The D-RISC—An Architecture For Use In Multiprocessors

T. J. W. Clarke*

Computer Laboratory,
Cambridge University.

June 19, 1987

Abstract

This paper explores the relationship between latency, bandwidth, concurrency and CPU performance using both theory and concrete design analysis. The results are applied to uniprocessors, where they suggest a new CPU architecture, and multiprocessor systems, where this architecture is shown to be particularly effective.

A model of asynchronous concurrency using threads which may be created, referenced and read is used to analyse and contrast dataflow and von Neumann CPU design. Dataflow CPUs are constrained by fetch bandwidth, von Neumann CPUs by fetch latency. A simple quantitative theory of VLSI CPU implementation shows that latency and bandwidth constraints may be balanced, increasing performance, by choosing the right concurrency. This motivates the D-RISC architecture, a hybrid of RISC and dataflow machines using dataflow-like thread-switching to hide cache miss latency.

1 Introduction

This paper comes from an attempt to demystify study of the performance of multiprocessor architectures. Its main content is the description of a new type of CPU: a hybrid of RISC and dataflow machines called D-RISC. Advocacy of this is supported by two strands of theoretical investigation. The first describes an execution model which allows analysis of both von Neumann and dataflow execution and shows that they are not incompatible. The second shows how a multiprocessor system will exhibit, as a function of loading, three different types of behaviour. CPU performance is related to these: dataflow and von Neumann CPUs perform badly at

*This work was supported by an SERC studentship

different levels of system loading. D-RISC CPUs, with a natural ability to alter from von Neumann to dataflow execution, perform well at all loadings.

The computation considered here is general purpose and concurrent, specified by a simple strict functional language. Section 2 contains an analysis of the ways in which strict functional languages may be compiled into concurrent threads of computation which are then executed on hardware.

In section 3 designs of dataflow and von Neumann CPUs are contrasted using this thread oriented execution model. They are shown to lie at two ends of a continuum of possible designs. The next section discusses the relationship between concurrency and the latency and bandwidth constraints which hardware imposes on CPU design: this suggests the consideration of a new CPU design. Section 5 describes this design, called D-RISC, which incorporates RISC and dataflow techniques offering a number of implementational advantages.

Section 6 extends analysis of the performance of single CPUs to that of a multiprocessor system and shows that here a D-RISC design is particularly attractive. Finally section 7 concludes and identifies areas for further research.

2 A Thread-based model of concurrency

This section outlines the ideas that will be used in the rest of this paper to analyse CPU design and performance. First hardware constraints of bandwidth and latency are described: these will prove to be of fundamental importance in the following analysis. Then an abstract specification is constructed of the hardware operations necessary to execute a strict functional program. This identifies possible concurrency and locality of data access. Finally concurrent execution is described using threads. A thread comprises a set of sequential operations and communicates with other threads asynchronously. This results in an appropriate model with which to consider the possible decomposition of a functional expression into parts which can efficiently be executed by hardware, for example by a von Neumann CPU.

2.1 Latency, Bandwidth and Concurrency

The most basic operation in any computer system is memory access. This is performed at many levels in a CPU, both for instruction and data fetch. A memory consists of a fixed number of locations each of which may be written and subsequently read, returning the contents last written.

The definition of memory as a unit of computer hardware thus invokes the notion of sequence. When discussing performance it is necessary to quantify this and consider time. Memory operation is constrained by hardware limitations in two different ways: **latency** and **bandwidth**. The bandwidth of a memory is the rate at which successive operations, read or write, can be performed. The latency is the delay between initiating a read operation by specifying the address of a location and recovering a location's contents.

A complicated unit of hardware may also be analysed in terms of latencies and bandwidths: latencies are the delays between data necessary to initiate an operation and data coming from

it; bandwidths are restrictions on the frequencies at which operations can be performed.

In simple memories of a given size bandwidth and latency are closely and approximately inversely related. The need to distinguish between them becomes apparent when describing constraints on the operation of more complicated blocks of hardware. This views the hardware as constraining the execution of a given algorithm and so is appropriate when considering the possible implementation of an algorithm. Thus a memory with low latency is good because it puts little constraint on the corresponding lookup. Throughout this paper it will be assumed that technology imposes upper bounds on attainable bandwidth and lower bounds on attainable latency.

For a complicated operation latency may be the sum of many small delays—pipelining the system makes the corresponding total bandwidth constraint much less severe than total latency, allowing several operations to proceed simultaneously. This is an example of concurrency. It illustrates an important principle: hardware performance measured by the achievable bandwidth of some operation may be constrained by latency and bandwidth. If the operation is decomposable into physically separate suboperations the bandwidth constraint for operation is likely to be less severe than its latency constraint. Optimal performance can then only be achieved by executing operations concurrently.

Even without pipelining, concurrency can be used to increase the value of a bandwidth constraint on performance by using multiple hardware blocks in parallel. The latency of an operation cannot be reduced in this way: latency is thus a fundamental constraint on hardware operation.

This paper is concerned with the performance of concurrent hardware, not the formal semantic description of concurrent communication. This can be found in Hoare's CSP [Hoa85] which provides a sound global framework within which to analyse the semantics of communicating systems. An elegant extension of this model which describes systems which are built up from local units can be found in [Mon86]; the natural representation of locality in this model makes it more suitable for discussion of asynchronous hardware.

2.2 Dataflow Execution Graphs

A **Dataflow Execution Graph (DEG)** describes abstractly the constraints on any hardware executing an expression in a strict functional language. It is a graph with nodes that represent strict ALU operations and arcs data dependence or control dependence.

If one ALU operation x has as operand the result of ALU operation y then a D-arc is drawn from y to x , signifying data dependence. Control dependence is more complicated. Every ALU operation in a program has a data dependent control flow change after which it may be specified and must at some time be executed, though this may not yet be possible. This is represented by a C-arc from the node determining the switch to the control dependent node.

The C-arcs from a node thus point to the subsequent nodes which are specified by that node, and must be remembered by executing hardware. Graph specification and graph execution happen concurrently during program execution, every node is first specified and then executed. C-arcs define sequence from node execution to node specification, D-arcs from execution to

Figure 1: Dataflow Execution Graph of $\text{fib}(2)$

execution. A node can also only be executed after it has been specified so C-arcs sequence execution also. Figure 1 shows the DEG of $\text{fib}(2)$, where $\text{fib}(n) = \text{If } n < 2 \text{ Then } 1 \text{ Else } \text{fib}(n - 1) + \text{fib}(n - 2)$.

It is important to note that a DEG corresponds to an executable program, not a function. The DEG cannot in general be derived by static inspection of a program: the data dependent parts of it develop as program execution proceeds. The data-independent part of a DEG is similar to a dataflow graph, the basis of an execution model for dataflow machines. In contrast a DEG specifies the temporal constraints that must be satisfied by execution of a particular expression on any hardware.

A single CPU, together with appropriate compiler, determines a map from the DEG of an expression onto a concrete execution model which has three components:

1. A temporal order on nodes corresponding to their execution sequence on an ALU.
2. A map from each arc to a function from the time interval between the two ends of the arc to a physical data location.
3. A way of remembering those parts of the DEG which have been specified but not executed.

The study of CPU design for strict functional languages is thus the study of the hardware constraints imposed on this map. Asynchronous concurrency in a CPU may be related to a partitioning of the DEG into appropriate subsets, these are called threads and discussed in the next section.

2.3 Threads

I will call the unit of computation associated with asynchronous concurrency a **thread**. Threads vary in size from single ALU operations in a dataflow machine to whole programs in a von Neumann processor. Threads are characterised by their internal execution, which is synchronous. Thus the amount of concurrency in a single thread is usually limited to that available in the execution of a single function body. The execution of compiled code on a von Neumann machine

defines a special type of thread which will be called a **VN-thread**. A VN-thread corresponds to sequential execution of a function application and its subfunctions.

The execution of a thread may require local temporary data, this will be called the thread's **context**. Threads may be composed lengthways, to form a longer thread, or in parallel to allow asynchronous concurrency.

To support multiple processes it will be assumed that threads may be created, exported and referenced. Creation of a thread immediately returns a unique handle which is a data object and may be used globally to refer to the thread. Reading this handle at any time results in either a synchronisation wait, if the thread has not finished execution, or its value. Requests for the value of an unfinished thread are queued with the thread local data and satisfied when it finishes.

This method of inter-thread communication may be optimised for various special cases. The most important of these is when the identity of a referencer of a thread is known at thread creation time. In this case the thread value can be forwarded directly to the referencing thread when it is ready. Other optimisations can result when it is known exactly who will reference a created thread.

Threads allow dataflow style communication but may also be incorporated into data structures by reference so maximising concurrency. They are thus units both of concurrent execution and inter-thread reference. These two operations are conceptually separate although concurrent execution requires inter-thread reference. The restriction of global access to heap leads to a number of hardware optimisations not discussed here for lack of space.

In a thread model of multiprocessor asynchronous execution the two operations that require global communication are *thread export* and *thread reference*. Different runtime strategies and communications hardware will determine the ways in which these two operations can best be optimised. Full discussion of this lies outside the scope of this paper, although section 6 explores the relationship between different export strategies and CPU design.

3 CPU design

The last section showed described a model within which the ALU operations required during execution of a functional language could be arranged into a number of sequential threads, communicating by creation, reference and synchronised reading. This section uses this model to look at the hardware needed to sequence and present to an ALU its operations: this is my definition of a CPU. In VLSI ALUs are generally smaller than banks of registers so total ALU bandwidth, the measure of CPU performance, is limited by the speed at which operands can be fetched.

Two techniques are available to speed up operand fetch: physical locality and operand prefetch. If ALU operands can be fetched from a small high speed memory close to the ALU this will have both low latency and high bandwidth. Prefetch, by allowing a number of fetches to be overlapped, decouples ALU bandwidth from fetch latency although not fetch bandwidth.

Different types of operand scheduling lead to very different uses of locality and prefetch. In multiple thread execution two obvious forms of scheduling are intra-thread and inter-thread.

These are used, respectively, in von Neumann and dataflow CPUs which are described below.

3.1 Von Neumann CPUs

In von Neumann execution a single thread uses the entire resources of the CPU. The operations to be performed are specified by compile time defined code which is sequenced by a program counter and a number of return addresses. Thread context is mostly stored in stack frames, these are created dynamically and reclaimed after use.

This system has two advantages. Firstly the use of local caches or registers for ALU data is highly effective. Both the temporal locality of data access and the possibility of garbage collecting stack frames from within a cache contribute to this. Secondly the specification and execution of future operations is controlled by a set of code pointers which are compact and easily managed.

Single thread execution is specifiable in advance only up to the next conditional branch. This limits the number of concurrent prefetches possible without risking the prefetch of large amounts of unwanted data. High performance CPUs are thus ALU operand fetch latency limited.

3.2 Dataflow CPUs

In this section a model of dataflow CPUs is constructed based on threads. It describes the features of tagged token dataflow machines and is clearer for present purposes than an exposition starting with dataflow graphs. Analysis and description of static dataflow machines is more complicated and not attempted here, so that in all that follows the words dataflow machine should be taken to mean tagged token dataflow machine.

This description is cursory and does not attempt to present all of the implementation problems of dataflow machine design.

Dataflow CPUs restrict the size of threads to one ALU operation and use inter-thread scheduling and prefetch. One advantage of this is that threads have no context except a name and operand data. Thread synchronisation is accomplished by waiting for all of this data to arrive at which point the thread is executable by a single ALU operation. All computation is data driven so names are needed only by data which must be matched and access the right static thread specification. Threads thus represent nodes in a program's DEG. With such small threads the overhead of thread management is considerable and a number of ingenious techniques minimise this.

In a dataflow machine DEG specification requires thread name and operand data creation. New threads may be created for a function body in its entirety at function application time. Within a function each thread consists of a name, ALU operation and the names of its referencers; all of this may be statically determined. Dynamic creation requires that this structure should be given a globally unique name prefix.

Conditionals may be represented by switch threads which send their data input to the one of their destination sets specified by their control input. The threads that never execute do not use machine resources since they have no context.

Note that this model results in one unique new thread name per function body. Any function call which is not recursive may be expanded in line further to reduce dynamic overhead.

The part of a thread name that is unique to a function call may be held as a tag on its data which is passed to its value. Each thread now has two components: the first is a static template consisting of ALU operation specification and the static part of the referencing thread names. The second is a dynamically generated tag which arrives on its inputs and is passed to its referencers. Prefetch of an entire thread now involves finding matching tags on input data and fetching the static instruction template defined by the static part of the tags. The limiting operation in prefetch is matching incoming data tags from values of executed threads with existing data in an associative matching store. It appears [Arv86a] that a waiting-matching store of typically 10K - 100K threads is needed. The large size of this makes a high match bandwidth, and hence high performance dataflow machines, difficult to engineer.

Heap control

Heap reference may be added to this model of dataflow without any loss of potential concurrency by restoring the function of thread reference. Cons may be implemented as a thread that returns a storage structure with references to its two input threads. These references are read by any subsequently created head or tail nodes with input this structure. The realisation of these semantics in a dataflow machine leads to a specialised storage module which may be written directly by destination tokens and read at graph specification time. If the data to be read has not yet been written the address of the reader is queued in the storage module to be satisfied as soon as the data is written.

This mechanism is exactly the I-structure proposed by Arvind [Arv86a], and seems to be the most satisfactory of those proposed for dataflow machines.

3.3 The possibility of compromise

The above analysis demonstrates that von Neumann and dataflow CPUs both suffer from hardware-imposed limits on performance and the problems are complementary. The limited prefetch available in a von Neumann machine makes performance very dependent on low latency operand access. In a dataflow design the temporal locality of single thread access is destroyed by thread switching so that local cacheing is difficult.

It is natural to ask whether simple modifications to either von Neumann or dataflow design exist which use both prefetch and locality. In a dataflow machine a small matching store cache close to the ALU is easy to realise and would with a high match hit rate localise most token traffic. High bandwidth instruction fetch is still necessary but as fetch latency is not critical this too is realisable. The problem with this design is that dataflow makes no distinction between execution of different function bodies so temporal locality and hence cache hit rate is lost.

The effectiveness of a cached dataflow machine has yet to be proved and will not be discussed further in this paper.

In Section 5 the addition of multiple thread prefetch to RISC von Neumann execution is

considered. This means that thread switch can be used to minimise the effect of cache miss latency.

The distinction between bandwidth and latency problems in CPU design becomes more significant in a multiprocessor system. Section 6 develops this showing that in a multiprocessor environment, as for a single CPU, the key to understanding performance lies in the relationship between latency, bandwidth and multiple thread execution.

4 Latency and bandwidth constraints on performance

The last section showed that in specific designs CPU performance could be limited by either fetch latency or fetch bandwidth. This section presents a general argument that quantifies the relationship between these constraints and concurrency under certain assumptions. The argument motivates the D-RISC design, outlined in the next section, and also quantifies the assertion that dataflow machines are more efficient than von Neumann CPUs in multiprocessor systems.

A CPU may be partitioned into an ALU, a local cache including all data and instruction fetch registers, and an interface to external memory. Suppose that the cache miss rate is a design constant determined by external memory communication bandwidth, and that cache miss latency is negligible: this last assumption will be relaxed later. Suppose also that the CPU and its local cache are implemented in VLSI on one chip.

CPU performance may be limited by either fetch latency or access (read and write) bandwidth to cache. In this analysis code is not distinguished from data; the execution of a von Neumann instruction thus requires a number of sequential fetches. Where these sequential fetches can be arranged to occur in different memory banks fetch latency becomes a more severe constraint than fetch bandwidth: in von Neumann design this extra bandwidth is used in pipelining concurrent instruction execution within one VN-thread.

Suppose that the hardware determined minimum instruction fetch time and maximum fetch bandwidth are t_{fetch} and f_{fetch} respectively. Suppose further that the CPU design allows concurrent execution of ν instructions. Then the CPU performance, f_{ALU} , is constrained as follows:

$$\begin{aligned} f_{ALU} &< f_{fetch} \quad (\text{bandwidth constraint}) \\ f_{ALU} &< \frac{\nu}{t_{fetch}} \quad (\text{latency constraint}) \end{aligned}$$

In von Neumann CPUs concurrency comes from pipelining different phases of instruction execution. For a given CPU concurrency may increase performance by altering the latency constraint on execution. It is possible, but not necessary, that these two constraints are only equalised by a concurrency greater than that obtainable by pipelining. If this is so then the extra concurrency can still be obtained by *inter-thread* concurrency. Suppose, pessimistically, that different threads have no data dependency. Then to keep the same cache hit rate cache size must increase by n , the number of concurrent threads. This is equally true for any special purpose registers associated with instruction execution or data fetch. The size increase has an affect on memory speed; for relatively small memories in VLSI it is reasonable to assume

Figure 2: Latency and Bandwidth constraints on CPU performance

that latency scales as the square root of memory size, see [Mea80]. Let f_{fetch} , t_{fetch} represent the unscaled hardware performance. Then CPU latency and bandwidth constraints may be rewritten as functions of n , supposing a fixed concurrency of ν to be available from pipelining:

$$f_{ALU} < f_{fetch} n^{-\frac{1}{2}} \quad (1)$$

$$f_{ALU} < \frac{n^{\frac{1}{2}} \nu}{t_{fetch}} \quad (2)$$

As may be seen from figure 2 the ALU bandwidth is maximised when

$$n = \frac{t_{fetch} f_{fetch}}{\nu}$$

Inter-thread concurrency, though less efficient than synchronous concurrency, will balance latency and bandwidth with a performance increase of the square root of their ratio.

It is unclear whether inter-thread concurrency is useful in von Neumann design, although it has been used, as would be expected from this argument, in some designs with long pipelines [Smi78].

Now consider the effect on the preceding analysis of a cache with a miss latency which is not negligible. Suppose this reduces average f_{ALU} to a fraction β of its maximum value. The optimal extra concurrency to compensate for this may be found by applying (1) and (2), assuming that η concurrent asynchronous threads each with a workload of $1/\eta$ result in a workload of 1:

$$\eta^{\frac{3}{2}} - \eta^{\frac{1}{2}} = \frac{1 - \beta}{\beta}$$

The complicated form of this equation comes from the effect of increased concurrency on ALU bandwidth and hence β . For small β this approximates to $\eta \approx \beta^{-\frac{2}{3}}$. The corresponding gain in performance is $\beta^{-\frac{1}{2}}$.

Thus extra operand fetch latency swings the design balance to execution with a larger number of threads. The next section investigates the realisability of a multiple thread CPU using a small cache and tolerating a high cache miss rate. This design performs better than is indicated by these figures for two reasons. First by using the right thread scheduling considerable correlation

Figure 3: D-RISC CPU schematic

can be expected in inter-thread data access. Secondly thread switches occur only on cache misses so that some of the context of a thread can be held in fixed registers.

These results may equally be applied to multiprocessor execution, where inter-processor synchronisation and global memory access delays may be treated as cache misses. This is one basis for the assertion, examined in Section 6, that dataflow machines perform relatively better than von Neumann ones in a multiprocessor environment.

5 Combining RISC and dataflow design. The D-RISC.

This section describes a modification to RISC architecture which attains high performance through dataflow-like thread switching of VN-threads. The basic idea is simple: the ALU uses a RISC multiple register file cache for fast operand access. Thread switching occurs whenever a cache miss is anticipated and is used to hide cache load, or flush and load, latency.

The rest of this section, including the final performance predictions, assumes that a D-RISC is fabricated on a single chip as a microprocessor, and hence that memory speed and size suffer trade-offs as discussed in [Mea80].

5.1 D-RISC Architecture

The organisation of a D-RISC is illustrated in figure 3. The single thread execution state (TES) is held in a triple of registers: WP1, WP2 and IP. The first two point into a register file cache and the last into a code prefetch cache. The caches have a line size of 8 or 16 words, the three blocks pointed to form the thread's **immediate context**. Instructions perform ALU and transfer operations on WP1, WP2 data, and may also use a number of scratchpad registers which are lost on thread switch.

The innovative feature of a D-RISC is the high speed cacheing of a small number of TESs to enable fast thread switch, and the automatic use of this whenever a change to the immediate context means that part of it is not cached. This restricts thread switches to instructions

altering the blocks pointed to by the TES, allowing the use of scratchpad registers between these instructions. The ALU bandwidth lost on thread switch depends on the ease with which thread changes can be fed through instruction fetch and decode pipelines without flushing. With a separate small cache from which to fetch code and given advance warning of impending switches it is possible to arrange for zero thread switch latency. Even with a cycle or two overhead thread switch latency is much less than expected time between thread switches.

To implement this automatic switching the D-RISC needs a prefetch control unit and a dataflow style matcher which operate concurrently with von Neumann thread execution. Each cached thread has a status of runnable or latent according to whether its immediate context is cached or not. The matching unit puts runnable thread TESs into a queue waiting for the next forced thread switch. This is an easy job since there are only a few cached threads. The prefetch unit controls external bus activity and fetches immediate context of latent threads. Care is needed to not to flush any block in an immediate context: this is always possible because the total size of all immediate contexts is less than the cache size.

5.2 Cache management

The design criteria for a D-RISC cache are relaxed from those of a normal RISC machine because cache miss latency is less of a design constraint than cache update bandwidth. This means that a low hit rate for read only access is more acceptable than a low hit rate for read/write data access. The first simplification that this allows is the use of a very small code cache which is used for prefetch only except in tight loops or tightly recursive code.

Most cached data in a D-RISC will be used for local function context which can be garbage collected on function exit. This is used in normal RISC designs to increase cache hit rate: it relies on most stack frames having a short lifetime. In a D-RISC the existence of n concurrent threads increases this lifetime and so cache size for a given hit rate.

The assumption that this increase is by a factor of n , made in the last section, is true if each cached thread behaves like a von Neumann executing program. That this need not be so can be seen by considering the thread dependence tree of a program, in which the threads created by a thread are dependent on it. Von Neumann execution of compiled code results in a depth first traverse of this tree, with stack frames alive only for functions on the path between the currently executing function and the root.

Picking a random runnable thread for execution on termination of an existing thread results in a jump to a random part of this tree: the traverse so obtained is not exhaustively depth first and gives increased stack frame lifetimes. For a D-RISC a better traverse can be obtained by caching the threads that would in von Neumann execution be executed immediately after some cached thread. This not only preserves depth first execution of stack frames but does better by maximising the data dependence of separately executing threads.

The cache space needed for a given hit rate in concurrent execution of n threads with this scheduling is a factor of between 1 and n times that needed in von Neumann execution.

The fully associative fixed block-size cache in a D-RISC is less space efficient than a single circular stack buffer, however the overhead in associative lookup is minimal because this occurs

only when the immediate context is changed.

A full analysis of cache behaviour in a D-RISC, together with implementation details of the proposed scheduling, can be found in [Cla87].

5.3 Synchronisation

In a von Neumann thread model of concurrency finishing threads send wakeup signals and results to reading threads. These have a similar function to dataflow tokens matching but as threads are allocated fixed external memory addresses they may be implemented without associative lookup. The synchronisation overhead is thus much less than on a dataflow machine where threads are smaller. It may be further reduced by special support for in-cache synchronisation.

5.4 Out of cache data transfer

In a D-RISC ALU bandwidth is not limited by cache miss latency, but may be by cache replacement data bandwidth. The transfer of all data in fixed blocks of 8 words allows higher data transfer bandwidths than sequential data access, for example by the use of column addressing for dynamic RAM, or wider memory data paths.

Thus even with a cache simplified to prefetch data only, without associative lookup, out of cache transfers can be less expensive than their von Neumann counterparts. The transfer of data in blocks without critical transfer latency allows another optimisation. All data can be translated to a new representation for external transfer. This transformation can result in considerable compaction, so reducing external bandwidth. In conjunction with unique reference pointer garbage collection [Sto84] it may make possible the efficient localisation and cdr removal of heap. This is of particular use in a multiprocessor system, where global heap pointers and transfers are expensive.

5.5 Heap Control

In a D-RISC the efficient caching of heap has yet to be investigated. A simple solution is to store all heap externally and thread switch on heap access. Single thread performance could be enhanced with heap structure dependent multiple prefetch. The writing of new heap by using a cache block as a window onto a number of cells of heap in external memory poses no problems.

Recent work on cdr coding [Li86] shows that a memory block size of 8 words can be used to preallocate store with no increase in memory use. This suggests that the space overhead of managing heap in fixed size blocks could be small: the advantages in faster access to lists are particularly significant in any parallel implementation.

5.6 Performance

The observation that a D-RISC with zero cache hit rate is not cache miss latency limited allows a lower bound to be put on D-RISC performance which is independent of the complexities of scheduling discussed in section 5.2. With no concern about hit rate the cache becomes a

simple fixed prefetch buffer for each immediate context block. The performance of this machine is limited by external memory bandwidth. A simple design with a 32 bit wide data path to dynamic RAM and using column only block access would achieve 32Mbyte.s^{-1} . A RISC CPU using these techniques, the Acorn RISC Machine [Fur85], has a word long instruction rate of 4 mips. Performance equal to this could thus be expected from a simple D-RISC.

6 Performance of CPUs in multiprocessor architectures

This section extends the analysis of CPU performance to that of multiprocessor systems. The significant difference is that communication latency across a high bandwidth bus is typically much longer than local operand fetch.

Balancing the various constraints on multiprocessor design is complicated and not attempted here. Instead a model of multiprocessor design is constructed which is sufficient to analyse the relationship between inter-thread CPU concurrency and performance.

A multiprocessor system, for this purpose, may be characterised by three independent design axes. Each axis is characterised by its two extreme design strategies: these are tabulated below.

Design Axis	Design endpoints	
Inter-thread CPU concurrency	dataflow CPUs	von Neumann CPUs
Export strategy	export on creation	export on demand
Processor coupling	strongly coupled	weakly coupled

In order to investigate the effect of CPU concurrency on performance two parameters which characterise execution are useful. The **loading**, L , of a multiprocessor system is the number of executable threads in the system per processor. Three execution domains may be identified, corresponding to different values of L :

- $L \gg 1$ Bandwidth limited computation.
- $L \ll 1$ Latency limited computation.
- $L \approx 1$ Load balanced computation.

Real programs exhibit dynamically widely varying amounts of concurrency. For any particular system programs may be chosen whose performance is mainly either latency or bandwidth limited. In the former case the multiprocessor is too large for the program: this is not a sufficient reason for ignoring this domain of performance. As the size of a multiprocessor scales up so latency limited computation becomes more likely. Conversely, for reasons of cost, it is always likely that many bandwidth limited programs will be run. In any case the wide dynamic variation in program concurrency makes it important to consider both domains. The third domain is transitional and of less interest: it is sensitive to details of scheduling and not discussed here.

