Department of Electrical
and
Computer Systems Engineering

Technical Report
MECSE-9-2007

An FPGA Based Implementation of the CSIRAC II Dataflow Computer

A. Sloan and G. Egan

An FPGA Based Implementation of the CSIRAC II Dataflow Computer

Adam Sloan
Dept. ECSE, P.O. Box 35
Monash University
Australia, 3800
adam.sloan@eng.monash.edu.au

Greg Egan
Dept. ECSE, P.O. Box 35
Monash University
Australia, 3800
greg.egan@eng.monash.edu.au

ABSTRACT

It has become common to translate applications to directed graphs that can be directly mapped to a programmable logic device. However, resource constraints force critical resources such as ALUs to be explicitly reused through switching interconnects. Dataflow computer architectures interpret and execute directed graphs in a general purpose manner. This facilitates the acceleration of highly complex graphs and the ability to execute concurrent applications. Dataflow computers are a well known machine architecture. However, an investigation is required to accurately measure performance using current hardware technologies.

This paper describes the low-level design and implementation of a hybrid dataflow computer called CSIRAC II. A multi-processor investigation was conducted by cycle accurate simulation. Results indicate that the architecture is capable of extracting implicit concurrency whilst maintaining high processor utilisation levels. Most importantly, it was determined that current FPGA technologies can make the overheads traditionally associated with dataflow architectures acceptable, particularly in streaming applications.

1. INTRODUCTION

The introduction of high density programmable logic devices (PLDs) has provided researchers with the means to investigate and implement complex architectures previously restricted to custom silicon fabrication. In particular, research based upon field programmable gate arrays (FPGAs) has become widespread in research groups from diverse areas of interest. It has become common to translate a wide variety of applications into directed graphs that can be directly mapped to PLDs. Despite increasing device densities, some kernels cannot be directly implemented due to resource constraints. Consequently, hardware reuse is required through switching interconnects to share critical resources such as ALUs. Dataflow computer architectures interpret and execute directed graphs in a general purpose manner. Such architectures can thus be utilised to accelerate highly complex graphs in their entirety. Additionally, dynamic dataflow architectures possess the ability to execute multiple concurrent applications, providing a more flexible system. Dataflow computers are a well known machine architecture, however little known work has been undertaken to determine the performance of pure machines on current hardware technologies. An investigation is required to accurately measure their performance using current PLDs. In particular, an assessment is needed to determine whether the overheads traditionally associated with dataflow architectures can be made acceptable.

A high-level hardware description language (HDL), Handel-C, was utilised to describe a hybrid class dataflow computer called CSIRAC II. The CSIRAC II dataflow computer is well specified at a functional level. However, a move from an architectural specification to an actual hardware implementation requires numerous design considerations. This paper details register transfer level (RTL) handling of operations not considered in the functional level design. Performance and resource tradeoffs are discussed and the subsequent resource utilisation results presented.

The performance of CSIRAC II running three benchmarks is discussed. Results indicate that the architecture is capable of extracting implicit concurrency from light workloads whilst maintaining high processor utilisation levels.

2. DATAFLOW COMPUTING

The original concept of the dataflow graph is usually credited to Karp and Miller in their 1966 paper [23]. This work was later built on by Dennis [10, 11, 12] to form a dataflow schema. Much enthusiasm and research in the area followed, however the initial expectations of the paradigm were never fully realised. Dataflow machines promised simple representation of parallel computations, performance limited only by data dependencies and implicit exploitation of parallelism [28]. However, practical considerations revealed the need for complex matching hardware, high bandwidth interprocessor networks, inefficient structure handling and a high overhead for fine-grained parallelism.

Dataflow computing is based upon the direct execution of a dataflow graph (DFG). A simple example of a DFG is shown in Figure 1. A DFG is a directed graph comprised of *nodes* which represent the instructions to be executed and *arcs* that represent the data-dependencies between nodes. Each node generally has either one or two *inputs* depending on whether the node is *monadic* or *dyadic* respectively. Data in the form of packets or *tokens* flow down the arcs. A node executes or *fires* once the tokens directed at each of its inputs have arrived. Generally, when a dyadic node fires, one token is absorbed from each input and one or more tokens are generated on the output. In this way a dataflow computer is said to be data-driven; instructions are executed once data dependencies are satisfied.

This paradigm differs greatly from the widely implemented von Neumann architecture. von Neumann based machines are required to adhere to an execution order specified by a compiler and scheduled at run-time by a program counter. Parallelism is thus hidden by a typical von Neumann tar-

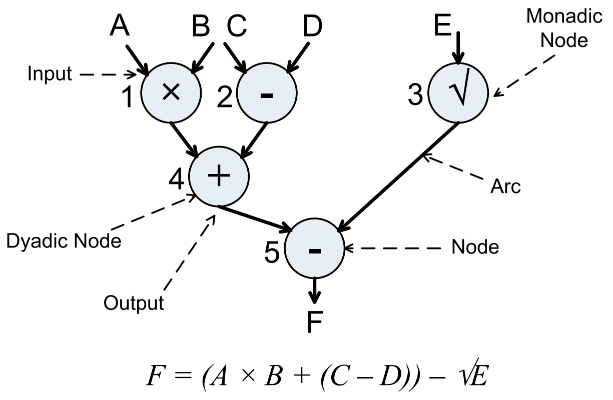

Figure 1: A simple dataflow graph

geted compiler because the data dependencies are expressed in a sequential stream of instructions. However by expressing a computer program in the form of a dataflow graph, the parallelism is inherently exposed. For example, in Figure 1, nodes one, two and three are all independent operations, thus these nodes can all fire simultaneously. Concurrency is extracted by a dataflow computer when more than one node fires simultaneously. Each node is a single instruction and thus dataflow computers expose fine-grained parallelism.

2.1 Dataflow Classes

The static class of dataflow architecture was suggested by Dennis in the mid 70's [13]. The distinguishing feature of static machines is that only a single token may be present on any arc. To achieve this, acknowledgement tokens are sent from the destination node (consumer) to the original firing (producer) node. The firing rule of a static machine thus becomes: a node can fire once each of its inputs become available *and* there are no tokens on any of its output arcs. One token per arc facilitates simple matching and compile time allocation of resources for each arc [22]. Arguably, this advantage is outweighed by the classes inability to expose the maximum available concurrency. Loop constructs can only execute in a pipelined fashion because each iteration needs to wait for an acknowledgement token. Additionally, acknowledgement tokens increase token traffic throughout the machine by up to a factor of two [6]. The static class restricts programming flexibility. Recursion is not supported because there can only be one invocation of a function at any particular time.

In order to overcome some of the problems associated with static dataflow machines, *dynamic* or tagged token dataflow machines were introduced. Groups based at the University of Manchester led by Gurd [21] and at MIT led by Arvind [7] began work on dynamic machines in the late 1970's. The primary distinction between the static class and dynamic class of dataflow machines is the support of multiple tokens per arc and as a result; recursion and multiple function invocations. To facilitate these additional features the dynamic class utilises tags or *colours*. A colour is essentially a context; a field carried by a token to differentiate it from other tokens produced by different instances of the same node. Because dynamic machines can differentiate between multiple instantiations of the same node, loops may execute in parallel thus exposing the maximum available concurrency. This however comes at a cost; a highly complex *matching unit*. The matching unit or matching store is responsible for enforcing the firing rule in dynamic machines. It determines when the inputs for a particular node are available and whether the colours match.

2.2 Other Important Dataflow Developments

This paper focuses on static and dynamic classes of dataflow as they are most relevant to CSIRAC II. However, there were several other important developments in dataflow architecture. In particular, Papadopoulos and Culler introduced the concept of the *Explicit Token Store* (ETS) [26, 9]. The ETS was introduced in an attempt to reduce the overheads associated with hash based matching units. Matching space within ETS was statically allocated at compile time and accessed through pointers. Also of note was P-RISC [6, 25]. P-RISC combined features of dataflow and standard RISC architecture in order to provide a more subtle transition from RISC like architectures to pure dataflow machines.

3. THE CSIRAC II DATAFLOW COMPUTER

The CSIRAC II dataflow computer is part of a research project that began at Manchester University in the 1970's. In 1986 the work was continued as part of the joint parallel systems architecture project at CSIRO and RMIT. In the early 90's the work was carried on at the Swinburne Institute of Technology and now again in 2005 to the present date at Monash University. Detailed material on the architecture can be found in [20, 5, 3, 14, 4].

3.1 System Overview

CSIRAC II is a hybrid class dataflow computer. The hybrid class combines the functionality and advantages from a modified static class and the dynamic class. The static class is modified to alleviate the need for acknowledgements and their associated overheads. This is achieved by allowing tokens to queue on arcs. If a token is directed at an arc that already contains a token destined for the same input, it is placed in a FIFO queue to maintain temporal ordering. For this reason, the modified static class is named static queued. CSIRAC II utilises a single tag to support the dynamic class of execution. The hybrid model facilitates the ability to utilise tagging only when required. Thus, if a particular graph is not coloured, the overhead of tagging is removed. In contrast to other dynamic class implementations, CSIRAC II maintains the arrival order of tagged tokens. A queue is maintained for each different tag, eliminating the need for data reordering.

CSIRAC II consists of up to 256 processors connected via a high speed interconnection network, although only four processors are simulated here. Similar to other machines, each processor consists of a matching unit and an evaluation unit. The matching unit forwards tokens destined for monadic nodes and determines a match for those tokens destined for dyadic nodes. The evaluation unit executes the instruction set and distributes tokens to their destinations.

Tokens are 130 bits long and consist of a control field, name, tag and data, see Figure 2. A name is a grouping of a processor field, process field, node field and two bit fields that indicate whether the token is destined for a monadic node and to which input. The processor field indicates the

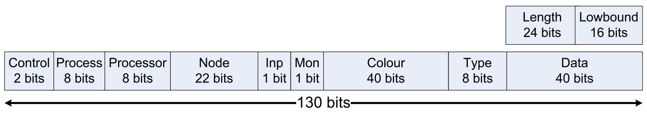

Figure 2: Token format

destination processor while the process field facilitates the simultaneous execution of multiple graphs (identifying tokens from each). The tag field is a single context identifier and is intensionally large to avoid the need for token recycling. Finally, the payload of a token is a data field of 40 bits in conjunction with an 8 bit type field. The data field is large enough to hold most data types including tags, integers and single precision floating point. If a datum is larger than 40 bits, a multi-word token can be formed such that the name and tag fields do not need to be copied many times. The first word contains name and tag fields, however length and lower bound fields replace the data field. The data is subsequently packed into 128 bit words. Each 128 bit word is accompanied by a 2 bit control field which indicates the header, body and end of multi-word tokens. This facilitates support for vector and compound (record) tokens whilst reducing tagging overhead and network traffic.

When a token arrives at a processor it is placed in a queue called the input queue. The input queue stores unmatched tokens arriving from the network. Sometimes the matching unit cannot match tokens as fast as they arrive from the network. This occurs particularly during long bursts of dyadic nodes. The input queue acts as a buffer to smooth out such bursts of tokens. Similar to the Manchester machine, CSIRAC II implements an evaluation queue. This queue is located between the output of the matching unit and the evaluation unit. Some functions in the evaluation unit consume more than a single clock cycle. Thus, during bursts of monadic tokens from the matching unit, the evaluation unit may become overwhelmed. The evaluation queue buffers this effect and can also reduce starvation if a sufficient buildup of tokens is available to hide high latency periods in the matching unit.

3.2 Matching Unit

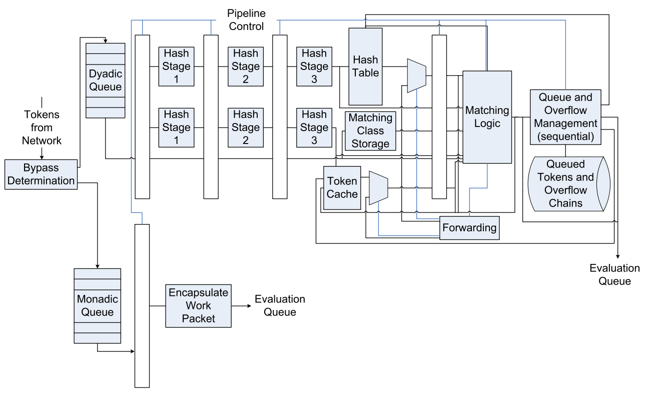

Figure 3: The matching unit

The matching unit consists of a monadic pipeline and a dyadic pipeline coupled with sequential modules which handle queuing (see Figure 3). Rather than implementing a single local queue, two queues were implemented; a dyadic queue and a monadic queue. The first pipeline stage is shared between the monadic and dyadic pipelines; it determines whether the token arriving at the processor is a bypass or not, and enqueues the corresponding queue. At the cost of managing two queues, bypass tokens can completely bypass the matching unit, irrespective of the state of the dyadic pipeline. The bypass pipeline is both short and simple. It consists of two stages; one to dequeue the bypass queue and one to encapsulate a work packet and enqueue it onto the evaluation queue.

In the dyadic case, an associative search is required to determine whether a matching token is present. The key fields (node and tag) are too large to utilise a fully associative memory, thus a hashing technique is implemented. The hashing function implemented is the same function implemented in the CSIRAC II simulator, completed in previous work [17]. The decision to pipeline the hash function into three stages was made to ensure high clock frequencies were achievable.

The token cache is implemented in block RAM as a two-way set associative cache. The hash table is similarly implemented in block RAM. A read issued to either of these memories incurs a single clock cycle latency before the data is ready. Consequently, a read is issued one pipeline stage before the data is required. Forwarding is required to avoid read after write hazards for both the token cache and the hash table. Stage four of the dyadic pipeline performs the final portion of the hashing function and then subsequently reads the hash table, token cache and matching class store. The matching class store holds the matching class definition for each node. Upon graph loading, the class of each node is written into the store which is a directly addressable block RAM.

Instead of using a combination of the node and colour fields directly, this implementation indexes the cache by the result of a hashing function. The same hashing function that indexes the hash table is utilised for the index of the cache. This scheme ensures that the cache is utilised effectively in a machine where mixes of uncoloured and coloured tokens exist.

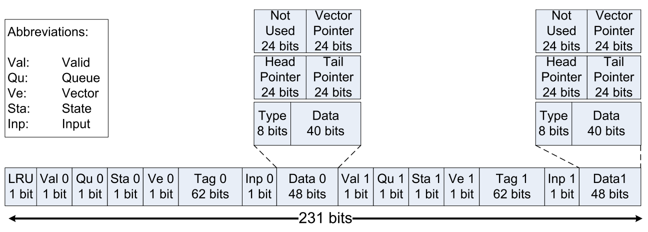

Figure 4: Cache line format

Each token cache line is 231 bits wide, see Figure 4. The majority of the space is consumed by the cache tag field which is 124 bits wide (total for both sets). 48 bits is reserved for data which can be stored in three different configurations. A standard type and data field, a head and tail pointer to a queue or a single pointer to a multi-word token may be present. Each set has a corresponding input bit that indicates which input the cached token(s) is directed to. A single LRU bit is implemented which, if asserted, indicates that set two is the most recently accessed set. Finally, each set has three state associated bit flags. These bits indicate how the data field should be interpreted. An asserted 'queue' bit indicates that pointers to a queue of tokens is present, whilst the 'vector' bit indicates a multi-word token is stored. The 'state' bit is used in conjunction with the matching class to determine the state of a stored token. In this implementation the only non-standard matching class utilised is *protect*. The protect matching class forwards the first token arriving on input zero, but then protects the input until a token arrives on input one. The state bit should facilitate the addition of most other complex matching classes if future expansion is required.

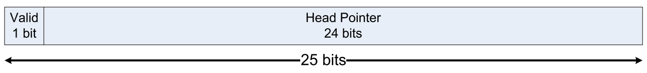

Figure 5: Hash table entry

Stage five of the pipeline is responsible for determining whether a match has occurred. Each valid set in the token cache line fetched by stage four is matched against the incoming token. A valid bit is maintained for each set which identifies whether the data payload is current (valid). A match is thus determined by ensuring the valid bit is true, that the node and colour fields match and that the incoming token is directed at a different input than that of the cached token. If a match is determined, a work packet is encapsulated and the evaluation queue is enqueued. The valid bit corresponding to the set whose token was removed is set to false and the LRU bit is updated. This process constitutes what is defined here as a cache read hit. Several different sequences are required if an incoming token does not obtain a hit with a stored token in the cache. The hash table entry that was fetched at stage four is inspected. Each hash table entry contains a valid bit and a pointer to main memory, see Figure 5. The valid bit indicates whether an overflow chain is present. If the valid bit is not set then the incoming token may be inserted into the cache (cache write hit). However, if the associated cache line already contains two valid sets then one of the sets must be retired to main memory. The LRU bit determines which of the sets will be retired. The set is then placed at the head of an overflow chain which is pointed to by the pointer in the hash table. The valid bit is also then set in the hash table. The incoming token can then be safely inserted into the cache and the LRU bit is updated. In the case that the valid bit is set in the hash table, before the incoming token is inserted into the cache a search of the overflow chain must be undertaken. If a match is located in the overflow chain then the stored token is removed and the node is fired. If a corresponding queue is located then the incoming token is placed directly into the overflow chain.

3.3 Overflow Chain Handling

The matching unit main memory has been structured such that it is as wide as the largest single data field it is required to store. This has the effect of minimising the number of memory accesses required for the largest data entries, however, it also introduces some storage overhead for data entries that do not consume the entire memory word length. The four different data configurations which can be stored in main memory are depicted in Figure 6. Every data entry in main memory is part of a linked list and as a result, the 24 least significant bits of each of the data configurations is a pointer. The pointer is 24 bits because this is the maximum length that the cache can store without significant wastage. The widest data entry in main memory is a 128 bit word of packed multi-word token data. Thus main memory is configured as a 152 bit wide memory. By organising the memory in this way, four 32 bit vector elements may be stored and retrieved simultaneously (in one memory access) and passed to the evaluation unit.

