design considerations

The major design consideration was whether to use RAM or arrays. In general this decision was already made with the size restriction the FPGA placed on the project. Thus the majority of storage was placed in RAM, with a lesser amount implemented using arrays and the program executing on the chip stored in external RAM. The use of external RAM was restricted to storing the program, which would only be accessed once, thus avoiding the performance degradation in the main section of the processor while allowing the processor to fit on the chip.

Another consideration was how Handel-C implemented some of its available features. A perfect example of this was the default division as it required the entire FPGA and reduced the maximum clock speed to 2MHz. Division was excluded for the instruction set and replaced with shift operations allowing the programmer to achieve the same results using different operations. Channels are designed for passing values between parallel sections of code using handshaking. While this is useful in the correct implementation, in many cases it was not practical to pass data this way, thus for some sections the data was passed using variables and signal.

A. Implementation restrictions

Due to the FPGA size limitations and desired performance requirements the following restrictions were placed on the processor.

The Linked List which stores the multiple tokens for a particular node in the Matching Store was placed in on-chip RAM, thus its size was limited to 256 locations which in most cases would be adequate as the probability of having more than three tokens for one node stored is extremely low.

The number of nodes allowed in the program was restricted to 125. This allows small programs to be executed on the processor, thus allowing the design to be tested while not creating problems caused by allocating excessive amounts of memory and therefore restricting the available space on the FPGA required for implementing the rest of the processor.

The instruction set and number of types were reduced. The reason for doing this was to simplify the ALU being used in the implementation as this part of the program could very easily be obtained from a conventional processor which implements a larger instructions set and more data types. Thus a select few instructions were chosen to allow more of the chip space to be dedicated to the data-flow sections of the processor while still allowing suitable programs to be written.

The branching factor for each node was restricted to two destinations. Since the design is for a single processor, which could be joined to other processors later by using multiple FPGAs. Thus, having a larger branching factor would only fill the Local List with tokens faster than they can be removed causing the processor to deadlock. In a multiple processor design these extra tokens would be spread between the processors thus avoiding this problem.

A. On-chip specifications

The Quartus compilation results for the architecture on the Altera Cyclone EP1C6T144C6 FPGA:

-- 5,260 / 5,980 (87%) LE's
-- 19 / 20 M4K RAM
-- 62,090 / 92,160 (67%) bits of RAM
-- 50 MHz clock.

Despite the fact that there were a limited number of M4K RAM blocks only 19 were used, as altering the design to increase the storage used required two RAM blocks. Most RAM blocks were not fully utilized due to the way the RAM was configured. Not all of the LEs available on the FPGA were used, as increasing the percentage of used LEs increases the interconnections on the FPGA, thus reducing the clock speed. A larger FPGA would have reduced this limitation on the clock speed while allowing sections of the program to be written differently to obtain a higher clock speed.

A. Problems encountered and solutions

Some of the problems encountered during this project were:

--Collecting the outputs from the Bypass and the Matching Store and inserting them into the Execution Unit Buffer. As it was decided to do this in one clock cycle signals had to be used, with variables used for the control logic. These variables had to be assigned in the previous clock cycle which was possible, as knowing if something will be sent is simpler than knowing what will be sent.

--The Node Store had a similar problem with obtaining the function and argument, however the destinations could use a channel as there was no control logic due to there only being one input to chose from.

--Being able to stall the pipeline if the lists and buffers were becoming full caused some problems with the signal used in the pipeline stages. If the stages were stalled when the list or buffer became full some of the information was lost, however if the pipeline stages were fed Nop and continued to run the processor operated correctly.

Timing of the hardware design was found to affect the output with minor adjustments in the design causing tokens to be gained or lost in different sections of the processor. The cause of these problems was very difficult to locate as the simulator in Handel-C was not affected by them and there were very few debugging features added to the implementation. Another possibility was some of the data stored in the external ram was being corrupted depending on the design being loaded onto the FPGA.

I. Operations

A. Compiling the test programs

To operate the processor the data-flow graphs had to be compiled into machine code. This process was done in two steps to allow the program to be checked between the steps. The first step compiled a representation of the directed graph into a list of node descriptions and tokens. Normally the data-flow machine would execute this directly however due to the restraints on the implementation a simple parser was used to convert this into serial code.

A. Description of the test programs

There were two test programs implemented: a modeling of a small heated 2-d mesh, and a filter. These programs were executed on a control-flow processor as well as the data-flow processor in order to obtain a performance comparison.

The mesh contained 9-cells heated from the top, cooled from the bottom and insulated on the sides. This mesh is continually heated until the center cell has a constant temperature for a set number of iterations. This mesh program should execute in approximately the same time for the two architectures as the programs have a similar number of instructions and the processors have the same clock speed. Also the data-flow processor has the blocking node primed to allow multiple iterations of the same value and branch predictions is enabled on the control-flow architecture. Taking these considerations into account the data-flow processor should perform as well as the control-flow processor. The reason for this is the mesh is very small thus allowing the values to be stored in registers in the control-flow processor, this alleviates the need to load and store data from cache therefore avoiding most of the data dependencies. Having the values in memory also reduces the number of instructions required to update the old values, thus the advantage of the data-flow processor keeping the values for each iteration separate will not be apparent. The control instructions to ensure the processors do the same number of iterations will only be slightly smaller in the data-flow design and the advantage of the data-flow processor continuing to execute nodes while the branching is being evaluated will probably not be obvious due to the branch prediction being used in the control-flow processor.