The **latency**, λ , of a multiprocessor system measures the importance of its bus latency. It is defined to be the single message global communication latency, or half of the round trip global reference time if this is significantly greater, divided by the average time between thread finishes or synchronisations on one processor. Although this depends on the structure of the executing code it is typically fairly constant, and will be regarded in what follows as a design parameter relating global bus latency to CPU performance. For systems with $\lambda \gg 1$ latency

limited performance is dominated by bus latency, and the number of bus latencies on the critical path of a computation must be minimised.

Export on creation optimises latency limited performance in two ways. First the delay from creation to execution on a different processor is one bus transfer, not the two or more for demand export. Secondly any thread reference can contain the identity of the processor to which the thread will be sent, so reducing the bus latency overhead of thread reference, which would otherwise have to indirect through the creating processor.

In contrast export on demand optimises bandwidth limited performance by minimising the number of threads that are exported. This, primarily, is useful in reducing bus bandwidth requirements but in a D-RISC the extra locality reduces cache miss rates. A newly imported thread is (almost certain) to have no data dependence with existing local threads. The *thread export probability*, E , a dynamic execution parameter, thus determines D-RISC performance in bandwidth limited computation.

Strong processor coupling corresponds to use of global shared memory rather than local memory. Actually all memory may be physically local to a processor at little cost. In a strongly coupled design this is of little use because working data is not likely to be local. Just as dataflow and von Neumann CPUs are constrained, respectively, by operand fetch bandwidth and latency, so strongly and loosely coupled designs are constrained by the cost of bus bandwidth and memory size. At this level of abstraction it is not necessary to distinguish between use of memory as cache and use of memory as locations that are absolutely referenced, but used locally, such as registers.

In the extreme of strong coupling all ALU operands are stored globally and so inter-thread switching costs nothing. My analysis of CPU performance has assumed that at least some high speed cacheing local to an ALU is useful: this implies that coupling cannot be strong. Intermediate coupling corresponds to an architecture in which processor communications and storage hierarchies are combined, with each level of cacheing supporting a larger number of sharing processors until it becomes appropriate for all memory access to be global. This is when the necessary global communications bandwidth is comparable with the cache miss rate, so that bus use is dominated by communications rather than cache update traffic.

This classification of multiprocessor systems is enough to allow analysis of the performance of different CPU types in multiprocessor systems.

6.1 An analysis of performance

The performance of a multiprocessor system is determined by that of its CPUs and its thread scheduling policy: when and to whom to export computation.

In bandwidth limited computation on dataflow CPUs the latency of inter-processor operand fetch can always be hidden by thread switching. Each CPU operates at its full bandwidth and the performance of the system is maximised. Scheduling to minimise processor idle time is easy because performance remains optimal with a considerable variation from average loading on any particular processor.

Von Neumann CPUs with slow thread switch perform badly in bandwidth limited computa-

tion. This penalty comes from processor switching time whenever a thread waits on another one or finishes. If the average computation time between these events is t_{comp} and the thread switch time t_{sw} then $t_{comp}/(t_{comp} + t_{sw})$ is the fraction by which total CPU bandwidth is decreased.

The performance of D-RISCs in bandwidth limited computation depends critically on export strategy. If this is such as to limit the number of independent concurrent threads per CPU the cost of thread switching in cache misses remains low, otherwise performance deteriorates as caches become useful only for prefetch. Even in this case performance remains better than that of a von Neumann CPU because the degradation comes from extra out of cache bandwidth rather than latency caused by context switching.

The benefits from a D-RISC multiprocessor system which maintains good cache hit rates are increased by the possibility of using intermediate coupling, and at each cache boundary connecting as many CPUs as typical cache miss bandwidths will allow on a single bus. This type of communication is extremely cheap and also low latency. Broadcast and snoopy behaviour on such a bus could perhaps be used to optimise both export and reference [Cla87]. Intermediate coupling used in this way creates multiprocessor units of anywhere between 2 and 20 processors which may then be strong coupled. This demonstrates that *cache* locality can be used to good advantage in multiprocessor design even where *communication* locality cannot be maintained.

During latency limited computation a program's execution time is bound by its sequentiality. There are always free CPUs to accept exported threads so execution time is determined by the longest path through the DEG, rather than its total size. This length is a sum of bus latencies and local operand fetch latencies: the latter may be called **thread execution time**. This path is called the **critical path** and it contains exactly those delays which sum to make the program execution time.

The performance of a CPU during latency limited computation is determined by its ALU bandwidth at low concurrency. A naive comparison between von Neumann and dataflow CPUs would assign to the latter its bandwidth executing function applications sequentially. Dataflow CPUs may give higher performance than this if they can tolerate a larger amount of concurrency without a significant increase in thread latency. This is because several functions may then be executed on one CPU without loss of CPU bandwidth; communication between these is lower latency than inter-processor communication. This is significant when $\lambda \gg 1$ and depends on how well critical communication between functions can be localised.