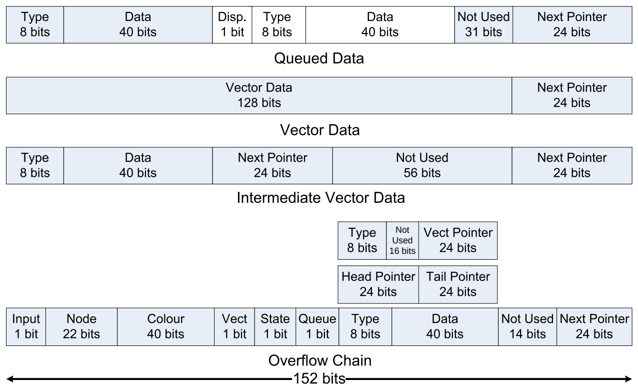

Figure 6: Main memory fields

Overflow chain entries contain the same fields as a cache set except they also store a pointer to the next chain entry. Each entry either holds its data payload directly or in the case of a queue or multi-word token, via pointers. The formatting of the entries is general enough to allow overflow chains with queues of multi-word tokens. Such structures are supported by storing a pointer in the overflow chain data field that points to an intermediate word in memory. This intermediate word then contains the length and lower bound fields of the multi-word token, a pointer that points to a list of the token data and another pointer that points to the next intermediate word in the queue.

Queue and overflow chain maintenance are not pipelined, rather they are performed sequentially. This is because multiple main memory accesses are required. Main memory must be allocated and deallocated by the hardware whenever a queue or overflow chain is created, destroyed, expanded or reduced. As such, a dynamic memory allocation scheme is required. Such a scheme must be fast as allocation and deallocation will occur often, particularly when tokens are queued which is expected to occur more often than overflow chain generation. A simple list of free memory locations is maintained with a global pointer that points to the first location. The memory is initialised so that each adjacent memory location points to the next in increasing order. Allocation is then simply performed by updating the global pointer by reading the address which is stored in the memory location pointed at by the old global pointer. The word to be stored is also written to the address pointed to by the old global pointer. Thus the allocation and writing of one word of memory consumes one read and one write. Similarly, the cost of a deallocation after a memory read is one read and one write. Initially, the data is read from the requested address. The global pointer is then stored in the requested address and the global pointer is updated to the address being deallocated.

3.4 Evaluation Unit

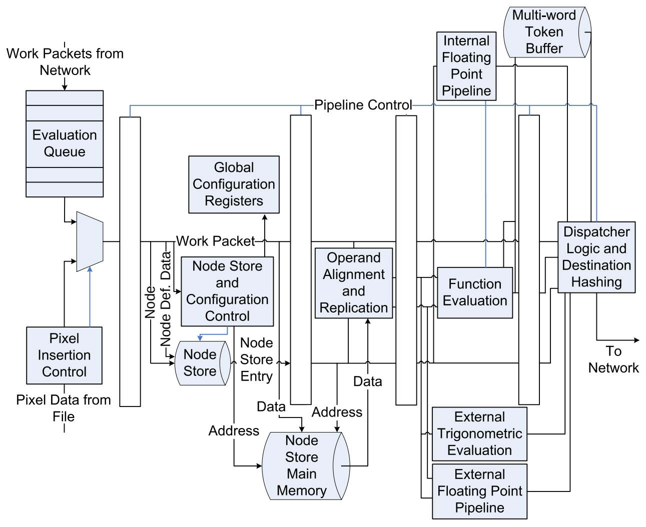
Work Packets from Network
Evaluation Queue
Pixel Insertion Control
Pixel Data from File
Pipeline Control
Global Configuration Registers
Work Packet
Node
Node Def. Data
Node Store and Configuration Control
Node Store
Node Store Entry
Data
Address
Node Store Main Memory
Operand Alignment and Replication
Data
Address
Internal Floating Point Pipeline
Multi-word Token Buffer
Function Evaluation
External Trigonometric Evaluation
External Floating Point Pipeline
Dispatcher Logic and Destination Hashing
To Network

Figure 7: The evaluation unit

The evaluation unit consists of a five stage pipeline coupled with numerous floating point and integer function pipelines. It also houses the node store, evaluation queue and multi-word output buffer, see Figure 7.

3.5 Work Packet Fetch and the Node Store

The evaluation queue is dequeued in stage one — if the queue is empty, a counter is incremented to keep track of evaluation unit starves. The evaluation queue is currently implemented in block RAM as a FIFO buffered channel with 8,191 entries[1]. However, this storage space may be moved to an external queue chip in order to minimise the amount of block RAM required. Each work packet facilitates two operands each with four 32 bit words (multi-word tokens). This allows vector operations which quadruple the maximum throughput of the evaluation unit (in comparison to work packets that only contain single word operands).

The node store is implemented as a directly indexable block RAM. The node field is extracted from the work packet in stage one is used to index the node store at stage two which then produces a node store entry in stage three one cycle later. There are 32,768 entries in the node store, supporting the same number of nodes in the graph being executed. Each entry contains a literal bit and a trace bit which indicate whether a literal is present or whether detailed execution information should be reported respectively. An 8 bit function field indicates which instruction is to be executed. The number of destinations is also stored in an 8 bit field, although this implementation only supports up to three destinations without a literal or up to two destinations with a literal. The field is maintained at its originally designed length to maintain compatibility with the assembler and future additions to the architecture. Finally, three destination name fields are present, although the third can either hold a 40 bit name or 48 bit literal field — see Figure 8. Multi-word literals are supported through an additional block RAM storage called the node store main memory (NSMM). The NSMM is 128 bits wide and can store up to 1,024 128 bit multi-word tokens. If a multi-word token literal is present, the node store entry stores a pointer to the location in the NSMM where the literal data resides. A length field is also present in the node store entry which indicates how many NSMM words the literal consumes. Because memory in the NSMM is only allocated at graph load time, it is static. Thus each word can be fetched by incrementing the address from the starting pointer in the node store entry until the number of words indicated by the length field have been fetched.