The filter was an integrator consisting of an add node and a feedback path consisting of a shift down node. This graph was fed with a list of tokens generating a step response. As a directed graph this program only contained two instructions compared to the six instructions for the control-flow implementation, thus the data-flow program was assured of executing faster. The data-flow machine had so few instructions as there was no branching required because the output would be stopped when there was no input, instead of checking if there is no input then breaking out of the loop.

A. Performance comparison of the two architectures

With the mesh the data-flow architecture was found to perform slightly slower than the control-flow architecture taking 125 clock cycles compared to 77 per iteration. The main reasons for this was that all of the advantages a data-flow architecture has over the control-flow architecture were not tested in this program. Also about 50% of the nodes in the directed graph were dyadic and as this implementation of the CSIRAC II architecture sends all of the tokens through the Bypass, the Execution Unit was only executing two-thirds of the time. However if the control-flow architecture had to store and retrieve the values from memory the number of clock cycles would increase by 54 per iteration causing it to execute slower than the data-flow implementation, assuming each memory access only takes one clock cycle.

The filter also had better performance on the control-flow architecture. However this was expected as the data-flow architecture reads the tokens from the external RAM which takes 16 clock cycles thus the processor would have computed the result before the next value was added to the Input List. As this did not give a fair comparison due to the hardware limitations dictating the processors performance, the loading section of the program was rewritten so these tokens were fired multiple times instead of multiple instances of them being read from memory. With this alteration the data-flow architecture's performance was found to exceed that of the control-flow taking an average of four clock cycles compared to seven clock cycles per input value. The reason the data-flow processor is able to output an almost continuous stream of results is that the proceeding values beginning execution before the previous values has been fully executed.

II. Conclusions

A subset of the CSIRAC II data-flow architecture was successfully implemented on an Altera Cyclone FPGA. However there were some limitations on the design mainly due to the large amount of memory required and the configuration of the available RAM. One of these restrictions was that the program, due to its size, had to be stored in external RAM. This increased the number of clock cycles required to load the node definitions by a factor of 16 as the data bus was only 8-bits. This also starved the processor while executing the filter program, as the previous token would have completed execution before the processor had retrieved the next token from the external RAM. The effect of this could have been reduced by using another device to implement the external network thus allowing a larger data bus to be used. Alternatively the token size could have been reduced however this would cause the processor to differ from the specifications of the CSIRAC II architecture. The other limiting factor causing the clock speed to be restricted to 50MHz was the number LEs available on the FPGA. This caused compromises to be made in the design, thus limiting possible performance improvements and reducing the performance gain obtained from the scalability of a data-flow architecture.

When comparing the performance of the processor to a control-flow architecture it was found that the performance advantages of the architecture were only apparent in some of the programs being executed. This was due to the size restrictions on the directed graphs being executed, thus the control-flow architecture was able to use registers for all of its data storage. However if the program was scaled up the control-flow architecture would have to access memory, drastically decreasing its performance, while the data-flow architecture would only be affected by the increase in the number of instructions being executed. The program that performed better on the data-flow architecture was the filter. This was an expected result as the filter requires a constant stream of data to be processed simultaneously which is possible in the data-flow architecture as another iteration of the graph can begin execution before the previous finishes. Also, unlike the control-flow architecture, no control logic is required, thus reducing the number of instructions being executed from seven down to four.

Some possible improvements to the processor to improve its performance would be:

--Move the retrieval of the function and literal closer to the Execution Unit so the dyadic tokens do not have to go through the Bypass thus increasing the average throughput of the Execution Unit.

--Distribute the tokens to both the Local and Input List, thus allow a larger branching factor to be used and therefore reducing the number of replicates used in the programs.

--Allow the Local List to send tokens to the Bypass and the Matching Store simultaneously.

--Collect the initialize tokens from another source so a larger word size can be used or alternatively have a different clock for this section to ensure the maximum clock speed to collect the tokens.

--Implement the architecture on a larger FPGA allowing it to be scaled-up, therefore the performance will be comparable to the control-flow architecture. Also this would allow the implementation of the design alteration that were previously disregarded because of the size restrictions of the FPGA.

Acknowledgments

We wish to acknowledge; Mr. A. Sloan for re-instating the compilers developed by the dataflow research groups at the Royal Melbourne Institute of Technology and Swinburne University of Technology; Dr. A. Price, Mr. T. Cornall, and Mr. T. Ramdas for their technical assistance and suggestions; and Miss. N. Brammer for her support and assistance with proof reading.

References

[1] J. Silc, B. Robic, T. Ungerer. Processor Architecture: From Dataflow to Superscalar and Beyond. Springer-Verlag Berlin Heidelbery, 1999.

[2] Gaudiot, J-L. & Bic, L., Advanced topics in data-flow computing, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[3] Egan, G.K. (2005), personal communication.

[4] Egan, G.K. (1990), The CSIRAC II Data-flow Computer: Token and Node Definitions.,Technical Report 31-001, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology.