Anaconda

A Real-Time Control-Flow/Data-Flow Hybrid Processor

Simon W. Moore

February 16, 1993

Abstract

Typically real-time applications are written in a multi-threaded language and are executed on single threaded control-flow (von Neumann) processors. A software scheduler is used to switch between the application's multiple threads, thereby simulating a multi-threaded processor. Unfortunately software scheduling is problematic if real-time constraints are to be guaranteed and therefore static (pre-determined) scheduling is often used even though this prohibits sporadic tasks.

Control-flow processors, being inherently single threaded, are unable to use concurrency to tolerate latency. Therefore, the faster control-flow processors become, the more they rely on caches to reduce memory access latency. Unfortunately, caches are also a significant source of temporal complexity, especially when scheduling is involved.

Data-flow processors are inherently multi-threaded and are able to tolerate latency. Unfortunately, current data-flow processors have very simplistic schedulers which prevent real-time requirements from begin met. Furthermore, assignment, and thus a sense of history which is vital for many real-time applications, is problematic.

Control-flow/data-flow hybrids have been proposed in the past but they do attempt the challenge of scheduling for real-time. This report presents a possible real-time control-flow/data-flow hybrid processor — Anaconda.

1 Introduction

The phrase "real-time" is used to describe a plethora of applications from *on-line* which just need to be fast to *control* applications which need to be hard real-time (i.e. time deadlines must be met). In between there are so called soft real-time systems where deadlines may occasionally be missed, for example, multi-media where the occasional video frame may be disposed of if the processor load becomes too high.

This report begins by identifying hard real-time processor requirements. Todays control-flow, data-flow and control-flow/data-flow hybrid processors are briefly introduced and assessed. Finally, a real-time control-flow/data-flow hybrid processor, Anaconda, is proposed and conclusions are drawn.

2 Hard real-time processing requirements

Performing any application requires more than just a processor. Real-time applications rely heavily on their input/output devices (from sensors, to actuators, and between other processing devices) and other facilities like timers. This section looks at the requirements of the whole processing system with particular emphasise on temporal predictability (i.e. the ability to compute in hard real-time).

2.1 Time, space and information

Temporal predictability

Hard real-time systems must have temporally predictable characteristics to ensure that input data may be collected and output data produced within specified time bounds. Therefore, it is essential that the computation time between input and output can be predetermined. Whilst computation can commence when input data is available, it is often essential that the production of output be delayed so that it appears within the required output time bounds.

Clock distribution

Space and time are inextricably linked by the laws of Physics in the form of spatial relativity [6] (i.e. timing signals take time to propagate which is proportional to the distance travelled in a given medium). Fortunately most distributed systems are spatially fixed, so that timing signal delays can be compensated via suitable hardware. The higher the timing clock frequency the more difficult it is to transmit the signal. Therefore, it is recommended to have high frequency, and thus higher accuracy, local time (at or near the processor's clock frequency which will be in the GHz region soon) and lower frequency global time (at most a few MHz). Since clock distribution paradigms could be included in any processor design the subject will not be discussed further. However, this does not seem to prevent real-time hardware designers from ignoring the problems.

Transmission and locality of data

The transmission time of data is partly a function of distance and the transmission medium but also (and often more significantly) a function of the interfaces it has to travel though, particularly buffering. To ensure temporal predictability, the communication mechanisms must be able to guarantee a minimum throughput. This is primarily a network problem rather than a processor paradigm problem and will not be discussed further.

To ensure sufficient performance, frequently used data must be stored locally. Intermediate results should be stored very locally (e.g. a register file).

Assignment and input/output

A sense of history is vital for many real-time algorithms. From a practical viewpoint this means that assignment is essential (i.e. the ability to order memory access transactions to each location).

Input and output (I/O) requires information to be read and written at the right times and in the right order. This reduces to the need for assignment and temporal predictability.

2.2 Performance

Parallelism and concurrency

Real-time systems have to interact with an parallel world. Software development is made conceptually simpler if parallel parts of the real world are controlled by corresponding parallel software routines. Whilst it would be possible to provide every parallel task with its own processor, this is usually not cost effective. However, simulating parallelism via scheduling is more cost effective and is acceptable provided temporal constraints can be met.

Latency and frequency tradeoffs

If a task takes input, performs a computation and produces output in a sequential manner then the maximum manageable frequency of input data is dependent upon the worst case latency (ℓ). However, if the task (T) is broken up into subtasks ($\{\tau_0, \tau_1, ..., \tau_n\}$) connected together in a pipeline ($\tau_0 \rightarrow \tau_1 \rightarrow ... \rightarrow \tau_n$) , then the input frequency is dependent upon the the worst case latency of one of the subtasks, $f \propto \max(\ell(\tau_0), ..., \ell(\tau_n))$, which is likely to be much less than $\ell(T)$. However, $\ell(T) < \sum_{i=0}^{n} \ell(\tau_i)$. Thus, it is possible to trade between latency and frequency.

Since some real-time applications require very high data throughput it is important to be able trade latency for frequency (e.g. using multiple processors in a pipeline). Supporting this requires high performance and reliable interconnection networks and an execution model which facilitates (preferably automatic) application partitioning.

Latency tolerance

Theoretically, as processors become faster, due to miniaturisation and other hardware technology improvements, memory can also become faster because it can be made from the same technology. However, faster processors typically demand larger memories because larger applications are written. Larger memories result in increased memory latency. However, by constructing large memories from smaller ones, and by pipelining the fan out and fan in of addresses and decoding signals to these memories, the access frequency can scale with processor speed. Thus, it is becoming increasingly important to be able tolerate memory access latency.

In practice static memories (SRAM) have scaled with processor speeds but dynamic memories (DRAM) have not. This is probably due to DRAM technology begin driven towards producing larger and cheaper DRAMs rather than improving throughput. This may well change in the future.

3 An assessment of control-flow

The wide scale academic and commercial development of the "general purpose" control-flow or "von Neumann" processor paradigm over more than 40 years has resulted in exponential performance improvements and wide spread use. This section assesses applicability of todays control-flow processors to real-time processing.

3.1 Assignment and input/output

The sequential nature of control-flow makes assignment and I/O relatively easy.

3.2 Latency tolerance

Main memory access latency is becoming an increasing problem as the performance increases. One or more levels of caching is typically employed to reduce the average memory access cost. Unfortunately the temporal characteristics of caches are dependent upon memory access patters which are difficult, if not impossible, to predetermine. This becomes even more complex if multiple tasks execute concurrently, unless each task has its own area of cache [5].

Another technique is to perform loads and stores concurrently with executing other instructions. However, data dependencies severely limit the latency which may be tolerated.

3.3 Latency and frequency tradeoffs

In order to exploit latency and frequency tradeoffs an application needs to be partitioned amongst a number of processors. This is difficult on control-flow processors and usually requires manual partitioning and insertion of message passing instructions. This makes adjustments to an applications execution performance difficult.

3.4 Concurrency

Control-flow processors are inherently serial. Concurrency is simulated by scheduling. The scheduling overhead prohibits fine grained concurrency unless a significant proportion of the processing time is to be allocated to scheduling. Whilst concurrency support may be added to the processors design (e.g. the Inmos Ltd. Transputer™ with its simple microcoded non-preemptive round robin scheduler and small context) the overhead is still significant. Of course the scheduler's temporal characteristics must be assessed; a process which is made particularly difficult when sporadic tasks are catered for.

3.5 Data locality

Control-flow processors are good at utilising very local memory (register or stack) to store intermediate results, thereby improving performance, although this impinges on concurrency. Longer lived data is kept in main memory. Distributed data is more of a problem and usually involves caches or explicate message passing.

3.6 Temporal predictability

The combination of cache unpredictability and simulated concurrency ensures that it is difficult to assess the temporal characteristics of a real-time application executing on a control-flow processor.

4 An assessment of data-flow

4.1 Introduction

Control-flow programs explicitly define the order of execution which ensures that data is manipulated in the required order. On the other hand, data-flow programs specify the data-dependencies and allow the processor to choose one of the possible orderings (see figure 1). Any instruction which has received its operands may be executed, thereby allowing parallel execution. Furthermore, logically separate data-flow routines may run concurrently without any need for software scheduling.

The next section presents a brief overview of different data-flow paradigms. Then the preferred paradigm, tagged-token data-flow, is assessed for its real-time applicability.

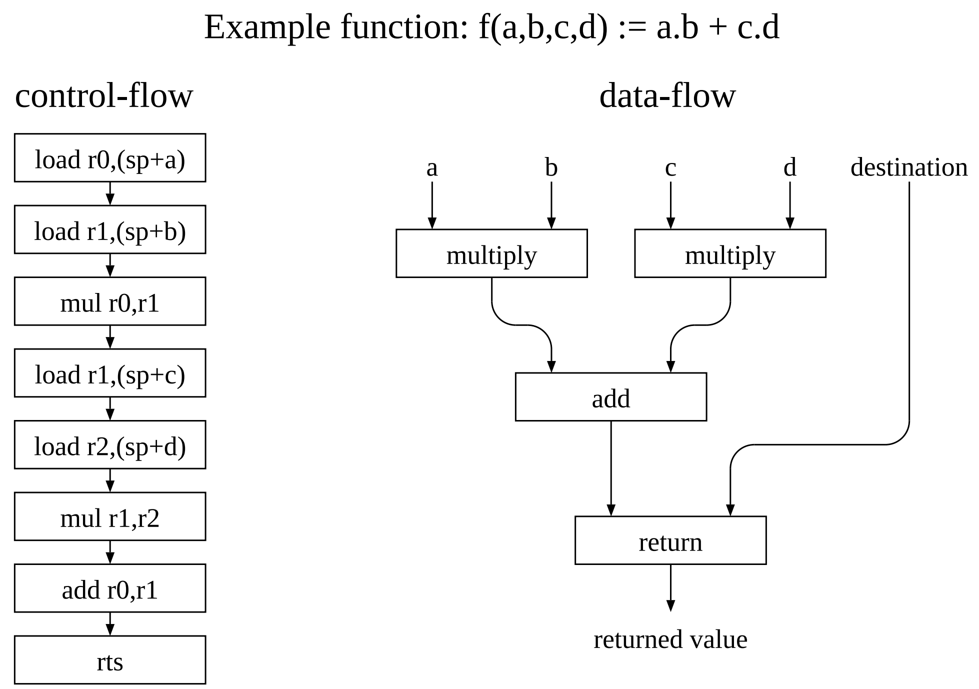

Figure 1: An example of control-flow and data-flow coding

4.2 Overview of data-flow

Static data-flow

With static data-flow [3] there is at most one token (datum) on an arc (path between one data-flow instruction and the next). The values on the arcs, or "operands", are stored with the instruction (see figure 2). To ensure operands are not overwritten before they are used there are backward signal arcs which inform previous instructions when the instruction has been executed (i.e. when the operands have been used and the destinations have been written to).

op-code	op1	(op2)	dest1 + dc1	(dest2 + dc2)	sig1	(sig2)

where () indicates optional parameters
op-code is the instruction identifier
op1 and op2 are the spaces for operands (op2 missing for monadic operations)
dest1 and dest2 are the destinations (dest2 being optional)
dc1 and dc2 are destination clear flags (initially clear)
sig1 and sig2 are the signal destinations (handshaking arcs)

Figure 2: Static data-flow instruction format

There are problems with static data-flow:

- Shared functions are difficult to implement because mutual exclusion must be enforced upon writing all of the operands to the function. This severely limits concurrency and is often inefficiently solved by replicating functions either statically or dynamically.
- The backward signal arcs double the number of tokens to be matched.

Coloured dynamic data-flow

With dynamic data-flow there may be many tokens per arc. In the coloured data-flow paradigm associated tokens are given an unique identifier, or "colour". This allows functions to be represented because each invocation of a function is given a unique colour. Only operands with the same colour may be matched in dyadic operations. An example of coloured data-flow is the Manchester prototype [4]. The RMIT hybrid [1] also utilises coloured data-flow to encapsulate functions but supports static data-flow within functions.

The main problems are:

- Matching colours is expensive and temporally unpredictable — often implemented using hashing.
- Uncontrolled fan-out can course a token explosion problem.

Tagged token dynamic data-flow

An alternative paradigm (used for the MIT Monsoon machine [10]) is to remove operand storage from the instruction and to place it in a data page, or "activation frame". Activation frames effectively store the context of a function. This may be implemented using a conventional flat memory store with the addition of an empty/full flag for each word in memory. The advantage is that each instantiation of a function has it's own separate activation frame which makes matching of operands easy:

- **instantiating a function** — obtain an unused activation frame and set all of the flags to empty.
- **matching a token to a monadic instruction** — the token's *statement pointer* (see figure 3) is used to lookup the instruction to be executed which forms an executable packet to be queued for execution.
- **matching a token to a dyadic instruction** — the token's *statement pointer* is used to look up the instruction to be executed. The instruction's r value (see figure 3) is then added to the token's *context pointer* value to form an address in the activation frame (i.e. r forms an offset into the activation frame). If there is a value in the activation frame then value is read and matched with the token's *data* value which is sent, with the instruction information, to be executed and the activation frame location is set to empty. If there was not a value in the activation frame then the token's *data* value is stored in the activation frame to wait for it's partner and the flag is set to full.

Example instruction format

op-code	(r)	dest1	(dest2)

where () indicates optional parameters
op-code is the instruction identifier
r is the activation frame offset number for dyadic operations
dest1 and dest2 are the destinations (dest2 being optional)

Example tagged token

context pointer	statement pointer	port	data

where	context pointer	=	address of the start of the activation frame
	statement pointer	=	address of the target statement
	port	=	indicates if the destination is the left or right operand
	data	=	a word of data

Figure 3: Example tagged-token data-flow instruction and token formats

The matching scheme may be extended (e.g. the EM4 machine [11]) to allow intermediate results to be held in a register file, rather than being passed via a matching store, whilst instruction are executed in a control-flow manner. In the EM4 implementation the control-flow sections are limited to input values into one instruction (dyadic or monadic) followed by a linear sequence of operations fetching their operands from the register file (i.e. the control-flow segments are limited to at most two input parameters).

The problems with tagged token data-flow are:

- Every time a function is instantiated an activation frame has to be emptied (128 words on the Monsoon machine [10]).
- Matching the first operand for a dyadic operation results in no operation to be performed and thus a bubble in the pipeline.

Conclusions about current data-flow paradigms

Static data-flow, though simple, is impractical because functions cannot be supported properly. Coloured data-flow supports functions but matching is complicated and it is difficult to envisage how it could be made temporally predictable. Tagged token data-flow supports functions and the matching is simple.

Since tagged-token data-flow seems to be the most practical, particularly from a real-time perspective, it is the paradigm which is assessed in the following sections.

4.3 Assessment of tagged token data-flow

Assignment and input/output

The highly concurrent nature of data-flow means that assignment is a problem, and hence the reason why data-flow languages are single assignment/functional. This makes I/O difficult, particularly when I/O devices have to be shared.

Latency tolerance

The concurrency of data-flow machines enables them to perform memory operations and execution in parallel. This allows them to tolerate memory latency and therefore caches are unnecessary. Furthermore, these parallel operations can be performing work for logically different threads. Therefore, data-flow machines can exploit concurrency in order to tolerate latency.

Concurrency

Data-flow machines are concurrent. However, to date the machines have had a simple scheduling scheme (e.g. LIFO or FIFO queueing of instructions waiting for execution) which is inappropriate for ensuring that deadlines are met.

Data locality

Data is often distributed in order to optimise multi-processing. However, there is no reason why data could not be made local if required, although this could impinge on multi-processor efficiency.

Temporal predictability

Whilst caches are unnecessary and, therefore, do not add to the temporal unpredictability, the simple hardware scheduling schemes are better suited to number crunching than real-time processing. Therefore, it would be difficult to construct temporally predictable tagged token data-flow programs on the Monsoon or EM4 machines.

5 Control-flow/data-flow hybrids

5.1 Introduction

Control-flow/data-flow hybrids have been sought in an attempt to combine advantages whilst attempting to avoid the disadvantages (e.g. [13]). This invariable leads to multi-threaded control-flow models, of which two examples are briefly discussed and assessed below.

5.2 D-RISC

Overview

The data-flow RISC, or "D-RISC", machine was conceived by T. Clarke [2][12]. It is a multi-threaded control-flow model where each thread has a thread execution state (TES) consisting of a triplet $\langle WP1, WP2, IP\rangle$. $WP1$ and $WP2$ point to data cache lines (8 or 16 words) for working variable storage (like a register file), and IP points into the code prefetch cache. Context switches occur at memory access points. They are perform by queueing the old TES and using a new TES from the queue. Therefore the D-RISC can tolerate memory access latency. Furthermore, a TES cache miss can also cause a context switch so cache line fill latency can also be tolerated. For efficient operation the TES cache must have enough lines for all of the active tasks.

Latency tolerance

Latency tolerance is good provided there are sufficient threads waiting to be executed which already have their data and code cache lines filled. If the cache lines for a task are not filled then it will be rescheduled and processor time will be wasted.

Assignment and input/output

Assignment of variables within one thread is easy because the thread is executed sequentially. Access to variables shared by multiple threads may be arbitrated using conventional mutual exclusion techniques.

Latency and frequency tradeoffs

Has the same poor properties as control-flow.

Concurrency

Concurrent but the scheduling policy is dependent upon inter task cache utilisation and is therefore temporally unpredictable.

Data locality

Reasonably good, although a conventional multiported register file cannot be used.

Temporal predictability

Scheduling policy problems due to caching result in temporal unpredictability.

5.3 SAM

Overview

SAM was developed by Peter Maurer and Wei Lin[8][9]. Its name is derived from two special instructions *Split* and *Merge*. *Split* forks off a thread and *Merge* merges threads together by terminating the first thread which executes it and allowing the next thread to continue unabated.

The context for each thread is stored in 256 words of "environment", EP, (often cached) and a separate stack, SP. Threads are represented by the triplet $\langle EP, SP, PC\rangle$ where PC is a pointer to the program to be executed. Threads are scheduled in a round-robin manner. Execution is highly pipelined where each stage of the pipeline is working on a different thread.

Assignment and input/output

As with D-RISC, assignment of variables within one thread is easy because the thread is executed sequentially. Access to variables shared by multiple threads may be arbitrated using conventional mutual exclusion techniques.

Latency tolerance

Possible provided the *memory access* parts of the pipeline are made long enough.

Latency and frequency tradeoffs

The same as control-flow.

Concurrency

Inherently concurrent but round-robin scheduling is inappropriate for meeting deadlines. Adding earliest deadline first scheduling, so that threads not in the pipeline were queued in earliest deadline first order, would not work very well because the pipeline would still act like a FIFO queue. Therefore, no benefit would be gained from earliest deadline first scheduling unless there were many more threads than stages in the pipeline. Since the pipeline must be long, if latency is to be tolerated, this scheduling policy would often be ineffective.

Data locality

Very local data (like a register file) is not supported and hence the performance is likely to be poor.

Temporal predictability

Reasonably predictable but round-robin scheduling is inappropriate for real-time work.