[1]Powers of two are avoided in FIFO buffers as Celoxica indicates a higher latency for these lengths

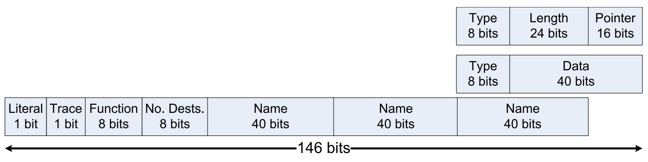

Figure 8: A node store entry

Stage three of the pipeline aligns arguments according to the input point specified in the work packet. However, it also handles the replication of single token literal arguments when they are being executed against a multi-word partner. Similarly, it replicates single token arguments against multi-word literals. The fetching of multi-word literals is also controlled by stage three. The functionality of the CSIRAC II instruction set is dependent on the type of arguments. Replication of arguments is required to ensure the correct function is executed. For example, the multiply instruction can either multiply two single arguments together, two vectors or a vector and a stream of replicated single tokens.

3.6 The Execution Stage

Stage four of the evaluation unit pipeline is responsible for executing the instruction set of CSIRAC II. In this implementation 78 instructions have been implemented. Most instructions execute in a single cycle, however complex multi-word token manipulation, sequencing and floating point operations consume several cycles. Floating point is implemented through the Celoxica floating point library. Multiply, divide, add, subtract, cast to integer and six comparison functions are implemented. Each of these (except comparisons) are fully pipelined and operate in parallel with the evaluation unit pipeline. Floating point operations are initiated in stage four of the pipeline. The evaluation unit continues executing instructions whilst the floating point instruction progresses through the relevant floating point pipeline. One completed, the floating point pipeline asserts a flag to indicate a result is ready. In the next available clock cycle, stage four of the evaluation pipeline forwards the floating point result to stage five. A round robin scheme is implemented to prioritise the switching between multiple floating point pipelines with results ready. In addition to the standard floating point instructions mentioned above, a 16 stage integer divider pipeline is implemented in a similar manner, as are the Sine and Cosine functions. The Sine and

Cosine functions however are not pipelined, they are sequential. The Celoxica fixed point CORDIC based trigonometric library is utilised to implement these functions.

Some multi-word token manipulation functions require one of the arguments to be buffered. For example, the 'form compound' function appends two arguments together to form a compound token. If both arguments are vectors, argument one must be buffered. This is because the incoming arguments arrive together (2 × 128 bit words) but they must be transmitted in serial (1 × 128 bit word). Any node that has more than one destination associated with it must also buffer its result if the result is a multi-word token. This is because the entire multi-word token must be sent to each one of the destinations in serial. For these reasons a multi-word token buffer was implemented in the evaluation unit. The buffer is stored in block RAM and has space for 256 entries (although this is easily adjustable depending on device resources).

The final stage of the evaluation unit pipeline (stage five) controls the transmission of tokens to the network. If one destination is present for the current node then the evaluation pipeline runs uninterrupted. In the event of a node with two or three destinations, the pipeline is stalled for one or two cycles respectively (output buffering is pushed through to the interconnection network input). This minimises the block RAM required for buffering within the evaluation unit and places the burden on the interconnection network, see Section 3.7.

3.7 Interconnection Network

A simple but fast interconnection network was implemented to facilitate inter-processor communication. In order to ensure the network does not become overwhelmed with tokens, input buffering is implemented through the use of queues in block RAM. As a token arrives at an input in the network FPGA, it is placed in a FIFO queue corresponding to its destination processor, see Figure 9. Each input on the network FPGA has its own grouping of FIFOs. Using this scheme and assuming there is always a token buffered for any particular destination processor, one token is guaranteed to be written to each output on every clock cycle.

Graph loading is performed by the network FPGA. It is envisaged that the graph to be executed would be available to the network FPGA either through a connection with a host machine or an off-chip non-volatile memory storage. A logic module on the network FPGA reads the memory or input connection and distributes nodes and priming tokens to all of the processors. An internal communication control code is attached to each token to indicate to the matching unit that the payload is either graph or machine configuration information. The matching unit then encapsulates this information into a work packet and enqueues it into the evaluation queue. Matching class information is also stored directly in the matching unit. The evaluation unit stores the node definitions into the node store and the node store main memory. In addition to node definitions and priming tokens, machine configuration is also passed from the graph. In particular, auto streaming, insertion rate and the archaic flag are defined by the graph. Auto streaming and insertion rate turn on automatic streaming from an external data source at a particular rate (samples per clock cycle). This is used primarily for streaming large data sets like pixels from an external source directly into processor number zero. The archaic flag indicates the old instruction set is being utilised; this changes the functionality of some instructions.

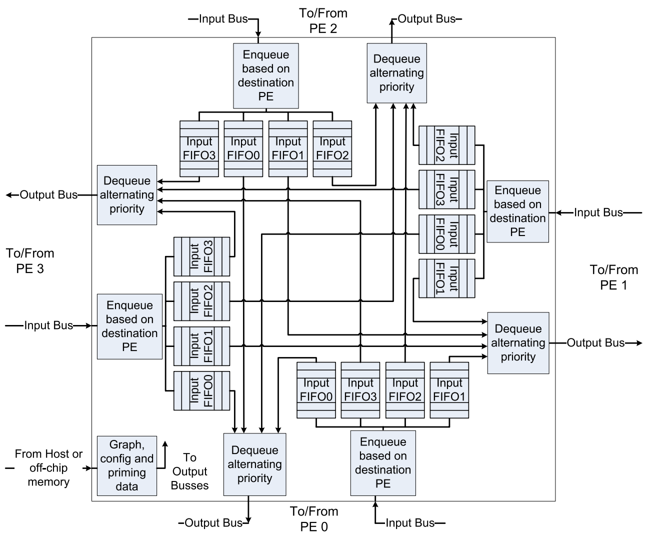

Figure 9: The interconnection network

3.8 System Summary

Table 1 summarises the storage capabilities of the implemented system. For the purposes of simulation, all of the tabulated storage units have been constructed from block RAM. When implemented on a suitable hardware system, it is expected that the dyadic and evaluation queues will be moved to external queue chips. The matching unit main memory is also expected to be moved to an off-chip memory. Relocating these storage units not only frees up valuable block RAM, it also allows them to be significantly expanded in capacity. The current storage capabilities would however fit entirely into block RAM (without external memories) if the processor was spread across two large FPGAs[2] (one for the matching unit and one for the evaluation unit).

