Technical Report

UCAM-CL-TR-174
ISSN 1476-2986

Number 174

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

General theory relating to the implementation of concurrent symbolic computation

James Thomas Woodchurch Clarke

August 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 1989 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

General Theory Relating to the Implementation Of Concurrent Symbolic Computation

The central result of this work is the discovery of a new class of architectures, which I call D-RISC, sharing some characteristics of both dataflow and von Neumann RISC computers, for concurrent computation. This rests on an original and simple theory which relates the demands of concurrent computation on hardware resources to the fundamental performance constraints of technology. I show that dataflow and von Neumann architectures have different fundamental hardware constraints to performance, and that therefore a D-RISC architecture, which balances these two constraints, is likely to be optimum for concurrent computation.

The work forms four related sections:

- A study of the nature of concurrent symbolic computation and the demands which it makes from any implementation. Two new results emerge from this. A model of computation which will be used extensively in subsequent sections, and a way of incorporating imperative updates in a functional language, similar but superior to non-deterministic merge, which captures locally sequential updates in a computation with minimum constraint on global concurrency.

- The computational model is used to contrast different policies for localising data near a CPU. A new type of cache is proposed which renames all of its cached addresses in order to reduce CPU word-length.

- CPU design is examined and a new class of architectures for concurrent computation, called D-RISCs, are proposed.

- The multiple-thread implementation problems encountered in the new architectures are examined. A new analysis of the relationship between scheduling and intermediate store use in a symbolic concurrent computation is presented.

Contents

Chapter 1

Introduction

This thesis approaches the problem of multiprocessor architecture design for symbolic computation from a standpoint which bridges two different fields of research: concurrent language implementation and hardware design.

My choice of this approach owes much to the SKIM project: my part in which was the specification and design of the two processors, so I will first describe this project and my interest in it.

SKIM was a simple specialised uniprocessor designed specifically to perform (fixed) combinator reduction. SKIM II was designed after development of the system software for SKIM, and benefited from this. The subsequent software development on SKIM II resulted in a highly sophisticated combinator reducer incorporating a number of innovative implementation techniques.

The relevance of SKIM to this thesis lies in its illustration of the intimate relationship between hardware and system software design. The architectural enhancements to SKIM II were of two distinct types:

- Balancing. The bandwidths of the various concurrent operations permitted within a microcycle were adjusted to optimise use of hardware resources.
- Generalising. Wherever possible without undue extra complexity hardware capabilities used on SKIM were generalised and made more flexible. This lead to a word tagged architecture with 4 bit tags and a rich set of tests and branches on tags.

The balancing enhancements arose from what we did understand about combinator reduction algorithms. The generalising enhancements from an awareness that there was much that we did not understand.

The most interesting results to emerge from SKIM II came from the use of generalising architectural enhancements in ways that were completely unanticipated at hardware design time. For example the richness of the tag structure allowed the efficient use of unique pointer one bit local garbage collection. The discovery of this technique would not have happened without both detailed development of the microcode and the fortuitous extra hardware capability that made it feasible.

This exemplifies a fundamental problem in research into innovative hardware for applications where the software is itself not well developed. The ideal simul-

taneous optimisation of hardware and software is impossible: instead a laborious process of experimentation is necessary.

What about the use of simulation to obviate this problem? SKIM was an unusual example of hardware where the total hardware design, construction and test time was small, and the working machine provided facilities that enabled otherwise impracticable software development. The preliminary study of high performance architectures can usually more easily proceed by simulation. However the instrumentation of a simulation fast enough for substantial software development is difficult and liable to preclude the architectural flexibility that would allow investigation of many different architectures. Furthermore the higher level, and hence faster, a simulation the less easy it is to relate its parameters to real hardware constraints.

Multiprocessor architecture investigation offers a particularly intractable example of this dilemma. Neither system software nor hardware are separately understood, whilst the enormous range of possible implementation techniques makes iterative solution extremely difficult.

Analysis of the way that SKIM hardware was used by its software led to general insight into the fundamental performance constraints in combinator reducing hardware and showed how much faster hardware could be designed. My aim has been to achieve the same sort of general insight into the much more difficult problem of multiprocessor architecture performance. To do this requires an investigation which is broadly-based and can identify the relationship between constraints on hardware and performance of the software that uses it. I will call this sort of analysis *implementation theory* because an implementation is composed of both hardware and system software.

It is generally the case that implementation theory is worked out retrospectively, as experience with existing designs shows the relationship between different implementation problems. The work from which this thesis started was such an analysis of SKIM's performance and its relationship to the SKIM hardware [Cla84]. The problem which I address in this thesis is more ambitious: how can the implementation theory appropriate to new hardware be investigated sufficiently to have some idea about the advisability of the new hardware design?

In uniprocessor design the underlying theory is relatively simple, so new designs can be investigated with some confidence. The corresponding central theory appropriate to designs for concurrent computation is the object of my investigation in this thesis.

The central result of my work is the discovery of a new class of architectures, sharing some characteristics of both dataflow and von Neumann computers, for concurrent computation. This rests on an original and simple theory which relates the demands which concurrent computation makes on hardware resources to the fundamental performance constraints of technology.

The results of my work thus form four related sections:

- Chapter 2 studies the nature of concurrent computation and the demands which it makes of an implementation. Two new results emerge from this. A model of computation which will be used extensively in subsequent sections,

and a way of incorporating imperative updates in a functional language, similar but superior to non-deterministic merge, which captures locally sequential updates in a computation with minimum constraint on global concurrency.

- Chapter 3 studies the low-level performance limitations of processor hardware, and uses the computational model to contrast different policies for localising data near a CPU. One new result of this is the discovery of a type of cache which renames all of its cached addresses in order to reduce CPU word-length.

- In Chapter 4 the preceding work is applied to CPU design and a new class of architectures for concurrent computation is predicted to have performance which is superior to conventional von Neumann or dataflow architecture. Without concrete design this theoretic prediction can be no more than a motivation for further work, however recent design work by Ianucci [Ian88] offers support for the proposition that these architectures are truly superior to existing ones.

- Finally the multiple-thread implementation problems encountered in the new architectures (and also in any multiprocessors for symbolic computation) are examined. A new result here is an analysis of the relationship between scheduling and intermediate store use in a symbolic concurrent computation. This provides an underlying theory which is confirmed by recent experimental results [RS87] on dataflow machine store use.

Chapter 2

Models of Symbolic Computation

This chapter describes the styles of computation to be considered in this thesis, and gives an overview of the assumptions to be made about programming languages and computational models.

It will assumed that computation is specified by an annotated **implementation language** (IL). The implementation problem which I address is determined by the type of computation expressible in this IL, and so issues of language design and compilation above the level of the IL need not be of concern in subsequent chapters.

The first section considers the extent to which the IL should be functional (free of side effects) and concludes that some imperative-style updates are necessary. A class of updates is introduced which are related to non-deterministic list merge, and which have particularly clear semantics.

The next section argues that the IL should use a parallel evaluation order which behaves semantically like applicative order evaluation but allows greater concurrency. Section 2.2.1 describes a set of annotations of the IL which cooperate with a run-time system to control export of computation.

The next section introduces a formal model of threads which describes the concurrency available in an IL program. The final section bridges the gap between IL and hardware by showing how any IL expression defines a set of ALU operations necessary for its evaluation. The temporal constraints on these operations may be derived from the IL directly and define the potential concurrency of the expression.

2.1 Side-Effects and Concurrency

Even when a fully parallel language is used to specify computation, at the level of hardware all execution occurs by updating the value of physical memory locations. Since there are a finite number of these and an arbitrary number of intermediate data in a computation some notion of sequence must be adhered to in order to permit location reuse.

The implementation of any IL must cope with this need for sharing and therefore compile to one or more sequential threads, each of which may then make sequential use of the resources allocated to it. The multiplicity of threads expresses the (asynchronous) concurrency which can be achieved: between different threads the sharing of resources is asynchronous and may be determined by the whims of a run-time system.

On uniprocessors the sophisticated compilation of purely functional languages leads to equivalent and more efficient sequential programs in which tail-recursive function calls turn into loops with assignment to local variables. In functional languages any recursion imposes a sequence on necessary ALU operations. If the same computation is expressed using structured loops and variable assignment then exactly the same constraints on execution order will be present; in this sense structured use of iteration and local variable assignment are exactly equivalent to the use of functional languages and recursion.

In order to maximise potential concurrency in the IL it is thus necessary to

limit side-effects only between operations that could lie on concurrent threads. If the IL is a purely functional language then, subject to the dataflow constraints examined in Section 2.4 below, any two different function applications could be concurrent. The uncontrolled use of global assignment by any part of an expression is thus dangerous and could lead to unnecessary specification of evaluation order in order to preserve well-defined semantics.

There are however important cases in which the addition of side-effects to an otherwise functional language allow algorithms to be expressed in a more concurrent way. A common instance of this is in the incremental construction of some global object (for example a symbol table). A number of function calls, perhaps from different and concurrent threads, must sequentially access and update the value of a variable. Initially some value is assigned to the variable and eventually, after all updates have occurred, the variable may be read. Between these two events the sequence of updates is arbitrary; however each update must be an atomic imperative operation. Without this restriction such a task could only be accomplished by specifying a predefined order in which the updates are to be handled, with it updates may be interleaved freely.

This lack of expressiveness in pure functional languages has long been recognised, and remedies have been suggested. The most popular of these, see for example as in [Hen82], is to allow a primitive non-deterministic merge operation which merges two lists in some order determined by the run-time system. In the rest of this section I will introduce and then consider the properties of a new operation, which I call a sequencer. This is similar to a non-deterministic merge, but more general. Furthermore the resulting language semantics are clearer because the effect of the non-determinism introduced is clearly demarcated.

Sequencers. I will call a globally updated variable a **sequencer.** Associated with each sequencer s of type S is an **update function** f_s of type $S \times X \rightarrow \langle R, S \rangle$, say. Each call of f_s has as a side-effect the modification of s and returns a value which may depend on the current value of s. The important property of f_s is that it be *semantically commutative*: that is that any permutation in the order of a set of calls of f_s does not alter the final result of the computation.

This can be proved when the S and R are such as to limit the operations that may be performed on values within them to ones which are invariant under these permutations. The proof may be local, without any reference to the global structure of the program, and it is this which makes sequencers an appropriate way of packaging limited side-effects, needed for either semantic or implementation convenience, within a concurrent program. Sequencers introduce into a program non-determinism which is provably (by local analysis) encapsulated within certain data types. If these are not printable the result of the program must be deterministic.

The semantic commutativity of sequencer calls means that sequencer updates can often be written using a semantically associative operation. In fact the semantic commutativity of sequencer calls means that this can in principle always be done, though the required operation may be expensive.

Call a set of objects each tagged with a positive integer a mset—msets are sets with possibly multiple members. For any mset of sequencer calls X, let S_X be one corresponding possible state of the sequencer. The associativity of mset union transforms to semantic associativity of sequencer state combination by defining $S_X \diamond S_Y = S_{X \cup Y}$. This operation can at worst be implemented by holding X together with each S_X and generating $S_{X \cup Y}$ from S_X and Y. Often (as in the example below) a simpler algorithm can be found for $S_{X \cup Y}$.

Now let

$$f_s(x) = \langle f_{\text{result}}(s, x), f_{\text{update}}(s, x) \rangle$$

If $f_{\text{update}}(s, x) = s \diamond x$, where $\diamond$ is an associative commutative operation, f_s can be implemented as a tree of sequencers which establishes a sequence over an arbitrary number of concurrent threads while never exceeding the spot bandwidth needed for two threads.

Consider a simple example: the distribution of a unique non-negative integer to every node of a tree in such a way that the maximum integer so used is O(the size of the tree). Three implementations of this are:

a) Sequencer with:

$$\begin{aligned} S &= \text{INT} \\ R &= \text{INT} \\ X &= \text{VOID} \\ f_s(s, x) &= \langle s + 1, s + 1 \rangle \end{aligned}$$

In this implementation the sequencer calls form a sequential thread within the otherwise concurrent tree-crawling threads which use the sequencer. Where the sequencer update bandwidth is less than the total sequencer call bandwidth this does not constrain program execution.

b) Pure functional program in which tagged subtrees are merged to give the right answer:

```
%(REDUCE-like fragment)

Procedure tag_tree( tree) =
 If atom tree
 Then tag_atom tree
 Else merge_tagged_trees( tag_tree( car tree), tag_tree( cdr tree))

Procedure merge_tagged trees( x, y) =
Let n = largest_tag_in x
In (  x . add_to_tags( y, n+1))

Procedure largest_tag_in x = ....   % return largest tag in tree x

Procedure add_to_tags( x, n) = .... % return copy of x with n added
```

% to each tag

The concurrency of the tree crawl is preserved but at the cost of an overhead O(average depth of tree).

c) Set of sequencers with:

$$\begin{aligned} S &= \text{INT} \\ R &= \text{INT} \\ X &= \text{INT} \\ f_s^i(n,x) &= \langle n+x, n+x+1 \rangle \end{aligned}$$

In the following description in order to simplify notation I will write $f_s^i(x)$ meaning an application of the function $X \rightarrow R$ defined by sequencer f_s^i to x.

Suppose that the f_s^i are used by a tree-crawling program in such a way that a call to any f_s^a has the effect of a single global sequencer call, irrespective of a. This can be achieved by combining the f_s^a together. To combine f_s^1 and f_s^2 to get a single sequencer f_s^3:

- If a call to just one of f_s^1 and f_s^2 is outstanding pass it on to f_s^3 unchanged and return the corresponding result.
- If two calls $f_s^1(x)$ and $f_s^2(y)$ are outstanding, let $r = f_s^3(x+y)$, and return r and $r+x$ as result of $f_s^1(x)$ and $f_s^2(y)$ respectively.

In this way sequencers can be combined in an arbitrary tree which preserves the sequencer semantics. If the tree is binary no sequencer has an update bandwidth greater than the maximum call bandwidth at a leaf of the tree.

It is interesting to contrast the operation of such a tree of sequencers with the pure functional algorithm, which also combines results in a tree using an associative combination operation. There are two differences:

1. The functional program does not allow sequencer state to propagate down the tree to the leaves and so must recompute a subtree's tags at each node.
2. The sequencer algorithm uses a combining tree whose shape is arbitrary instead of being exactly specified by the program [1].

The first of these demonstrates the way in which sequencers can use (local) side-effects, a different sequencer with no side effects could be used which would more closely mimic the functional program. The second of these demonstrates the great freedom which explicit acknowledgement of the associativity of a sequencer operation gives the implementor.

The number and tree structure of sequencers can be chosen so as not to limit overall concurrency while minimising sequencer combination overheads. This

[1] In fact following the program's Thread Creation Tree, see Section 2.3

choice may be made with due consideration to hardware connectivity or execution dynamics (new sequencers can be added dynamically as necessary to prevent saturation of any one sequencer).

In a binary switching network this type of sequencer can be distributed across switching nodes so enabling global sequencing without any sequencer call bandwidth hot-spots. This example has been used in a number of architectures for this purpose, where atomic primitive "fetch-and-op" instructions provide concurrent communication for different (associative) arithmetic and logical operators, and are combined in special "combining" communications networks. For example the IBM RP3 multiprocessor [AH88], where processors communicate via either a fast read/write network or a slower combining network.