6 A brief assessment of different processor paradigms

The following observations can be made about existing processor architectures (with reference to the previous sections):

Advantages	**Disadvantages**
Control-flow	
good at executing sequential code efficient use of a register file assignment and I/O are easy	latency intolerant simulated concurrency
Data-flow	
concurrent latency tolerant	assignment and I/O are a problem the usual LIFO or FIFO scheduling policies are too simplistic for real-time inefficient at using a register file
Multi-threaded control-flow	
concurrent some latency tolerance possible assignment and I/O are practical	Conventional register files cannot be used and so caches are used instead. These are less efficient and their temporal characteristics cannot be guaranteed. Scheduling policies inappropriate for real-time.

Figure 4: A brief assessment of different processor paradigms

7 Anaconda

7.1 High level design proposals

Scheduling for real-time is not satisfactorily supported by any of the architectures discussed so far. Earliest deadline first and least slack scheduling are optimal provided only one resource is being scheduled [7]. Earliest deadline first scheduling has the advantage that it can also be used for fixed priority scheduling (assuming the lower the priority number the higher (greater importance) the priority).

Data-flow is concurrent and latency tolerant; vital properties which control-flow machines do not exhibit. However, to schedule for real-time, deadlines need to be associated with a task. In tagged-token data-flow a task can be considered as a function executing in a particular activation frame. Each activation frame could then be assigned a deadline thereby allowing tokens to be queued in earliest deadline first order. Unfortunately the queueing process would be very difficult because a large number of tokens would have to be placed in a structure which would allow insertion and a *find minimum deadline operation* to be performed every clock cycle.

Since fine grained (instruction level) earliest deadline first scheduling is impractical, a larger data-flow node model is proposed. Since control-flow is efficient at executing small functions and is able to make efficient use of a register file to store intermediate results, it is proposed that these large data-flow nodes be small control-flow functions.

The direct matching store of tagged-token data-flow is efficient and temporally predictable. A variation on the same idea could be used to match parameters to each function node, where each instance of a function is assigned its own activation frame. Parameters could be written directly to the appropriate activation frame.

The activation frame needs to store the deadline, pointer to the function to be executed, data parameters and empty/full flags to indicate which parameters are still to be written (see figure 5). For implementation reasons the parameters are 32 bits long and the deadline is 48 bits long so needs to straddle two parameter storage areas but can conveniently share half a parameter with the 16 full/empty bits.

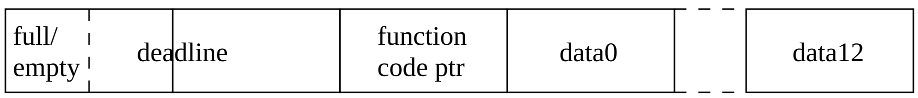

Figure 5: Activation frame format

7.2 High level functional description

An instance of a function is created by initialising an activation frame in the matching store by setting the full/empty flags, deadline time and function pointer. When all of a functions parameters have been matched then the deadline time and activation frame pointer are passed to a scheduler. The scheduler keeps feeding the processor with parameters and code of the functions with the earliest deadlines. The parameters could be transferred directly into one page of the register file whilst another page is being use for execution; similarly the code could be transferred to a small processor local memory (i.e. a pre-fetch cache). Execution of a function results in values (an address + data + write token $\langle a, d, w\rangle$) being written to the matching store main memory and I/O devices.

Values may be read by specifying an address to be read and a destination address for the value to be written to (i.e. a token $\langle a, d, r\rangle$ where d is the destination address rather than data).