Monadic Queue Length	4,095
Dyadic Queue Length	16,383
Matching Class Store Entries	32,768
Hash Table Entries	32,768
Cache Lines	8,192
Matching Unit Main Memory Words	32,768
Total Matching Unit Memory (Bits)	10,551,036
Evaluation Queue Length	8,191
Node Store Entries	32,768
Node Store Main Memory	1,024
Vector Buffer Entries	256
Total Evaluation Unit Memory (Bits)	7,078,140
Total Memory Bits	17,629,176
Total block RAM Bits (with off-chip mem.)	8,388,990

Table 1: System storage specifications

[2]This is an approximation that depends on the target device and how effectively the vendors fitter allocates individual block RAMs. However it provides a crude idea of the amount of block RAM space that would be required for these specifications.

In order to gain hardware resource utilisation estimations, the Handel-C code was synthesized to an EDIF netlist. Table 2 summarises the results. A Xilinx Virtex 4 was chosen as the target device as it is the most recently released FPGA supported by DK4. The Handel-C technology mapper is aware of the Virtex 4 architecture and can thus provide an accurate LUT count. Recent research determined that an Intel Pentium processor consumed approximately 50% of one Xilinx Virtex 4 LX200 FPGA[24]. Results from this work indicate that one CSIRAC II processor would consume approximately 30% of the same device.

System Module	Virtex 4 LUTs	% of Total
Matching Unit	16,221	27.17%
Evaluation Unit	25,239	42.27%
Floating Point Unit	18,246	30.56%
Single Processor Total	59,706	100.00%

Table 2: Single processor resource utilisation

System Module	Virtex 4 LUTs
4 Processors	238,824
Interconnection Network	16,385
Four Processor Total	255,209

Table 3: Four processor resource utilisation

Table 3 depicts the resource requirements for four processors and the interconnection network described in Section 3.7. The total LUT count exceeds the resources available on the largest Virtex 4 LX device (LX200) [1]. However, the logic of the design would most likely fit into the largest Virtex 5 LX device (LX330), which Xilinx estimates to contain 331,776 LUTs[3] [2]. There is insufficient block RAM on a single Virtex 5 LX330 to implement four processors and the interconnection network. Consequently, off-chip memories would be required. The Virtex 5 LX300 device has 1,200 I/O pins or 300 per processor. This is also insufficient to implement memory busses as wide as those assumed in this work. Bus widths could be reduced and accessed multiple times to reduce the number of required I/O pins. This would have the effect of reducing the performance of the system, however would maintain the flexibility and convenience of a single FPGA implementation. This may be an acceptable price to pay for an implementation targeted at evaluation, eliminating the need for five FPGAs (one for each processor and one for the interconnection network).

4. SIMULATION RESULTS

This section presents detailed cycle accurate simulation results on a four processor implementation of the CSIRAC II dataflow computer.

4.1 Benchmark Overviews

The first benchmark investigated was a portion of the Canny edge detection algorithm [8] written in i2 [19]. In particular, the first two steps of the algorithm were implemented. One 5×1 and one 1×5 convolution mask are first applied to an incoming image stream. These vector masks are a discrete approximation to a Gaussian function and are applied to reduce noise in the original image. Next, two 3×3 masks are applied to estimate the gradient in the `x` and `y` directions. The edge strength is then computed by summing the absolute values of the two gradient components. In order to keep simulation time reasonable, a 128×128 image is streamed into processor zero using the auto-stream functionality described in Section 3.7.

The second benchmark under consideration is the solution of shallow water equations. This benchmark is an iterative 2D grid based application that models a square layer of fluid. In particular, equations involving fluid velocities, pressure, field height, cartesian mass fluxes and potential velocity are modeled [27]. An 8×8 grid of reals is utilised and 11 iterations are executed. The program is written in Pascal [16].

The final benchmark presented is the numerical computation of π. This is a program written in Pascal that demonstrates the recursive performance of CSIRAC II. The program computes π through rectangular integration using a fixed number of rectangles [15].

4.2 Single Processor Results

The single most important measure in determining the performance of the matching unit is the evaluation unit utilisation (EUU). The EUU is determined by recording the proportion of clock cycles in which the evaluation unit is kept busy. The evaluation unit is fully pipelined and thus can produce a result token each clock cycle. If CSIRAC II can keep it busy, the overheads usually associated with matching units have been made acceptable.

	Canny	Shallow	Pi
Clock Cycles	1,443,351	15,086,251	305,051
Nodes Fired	1,053,959	1,240,407	149,516
Monadic Fires	68.82%	45.66%	53.43%
Tokens Generated	1,417,394	11,459,281	223,241
Multi-word Tokens	0.00%	83.12%	1.84%
Max. Main Mem.	2,085	10,918	14,240
Max. Cache Sets	19	4,011	12,310
Avg. EUU.	98.28%	95.74%	83.49%

Table 4: Single processor benchmark summary

Table 4 summarises the single processor benchmark results. In particular it can be seen that the EUU for all three benchmarks is very high.

The average EUU for the Canny benchmark was determined to be approximately 98%. Several factors contribute to this high utilisation. Approximately 69% of the nodes fired were monadic, significantly more than the monadic/dyadic mix observed in the other benchmarks. This is due to the large amount of replication required — each incoming pixel must be copied numerous times because at some time they must appear in each cell of each mask. Such a high proportion of monadic fires helps to keep the evaluation unit busy as bypass tokens do not require matching.