Sequencer Implementation. Each sequencer s must be given an initial value, updated, and then read. The reading of a sequencer can be synchronised so as to be after all possible updates by the use of *update permit* thread reference counts. A thread which contains a sequencer initialises it and then has update permission until it tries to read the sequencer. Any thread created by a thread with update permission may be given its own update permission: a global reference count associated with the sequencer must return to 0 before the sequencer is read.

The run-time overhead of these reference counts can be reduced by careful static code analysis: for example a thread which updates a sequencer and then creates just one update-permitted thread may pass on its own count to this thread. The use of reference counts is only necessary when a sequencer *is* read. This need not be so, consider for example a simple sequencer used to distribute globally unique tags.

For the purposes of implementation two subclasses of sequencers can thus be identified:

1. Sequencers whose final state is never read: thus communication is always downwards to subthreads. Reference counts are not needed.

2. Sequencers which do not return a useful result on individual calls: thus communication is always upwards from subthreads to some parent. In this case, if the sequencers are also associative, local sequencers can be used as required to reduce latency or increase bandwidth. This case corresponds to algorithms which can be expressed directly (but less efficiently) in pure functional languages.

In general a sequencer will use both types of communication, as in the implementation of a symbol table returning access keys for names in individual calls, and whose final value is the symbol table itself.

The implementation of sequencers in multiprocessor hardware requires atomic read-modify-write access to shared store which must be carefully considered during architecture design, and perhaps incorporated into combining switching networks for higher global bandwidths.

Sequencers express a type of inter-thread communication which occurs naturally in many algorithms in a way which minimises the requisite sequentiality. Where a sequencer can be represented by an associative update this information can be used by the hardware or run-time system to optimise communication. They are thus an indispensible element of a concurrent implementation language.

2.2 Evaluation Order

Thread creation and reference results in **parallel evaluation order.** In a functional language this has the same semantics as applicative evaluation order but allows more concurrency. **Normal evaluation order,** in which function bodies are evaluated first and arguments are then evaluated by need, is often confused with **lazy evaluation,** in which every expression is evaluated only if it is subsequently needed and at most once. The difference lies in whether repeated references to an expression result in repeated evaluations of it.

One reason for this confusion is that normal order combinator reduction is efficient and results fortuitously in lazy evaluation. Combinator reduction has recently been shown (for example in [FW87]) to be an efficient implementation technique for uniprocessor lazy evaluation, and this has encouraged parallel graph reduction machine architects. The words *lazy, combinator, parallel architecture, graph reduction machine* are not synonymous.

Normal order semantics result in the best possible termination properties for programs, in the sense that if any evaluation order for an expression will terminate then the normal order one will. This results in greater freedom for the programmer, who can represent infinite objects explicitly in a program without worrying about their unnecessary and fatal evaluation.

Normal order semantics has some problems, and these are to do with implementation. The evaluation of expressions only on need often leads to large intermediate expressions, consisting of nested suspensions, in evaluation of recursion. In a large program it is easy to produce 'space leaks' where very large suspensions accumulate during execution. This is undesirable and results in a use of store that is much higher than expected by the programmer.

It is in general very difficult to infer from the structure of a Normal order program where a space leak will occur. This is the other side of the programming freedom introduced by Normal order. It appears that reasoning about size of intermediate results is very difficult with the highly data-dependent specification of evaluation order necessitated by lazy semantics.

Strictness analysis is a technique used to ameliorate this problem. From static analysis of code it is often possible to infer that an expression will always be needed. Then it may be evaluated in applicative or parallel order without disturbing Normal order semantics. A Normal order semantics evaluation can only make use of concurrency by the use of strictness analysis to identify expressions which can be evaluated eagerly.

The implementation issue here is important. If code can be evaluated ap-

plicatively then architectural investigation should concentrate on strict functional languages and use conventional implementation techniques. Otherwise architectures which optimise combinator reduction must be considered. The differences in run-time execution behaviour between the two are significant.

I choose to base this analysis on applicative (which in a concurrent architecture may be implemented as parallel) order languages. There are two reasons for this.

First, parallel order semantics result in more concurrent implementation than normal order semantics. Therefore they will be used if either strictness analysis allows this or, in the interests of high performance, the freedom of normal order semantics is foregone. A pessimist would require the latter, an optimist hope that developments in strictness analysis result in the former.

Second, in a parallel system applicative order must be used sometimes, so the additional consideration of Normal order evaluation merely complicates the analysis. The problems of concurrent implementation are large enough themselves without solving other implementation problems as well.

The sequencers described above allow explicit implementation of combinator-like lazy evaluation: this extension to the IL thus allows lazy evaluation should this ever be required.

Logic programming languages present a different set of implementation problems from functional languages and are not directly considered by this thesis. The efficient implementation of non-deterministic concurrent computation, as in breadth first searches, where termination of one computation may make necessary the killing of a large number of sibling computations, is an extra computational requirement which does not exist in a thread world. However the results of this work do extend to the larger problem, though extra work may be needed fully to understand the requirements of these programs.

2.2.1 Annotations

An annotation of the IL constitutes added information that does not affect its semantics but is used to optimise code generation. A well-known example of annotation is the **register** declaration in C.

Two sorts of annotation may be identified: critical and advisory. A critical annotation, if present, must be correct; an example of this is type information used to optimise code generation. An advisory annotation can never result in incorrect program execution. In a functional language export of computation is semantically invisible so all annotations controlling this are advisory. This has an important implication: export annotations may be adjusted empirically by unsophisticated programmers without catastrophic results. Alternatively, annotations may be generated automatically be empirical rules that give good results most of the time. The formulation of such a rule is much easier than that of a rule which will *always* give good results.

The export annotations chosen for the IL are architecturally independent. They specify the export of computation, but give no indication of where such computation is to go. One possibility is for computation annotated for export

to be scheduled on its creating processor. The export annotations, in addition to specifying thread boundaries (and hence possible units of exportable computation) may give additional information which the run-time system will take into account when deciding whether or not to export a thread.

2.3 A Computational Model of Threads

The preceding sections motivate and describe an IL in which concurrent symbolic computation can be expressed. The rest of this chapter is concerned with a more formal description of one aspect of the IL of interest to an implementor: its potential concurrency. The notion of a thread of computation has been used above informally: I now introduce a formal definition of the word.

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 execution of a thread defines a set of clock-ticks which are constrained to occur in a particular sequence. Each clock-tick may have associated with it a number of operations which happen concurrently. This is called synchronous concurrency[2]: an array processor may have a large amount of synchronous concurrency, von Neumann compiled code has an amount limited to that available within one function body.

Perhaps surprisingly this amount can be considerable if careful compilation is used to unroll loops and treat as exceptional conditional exits from loops. Fisher [JAF88] has demonstrated that most von Neumann programs exhibit high (> 5) fine-grain synchronous parallelism when appropriately compiled. The cost of this is a big increase in static code size.

The execution of compiled code on a von Neumann machine will be called a **VN-thread**, and corresponds to sequential execution of function bodies.

In general the execution of a thread will require local temporary data which will be called the thread's context. The size of a thread's context may vary with time: the dynamic part of a VN-thread's context is usually held in a stack.

This notion of a thread is similar to the familiar one of a task. I use the word thread because it evokes a sequential strand of computation and in order to be precise about the communication allowed between different threads. 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 in general queued with its context and satisfied when it finishes.

It may be expedient to ensure that synchronisation waits occur only at the start of a thread's execution, so that data availability becomes an enabling condition for execution of the thread. This can be accomplished simply by requiring a thread

[2] An exact definition of synchronous concurrency will be given in Chapter 3.

to create a new thread on waiting. The thread's value will be returned by the new thread, which is just the waiting continuation. Semantically this distinction is trivial, but it results in differences in implementation strategy which are important and will be considered later on in Chapter 5.1.

This method of inter-thread communication may be optimised in various special cases. The most important of these is when the identity of the sole 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 may result from less complete information about how a thread may be referenced.

Threads thus allow dataflow style communication but may also be incorporated into data structures by reference so maximising concurrency. They are units of both inter-thread reference and concurrent communication. These two operations are conceptually separate although (asynchronous) concurrent execution requires some form of inter-thread reference. In Chapter 5.3 I evaluate the usefulness of different strategies of global heap reference.

Thread Creation Trees. A thread decomposition of a particular symbolic computation has one important associated structure: its **Thread Creation Tree** (TCT). The TCT is a structure describing the thread structure of a computation: each node represents a thread and arcs point from parents to their children. Different static decompositions of a computation into threads will therefore have different TCTs.

The TCT captures the relationship between threads. Chapter 5 will use TCTs when describing the dynamics of executing programs. For example the likelihood that two executing threads are dynamically executed on the same processor may be related to their distance apart in the TCT.

2.4 Modelling Temporal Constraints with DEGs

A thread decomposition of an IL program specifies asynchronous parallelism. I will now introduce a new and related model which captures exactly the algorithmic constraints on parallelism: a **Dataflow Execution Graph** (DEG). We will see below that implementation can be viewed as an appropriate map from DEG to hardware execution model, determined by a compiler.

A Dataflow Execution Graph describes abstractly the constraints on *any* hardware executing an IL expression. It thus specifies the task of code generation for the expression in a way which is architecture independent.

A DEG is a convenient way of representing two partial orders which define the sequencing intrinsic to a concurrent computation. Throughout this thesis a number of structures will emerge which are, mathematically, order relations on finite sets. For notational convenience these are described by the corresponding canonical directed acyclic graphs—any dag establishes a partial order on its nodes through path connectedness; the minimal dag representing a given partial order (the one with the minimum number of arcs) represents the order canonically.

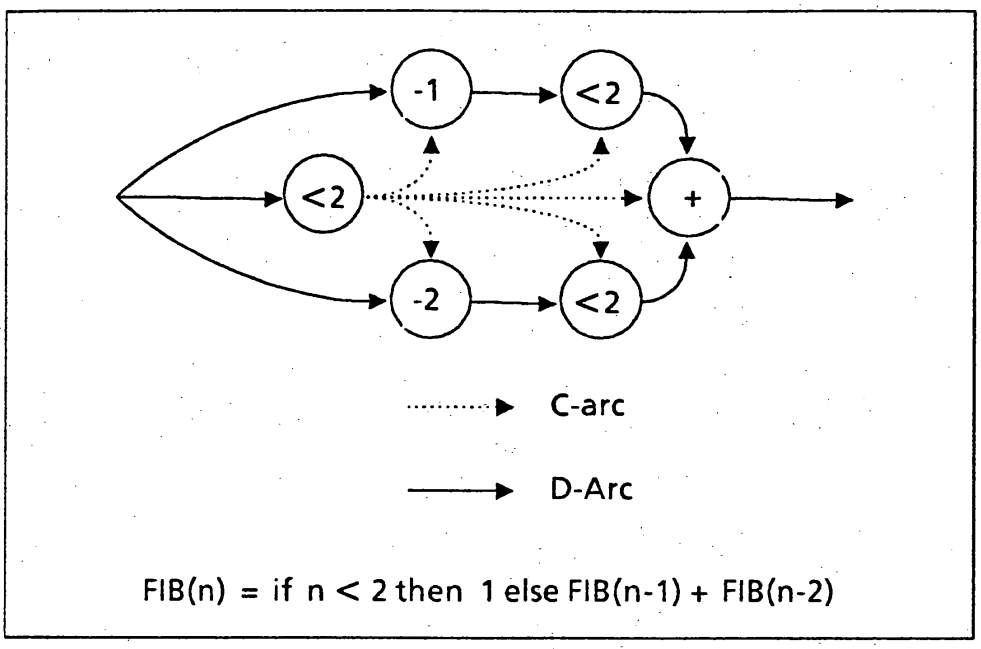


Figure 2.1: DEG of **FIB(2)**

A DEG is constructed from a (strict functional) expression by noting the ALU operations performed in its execution, these form the nodes of the DEG. Directed arcs between the nodes represent causal relationships so that $a \rightarrow b$ means 'a must happen before b'.

ALU operations must first be *specified* and then *fired.* During evaluation the specification of an ALU operation determines the compile-time information associated with it and guarantees that, at some future time, the operation will be executed. Firing of a specified operation requires the availability of its source data. Every ALU operation must first be specified and then fired. After firing of an ALU operation its result becomes available as source data for subsequent operations.

The DEG thus consists of two separate directed acyclic graphs sharing a common set of nodes. One represents dependence of specification on firing, the other dependence of firing on firing. Call these two graphs, respectively, the C-graph and D-graph and their arcs C-arcs, D-arcs. It is useful to think of C-arcs as representing control flow, and D-arcs data flow.

Arcs that do not signify immediate dependence, and so are implied by other arcs, are omitted from the DEG.

The DEG for fib(2) is shown in Figure 2.1. Note that the D-arcs to the final $\boxed{+}$ node source compile-time constant ones, available only after the firing of the associated test.

A DEG is closely related to a dataflow graph, but should not be confused with it. Whereas dataflow graphs may be used as a particular execution model for a program DEGs are a general abstract description of a computation. The DEG is not determinable from static inspection of a program because it describes all of a computation's execution dynamics.

A single CPU, together with appropriate IL compiler determines a map from the DEG of any IL computation to a concrete execution trace which has three components:

- A temporal order on nodes corresponding to their execution sequence on the ALU.
- A map from each D-arc to a function from the time interval between the two ends of the arc to the set of data locations in the CPU. These functions must satisfy the constraint of unique location occupancy as well as data transfer constraints determined by CPU architecture.
- A way of remembering those nodes which have been specified but not yet fired.

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 or multiprocessor system may be related to a partitioning of the DEG into appropriate subsets which define threads.

2.5 Related Models of Computation

This chapter has now established a number of different models of the style of symbolic computation which this thesis considers. This work lies in between implementation oriented descriptions of computation, such as dataflow graphs ([Den80]), and programming languages: for example Multilisp [Hal85].

The Multilisp language design contrasts with this work by allowing arbitrary imperative constructions within a mainly functional language—it is thus close in spirit to LISP and larger than my IL. However with this exception the type of computation which I consider is more general than other proposed models of parallel computation.

The models of parallelism which correspond to successful parallel architectures and are widely used differ from my definition of symbolic computation by using concurrency which is (to some extent) well-defined at compile time. This is much easier to map efficiently onto parallel machines than the general case of highly dynamic concurrency. Two directions in which work has resulted in some success are vector parallelism, in which parallel computation is highly regular; and models which define concurrent processes and communication statically (for example OCCAM [Hoa]).

One interesting new model suggested by Sabot ([Sab88]) introduces coarse-grain dynamic creation of concurrent array operations: this is nearer to symbolic computation than pure vector models.

One other set of execution models is based on lazy functional languages, for example Hudak's "parafunctional programming" [HS85]. These are rejected by me for the reasons given in Section 2.2 above.

Figure 2.2 illustrates the relationship between different styles of computation. Those occupying a smaller area are in general easier to implement.