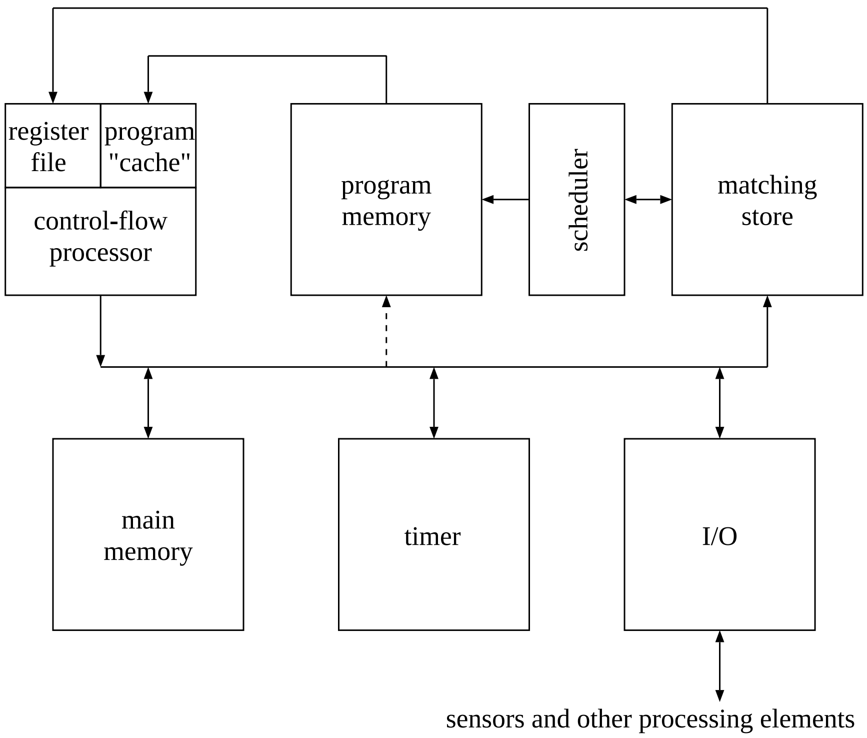

Figure 6: Idealised Anaconda

7.3 Example execution models

Multi-threaded

One possible execution model is multi-threaded control-flow. This may be supported by joining function nodes in a sequential order (see figure 7). Note that all memory and I/O accesses are passed to the next function node so that other threads may run concurrently rather than waiting for responses. Also, the function nodes in the thread may use the same activation frame (same area of matching store) so values which may not change from one function node to the next (e.g. the deadline) do not need to be written back to the matching store.

All code and data are pre-fetched by the scheduler so the processor never needs to stall, provided there are a sufficient number of threads waiting to be executed (around about four active threads to prevent stalling).

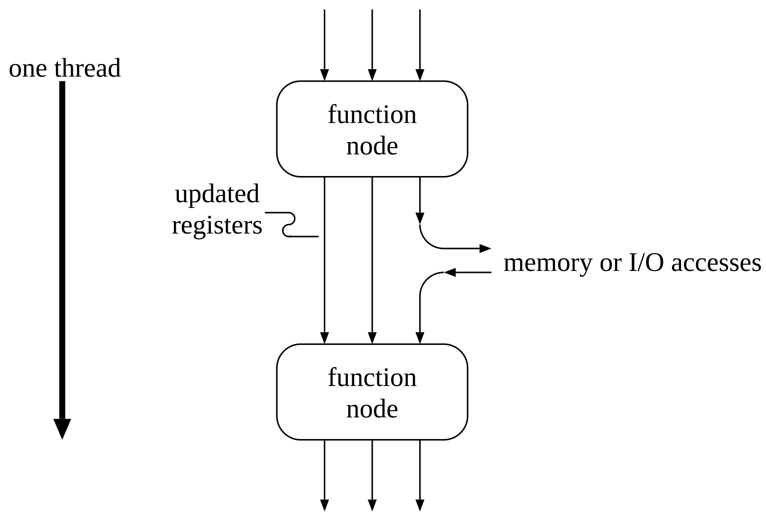

Figure 7: Multi-threaded control-flow model

Loop unfolding

Many applications will have a sufficient concurrency to ensure that there is enough work to be performed whilst waiting for memory responses. However, techniques like loop unfolding may be used to increase the amount of concurrency. Figure 8 illustrates loop unfolding where each iteration of the loop is independent. Interdependencies may be catered for but this makes the graph more complicated.

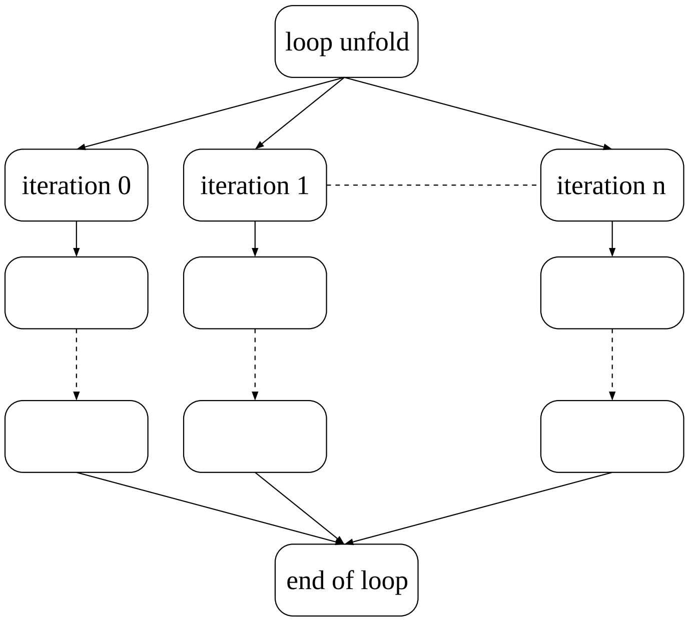

Figure 8: Simple loop unfolding

Data-flow

Data-flow like operations may also be performed. For example, a bubble sort could be encoded as a series of compare and swap operations (see figure 9); since Anaconda is a large data-flow node processor it is more efficient for each node to compare and swap ten values, rather than two, at each node.