All of the accesses in the matching unit were performed in cache — no overflow chains were formed (see Table 6). This is the expected result because the graph is not coloured. Thus the cache is indexed directly by node, and given there are significantly less nodes in the graph compared to cache lines, no retires will occur. However, a significant propor-

[3]The Virtex 5 resource count is a Xilinx estimate; the Virtex 5 uses six input LUTs whereas the Virtex 4 uses four input LUTs

	Canny	Shallow	Pi
Avg. Mon.	1.49	1.70	29.99
Max. Mon.	14.00	92.00	1,608.00
Avg. Dya.	452.92	262.45	3,419.50
Max. Dya.	920.00	2,237.00	13,897.00
Avg. Eval.	46.89	128.40	2,226.90
Max. Eval.	1,302.00	1,673	7,720.00

Table 5: Queue Lengths

	Canny	Shallow	Pi
Cache Read Hits	7.87%	39.90%	42.46%
Cache Write Hits	11.79%	43.67%	40.02%
Cached Queue Accesses	80.35%	11.63%	0.00%
Cached Prt Accesses	0.00%	3.25%	0.74%
Total Cached Accesses	100.00%	98.45%	83.22%
Cache Retires	0.00%	0.44%	6.80%
Failed Chain Searches	0.00%	0.54%	3.18%
Successful Chain Searches	0.00%	0.56%	6.80%

Table 6: Cache Performance

tion (80%) of the accesses were cached queue accesses. This is expected as one line of the image must be constantly queued, ensuring the data dependencies are satisfied for the 3×3 mask. Such a proportion of queued accesses places a burden on the matching unit as continuous main memory accesses are required. This is supported by the high average dyadic queue length — approximately 453 tokens deep.

The queue lengths observed (see Table 5) indicate that the load on system storages remain well within the limits of the machine (specified in Table 1). An initial spike in the evaluation and dyadic queues occurs due to the proliferation of priming tokens in the initial stages of the computation. If the benchmark was scaled up to higher resolutions, the matching store main memory would become highly utilised. This is a consequence of queuing at least one horizontal line of pixels to satisfy data dependencies. However, it is expected that the main memory would be moved to an off-chip RAM in a full hardware implementation, expanding its capacity and ability to process higher resolution frames/images.

Shallow is the closest benchmark to a real application; it consists of several procedures which all perform a reasonable amount of work. This is apparent from the queue and memory statistics. At one point in the computation, approximately 33% of the matching unit main memory is consumed. Maximum queue lengths grow into multiple thousands, although remain well within the limits of the machine, specified in Table 1. Despite the modest storage capacities, reasonably complex computations can be comfortably performed.

A large percentage (83.12%) of the tokens transmitted are multi-word tokens. This is because the grid is maintained as one compound token. The single processor average EUU is approximately 96%. Thus the use of multi-word tokens has not detrimentally effected the machine utilisation. This indicates that multi-word token maintenance within the machine is working effectively and efficiently.

A small percentage of matching unit accesses involve overflow chains (1.5%). This is not unexpected due to the size of the benchmark. There are 3,374 nodes in the graph and a great number of unique colours are utilised. This places a high load on the token cache. Despite the overflow chain accesses, the latencies involved in matching are not sufficient enough to greatly effect the EUU.

The single processor EUU for the Pi benchmark was measured to be approximately 83%. Whilst this is a reasonable result, either the graph or the machine is holding back the potential performance. An observation of Table 6 indicates that approximately 17% of accesses to the matching unit involve overflow chain searching or manipulation. Despite the small number of nodes in the graph (102), many colours are generated due to the recursive nature of the algorithm. Consequently, some sets in the cache are retired to an overflow chain when more than two node/colour fields hash to the same cache line.

A small percentage of cache retires only accounts for some of the EUU reduction. The final part of the computation does not involve any monadic fires. This is because the base case has been reached; the results are getting passed back to each invoking context. The tokens being matched are the return context addresses that were generated as each level of recursion was invoked. The evaluation unit has little to compute during this period, only the summation of the rectangle areas. Thus the reduction in utilisation is due to an artifact of the tagged recursion graph which causes a long burst of dyadic fires at the end of the computation. The overhead involved in the recursive graph isn't prohibitive in this benchmark. It could however be made less predominant if the amount of work computed at each level of recursion was increased.

This benchmark places a significant load on system storage units. The dyadic and evaluation queues actually come close to overflowing and the matching unit main memory is 43% utilised. This is due to the massive exposure of parallelism. At the beginning of the computation the recursive tree is fully unraveled, exposing maximum parallelism. This benchmark demonstrates the need for throttling of graphs where there is far too much parallelism for the number of processors involved. Such a throttling mechanism is part of the CSIRAC II architecture [18] however was not implemented here.

4.3 Multi-processor Results

Num. Proc.	Pi EUU	Shallow EUU	Canny EUU	Pi Cached Matches
1	83.49%	95.74%	98.28%	83.22%
2	92.31%	88.11%	94.07%	92.76%
3	93.39%	74.35%	89.32%	96.25%
4	93.50%	76.00%	88.42%	98.53%

Table 7: Multi-processor benchmark summary

Table 7 summarises the multi-processor results. In general, the EUU decreases as more processors are added due to a reduction in the work available per processor. This trend is reversed in the Pi benchmark due to added cache performance. The Pi results indicate that the average EUU actually increases with the addition of more processors. This is a result of distributing the load on the token cache across multiple processors and thus achieving a higher proportion of cached accesses. The speedup achieved is super-linear (see Figure 10) due to the increase in cache performance and consequently evaluation unit performance. This clearly demonstrates that linear speedup is easily achieved by the machine when sufficient parallelism is exposed.

Multi-processor results from the Shallow benchmark are impressive; the speedup achieved is not linear however four processors manages a speedup of well over 3×. The grid size utilised is very small. It is unlikely a conventional multi-processor would gain any speedup at all (due to the small amount of work computed compared with communications overheads). CSIRAC II demonstrates the ability to gain speedup in applications with little work. Inter-processor communication is an integral part of the architecture and thus incurs far less latency than that in a conventional multi-processor. If a little more parallelism was exposed through a larger grid size or by the compiler chain, this benchmark would scale much higher (beyond four processors).

The addition of multiple processors for the Canny benchmark reduces the average EUU due to a reduction in available work per processor. Despite the reduction in EUU, the speedup achieved over two, three and four processors is close to linear. A four processor system consumes 407,251 clock cycles to process 16,384 pixels. This equates to approximately 24.86 clocks per pixel, or if a clock frequency of 250 MHz is assumed, 10.06 million pixels per second. There is sufficient work in this benchmark to suggest that it would scale to at least eight processors. The queuing nature of the benchmark places a high burden on the matching unit. However, a high proportion of bypass matches combined with the efficient implementation of queuing within the matching unit maintains a high utilisation. Thus the benchmark demonstrates that high performance stream based computing is achievable on CSIRAC II and that the matching overheads involved are more than acceptable.