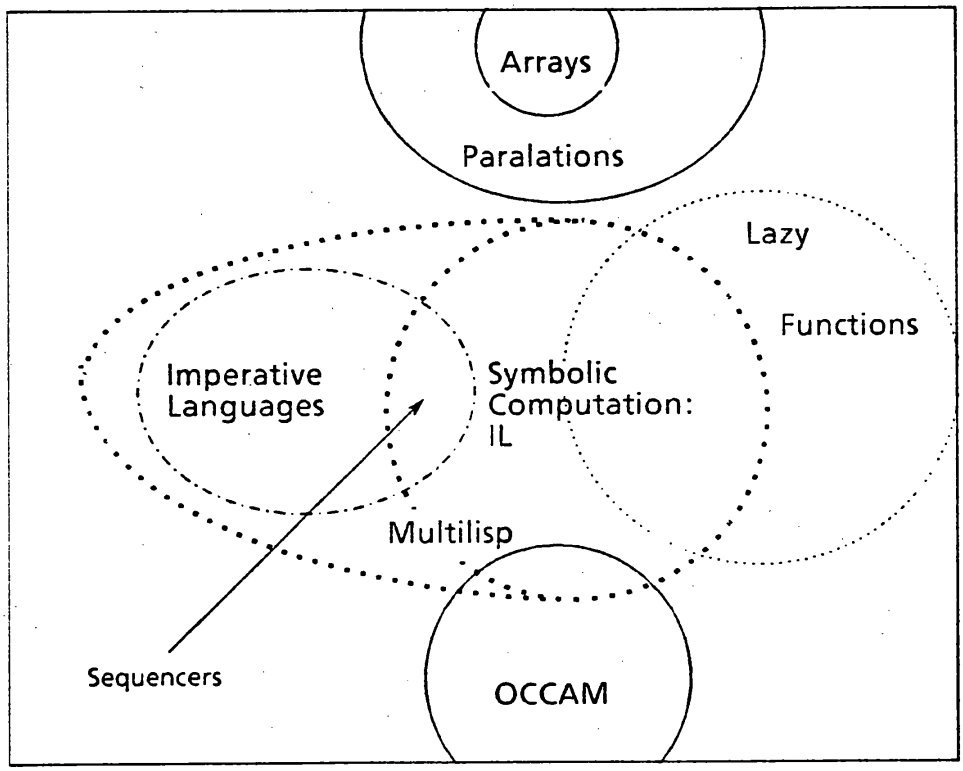


Figure 2.2: Types of Computation

Chapter 3

Models of Hardware

This chapter provides a low-level description of hardware which complements the description of computation by Dataflow Execution Graph in Chapter 2. The next chapter will relate these ideas to CPU design by discussing the different ways in which hardware can be used to localise data near an ALU.

The performance of a CPU can only be established by considering the characteristics of the hardware from which it is constructed. This can be done in many ways: in VLSI design it is usual for a circuit to be analysed at different levels of abstraction: solid state physics, transistors, gates, functional blocks. Low level descriptions are more detailed: in establishing the characteristics of a particular circuit a global low-level analysis would be ideal, this is seldom possible because the computation required to do it is not feasible. Instead a number of levels of analysis are used; each level calculates the operational parameters which are used by the next level up.

The rôle of abstraction in hardware design is similar. An ideal design algorithm would be to analyse all possible designs at a low level and select the one with the best performance and cost. This is impractical, the number of possible designs is exponential in design size and proving even one low level design to be correct without recourse to high level abstraction is usually a combinatorial nightmare.

Hardware design must thus proceed by making some high level decisions about structure and then working through the necessary low level details. It is difficult because until these have been worked out it is difficult to be sure that the high level decisions made in the design lead to high performance. Formal design tools for verifying and estimating the performance of a particular design do not help to formalise high level choices between design strategies. Computer design still proceeds by trial and error: a computer designer makes informed guesses about the shape of a new architecture based on what is understood about the performance of existing architectures.

This chapter examines a set of models of concrete hardware which facilitate understanding of the performance of multiprocessor architectures. The models here are not formal languages for the description of hardware operation, such as Hoare's CSP [Hoa85] or a model proposed by Monteiro and Pereira in [MP86] which describes clearly the formal properties of general communicating *asynchronous* systems. I propose models of hardware which explain the performance of differing implementations.

The lowest level of these is that described below, where the operation of hardware is related to time, and hence performance.

3.1 A Low Level Model

3.1.1 Synchronous and asynchronous hardware

Hardware may be modelled by functional blocks which perform operations *sequentially.* The operation of a memory requires this description, the value of a read is defined to be that remembered from the most recent corresponding write. Something which happens at an instant will be called an event. A memory may be

described by a sequence of read and write events.

If the operation of the block can be described by a single sequence, defining a total temporal order on events, the block will be called **synchronous**. Synchronous blocks may be composed synchronously to form larger blocks. Operation is still described by a totally ordered set, each object in the set is a tuple of operations, one for each of the composed blocks.

In contrast blocks may be composed asynchronously, with communication between blocks imposing only a partial temporal order on all operations. The operation of an asynchronous system may be described by a partially ordered set of events.

The performance of a synchronous system is defined by the rate at which events can happen: its **bandwidth**. Simple hardware may be described synchronously with reference to a fixed frequency global clock. Here the bandwidth of the system is constant. A more complicated synchronous description, for example a von Neumann instruction sequence, may have a variable bandwidth with different operations taking a different length of time.

Analysis of the performance of an asynchronous system is more complicated. Performance is limited by the **latency** between ordered events. Consider the minimal directed acyclic graph that is equivalent to the partial order on events defining the system. Each arc on this graph may be labelled with a latency, the real time between two events is given by the length of the **critical path** between the events. Here the length of a path is the sum of the latencies of its arcs, and the critical path between two events is the longest path connecting them.

It should be clear from this that synchronous systems are just a subset of asynchronous systems. The words *synchronous, asynchronous* may be appropriate at different levels of abstraction in the description of the system. All hardware is at a low level asynchronous: operation is determined by logic levels and delays through gates—in the limit their behaviour must be analysed with analog rather than digital electronics. When gates happen to form flip-flops with a common global clock the system may most usefully be described synchronously—however any determination of clock rate (and hence the system's bandwidth) must refer to critical paths through gate delays in the lower level asynchronous description. A large functional block which is globally clocked may also be given a simple approximate asynchronous interface description.

For example a memory system may be clocked synchronously in common with a CPU and take 10-12 cycles of delay from presentation of address to emergence of the appropriate data. Its interface with the CPU comprises a handshake, and so the memory may be described at a high level as an *asynchronously* cooperating unit. Successive levels of abstraction require a description which is asynchronous, then synchronous, then asynchronous again.

When analysing the performance constraints on a high level description of a design both asynchronous and synchronous styles of analysis may be appropriate. Synchronous performance limits are determined by the bandwidths of particular functional blocks, asynchronous performance limits by the latencies on critical paths in the system.

The DEG description of concurrent computation relates operations to time by specifying partial temporal orders that individual operations must satisfy and so is an asynchronous description. However the local descriptions of the operation of individual CPUs may most appropriately be synchronous. Reasoning about the performance of a multiprocessor system will thus use both synchronous and asynchronous models, corresponding to separate latency and bandwidth constraints.

A central problem in multiprocessor design relates to the use of synchronous units. Whenever computation can be synchronised the extra information that interacting blocks have about each other can be used to simplify design and reduce communication costs. The asynchronous nature of the concurrency in symbolic computation makes global synchronism extremely inefficient: in contrast the synchronous concurrency which can be exploited in vector and array computation is relatively efficient. The design of a multiprocessor system must balance the benefits of local synchronicity against its costs. The choice between synchronous and asynchronous use of resources must be made at many different levels in system design and is a compromise. An important example of this is considered in Chapter 4.

The next section looks in more detail at this tradeoff between synchronous and asynchronous use of resources.

3.1.2 Balancing synchronous and asynchronous resource use

In a CPU design the performance of some unit, for example operand fetch, may be described by both latencies and bandwidths. The latency of the unit is the time from the event that defines an operation to the event that completes it. The bandwidth of an operation is the maximum rate at which operations can be performed. Both parameters are important, and either may be the one which limits system performance.

If a unit x is used concurrently by n operations its bandwidth, f_x, and its latency, t_x, separately constrain its use. The frequency with which it can be used is $\min(f_x, \frac{n}{t_x})$. Concurrency relaxes latency constraints on throughput.

In synchronous systems concurrency is usually limited, since it requires organised correlation of different activities. An example of synchronous concurrency is tightly-coupled pipelining of von Neumann instruction fetch and execution, which works best when it is known a priori that these operations take the same time. In asynchronous systems resources may be used concurrently with more freedom, activities are unconstrained except by their competition for the shared resource.

This illustrates a beneficial effect of asynchronous use of resources: increased concurrency means that the design constraints on latency of the resource may be relaxed allowing a faster design.

An intrinsic disadvantage of asynchronous concurrency can easily be identified. Suppose that N asynchronous activities are each using a hardware resource X of bandwidth f_X. Each activity consists of a sequence of operations, some of which are requests to use the resource. Suppose that resource requests are made

randomly at a constant rate f_A, the distribution of time between requests is thus exponential. This simple model is a reasonable first approximation to the nature of resource use in arbitrary computations.

When an activity requests use of the resource it suspends until the request is granted, this wait does not alter the expected length of time between resumption and the next resource request.

Now consider the average use of the resource by these activities. The resource will be used for as long as the number of queued activities is not 0. Let

$$q_i = \frac{\text{Probability queue length is } i}{\text{Probability queue length is 0}}$$

and

$$\alpha = \frac{N f_A}{f_X}.$$

α is the amount by which the total rate of request of the resource exceeds its bandwidth.

Equating the transitional probabilities from and to each queue length we have:

$$q_1 = \alpha$$

$$(1 + \alpha(1 - \frac{i}{N})q_i = q_{i+1} + \alpha(1 - \frac{i-1}{N})q_{i-1} (1 \leq i \leq N)$$

From this we find that

$$q_i = (\frac{\alpha}{N})^i \frac{(N)!}{(N-i)!}$$

Let $S = \sum_{i=0}^{N} q_i$, then the fraction of time for which the resource is used is $1 - S^{-1}$ and the amount by which the activities are slowed down because of queuing is therefore $\alpha/(1 - S^{-1})$.

By inspection if $x = \frac{\alpha}{N}$ then $\frac{d}{dx}(x^{-N} q_i) = -q_{i+1}$, so:

$$x^{-N} S' - Nx - (N+1)S = -S + 1 \Rightarrow$$

$$S' = \frac{N}{x} S + x^N (1 - S)$$

This equation does not have a closed analytic solution, so neither does S. For the purposes of this investigation it is sufficient to observe that $\forall k < n$ S is bounded from below by $\{(1 - \frac{k}{N})\alpha\}^k$, so that for large N the resource is nearly fully used when α is near to and above 1, in other words when the total resource demand bandwidth is slightly more than f_X. However for small N the cost paid in some combination of high α or low resource use is considerable: in Figure 3.1a the utilisation of resource X is tabulated for different α and N.

This behaviour is characteristic of any resource shared by independent activities. If the resource is expensive *extra* concurrency must be wasted on it in order to ensure its nearly full use. An example of this can be found in Chapter 4 where the resource in question is CPU activity. In this case resource idleness impacts

α	N				
	1	2	3	4	5
1.0	0.67	0.80	0.86	0.89	0.91
1.2	0.74	0.88	0.93	0.96	0.98
1.4	0.79	0.92	0.97	0.99	0.99
1.6	0.83	0.95	0.98	1.00	1.00
1.8	0.86	0.97	0.99	1.00	1.00

(a): N activities competing for single resource

α	N				
	1	2	3	4	5
1.0	0.60	0.69	0.74	0.76	0.79
1.2	0.67	0.77	0.82	0.86	0.88
1.4	0.72	0.83	0.88	0.91	0.93
1.6	0.77	0.87	0.92	0.95	0.97
1.8	0.80	0.90	0.95	0.97	0.98

(b): Single activity with up to N queued requests for resource

Figure 3.1: Concurrent resource use

CPU performance directly, and the extra concurrency required to reduce this idle time results in higher CPU register storage requirements.

This calculation may be repeated in the different case of a single activity using a resource but which does not wait on requests until its queue of outstanding requests is full, for example a CPU fed by an asynchronous prefetch unit. With a maximum queue length of N and ratio of request bandwidth to resource bandwidth of α elementary queuing theory gives S and q_i, defined as above:

$$q_i = \alpha^N$$

$$S = \frac{\alpha^{N+1} - 1}{\alpha - 1}$$

This is tabulated in Figure 3.1b.

Another difference between synchronous and asynchronous systems may be identified. The low-level non-determinism introduced by asynchronism has its cost in the consequent extra book-keeping to keep track of which user each operation belongs to. I will be considering hardware where the number of users is relatively small (the reason for this will become apparent in the next chapter), so the storage overhead of appropriately tagging requests is low.

However a problem arises with contextual data local to a user which is needed by a shared resource. This must either be cached near the resource and randomly accessed by the tag on each request, or queued with each request. Either

strategy requires extra storage at least proportional to the maximum number of queued activities. This contrasts with a synchronous system in which no activity is queued and storage is minimal. The costs of asynchronism are illustrated by for example the Am29000 RISC CPU design [amd88]. In this processor throughput in a synchronous 4-stage pipeline is maintained by sophisticated hardware which keeps track of inter-instruction data dependencies and maximises instruction overlap. The registers to do this represent a large context which must be saved and restored on interrupts.

The designer is thus presented with a fundamental dilemma. In order to achieve freedom from latency design constraints where operations take variable lengths of time it is desirable that units cooperate asynchronously, however this results in extra overheads.

3.2 Data Storage

How can the data needed by a computation most efficiently be stored? It is possible to address this question by considering different ways in which the data associated with a DEG can be mapped onto physical storage locations.

Abstract data and physical locations

In this dissertation I distinguish between *abstract data* and the use of physical locations. The use of an abstract datum is defined by two operations, creation and lookup. The creation of a datum associates with the value of the datum a *name* which may be used later to look up the datum. In this model of abstract data it is impossible to update a datum. This makes the use of abstract data functional (free from side effects) and so is particularly appropriate to concurrent computation. The update of physical locations in a computer will correspond to the reuse of a location by different abstract data. Efficient reuse of locations is part of the optimised compilation of single threads of computation, not part of the specification of concurrent computation, because any reuse of a location imposes some sequentiality on the computations that use it.

Static and dynamic names

A name may have both a static and a dynamic component. In the creation of heap a unique dynamic tag is used to access a tuple of data. Each datum in this tuple is thus named by the concatenation of the tuple's dynamic tag and a static tag that distinguishes between elements of the tuple. A dynamic name is generated non-deterministically[1] at data creation time, a static name has a value that is known at compile time and will be correlated with other static data that make

[1]More precisely: dynamic names are allocated in such a way that no name is reused, but other than this the client of name allocation can make no assumptions about the sequence of dynamic names which is released to it.

up a program. Every random access location in a computer has a fixed absolute name. This name will allow direct access to, or writing of, the contents of the location. The advantage of static name definition is that absolute names can be used directly resulting in efficient physical access to locations.

The distinction between static and dynamic data allocation, made here, is fundamental to any meaningful use of data in a computation. The specification of the computation consists of statically named data produced by a compiler, its execution will require extra dynamically named data. Static names are static with respect to a particular compilation, and associated other static data. There is in general no reason why in the course of a computation intermediate data should not be analysed and perhaps transformed, with newly static specification of subsequent computation resulting. The cost of global data analysis is such that this is not often done. Typically a compilation step will take more than a million times longer than a single subsequent data read, so it is not worth applying compiler techniques to intermediate results likely to have a limited lifetime.

Dynamic cacheing of frequently used data can be seen to be a way of utilising some of the advantages of static names without the cost of global compilation. The absolute cache entry tags associated with a global name may index directly into a small fast data store. However the cost of cacheing a new global name is relatively small. A general theory of data use must allow the same abstract datum access by different names within different local contexts, where the datum may be cached in different physical locations. A way of formalising this is described in the next section.

3.2.1 Cache theory

A **cache** is a local memory within which frequently used data can be stored. This is managed by using local names (corresponding to the absolute names of the cache locations) and maintaining a dynamic mapping from local to global names. With respect to a particular cache at a given time any datum may be local, global, or shared. If the latter then its global and local names must be associated by the cache. Cache will be used here for both compiler-named local state (registers) and programmer-invisible fast storage (often called cache memory) which is automatically allocated. This will allow different strategies for cacheing to be compared and contrasted. The distinction between these two methods is very apparent to a compiler-designer, less so to a CPU designer where they represent methods of achieving the same end.

A datum may be created inside a cache by an ALU and return a dynamic local name which is also its absolute location name. This datum remains local until either its name is passed out of or its location is flushed from the cache. It must then be associated with a global name, and becomes a shared datum. When both the datum has been flushed and no local names to the datum remain in the cache the datum becomes global and uses no resources inside the cache.

Conversely a global datum may be cached by finding a free cache location and copying its value to the location. While a datum is shared, within the cache, its

global and local names are synonymous. However access from a local name is quicker because it does not require associative name mapping. The disadvantage of holding local name references in the cache is that they must be converted to global names when the datum is flushed from the cache.

Optimisation of local data in a cache becomes of interest when a high proportion of data created in the cache is of limited life and so can be garbage collected while still local. This sort of garbage collection within a cache has the function of localising data in the cache: the management overhead of the data is invisible outside the cache.

Single thread uniprocessors do not make use of garbage collection within caches. This would be surprising to an unbiased naive investigator, since a high proportion of intermediate data in most computation are short-lived. The reason lies in the very special optimisation that single thread compilation can make of data use. The study of computation has a heavy historical bias towards von Neumann sequential processing. The explicit reuse of datum locations is built into the structure of most high level languages as assignment to variables and so what is in single thread computation a useful static (compile-time) optimisation of data use has become a standard execution model. A major part of the technology of optimising compilers for von Neumann machines is concerned with the management of this mapping (see Aho, Sethi and Ullman [ASU86] pp 513-722).

In arguing this I am neither advocating the use of functional programming languages on aesthetic grounds, nor diminishing the importance of explicit static location reuse to all implementation. It is however a technique which works only for locations used by a single thread and it may not be so useful in the execution of concurrent computation. Certainly the model of non-destructive data use, presented here, is most appropriate for *describing* concurrent computation. Within it the reuse of locations can be evaluated as an optimisation free from the preconceptions that single thread implementation has imposed. When thinking in general about the use of data in a computation it is more natural to use a non-destructive abstract model of data use to specify computation, and then consider desirable optimisations.

3.2.2 Cache hierarchies

In uniprocessor designs we can sometimes consider cache hierarchies to be totally ordered sets (or equivalently linear unidirectional graphs). Separate instruction and data caches require a more complicated model, as do multiprocessor designs, where more than one ALU must be fed with data. Two possible extensions to this, in order of increasing generality, are for the hierarchies to have the structure of trees or directed acyclic graphs.

A **tree hierarchy** of caches contains **nodes** each of which is equivalent to a number of separate caches with global name spaces identified. A datum is cached by a node if it is local to any of its constituent caches. If a datum is cached by any two nodes it will be cached by every node on the unique path that connects them in the tree.

A node may be implemented by a number of local caches communicating with a shared global memory. In the tree structure communication between processors has much the same implementation as global data access. Any access which does not hit in a node will proceed towards the root of the hierarchy until it can be satisfied.

A complication results from the nature of inter-thread reference. In order to maximise concurrency it must be possible to export references (names) of threads which have not yet finished. These may be read and will result in synchronisation waits for the reading ALUs until they complete. The synchronisation management will happen at the level of the outermost cache node within which the thread has been exported, in other words the innermost cache node which covers both source and destination ALUs of the thread result.

This necessitates that the value of a thread be transferred to all nodes that cache it on the path between its ALU and the hierarchy root as soon as it is available. In the common case that a thread is read only once this method of communication is inefficient. Efficient communication within one node can happily be accomplished by a **snoopy bus** . Snoopy bus is a vague term used to cover busses in which each access is monitored by all bus users who source their own data as necessary. In non-destructive data use there is no need to share modifiable data and so snoopiness is merely a way of replacing two shared memory accesses by one user-to-user bus transfer. see

A tree cache hierarchy is only scalable if most data reference can be kept local to twigs of the hierarchy. This is not in general possible, as will be seen in Chapter 5. A scalable cache hierarchy must have multiple path, and hence high bandwidth, global data transfer.

A **Direct Acyclic Graph** (DAG) cache hierarchy may be used to provide this. The basic component of a DAG hierarchy is a node as in a tree but with multiple outgoing (from the ALU) connections. Each shared datum will have a global name on just one of these connections. A DAG hierarchy thus behaves like a switching communications network. A datum cached in a node of the hierarchy will be accessible by the processors that the node covers, nodes in outer levels of the hierarchy cover increasing numbers of ALUs—every outermost node covers all ALUs. An example of such a system would arise if cache nodes were to be integrated into, for example, a delta network [JHP79] as proposed for ALICE [MDF*87] or Project Flagship [WW87].

In a DAG hierarchy different data follow different paths through the hierarchy. Names of threads may thus be exported along a different path from that required in their reading, so every name must contain itself enough information to find where it is cached. This means that names must be passed outwards through the hierarchy by hashes on their addresses, at the outermost level each name must cache in just one node. It is difficult to predict what level in the hierarchy will be needed in reading of a datum, so all data must be written through to the outermost level of the cache. Storage in a dag hierarchy has two uses: a hit early in the hierarchy has lower latency than one at the outermost level, and multiple reads may be satisfied with traffic at a low level in the hierarchy. This use of

storage may be compared with broadcast bandwidth optimisation, where node storage is used to queue requests rather than data.

Mapping between local and global names

In a cache the administration cost of name mapping is critical. In general the more flexible the cache name mapping the more its hardware cost. A simple strategy is for each global name to map to a unique local name got by hashing its address. This hash can be simple, usually it is just the global address modulo some power of two. This allows efficient lookup directly from the global name, but suffers from a poor hit rate when the hashes of global names which are used together clash.

A more complicated strategy is to have a local cache content addressable on the full global name of the cached datum, and a cache replacement policy which tries to keep frequently used data in the cache. A typical high performance uniprocessor cache will use global name hashing into sets of a small number of content addressable locations.

An interesting possibility is for local to global name mapping to occur only on the interface between the cache and outer caches. Inside the cache all shared names are local and access the cache directly. Flushing of a name from the cache requires the renaming of all of its instances within the cache. This renaming can be accomplished in a single content addressable update operation. Section 3.2.4 investigates the advantages of a small cache of this sort close to an ALU—one of these is a reduction in word length of the ALU. Note that for this to be possible *all* internal CPU address-holding registers must be regarded as part of the cache and so modifiable as described below.

3.2.3 Data management

In uniprocessor CPUs data use is classified into, for example, **code**, **stack** and **heap**, for each of which different storage strategies are used. Stack data is taken to include all purely local or temporary values which are of limited scope, and heap includes permanently allocated (static) data. The efficient use of data depends on three things:

- How many times is the data to be referenced?
- How can it be established that the data is no longer needed?
- Can access of abstract data be mapped onto a small static name space corresponding to physical registers?

In order to investigate data use in concurrent computation we can look at the characteristics of the data used in a DEG.

Data in a DEG

The data needed by an ALU in the execution of a DEG may be classified into three categories:

Control data. Data needed to identify those nodes in the DEG which have been created but not yet fired. The control data must also distinguish between those nodes which are waiting on data and those which have data available and so may be fired.

In a conventional von Neumann compiled program control data comprises a code pointer and set of return addresses, managed dynamically, together with static flow control information embedded in code. Control data is thus highly static with only a small amount of dynamic data from conditional branches and indirect jumps. The disadvantage of this is the consequent statically defined synchronous execution order of ALU operations.

Code. Static data used to define the DEG and perhaps determine explicitly data allocation to locations or ALU operation order.

Intermediate data. Data on D-arcs in the DEG that is dynamically created and must be stored until it is no longer needed. The data associated with heap cannot usually be associated in any way with the ALU operations that use it because it is difficult to establish where, if anywhere, in the future evolution of a DEG a datum will be used.

In single-threaded compilation of a DEG most intermediate data can be stored in a FILO stack with data access happening only at the top of the stack. This leads to optimised storage in registers or a RISC multiple window register bank [TS83]. The tradeoffs here between register, shift register, and random access register bank cacheing are between cost of register access and cost of register update on stack push or pop.

In multiple thread compilation each thread requires a separate stack, since by definition different threads are asynchronous. These are much more difficult efficiently to localise, because localisation in fixed registers or a single stack incurs a high context switch cost on thread switching. *This cost is fundamental where CPU cacheing relies on the FILO nature of intermediate data use to optimise data storage.* The latency *vs.* synchronisation dilemma examined by Arvind and Ianucci [AI86], see Section 4.5.1, is a direct consequence of this and the use of thread switching to hide latency.

The structure of intermediate data use in multiple thread computation is that of a dynamically constructed acyclic graph. Each thread uses local data on a stack which may at any time be forced to wait on the completion of one or more other threads. Access to data on the active ends of stacks requires random access of a cache, however physical locations can still be reused.

The concurrent equivalent of a single thread stack is thus a random access cache of stack frames with explicit garbage collection of frames on function exit.

The efficiency of this sort of cache, contrasted with a von Neumann cache, depends on the way in which threads are scheduled and is examined in Chapter 5.1.

Garbage collection

In this thesis garbage collection will be used to refer to any scheme which recycles memory names, including stack allocation and reference count methods as well as mark-and-sweep, stop-and -copy, or scavenging types of the sort discussed in [Coh81].

In uniprocessor designs garbage collection is usually associated with the freeing of global memory resources rather than the reuse of local memory—the latter happens implicitly through the use of a stack. Languages which make heavy use of heap may use local garbage collection to preserve cache locality of this data. Reference count garbage collection is expensive to implement but other strategies may be more appropriate.

One bit reference counts can be held in the (unique) pointer to an object. The copying of this pointer destroys its uniqueness, the destruction of a unique pointer reclaims its associated store. Implementation of unique pointer garbage collection is very cheap with the right hardware support. Effective use of this depends on static analysis of code to identify those copies of a pointer which are 'ghosts' and will never be passed out of the current thread. Copying a pointer to form a ghost will not destroy its uniqueness, nor does destroying a ghost release store.

In a small local memory a possible form of garbage collection is the use of content associative scanning of local names, this is investigated in Chapter 3.2.4.

Explicit garbage collection of stack frames is the concurrent equivalent of reuse of locations in a stack; it presents no implementation difficulties when integrated with a fixed size allocation block cache design.

Global garbage collection is necessary in any language that uses heap and does not have 100% effective local garbage collection. Copying global collection methods become less of an overhead as physical memory sizes increase, this dissertation does not consider global garbage collection performance to be an important implementation problem. It is worth noting that any local garbage collection further decreases the overhead of global collection, and this is used in scavenging garbage collectors for greater efficiency [Ung84], but this is not the main motivation for considering local garbage collection. Local reuse of names results in decreased data bandwidth in the outer levels of a cache hierarchy: a particularly effective example of this is the reuse of register windows in a RISC.

Location reuse

The different implementation techniques listed in this chapter are all ways of ensuring the efficient reuse of local store; in Figure 3.2 they are listed in order of increasing flexibility and implementation cost. Concurrent computation makes reuse of store more difficult and so weights design towards the more flexible forms

Method	Drawbacks
Static allocation of registers	Limited to non-recursive threads
Dynamic allocation on a stack	Limited to a single thread
Explicit collection of stack frames	Complicated space allocation
Unique pointer local collection	Requires careful code analysis
Content associative collection	Needs local name translation in the cache

Figure 3.2: Location Reuse

of storage reuse. The cost of these techniques must be balanced against the increased efficiency of dynamic data cacheing that they allow.

3.2.4 Name-translating caches

In symbolic computation the use of a long word length is necessary in order to distinguish a large number of global pointers. If arithmetic, except counting or operations on small integers, is rare then pointer indirection and comparison is the only reason for having a long data word length. If the ALU is fed by a small local cache then local names in this cache would do just as well, and a reduction of data path length of 50% or more is possible. This may yield significant benefits in CPU complexity and speed. The cost of this strategy lies in the necessity of garbage collecting local names and translating between global and local names on all data transfer outside the local cache. In VLSI the cost of a content associative cache is small providing that name length is short; to implement local name garbage collection a content updatable cache is needed. This is not significantly more complicated than a content associative cache, and allows all local references to a local name to be changed when the name is flushed. Figure 3.3 shows this arrangement. In a technology such as VLSI which allows cheap associative caches

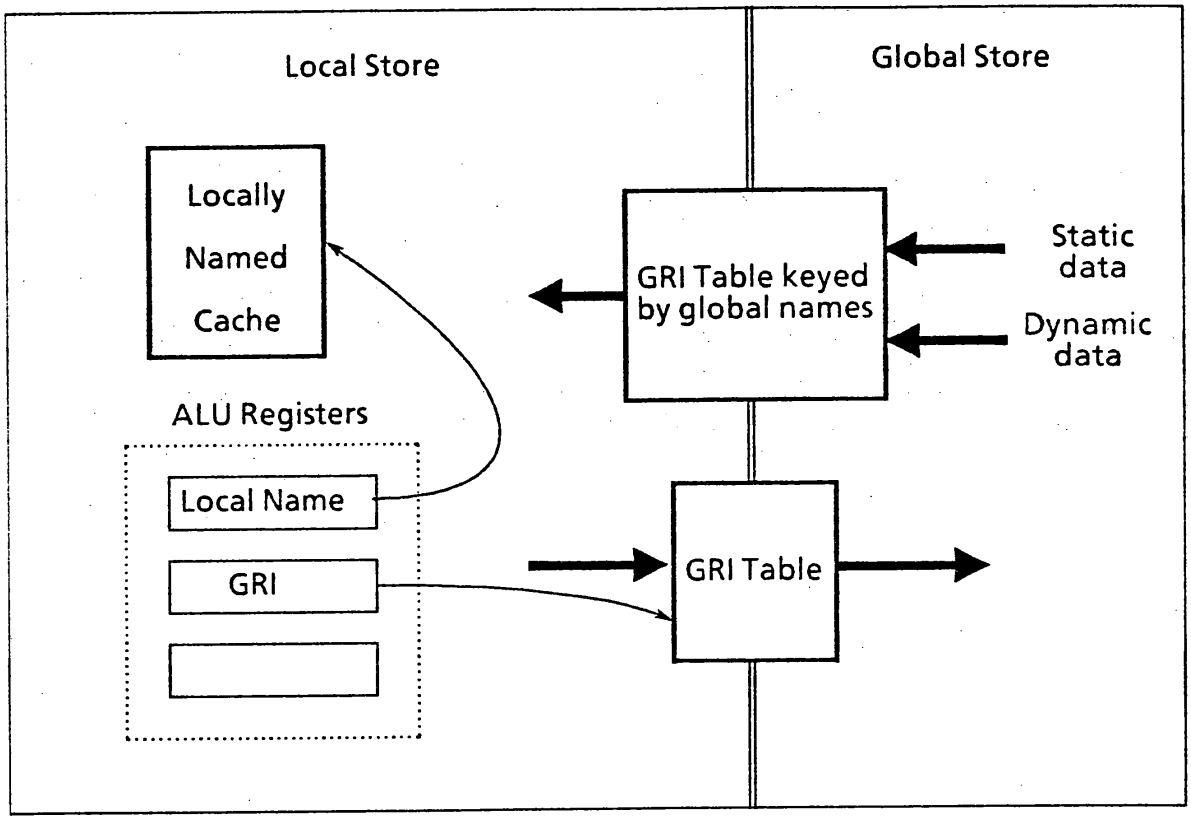


Figure 3.3: Name-translating Cache organisation

the design tradeoffs of extra complexity against smaller word length and flexible address management may favour this arrangement over a more conventional von Neumann CPU using an associative cache for on-chip memory location data cacheing.