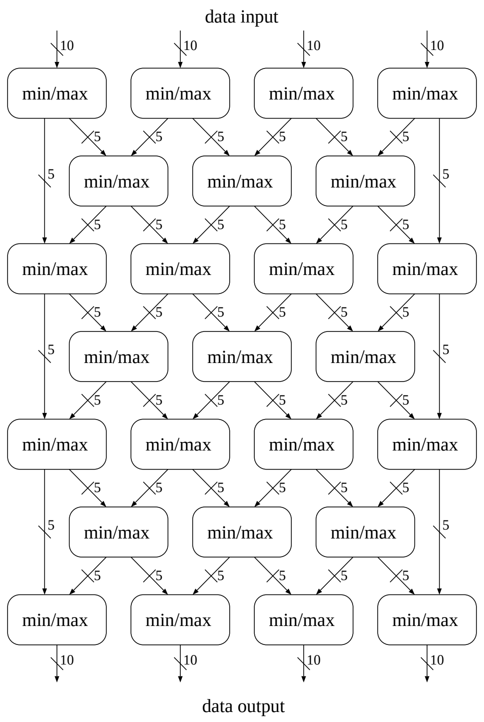

Figure 9: Data-flow bubble sort

7.4 Towards an experimental implementation

Processing element

Although the idealised Anaconda processing element would be control-flow its instruction set would be a little unusual because:

- the conventional *load* instruction (with its associated scoreboarding in the case of load posting) would be replaced by a *send address and destination* instruction, where the destination would be a location in the matching store
- the conventional *store* instruction would remain the same but additionally any computation operation should be able to write a value to an activation frame
- a *next* instruction or *this is the last instruction of this function* bit would be needed to indicate function termination

- branches/jumps would only need to work within a small control-flow function (say a maximum of 32 instructions)
- the register file and program pre-fetch storage areas need to be written to under the scheduler's control
- conventional caches are unnecessary
- interrupts are not required

However, in order to simplify the experimental implementation (and avoid large financial costs) it was proposed that an off the shelf microprocessor should be used. Anaconda's load and store instructions can be implemented using a conventional processor's store instruction where the top most bit of the address is used to indicate whether an Anaconda load or store request is to be performed. Performing an operation and write to the activation frame could be done with two instructions. The *next* instruction can be simulated with a suitable jump. Caches and interrupts may be ignored.

Program pre-fetch "cache" could be implemented but to simplify the first version the program memory will be "all cache" (i.e. all fast static memory) local to the processor. Therefore the *next* instruction will just be a jump to the appropriate piece of code.

The primary difficulty is in loading the register file directly. This may be solved by using two processors: one pre-loading its register file whilst the other is executing code. One of the processing elements is illustrated in figure 10. A small amount of dual-ported memory is provided to allow shared variables (e.g. a FIFO queue of unused activation frames).

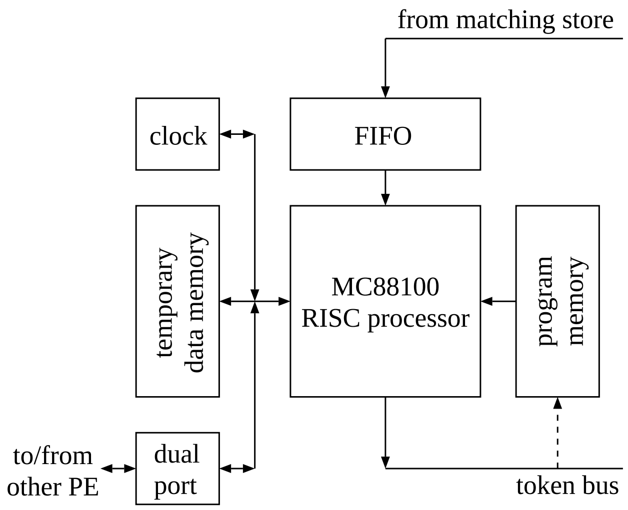

Figure 10: One of the two processor elements

The MC88100 microprocessor was selected because it has the following features:

- Harvard architecture (separate program and data busses) both internally and externally. This allows high throughput at modest clock frequencies.
- The program and data caches are not on chip (usually provided by MC88200 chip) so there is no problem avoiding cache unpredictabilities because the caches can be removed.
- No unneeded virtual addressing hardware (normally performed by the MC88200).
- 31 general purpose registers (R0 is zero) — 17 are required to store the activation frame + activation frame address and a few temporaries are required.
- Reasonably cheap (around £74).

Matching store and scheduler

The matching store is just a memory with an empty/full flag for each word (see figure 11). When an activation frame is written to the corresponding flag is set and the other flags in the activation frame are read. If all of the flags are set then the scheduler is sent the address of the activation frame and the deadline. Later on the scheduler requests the activation frame to be sent to a processing element.

Scheduling is performed using a custom *find minimum* chip which is able to accept or extract a value every other clock cycle.