Arvind and Ianucci [Arv86b] have suggested that for von Neumann machines the costs of synchronisation and latency cannot simultaneously be reduced and concluded that therefore they are not suitable for scalable multiprocessors. Both parts of this argument have been considered in this paper from a different perspective. I showed in section 3 that high performance single processor machines should use both context locality and inter-thread prefetch. This blurs the distinction between von Neumann and dataflow. The use of thread switch to minimise the effect of inter-CPU latency certainly modifies design criteria by making desirable a larger number of prefetched threads. This increase need not be larger than $1 + \lambda$ —a D-RISC architecture which can cache a small number of threads will accomplish this. High performance in bandwidth limited computation then depends on the degree to which cache locality can be preserved which in turn depends on export strategy.

7 Conclusions

The unifying theme of this paper is a study of the relationship between latency, bandwidth, concurrency and performance. Section 3 demonstrates that in CPU design latency and bandwidth constraints are both important. Section 4 shows that using the right concurrency they can be balanced, and section 5 outlines a new CPU architecture which does this. Finally the separate consideration of latency and bandwidth in multiprocessor systems leads to some insight into their behaviour.

This paper presents a mixture of abstract theory and more concrete design analysis. Of these two I believe the theory to be the most interesting, because of the insight that it sheds on the complexities of concrete design. Inevitably any general analysis of a problem loses detail used in specific study. The discovery of a useful theoretical framework is a process of simplification: the problem is to find a set of simplifications large enough to abstract from unnecessary detail while retaining the essential features of the problem.

An attempt to understand hardware design tradeoffs in the SKIM [Cla80] and SKIM II [Sto84] CPUs lead to the theoretical description here. Its application to D-RISC design demonstrates its generality.

It has been shown that a multiprocessor system can have latency limited, load balanced or bandwidth limited behaviour, and that in general large programs will exhibit all three. The D-RISC architecture can adapt dynamically its use of a RISC-like register bank and execution strategy to different load levels, behaving like a von Neumann or dataflow machine as appropriate. It is thus particularly advantageous in multiprocessor systems.

The theory described here provides for an analysis of D-RISC design, based on proven results about VLSI memory performance and register window cache behaviour, to determine the optimal concurrency. Previous experience with the design and construction of SKIM II suggests that a D-RISC design is feasible and will deliver high performance, although verification of this requires a more detailed design study.

This paper indicates several areas for further research. The study of the behaviour of caches during concurrent execution has been shown to be of importance in single and multiple processor design. Implementation techniques to minimise stack lifetime in a D-RISC are investigated in [Cla87]. The optimisation of locality in caches during concurrent execution is important in dataflow CPU design: it is possible that attention to this could result in a more dataflow-like compromise between von Neumann and dataflow execution characteristics. The study of a D-RISC, where concurrency is limited to a small number of VN-threads suitably chosen to minimise temporary data lifetime, gives one line of approach to this.

8 References

- [Arv86a] Arvind and Culler, D. E.
Dataflow Architectures.
Annual Reviews Of Computer Science, 1986
- [Arv86b] Arvind and Ianucci, R. A.
Two Fundamental Issues in Multiprocessing.
Computation Structures Group Memo 226-4.
- [Cla80] Clarke, T. J. W., Gladstone, P. J., Maclean, C. D. and Norman, A. C.
SKIM - The S, K, I Reduction Machine
Proc. 1980 ACM LISP conference
- [Cla87] Clarke, T. J. W.
Forthcoming Ph. D. Thesis October 1987.
Cambridge University.
- [Fur85] Furber, S. The Acorn RISC Machine.
Microsystem Design, November 11, 1985.
- [Hoa85] Hoare, C. A.
Communicating Sequential Processes
Prentice Hall International.
- [Li86] Kai Li and Hudak, P.
A New List Compaction Method
Software - Practice and Experience Vol 16(2), pp145-163 (Feb 86).
- [Mon86] Monteiro, L. F. and Pereira, F. C.
A Sheaf Theoretic Model of Concurrency
Proc. Logic in Computer Science Symposium June 16-18 1986 Cambridge, Mass.
- [Mea80] Mead, C. and Conway, L.
Introduction to VLSI Systems
Addison-Wesley 1980
- [Smi78] Smith, B. J.
A Pipelined, Shared Resource, MIMD Computer
Proc. 1978 International Conference on Parallel Processing.

[Sto84] Stoye, W. R., Clarke, T. J. W. and Norman, A. C.
Some Practical Methods for Rapid Combinator Reduction
Proc 1984 ACM LISP Conference pp 197-204.