Objects in the local registers and data caches of the ALU, all of which are content updatable, are one of the following:

- Small integers.
- Characters.
- Specialised markers (e. g. Nil).
- Local names pointing to locations in the cache.
- Global reference indirections (GRIs)pointing to a table of global objects which is held in the next outer cache and used to control name mapping.

The important characteristic of global indirections is that they be unique, so that local equality tests can be used to compare them. This necessitates a hash table of cached global names so that new access to global names already referenced in the cache maps to the same GRI.

The two tables to support name mapping are of roughly the same size as the local cache and only accessed on transfer out of it, so their overhead does not necessarily incur a large hardware cost.

In a dataflow CPU local name mapping is particularly beneficial because it replaces long dynamic tags on data by short ones. Static information accessing instructions can be mapped to an offset in code controlling one function body keeping the total tag size short.

Note that the benefits of having a small ALU wordlength are only visible when code and data data-paths are quite separate, this is likely to be true in a high performance design with substantial specialised cacheing local to the ALU.

Is this sort of radical name mapping viable? This depends partly on the size of local cache which can sustain a hit rate high enough to make it worthwhile. Concurrent computation makes necessary the cacheing of data that would in single-thread computation be stacked and that has typically a short lifetime. Thus name-mapping may be more appropriate in concurrent systems. The feasibility of VLSI name-mapping caches depends on details of cache design and will require further work to determine feasibility. This chapter shows that name-reuse is a fundamental requisite of data locality, so the low-level hardware support for this made available by a name-mapping cache should remain as a possible future design option whatever the implications of current feasibility studies.

Name-translating caches are based on the concept of indirection in algorithms, with the same advantages of flexible data management and compact data representation, and disadvantage of increased data access time. The work in the next chapter will show that given fine-grain concurrency in a CPU design latency becomes a less punishing design constraint. It may be that this will alter the design balance towards these caches.

One related application for indirection in data management can be found in RISC CPU design. The large number (for example 256 in the Am29000 [amd88]) of general registers in modern RISC instruction sets allows global dataflow optimisations of code. However optimal implementations of this put constraints on the registers used for inputs and outputs of subroutines with multiple entry points which cannot in general be satisfied without multiple copies of the subroutine code. The problem is a lack of register reorganisation bandwidth. One solution to this problem is to use register exchange instructions to interface one subroutine with a number of different calls with different entry and exit data registers. However register exchange does not match the usual bandwidth constraints on RISC register files (read bandwidth twice write bandwidth). One level of indirection in register access would allow high bandwidth register exchanges, and a data reorganisation bandwidth much higher than that achievable by register moves.

3.3 A Model Of Processors

This section describes the general model of architectures that emerged from the detailed investigation described later in this dissertation. It is motivated by a need to answer questions such as *'What is a multiprocessor?'* and unify the many apparently diverse design principles embodied in different multiprocessor designs.

The level of abstraction that I choose for this model is one in which the *band-*

widths and latencies of data access in different parts of the architecture are regarded as fundamental parameters and all other details of data access are not considered. This conceptual simplification need not lead to any unrealistic loss of detail in analysis because in a concrete design the calculation of the various system bandwidths and latencies will involve many low level details. By deferring consideration of these details it is easier to examine the fundamental principles that govern any multiprocessor design. Note that R. Wilson [Wil] observed that over a range of microprocessors memory bandwidth alone was a good predictor of complete CPU performance.

The need for an abstract analysis of multiprocessor hardware comes from the unusual complexity of this problem. My attempts at concrete design of uniprocessors for symbolic computation in the SKIM project [CJMN80,SCN84] were facilitated by a sense that well understood principles of uniprocessor CPU design could be applied with due consideration given to the details of a particular problem. When first studying multiprocessor design I had no such sense of what the fundamental design constraints are. This chapter addresses part of this problem.

My model starts with a simple description of a uniprocessor. This may be regarded as an ALU surrounded by a **storage hierarchy**. The function of the ALU is to perform those operations on data specified by some static program and initial data. This in general requires repeated access of not only the static program definition but also dynamic intermediate data generated and then used in the execution of the program. The job of the storage hierarchy is to aid the ALU by presenting it with this data in a way which allows a high ALU bandwidth. This bandwidth, f_{ALU}, is the performance of the uniprocessor. The task of architecture design is to maximise, over the class of computation specified by the design problem, this bandwidth for a given hardware cost.[2]

The function of the ALU itself is usually less critical (although in single thread RISC computation ALU latency can be a significant overhead) than that of the associated storage hierarchy. Heap access may therefore be regarded as an ALU operation on par with strict arithmetical operations—the function of an ALU in this is just that of a single physical point from which addresses are issued, and to which results return, in the physical access of store. In a dataflow machine this sequentiality is imposed by matching store access rather than program specification [AC86]. Chapter 4 looks in more detail at this difference which is not found to be fundamental.

The operation of a uniprocessor may be analysed by considering data fetch bandwidths and latencies at each point in the storage hierarchy. Going away from

[2]Hardware cost is not easy to specify: it usually contains separate component cost and design time constraints. Over the last ten years the exponential increase with time in VLSI density for a given component and packaging cost means that designs which are either very quick or quickly and efficiently scalable to different technology have in real terms (quite apart from amortisation of design cost) been better than designs which make painstakingly optimal use of particular technology. Whilst this seems likely to continue for the forseeable future the rate at which available technology improves, and hence the amount of low-level optimisation appropriate in a high performance design, may change. Consideration of this and of the ways in which design tools and methodologies may alter design efficiency is not the subject of this dissertation.

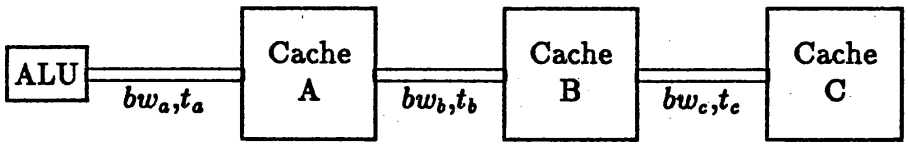


Figure 3.4: Simplified Model of Uniprocessor Architectures

the ALU fetch bandwidths decrease as successive memories localise more of the data (physical memory bandwidths may increase, for example in a memory system with interleaved banks of memory). As observed in Section 3.2 the memory in a storage hierarchy may be used either as a cache or a set of absolutely referenced locations, however these methods of localisation are not fundamentally different, so here any memory that localises data will be called a cache irrespective of its structure. This model of uniprocessor architectures is illustrated in Figure 3.4.

A number of problems arise from the simplicity of this model. No distinction is made between read and write bandwidths and latencies. This is a reasonable first approximation because read operations usually predominate over write operations, and in any case the respective bandwidths and latencies are usually quite similar[3]. The model can where necessary be extended by distinguishing between reads and writes, or stack and heap memory access. The freedom to do this will be used whenever hardware considerations make it important to do so. For example if it can be shown that data which is defined at compile time, like code, can be most efficiently be cached separately from dynamic data, *and that this increase in efficiency results in a significant overall increase in efficiency*, then the distinction of code from other access will be useful.

A more fundamental limitation of this model results from its implicit assumption that cache access can reasonably be described by constraints on overall bandwidth and latency. Often the cost of an access will be determined in a complicated way by previous and subsequent accesses. The detailed study of any particular architecture will throw up important hardware constraints which limit performance and cannot be explicitly described by a simplified model. This sort of complexity is architecture-dependent and cannot easily be used in the comparison of different architectures. Later work in this thesis will demonstrate that the differentiation between access latencies and bandwidths, fundamental to my model, is necessary

[3]Note that the costs of reads and writes especially in multiprocessors, will not necessarily be equal.

to encapsulate those features of low-level design which substantially affect high-level design.

The relevance of this model to uniprocessor design analysis, and its limitations, are considered in Chapter 4. It is at first surprising to see that such a simple model can lead to interesting analysis of design. The importance of separate consideration of system bandwidths and latencies is related to the nature of asynchronous concurrent computation, described abstractly by DEGs. Understanding of this is not important in the design of single processor computers where conventional design techniques lead to optimal designs. It becomes important when considering possible multiprocessor architectures.

An ALU in a multiprocessor system can equally be fed by a storage hierarchy. Extra complexity is introduced by the necessity for inter-thread export and reference. These two operations have no counterpart in a uniprocessor computer. They may simply be incorporated in a tree-structured storage hierarchy by communicating between ALUs with accesses to the first cache that feeds both ALUs.

This naive approach to multiprocessor organisation does not scale to large numbers of processors easily. Near the root of the storage hierarchy very large bandwidths are needed to accommodate highly global inter-thread communication. If reference and export could be successfully localised to part of the hierarchy this would not be so. Unfortunately it turns out that although most computation can be efficiently mapped onto multiple ALUs with local export, thread reference remains (usually) a highly global operation. Chapter 5.3 investigates this unfortunate fact of life for the multiprocessor designer.

The impossibility of maintaining locality has lead to a disenchantment with the use of hierarchically organised general multiprocessor systems. This thesis re-examines the question and concludes that storage hierarchical models of architectures need not be completely abandoned. To analyse this I remain with a tree-structured model of storage. However the connection to an outer cache may be high bandwidth and correspond to a global multiple-path message-passing network. Where most messages in this network are global it may be inappropriate to have any shared global storage. If this does exist it must be arranged in multiple banks whose access is interleaved. The distinction between high bandwidth interconnect and limited bandwidth interconnect can thus be seen as a separate issue from the amount of storage desirable at any point in the hierarchy.

Figure 3.5 shows this model of multiprocessor architectures. A connection connects a number of sub-hierarchies to a shared cache, which itself connects at the next level of the hierarchy. At each connection the outer cache may be of size 0, indicating that all shared memory is physically local to one of the subtrees. The model does not specify the details of how inter-thread synchronisation (export and reference) is accomplished. This is because different strategies may be appropriate for different architectures and at different levels of the hierarchy. These details are considered later on and will be important.

Without proceeding to any more detailed description of architecture two fundamental principles may be identified which are important when mapping multiple thread computation onto a multiprocessor architecture.

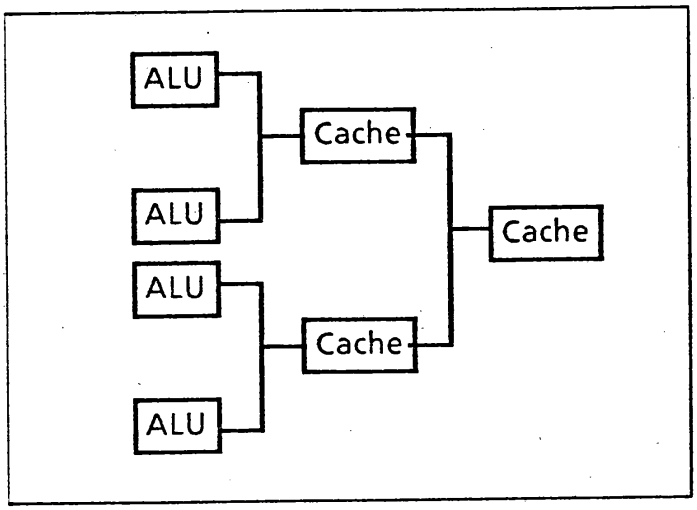


Figure 3.5: multiprocessor hierarchy

1. Latency Hiding Optimisation.

   In single thread computation access latency reduces performance by putting latency waits in the critical path of the computation. In conventional uniprocessor architectures much ingenuity is expended in avoiding these by prefetching data which is either certain or likely to be required before it is demanded. In multiple thread computation these techniques may still be used, but another way of increasing performance is available. Whenever execution is waiting on some operand with a long latency a switch to another thread can be used to hide the latency. If thread switching time is less than the latency this results in higher ALU throughput.

   This optimisation is used in conventional multi-tasking systems where disc access latency is hidden in this way, it works because the total cost of task switching is less than the expected disc latency. The real cost of task-switching is difficult to predict because as well as the direct cost of administration and static location flushing (of registers for example), there is an impact on dynamic data cacheing. The observation that system code exhibits typically lower cache hit rates than user code in a multitasking system [Smi82] can be attributed to the high level of task switching in system code.

   At each connection in a storage hierarchy concurrency may be used to decouple access bandwidths from access latencies. The product of access bandwidth and access latency for a connection will be called the storage of the connection.

2. Broadcast Bandwidth Optimisation.

   The existence of multiple access requests at any level in a storage hierarchy gives rise to another possible optimisation. Where two outstanding read requests are for the same data they may be satisfied simultaneously, so reducing the total necessary bandwidth. This technique lends itself to systems

which interconnect with shared busses because the inherent ability of such a bus to broadcast data can be used to reduce demands on bus bandwidth. In multiple path interconnection this optimisation can still be used by making the routing of data requests dependent on a hash of the name of the requested data. Thus at all levels in a multiprocessor hierarchy system bandwidths can be reduced in this way.

This optimisation is interesting because the likelihood of bus sharing is the product of the access latency at a given connection and the number of processors using it. This is one way in which long latencies can increase the performance of a system whose throughput is limited by a bandwidth. Certainly it can be used to mitigate the effect of necessarily long latencies, and of necessarily global accesses.

The principle of broadcast bandwidth optimisation can be seen in very large multiple user database designs where data access requests have a long latency and are satisfied multiply whenever this is possible.