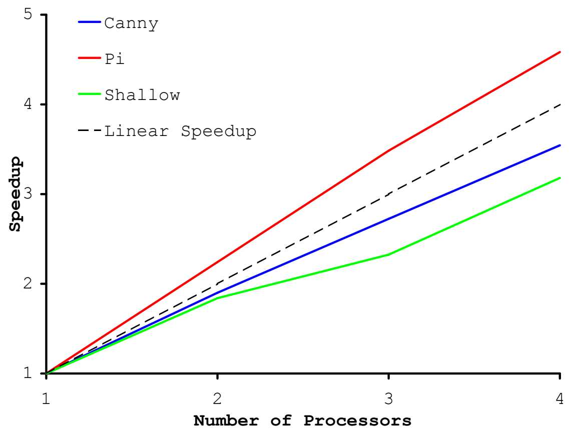

Figure 10: Multi-processor Speedup

5. CONCLUSION

The use of DK4 and Handel-C enabled the rapid development and verification of a complex architecture. It was found that Handel-C operates at a sufficiently low abstraction level in order to optimise complex machine architectures. However, it also provides sufficient higher level flexibility to eliminate verbose and repetitive expression of fundamental elements.

Dataflow computers are a well known machine architecture, however little known work had previously been undertaken to determine the performance of pure machines on current hardware technologies. Consequently, an investigation was undertaken to accurately measure their performance using current PLDs.

The results presented demonstrate that the architecture is capable of extracting implicit concurrency whilst maintaining high processor utilisation levels. The storage units within the processor were found to be sufficient for each of the benchmarks. However, image processing of higher resolutions or more complex, highly parallel graphs would require the expansion of the available queue and memory space. This is expected to be facilitated through off-chip memories in a complete hardware system. The multi-word token manipulation capabilities were demonstrated to work effectively and efficiently. The architecture was found to be capable of extracting parallelism from very light workloads. This was demonstrated through multi-processor speedups on small benchmarks. Such small workloads would be unlikely to show any speedup on conventional machines due to inter-processor communication overheads. Finally and most importantly, it was determined that current FPGA technologies can make the overheads traditionally associated with dataflow architectures more than acceptable. This was particularly evident in the stream based image processing benchmark which achieved close to linear speedup despite heavy queuing of tokens.

6. REFERENCES

[1] Virtex-4 Product Tables. Xilinx Incorporated, http://www.xilinx.com, 2007.

[2] Virtex-5 Family FPGAs. Xilinx Incorporated, http://www.xilinx.com, 2007.

[3] D. Abramson and G. Egan. The RMIT data flow computer: a hybrid architecture. *Computing Journal*, 33(3):230–240, 1990.

[4] D. Abramson, G. Egan, M. Rawling, and A. Young. The RMIT dataflow computer the architecture. Technical Report TR 112 061 R, Commonwealth Scientific and Industrial Research Organisation, 1990.

[5] D. Abramson and G. K. Egan. Design of a high-performance data-flow multiprocessor. In *Advanced Topics in Dataflow Computing*, pages 121–141, New Jersey, USA, 1991. Prentice-Hall.

[6] Arvind, L. Bic, and T. Ungerer. Evolution of data-flow computers. In *Advanced Topics in Dataflow Computing*, pages 3–33, New Jersey, USA, 1991. Prentice-Hall.

[7] Arvind, K. P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report TR-114a, University of California, Irvine, 1978.

[8] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[9] D. E. Culler and G. M. Papadopoulos. The explicit token store. Technical Report Memo 312, Massachusetts Institute of Technology, 1991.

[10] J. B. Dennis. Programming generality, parallelism and computer architecture. In *Information Processing '68 (Amsterdam)*, pages 484–492, North-Holland,

Amsterdam, 1969.

[11] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[12] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *Proceedings of the International Symposium on Theoretical Programming*, pages 187–216, London, UK, 1974. Springer-Verlag.

[13] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132, New York, NY, USA, 1975. ACM Press.

[14] G. K. Egan. The RMIT data flow computer: Token and node definitions. Technical Report TR 112 060 R, Royal Melbourne Instititute of Technology, 1987.

[15] G. K. Egan. Parallel computing: Easy as Pi. Technical Report TR 118 084 R, Royal Melbourne Institute of Technology, 1989.

[16] G. K. Egan. Some shallow experiences: The shallow water numerical weather prediction program. Technical Report TR 118 086 R, Royal Melbourne Institute of Technology, 1989.

[17] G. K. Egan. The CSIRAC II simulation suite. Technical Report 31-010, Swinburne Institute of Technology, 1990.

[18] G. K. Egan. The delta throttle. Technical Report 31-019, Swinburne Institute of Technology, 1990.

[19] G. K. Egan, M. Rawling, and N. Webb. i2: An intermediate language for the CSIRAC II dataflow computer. Technical Report 31-002, Swinburne Institute of Technology, 1990.

[20] G. K. Egan, N. J. Webb, and W. Bohm. Some architectural features of the CSIRAC II data-flow computer. In *Advanced Topics in Dataflow Computing*, pages 143–173, New Jersey, USA, 1991. Prentice-Hall.

[21] J. R. Gurd, I. Watson, and J. R. W. Glauert. A multilayered data flow computer architecture. Technical Report 80-3-1, University of Manchester, 1978.

[22] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[23] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Mathematics*, 14:1390–1411, 1966.

[24] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium; in a complete desktop system. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field programmable Gate Arrays*, pages 53–59, New York, NY, USA, 2007. ACM Press.

[25] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *ISCA '89: Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, New York, NY, USA, 1989. ACM Press.

[26] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, New York, NY, USA, 1990. ACM Press.

[27] D. F. Snelling. Experimental guidelines for a shallow mapping study. Technical Report 21, University of Leicester, 1989.

[28] T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, and S. Sakai. Dataflow computer development in Japan. In *ICS '90: Proceedings of the 4th International Conference on Supercomputing*, pages 140–147, New York, NY, USA, 1990. ACM Press.