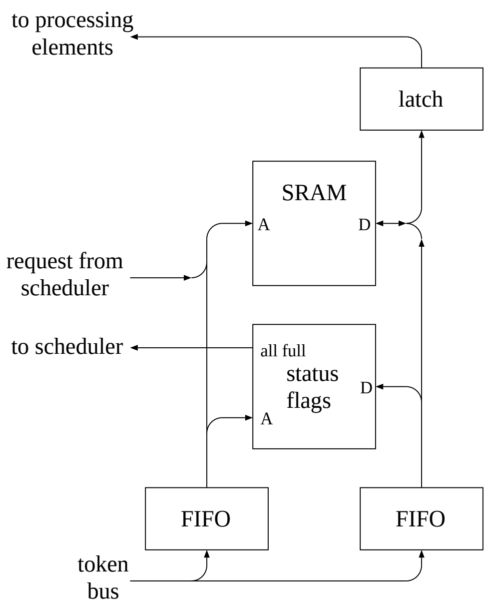

Figure 11: High level view of the matching store

Input/Output

The I/O interface is needed to connect to other Anacondas for parallel processing, other processors, disks and shared memory devices. Messages will be usually be short (e.g. remote memory requests and remote function invocation) and the bandwidth high. Our own ATM switch network was a natural candidate for this task (please refer to the rest of this document collection of details).

Conveniently an ATM cell (48 bytes of data + header) may be mapped, via the virtual circuit identifier, onto the 13 general purpose words of an activation frame. Thus, when a complete cell arrives it may be scheduled and then transferred to a processing element. This effectively means that the cell will be transferred directly to the processing element's register file.

Cell transmission is just a matter of sending 13 words to the ATM interface. This may be performed atomically if the complete cell is sent by one function node. Thus, a cell would be constructed in an activation frame and when complete it would be transmitted by the appropriate function.

Partitioning function nodes between a number of Anacondas may be achieved by adding function node stubs which transparently handle interprocessor communication.

The ATM interface will also be used for diagnostic reads and writes, and for booting the system. A cell on a predetermined virtual circuit will have its first 4 data bytes interpreted as an address + read/write flag and its next 4 bytes as data. A read will have its destination address set to a special address on the ATM interface which transmits the read data back over the predetermined virtual circuit.

8 Assessment of Anaconda and conclusions

8.1 Assignment and input/output

Multi-threaded control-flow nature means that sequencing assignments for one thread is easy. Sequencing assignments between multiple threads may be performed either by time ordering or by conventional mutual exclusion techniques. However, the matching store may often be used for inter-thread synchronisation and communication.

Input/output is well catered for with the ATM interface. Since assignment is not a problem, ordering values to be output is not a problem.

8.2 Latency tolerance

Inherently concurrent so is able to tolerate latency. This implies that caches are unnecessary.

8.3 Concurrency

Medium grained concurrency. Scheduling is temporally predictable and dynamic so both periodic and sporadic tasks may be catered for. There is no processor overhead involved with scheduling.

8.4 Data locality

Makes good use of a register file for storing intermediate results. Processor local memory and interprocessor shared memory are supported.

8.5 Temporal predictability

The latency tolerance means that there is no need for temporally unpredictable caches. The temporally characteristics of function nodes is easy to access statically. This, and the earliest deadline first scheduler, means that the machine is highly temporally predictable.

8.6 Conclusions

Anaconda — a temporally predictable multi-threaded processor with closely coupled I/O.

We hope to use Anaconda for high speed real-time multimedia experiments in conjunction with the desk area network project (please refer to the rest of this document collection for details).

A hardware experimental implementation begins in April, 1993.

9 References

[1] D. Abramson and Egan G. The rmit data-flow computer: A hybrid architecture. *The Computer Journal*, 33(3), 1990.

[2] T.J.W. Clarke. General theory relating to the implementation of concurrent symbolic computation. Technical Report 174, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, August 1989.

[3] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.

[4] J.R. Gurd, et al. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[5] D.B. Kirk. SMART (Strategic Memory Allocation for Real-Time) cache design. In *10th annual real-time system symposium*, pages 229–237, 1989.

[6] L. Lamport. The mutual exclusion problem: part I — a theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2), April 1986.

[7] S. Levi and A.K. Agrawala. *Real-time system design*. McGraw-Hill International Editions, 1990.

[8] W. Lin and P.M. Maurer. Sam: A multithreaded pipeline architecture for dataflow computing. emailed from wlin@arrakis.csee.usf.edu, 1993.

[9] P.M. Maurer. Mapping the data flow model of computation into an enhanced von Neumann processor. In *Proceedings of the International Conference on Parallel Processing*, pages 235–239, 1988.

[10] G.M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[11] S. Sakai, et al. Prototype implementation of a highly parallel dataflow machine EM-4. In *5th International Parallel Processing Symposium*. IEEE, 1991.

[12] J.A. Sharp, editor. *Data flow computing — theory and practice.* Ablex Publishing Corporation, 1992.

[13] P.C. Treleaven, R.P. Hopkins, and P.W. Rautenbach. Combining data flow and control flow computing. *The Computer Journal*, 25(2), 1982.