These principles are potentially ways in which the concurrency available in computation can be used to *reduce* the effect of ALU demands for bandwidth on design in a multiprocessor system. This is welcome because the necessary loss of locality in these systems is a large and intrinsic extra cost. Two questions arise: how can these principles best be implemented, and, having done this, do they lead to significant optimisation?

This dissertation examines in detail the first and most clearly applicable of these principles. The second is considered only briefly in Section 5.3.6.

3.4 Summary and Related Work

This section develops some natural models of hardware, data localisation, and multiprocessors which are extremely general in application. This work bears some resemblance to dataflow graph based analysis of computation, however there is a crucial difference. I base this work on no assumptions about the nature of computation except that it should be in a style which is highly concurrent. Within this framework it is therefore possible to consider all of the different possible ways of implementing a concurrent program.

Together with Chapter 2 the material in this chapter forms an introduction to computer design and programming which is consistently free of descriptions which restrict possible styles of implementation. The usefulness of this will be seen in the next chapter, where we will use these ideas to contrast dataflow and von Neumann CPUs.

In Section 4.5 we will consider further the importance of synchronism in CPU design, and distinguish between (temporal) synchronism and (spatial) coherence.

The investigation of name-translating caches here is tantalising. The concept of such a cache develops naturally from consideration of what caches are, however it has not been shown that these caches are ever useful. Symbolic computation

spends more time rearranging data structures on a heap than other styles of computation, the possibility of naming all CPU data objects with short names and using this level of indirection to implement high bandwidth data reorganisation and local garbage collection is therefore attractive.

Chapter 4

CPU Design

This chapter applies the ideas discussed in previous chapters to an analysis of the problems encountered in CPU design.

In the first half of the chapter von Neumann and dataflow CPUs are contrasted and a space for hybrid designs is identified. On very general arguments the new designs are shown to balance fundamental hardware design constraints of latency and bandwidth.

The second half of the chapter looks in more detail at this new design space, considers how current RISC design evolution has coped with fundamental design constraints, and shows that faster CPU technology pushes the design balance increasingly from RISCs towards D-RISCs.

4.1 Von Neumann and Dataflow CPUs

Two distinct issues arise when considering the execution of DEGs on a CPU. Chapter 3.2 defined the issue of storage hierarchy design—how to present operands to an ALU as fast as possible by efficient use of local store.

The other issue is that of execution sequencing (or *scheduling*), and it has two parts. What is the overhead in managing a particular scheduling strategy, and what constraints on scheduling, beyond those defined intrinsically by the DEG, should be imposed? Deterministic scheduling of operations allows better static reuse of registers, so this relates directly to the use of storage.

These two issues encompass all of uniprocessor design, and are an essential preliminary to the understanding of multiprocessor design.

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.

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.

There are corresponding disadvantages to this locality. As CPUs (for example SPARC [SUN]) have come to use more local context in registers close to a CPU the cost of asynchronous context-switching has increased: this is one undesirable result of locality in a RISC.

Another undesirable feature results from the sequential evaluation order imposed by von Neumann instructions. 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.

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 than an exposition starting with dataflow graphs. The execution model that tagged token machines use may be derived directly from a DEG description of computation, just as that of static dataflow machines relates to dataflow graphs. However dataflow graphs do not describe concurrent computation in a natural way, whereas DEGs do, so although historically tagged token machines are a development of static machines, conceptually they are both simpler and more fundamental.

Static dataflow CPUs may be seen as a development of tagged token machines in which the order of tokens on a statically defined dataflow graph arc determines their associated dynamic DEG arc. This can lead to implementation advantages, it also reduces potential concurrency, by requiring that all the DEG operations which are identified by sequence in this way occur in a fixed order.

Because of these problems static dataflow machines do not appear to be a useful departure from tagged token machines, they are not considered further in this dissertation and the word dataflow will be used to describe tagged token machines.

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 (see [AC86]) 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 [AC86], and seems to be the most satisfactory of those proposed for dataflow machines. The usefulness of a thread description of computation is demonstrated by its direct prediction of this implementation method.

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 4.3 below 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.

4.2 Latency and Bandwidth Constraints on CPU 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.

Let t_{fetch} and f_{fetch} be the respective latency and bandwidth used in operation fetch. In order for ALU bandwidth to be optimal a concurrency of $\sigma = f_{\text{fetch}} t_{\text{fetch}}$ is needed. If σ is higher than the concurrency obtainable from pipelining the CPU is limited by a latency design constraint. Performance may perhaps be increased by concurrent asynchronous execution of multiple threads. Certainly extra concurrency is available, and this will reduce the effect of fetch latency on performance.

The problem with using asynchronous concurrency in this way is that extra store is needed local to the ALU to hold the working context of the threads which

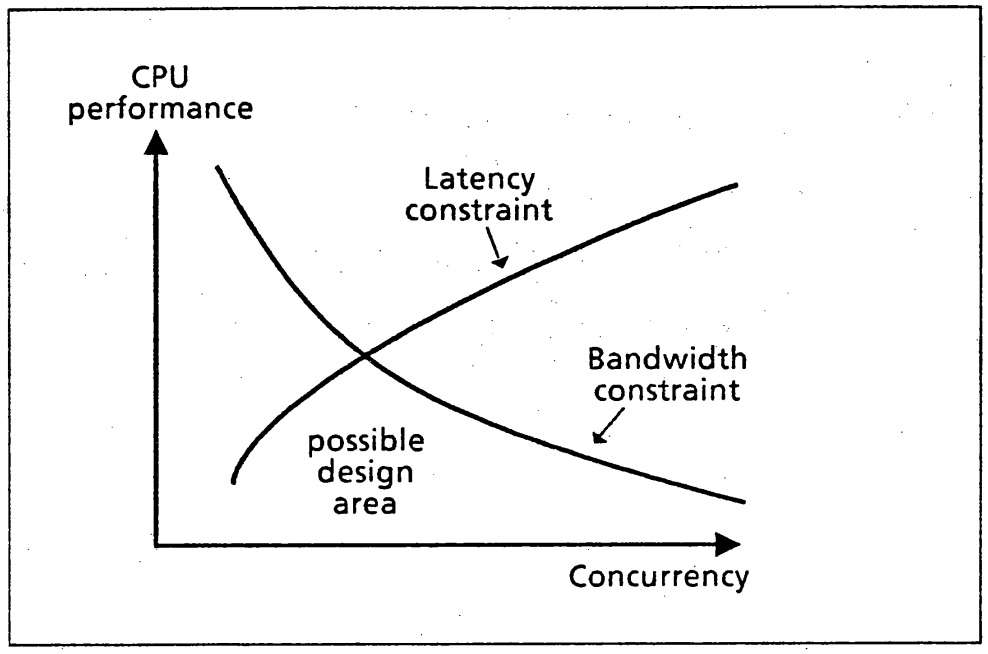


Figure 4.1: Latency and Bandwidth Constraints on CPU performance

are concurrently executed. If asynchronous concurrency is n this extra space is no more than a factor of n.

I assume that the performance of small caches is inversely related to the square root of their size, which [MC80] shows to be realistic in VLSI technology. Then the cost of supporting concurrent execution of n threads is an increase in fetch latency and decrease in fetch bandwidth by this factor. The ALU performance is thus separately constrained by latency and bandwidth as a function of n:

$$f_{ALU} < n^{\frac{1}{2}}/t_{\text{fetch}}$$

$$f_{ALU} < n^{-\frac{1}{2}} f_{\text{fetch}}$$

This shows that there is an *optimal* concurrency at which ALU performance is maximum, this is the square root of σ, see Figure 4.1.

In conventional uniprocessor CPU design pipelining can usually balance latency and bandwidth constraints without recourse to the less efficient asynchronous concurrency. This is not true in multiprocessors, where bus latency alters the design parameters by making $t_{\text{fetch}} f_{\text{fetch}}$ larger. This product measures the relative size of the latency constraint on CPU performance, it is of fundamental importance in CPU design and will be called the **access-overlap** of the CPU.

To see why access-overlap is different in a multiprocessor note that caches usually[1] have access times and bandwidths which are approximately inversely related (access-overlap of one) whereas multistage switching networks have message latencies which are necessarily longer than this. Of course this can be partly obscured by the design of networks which pass messages bit sequentially. Similarly

[1]See Chapter 4.3.5 for possible exceptions

busses which interconnect n processors have when fully loaded an expected message latency which is $n/2$ times longer than their cycle time. Thus a fundamental characteristic of processor interconnection is that data access latencies are long.

High global memory access latency thus moves the ALU design constraints so that a higher concurrency is necessary for balance. In this simple model local cache misses may be accommodated by a corresponding increased t_{fetch}. The cost of this can be absorbed by switching to a new thread, subject to the caches update bandwidth constraint.

This identifies a fundamental principle in CPU design. Thread switching is useful to hide latencies caused by memory access when the storage (see Chapter 3.3) of the memory connection is high. In single thread uniprocessor designs this is unusual because all design tries to minimise latencies.

One problem with asynchronous concurrency was described in Chapter 3— extra concurrency is needed to ensure that an ALU stays fully loaded. In any asynchronous loading of a device the device loading is $1 - e^{-l}$, where l is the amount by which the combined asynchronous load exceeds the device bandwidth. This was ignored in the analysis above— it does not greatly change the estimate of optimal concurrency.

This section shows how in theory context switching relates to performance. In real design the granularity of context switching is important: data which need not persist between thread switches can be allocated storage which is local to the ALU and not thread-context related, this is clearly important.

Analysis of the compiled LISP test programs detailed in Section 5.5 show that the typical time between thread-switching in symbolic computation is about 50 RISC CPU cycles. This time is long enough for single thread optimisation to remain important.

One way of using asynchronous concurrency at this granularity is by using a RISC-like register bank with windows corresponding to different threads. Thread switching can be very fast and the cache performs well in both single thread and multiple thread computation. This sort of CPU is considered in the next section.

Dataflow machines and RISCs

In the light of this analysis what can be said about the relative merits of dataflow and RISC CPUs? The poor ALU performance of simple dataflow machines is explained, they are hopelessly bandwidth limited. The synchronisation latencies in multiprocessor systems mean that conventional von Neumann CPUs also perform badly, a machine is needed which executes a small number of threads concurrently.

Interestingly this conclusion does not resolve the debate between dataflow and von Neumann machine designers about the relative merits of their architectures. A RISC can be optimised for multiprocessor applications by suitably designing its cache. Equally a dataflow machine can be optimised by local cacheing which optimises the execution of a small number of threads. Whereas in RISCs the context associated with a function is explicitly localised in dataflow machines it is not. However tag information could perhaps be used to control cache update.

One theoretical advantage of dataflow is that abstract data are allocated to physical locations for the shortest possible time, in compiled von Neumann code stack frames are often larger than the amount of usable data that they contain.

The possibility of designing dataflow machines which localise a small number of threads identifies an exciting area for further research at a level more concrete than this dissertation. For example name-translating caches (see Section 3.2.4) could be used to reduce the overhead of local data in a dataflow machine. My own familiarity with the realities of dataflow design is negligible so the extension of my theoretical ideas to this area is not something which I can address here.

4.3 D-RISC CPUs

A RISC CPU optimises function calls by using a cache of stack frames which is arranged as a circular buffer onto a conventional general purpose cache. A subsidiary issue is whether all ALU operations should be cache to cache, or whether data which have been read in the current stack frame should be separately stored in a volatile register bank which is transparent to the software model.

The necessary modification to accommodate concurrent threads is clear from the description in Chapter 3.2. Cache management must be associative with explicit garbage collection of frames on exit. This is more complicated to administer than a circular stack cache and probably means that a fixed block size must be used for each separately reusable chunk of store—either 8 or 16 words is an appropriate size.

This type of cache is not just a particular neat design idea, it is the only way efficiently to localise concurrent threads, with the resulting tree-structured dependence of intermediate data.

DRISC CPU design in VLSI is similar to RISC design, with the same concern for critical latencies in data paths and appropriate on-chip cacheing to meet necessarily limited off-chip bandwidth limits (see [Kat85]). The differences are described below and relate either to efficient management of the block associative cache—this will be called the **frame cache**—or to scheduling of multiple threads of computation.

4.3.1 Frame cache design

In a frame cache blocks are allocated and deallocated explicitly by executing threads, although on cache misses dynamic block replacement is necessary. However block allocation happens frequently (roughly every 20 cycles) whereas block replacement happens only every 1000 cycles. These two times set the parameters for frame cache design.

The frame cache could use an least recently used (LRU) replacement policy to maximise hit rate for a given cache size since this is a constraint on cache access latency and hence ALU cycle time. Where most object lifetimes are short, as in a frame cache, LRU will considerably more effective than random cache replacement, as I show below.

Assume that allocation of new data from the cache happens at a constant rate and let T = the time taken for store allocation equal to the size of the cache. Consider the probability that an object will be in the cache at a time L after its creation (with no previous accesses). For an LRU cache this is 1 if $L < T$, otherwise 0. For a random replacement cache, if the cache contains N objects, the corresponding probability is:

$$(1 - \frac{1}{N})^{-\frac{NL}{T}} \approx e^{-\frac{L}{T}}.$$

This shows that, as would be expected, LRU performs better than random replacement as the lifetime of cache data decreases. In order to simplify the subsequent calculation choose units of time such that $T = 1$. Now suppose that lifetime frequency varies as $e^{-\beta t}$, so that large β implies short average lifetime. Real data lifetime distributions will not be exactly exponential, however this is a reasonable first approximation for short-lived intermediate result data.

The probability of a data hit from a random replacement cache is now

$$p_{RR} = \frac{\int_{t=0}^{\infty} e^{-(1+\beta)t} dt}{\int_{t=0}^{\infty} e^{-\beta t} dt} = \frac{\beta}{1+\beta}$$

whereas that of a hit in the LRU cache is

$$p_{LRU} = \int_{t=0}^{1} e^{\beta t} = 1 - e^{-\beta}.$$

Now consider a random replacement cache of given hit-rate p_{RR}. If the corresponding LRU cache size for the same hit-rate is k times that of the random replacement cache,

$$1 - e^{-k\beta} = \frac{\beta}{1+\beta} \Rightarrow k = \frac{\ln(1+\beta)}{\beta} = \frac{(1 - p_{RR}) \ln(\frac{1}{1-p_{RR}})}{p_{RR}}$$

so as $\beta \rightarrow \infty$ (and hence the cache hit rate $p_{RR} \rightarrow 1$) so the ratio of LRU to random replacement cache size varies as

$$\epsilon \ln(\epsilon^{-1}) \quad \text{where} \quad \epsilon = 1 - p_{RR}.$$

Thus in a cache of this type with a high hit-rate LRU is a much better strategy than random replacement. Figure 4.2 lists some values of k for different p_{RR}.

In a D-RISC frame cache each block is associated with a global name pointing to a globally consed block of external memory. New pointers to free store are required on block replacement by a new local block, or when local data becomes externally reference and so must be flushed (this is controlled explicitly by the executing program).

Block allocation and deallocation is managed by a special purpose hardware stack of free block indexes: free blocks keep their references to free external memory with no consing overhead on block allocation.

Hit Rate	LRU/RR size ratio
0.5	0.69
0.75	0.46
0.9	0.26
0.95	0.16

Figure 4.2: Comparison between LRU and Random Replacement (RR) cache sizes for given hit-rate with exponential data lifetime distribution

References to cached blocks do not exist except in other local cached blocks. When a block is flushed its references to other blocks must be flushed as global names, on its re-cacheing these if read must be associated with current cache contents, and initiate an allocation if the match misses. However this allocation is unusual in that it replaces the external store previously associated with the free block with that pointed to by its own external name. Thus block allocation can lead to external store freeing and therefore consing and reclamation must be managed by a stack (rather than a FIFO) of free external blocks, this can itself conveniently reside in the cache as a cdr-removed list.

In general a queue of objects which may be extended at either end and collapsed at its head is a structure well suited to frame cache cacheing. The queue is held as a tail-pointer-removed list with pointers to its two ends which point to words in frame cache blocks. The efficient support of such queues can help with optimisation of scheduling, synchronisation, and pipeline communication between threads.

4.3.2 Scheduling

In a DRISC multiple concurrent threads are available for execution, and on any cache miss it is desirable to switch to a new thread. Threads may thus either be waiting for a cache update, which may if it comes from the reading of another thread take arbitrarily long, or ready. The simplest way to manage threads is with a LIFO of ready threads which is popped when a context switch is required, and pushed when a cache update for a waiting thread is satisfied.

Threads which are ready may not have all of their immediate data cached since this may be flushed before they are restarted. Thus thread switch proceeds by popping ready threads sequentially from the LIFO, checking if their immediate context is cached, and if not requesting a cache update.

There is a difference between a wait on another thread, and a wait on a local cache miss. If the cache has an extra bus for updates these may happen asynchronously with execution, but the cost of such a bus may be more than its worth. It is simpler for all cache update to be handled synchronously with execution, and no less efficient if total cache bandwidth is the main design constraint.

Waiting on a thread must however initiate global communication (if the thread is not local) and a wakeup signal on availability of data that is truly asynchronous

from program execution. Thus design optimisation through local cache miss latency hiding is a *uniprocessor* design issue and worthwhile only if the latency bandwidth product of operand fetch is high. In RISC VLSI design with memory off chip the bandwidth of off-chip busses is an important design constraint, the latency bandwidth product of these is currently about 2 (for example, see [amd88,mc888]). In contrast the hiding of global synchronisation latency will typically will result in longer delays and so a higher latency bandwidth product. In RISC CPUs fetch latency is hidden by pipelining instruction fetch and execution, as technology allows faster CPU speeds this requires a longer pipeline.

The analysis in the last section showed that very general bandwidth and latency constraints on CPU design make thread switching to hide a latency in a system attractive when the product of the latency and the corresponding access bandwidth is sufficiently large. In a DRISC cache this depends on the relative bandwidths of cache and local memory access. If local memory is significantly slower than cache then cache bandwidth can be conserved by using a buffer between memory and cache, and transferring between this and cache synchronously with program execution. Each block in this buffer used for a memory read has an associated thread which must be executed to complete the transfer and release the buffer space. Thus two lists of ready threads now exist, with priority given to the buffer list.

In a multiprocessor system a third list of ready threads is needed containing imported computation from other processors: this must be of lower priority than the local threads to maintain locality of computation (in as far as this is possible) in highly loaded systems.

It will be shown in Chapter 5.1 how scheduling of threads can best be managed by LIFOs, this requires little extra hardware cost since they can be held in the frame cache. It is particularly fortuitous that the hardware resources necessary for efficient thread context switching in a multiprocessor system can also support the structures needed for scheduling. This is one reason for making cdr-removed lists a low-level supported object in a DRISC.

4.3.3 Arvind's argument

Arvind and Ianucci's discussion of latency and synchronisation in von Neumann and dataflow CPUs may be found in [AI86]. I will examine this argument using the models of computation and hardware developed in chapters 2 and 3. Sequential thread execution on a von Neumann machine uses stacks for dynamic allocation and reclamation of physical locations for intermediate data. This sort of data use can efficiently be localised in a small register cache, leading to the high performance attained by simple RISC CPUs. The data localisation reduces *latency* of data access which in a RISC is an important determinant of performance. It will be shown in the next chapter that in such a machine discontinuities (see page 70) in thread execution result in stack flushing and consequent loss of efficiency.

First I will investigate this argument in the case of a simple CPU, the Acorn RISC Machine [arm87]. This single chip CPU has a well balanced design and is

highly synchronous. For the purpose of this analysis I use a simplified description of the ARM architecture.

The ARM has on-chip a set of 16 registers, the instruction set provides three types of data manipulation instructions:

- Three operand register to register ALU operations.
- Single word moves between a register and a register indirect indexed memory location.
- Multiple word moves from a set of registers to consecutive memory locations.

No data is cached on-chip except for the registers: code is fetched in a single instruction pipeline from external memory. This memory is dynamic RAM which uses column addressing whenever possible, with memory access taking 1 CPU cycle, full row and column addressing takes 2 CPU cycles. Within these external memory access constraints instruction times are optimal:

ALU instructions	1	
Single moves	4	
Multiple moves	$3+n$	(n = number of transferred registers.)

Thus the ARM has a very simple cache hierarchy: data is held either in the 16 CPU registers or in main memory. Register data access is faster than memory data access by a factor which varies depending on the use made of data. External memory read or write bandwidth is 0.25 for single transfers, nearly quadrupling for multiple transfers. Register access equivalent bandwidth is between 1 and 3, depending on whether good use is made of the 3 operand ALU instruction. However datapath latencies in the CPU are evenly divided between ALU, barrel shifter, and register access, so the true register access latency is 0.3. From these figures the ARM use of data locality can be inferred. Data in memory needs between 5 and 13 cycles for random ALU operations, compared with 1 cycle for registers. However multiple word moves to or from memory are much faster. A stack frame of 8 words can be transferred between registers and memory in 11 cycles. On the ARM static register change time is thus short: 22 cycles for 8 word frames. The cost of thread switching on an ARM is thus the loss of a fraction $\frac{n}{n+22}$ of CPU time, where n is the average time, in cycles, between thread switches.

The value of n is dependent on application, in the compiled LISP programs which I examine in Section 5.5.2 it never falls below 50 cycles and is typically much larger than this. This puts an upper limit on the advantage that any D-RISC or dataflow architecture can get over an ARM in multiprocessor computation due to faster thread-switching. If the multiprocessor has a high bus latency this makes necessary a larger number of available threads to switch between just to hide this latency but does not alter this bound, because the average time between thread switches is not substantially changed. This simple example shows that the loss of locality in multiprocessor computation need not necessarily be so large as to make either dataflow or D-RISC design much more attractive in multiprocessor

than uniprocessor computation. It does not however give any information about the relative merits of D-RISC, dataflow and RISC architectures. An extra argument for using D-RISC or dataflow design in a multiprocessor is that since here concurrent computation must be supported concurrency can usefully be used to hide CPU latencies.

This example shows RISC CPUs at their best in concurrent computation, because the ARM uses so little locality. The relative disadvantage of locality loss becomes more severe as more data is localised and the block move bandwidth to main memory scales down compared with local register access bandwidth. Both of these may be expected in higher performance CPUs with more cacheing.

The relationship between multiprocessing and the use of locality to obtain high computational efficiency may be addressed more directly by considering the amount of D-RISC cache used by a computation. Define λ as in Chapter 5.4:

$$\lambda = \frac{\text{average inter-processor thread read latency}}{\text{average time between thread-critical inter-processor thread reads}}.$$

The concurrency introduced by thread-switching to hide bus latency is on average $1 + \lambda$. This may be related to cache use by the arguments in Section 5.1.3. Let S_λ be the cache size needed for a given hit-rate in a particular multiprocessor computation as a function of λ. Then

$$1 < \frac{S_\lambda}{S_0} < 1 + \lambda.$$

The value of S_λ depends on the degree of independence of the $1 + \lambda$ concurrent threads in the cache, which in turn relates to cache size: as this increases so does the likelihood that any particular intermediate datum which will result in a cache hit corresponds to a part of the computation's TCT which is shared between concurrent threads.

The multiprocessor scheduling optimisations for highly bandwidth limited computation all work by removing bus latencies from the critical path of a sequential thread, for example by prefetching data, so decreasing λ.

In a similar way relative cache size on a multiple thread D-RISC using average concurrency of n to hide thread execution latencies varies between 1 and n. It is instructive to compare cache sizes on a sequential RISC processor and a multiprocessor with $\lambda = 0$.

The sequential processor has a number of advantages:

- Code may be compiled for a deterministic execution order, so reducing program control flow non-determinacy and increasing static reuse of frame locations.

- Data flushing on a stack is optimal: in a D-RISC cache LRU replacement is almost as good but difficult to implement, LRC (least recently created) replacement is less good.

- It is difficult to allocate data in arbitrary sized vectors in a D-RISC cache. Two allocation strategies which seem feasible are single fixed size blocks, and a small number of binary scaled sizes of vectors, e.g. 4,8,16,32,64 words, using the binary buddy system. Neither of these have the storage compactness of arbitrary sized vectors, although compact list representation reduces this disadvantage by making desirable all list storage allocation in fixed size (8 or 16 word) chunks.

The comparison between dataflow machines and RISCs or D-RISCs is complicated by two differences which are orthogonal to arguments about locality:

- Dataflow machines use export on creation because the processor on which a thread is executed is always part of its tag, used to control data flow.

- Dataflow machines have a highly dynamic fine grain method of intermediate data allocation which makes name storage a large system cost but also simplifies data access. Because token storage is associative and matched tokens represent executable ALU operations data access for presentation to an ALU requires only unidirectional information flow.

The first of these differences is not fundamental: a dataflow machine can be augmented with thread references through which token supply indirects to allow deferred decision about the processor on which an ALU operation executes. The second one needs deeper analysis.

Figure 4.3 Shows the sequence of operations which occurs in dataflow and von Neumann instruction execution. In dataflow machines each ALU operation has its own dynamically allocated local storage, and is enabled for execution by the writing of the data that it needs. Instructions may be fetched concurrently at the grain of a single ALU operation, since a single completing ALU operation can result in several destination tokens each of which enable (by matching with existing tokens) a new ALU operation.

In von Neumann machines and D-RISCs local data is allocated on a coarser grain (by stack frame) and instructions operating on a given frame are totally ordered.

A compromise between these two strategies can now be identified. Keep storage allocation (by stack frame) as in a von Neumann processor so enabling the data storage locality optimisations which are now well understood, for example [ASU86], but allow multiple *control* threads operating on the same frame and contributing to a single final thread value. This is described in the next section.

4.3.4 Separating control And data concurrency

In Section 5.1.1 I will show that at the level of implementation a thread could be represented by a vector of local store locations and an instruction pointer, together with links either to other vectors (in a manner similar to conventional stack frames) or other threads. This structure when executed on a Thread Execution Unit

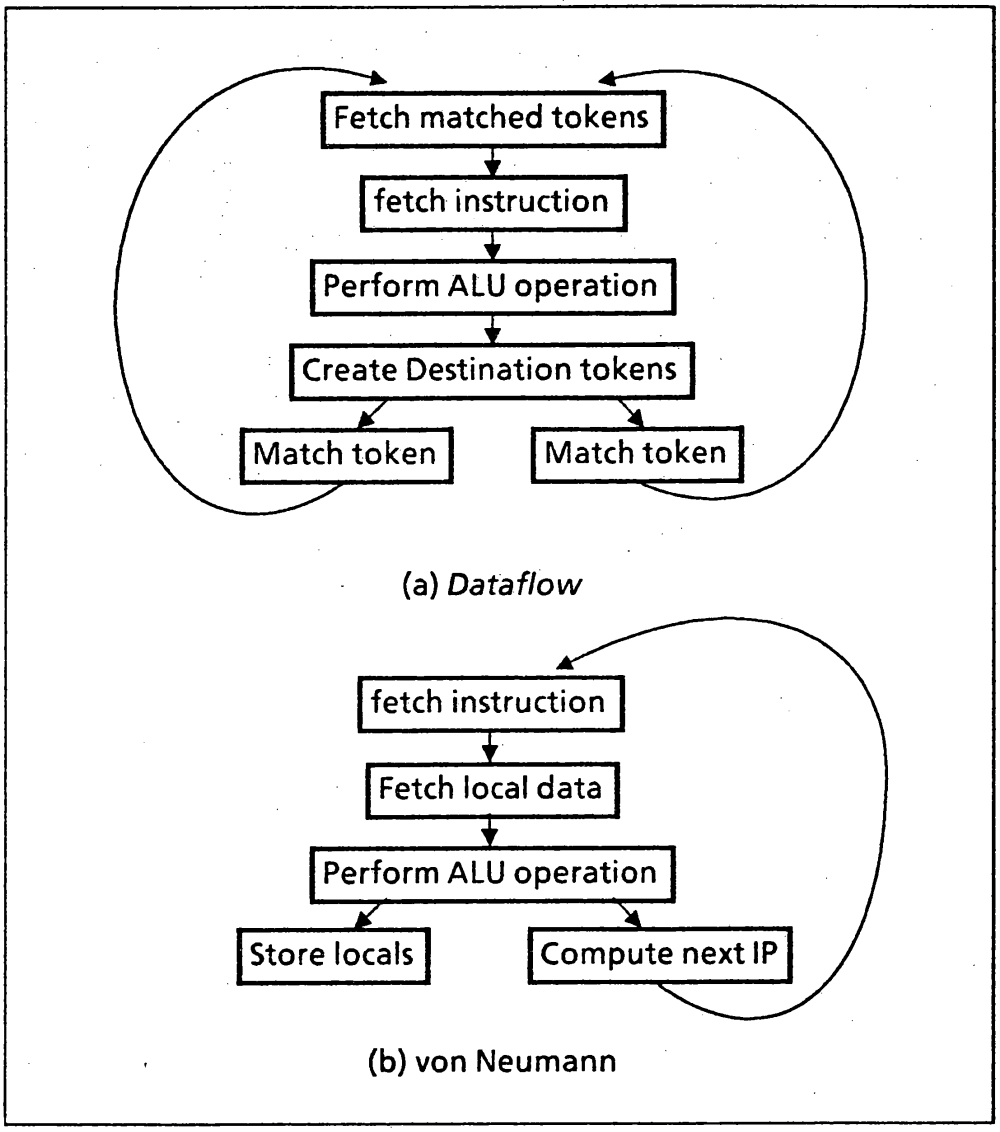


Figure 4.3: Instruction execution sequence

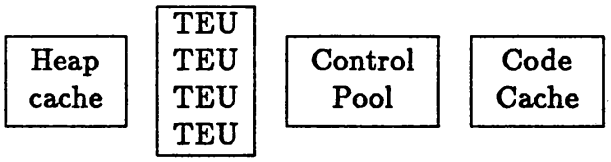


Figure 4.4: D-RISC with Separate control and data concurrency

(TEU) specifies a sequence of data changes in physical locations which calculate the thread's value.

The modification to allow control concurrency within one data frame is simple: separate instruction fetch from data fetch and have a pool of control pointers, each sequencing instructions and identified with a single TEU determining local data access. The resulting architecture is illustrated by Figure 4.4.

The number of control pointers on any data thread may be limited at compile-time in order to ensure that control-pool overflow never occurs. The size of the control pool is an important parameter of the design which must be minimised, subject to providing enough concurrency to hide system latencies, so the ability to queue surplus control concurrency for future execution is useful.

Pooled instruction pointers are available for asynchronous instruction execution, whereas queued pointers must be returned to the pool in a particular order. This can lead to deadlock if the pool fills up with join (see below) instructions which are waiting on queued instructions. One solution is to limit at compile-time the number of possibly simultaneous join instructions within a data thread to less than its share of the control pool. Optimal management of a control pool needs careful analysis of all possible compile-time and run-time measures to control concurrency.

Note that asynchronous fetch of instructions and data is intrinsic to efficient implementation of this type of design. It thus combines the separate advantages of dataflow and von Neumann designs and so is an appropriate schema within which to investigate optimal multiprocessor architecture.

4.3.5 Latency-bandwidth tradeoffs in memory design

Trading latency for bandwidth in memory design. The fundamental result of this chapter is that the CPU design balance between latency and bandwidth constraints on operand fetch is determined by latency-bandwidth product, or *storage* of the connection over which data is fetched. Multiprocessor interconnection is inherently of high storage, simple cache memories are not. However efficient memory access is theoretically a high storage operation because address decoding and data reading can be pipelined through a hierarchical memory structure to increase bandwidth—the storage of of such a memory is just the total length of its pipeline.

The detailed design of highly pipelined memories is dependent on particular

technology and beyond the scope of this thesis, however an approximate estimate of the benefits obtainable from pipelining can easily be made. Consider a 2^n word memory, pipelined with selection of (or data reading from) 2^m banks in each stage. This has $\frac{n}{m}$ stages each for address decode and reading and so potentially a speedup of $\frac{2n}{m}$ over the unpipelined memory.

Four factors may reduce this performance.

1. If m is too small the multistage decode circuitry will be slower than a circuit with fewer stages.

2. Clocked dynamic logic, necessary at each stage in the pipeline, will be slower than dynamic domino logic for the complete decode circuit.

3. The performance of the pipeline is limited by the maximum propagation delay through any stage, whereas that of the unpipelined memory is limited by the sum of delays through each stage. The former is thus likely to be less than the latter whenever technology introduces random inter-stage performance variations.

4. A highly pipelined circuit has more simultaneous signal transitions than the corresponding sequential circuit, as well as more global clock distribution, both these factors increase noise problems.

Suppose that these factors result in performance degradation by a factor of 2 for $m = 3$ (smaller m resulting in too great a pipeline overhead). The performance which may be expected of a cache memory of 2^n words is then:

Bandwidth:	$\frac{n}{3}$
Latency:	2

relative to that of an equivalent unpipelined memory.

The important caches in a D-RISC are a code cache (size 4K–16K words), and a frame cache (size 128–1024 words). The increased bandwidth from pipelining is therefore a factor of between 2 and 5. This must be set against worse single thread performance due to increased access latency.

One way of viewing a D-RISC uniprocessor with highly pipelined cache is to regard the shared use of cache memory by multiple threads as an extremely cheap sort of high bandwidth inter-processor communication, with the hardware of the processors themselves folded into a single CPU. This, when compared with a more conventional implementation, has higher system ALU bandwidth for given hardware size, but also worse single thread performance by a factor of very roughly 2. The latency limited concurrent performance is not necessarily so bad because the single-thread per processor implementation has bus latency which the D-RISC uniprocessor does not.

Trading bandwidth for latency. In contrast, RISC designs, for example the ARM [arm87] typically use precharging of register bank data lines to reduce memory latency at the expense of decreased bandwidth. Von Neumann machines have long pushed memory technology developments in this direction towards memories with a low latency-bandwidth product, because these are more effective in non-concurrent computation. Highly pipelined memories of the sort which would be useful in D-RISC design are not available as highly tuned off-the-shelf components. An investigation of the implementation of such designs in VLSI must therefore proceed with D-RISC architecture design.

4.4 What is a Uniprocessor?

In this chapter we have discussed uniprocessors which may run multiple concurrent threads of computation. Furthermore any operational block in a D-RISC may be either replicated or pipelined, increasing the corresponding bandwidth. There is thus no straightforward structural distinction that can be made between a uniprocessor and a multiprocessor.

For example a uniprocessor could be defined to be the hardware communicating through access to a single bank of shared cache. Since all D-RISC designs have some private memory for each executing thread the definition must not exclude this, so it will necessarily classify as uniprocessors machines with multiple CPUs with large private caches and memories communicating however through a single bank of shared memory. These hardware resources may be used differently by eliminating the global shared memory, and allowing shared access to some portion of each processor's physically local memory: this machine is undeniably a multiprocessor because multiple asynchronous memory accesses may be overlapped. Now consider a machine with a single shared memory which achieves high bandwidth through interleaving. This machine must accordingly be called a multiprocessor even if the interleaved memory is a component in a single D-RISC CPU.

The definition which I adopt is necessarily therefore somewhat arbitrary. A uniprocessor will be hardware which cannot sensibly be decomposed into smaller physically separate branches in a cache hierarchy of the sort posited in Section 3.2. In most of this work I use an idealised model in which a multiprocessor is a set of uniprocessors connected by some global message-passing bus. The topology of this bus is discrete: any node may communicate with equal ease with any other node. Furthermore the bus is characterised by a maximum global bandwidth and minimum message latency. Actual message latency increases with congestion only as necessary to limit the global bandwidth.

This thesis deals with CPU architecture design, therefore communication bus bandwidth is regarded as a specifiable design parameter, with a corresponding technology-dependent increase in latency for an increase in either number of connected nodes or bandwidth. Bus bandwidth itself is not seen to be a fundamental limitation to performance, although it may largely determine hardware cost. In any specific design a particular bus will be chosen which will then limit global

bandwidth. In some instances, where a certain bus is part of the design specification, this global bandwidth limitation will determine performance: this is a technology-dependent issue and not assumed here.

This model of inter-processor communications is extended in Section 5.3.6 below.

Shared and private memory. It should be clear from the preceding that the usual distinction between loosely and strongly coupled multiprocessors is one which is too coarse for this work. Inside a single uniprocessor memory may be shared or private to a TEU, as specified in the D-RISC design. This may therefore have some of the characteristics of a conventional strongly-coupled multiprocessor.

I assume a model in which inter-processor communication is physically via messages, since long bus latency, and, in a multiple thread system, arbitrary waits for threads to terminate, will rule out atomic two-way transactions. This corresponds to inter-processor coupling which is not directly modelled by global memory transactions: the usual definition of strong coupling). However it is a program's thread decomposition which determines when these messages coordinate shared access to local processor frame memory. Export of a thread reference always implies possible sharing, at least until the thread has finished: the exact implementation of this is discussed in Section 5.3.2.

4.5 Multiprocessor Design Taxonomy

How do these observations about processor architecture relate to historical classifications of multiprocessors? Gurd has recently suggested [Gur88] that the historical classification of multiprocessors is no longer helpful in view of recent developments, and proposed a new classification. I will first summarise his work and then relate these ideas to my work on low-level hardware organisation.

In [Gur88] Gurd argues that the comparison of multiprocessor architectures in a way that leads to understanding of performance requires a number of different levels of abstraction, and then proceeds to investigate the lowest (nearest to the hardware) such level. He observes that historically multiprocessor hardware has been characterised according to the tree in Figure 4.5(a). He then argues that a more helpful taxonomy would give primary importance to the nature of data transactions in the computation, and particularly to whether the memory used by a processor is integrated with the processor, or separated from the processor by a global communication network. More precisely, since all processing must have some local memory, he makes the distinction between machines whose work consists of mutating global memory, and those whose work consists of mutating only local memory, in cooperation with other processors.

He presents the new classification of Figure 4.5(b), with integrated/separated memory as a primary key, and MIMD/SIMD as a secondary one, and argues that this more accurately reflects behavioural differences in machines than the historical classification.

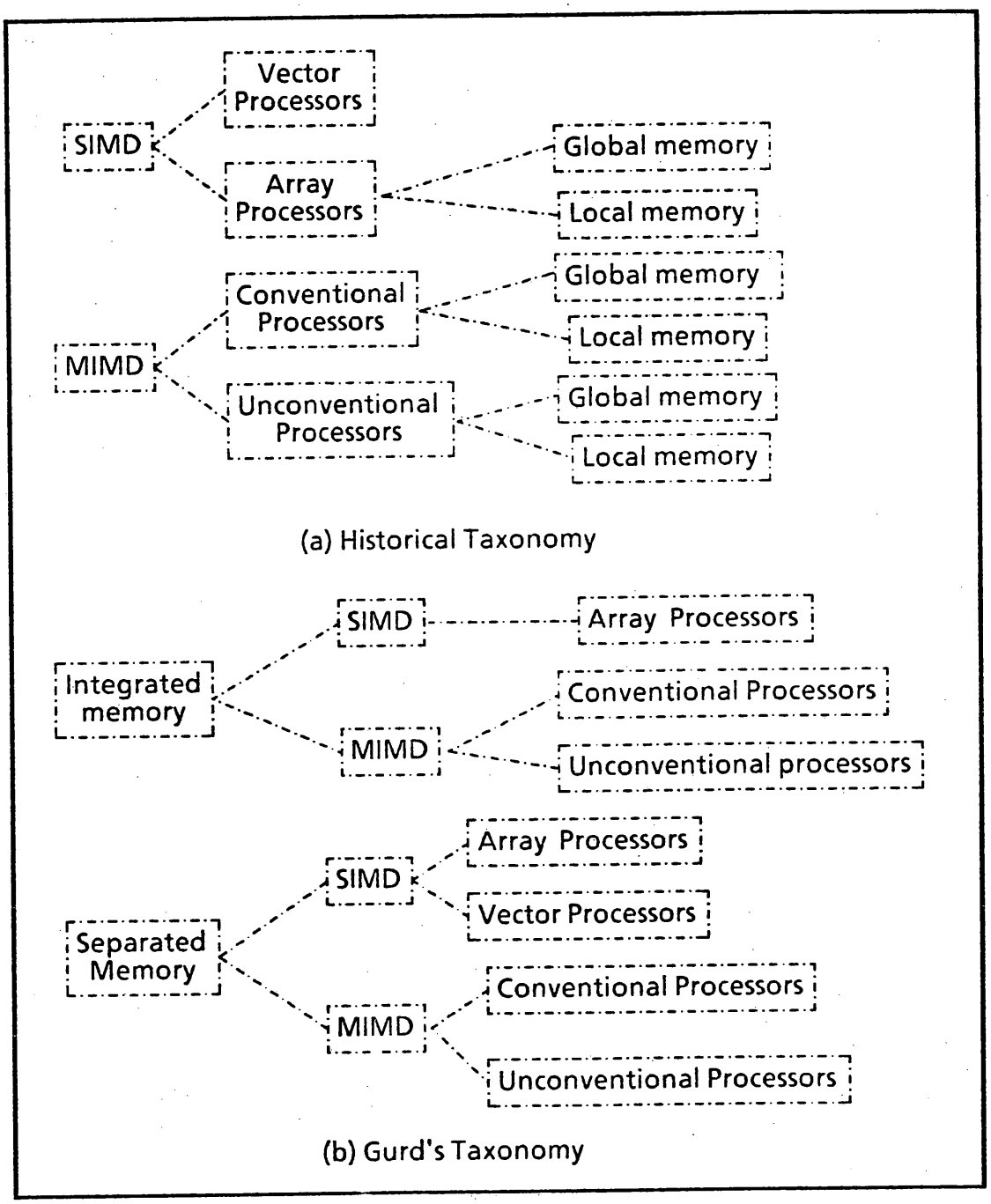


Figure 4.5: Multiprocessor Taxonomies

In this dissertation we have considered a number of fundamental characteristics of hardware, and tried to relate these to the different ways of executing a computation's Dataflow Execution Graph. Because this analysis started with no prejudice in favour of single-threaded computation we have been able to discuss processor design in a way which makes no fundamental distinction between intra-processor and inter-processor concurrency. As argued in Section 4.4, the decomposition of a concurrent processor design into single or multiple processors is not always meaningful.

The low-level characteristics of hardware which we have considered are:

- Latencies and bandwidths of connections, and in particular the *storage*, or latency-bandwidth product, of a connection.
- Synchronous and asynchronous operation.
- Cacheing, and naming strategies for data used in a calculation: either static, determined by compile-time analysis of a program's possible DEGs, or dynamic. Static names can refer directly to registers, dynamically named data must be localised in a associative cache.

These two last items can be related by considering the relationship between *synchronism* (discussed in Section 3.1.1) and **coherence**, which I will define as here.

In synchronous systems events at different *times* are related in an organised way. For example in a RISC pipeline different phases of one instruction are normally executed on successive clock ticks. This organisation is a type of a temporal granularity: blocks of operations occurring at different times are determined together.

There is a corresponding concept of physical granularity, in which the data in different registers at a given time has some coherent relationship. I will call this *data coherence*. The use of static names to address a register bank involves just this type of coherence, as does the arrangement of a dynamic cache memory into lines of contiguous data. Data coherence also imposes some granularity, and requires some *a priori* organisation.

The task of a processor design is to make whatever use possible of concurrency, synchronism, and coherence. We have seen that concurrency is needed in a design whenever the storage of a connection is greater than 1 to balance latency and bandwidth constraints on performance. A computation's DEG will provide concurrency which is partly susceptible to static analysis (the simultaneous creation of a new subgraph of predetermined shape in the part of a DEG with C-arcs from a conditional) and partly dynamic. The granularity imposed by synchronism and coherence requires some static execution information, and hence these desirable characteristics are limited by the nature of the executing algorithm.

Conventional RISC von Neumann CPUs use both synchronism (pipelining successive instructions) and coherence (storing data in a register bank with static register ids). Vector processors obtain high concurrency by using extra synchronism, with highly pipelined ALUs, and are consequently useful in programs with very

regular control structures. Array (SIMD) processors in contrast use data coherence to obtain high concurrency by performing an identical operation on a large number of data items.

D-RISC CPUs preserve some data coherence but use minimal control synchronism. The advantage of this is a chance to use more concurrency, even in irregular symbolic computation, than is available to von Neumann processors. Dataflow CPUs are even more radical and abandon both synchronism and coherence in the pursuit of fine-grain resource use.

These issues of concurrency matched to latency-bandwidth product, synchronism and coherence can be observed in multiprocessor architectures. They distinguish between dataflow, von Neumann, array and vector processors very well, in a way which subsumes the SIMD, MIMD distinction. How then is Gurd's integrated/separated memory dichotomy to be related to this?

The answer lies in the nature of local cacheing on each processor, and the type of computation which is executed. We have been considering symbolic computation where data-structures and parallelism are highly dynamic. Therefore concurrent execution requires substantial global access to data. This can always be implemented within a local cache hierarchy so that repeated access to global data is a local operation.

The optimum level of local cacheing is a critical design issue, and I think this is why Gurd's distinction has some merit. Integrated memory (in a symbolic computation) may be seen as local cache in contrast with separated memory. Some styles of computation (for example OCCAM, [Hoa]) limit communication between concurrent processes to predefined serial data channels and map processes statically onto processors. This computation may proceed using integrated memory without any model of global data cacheing.

The argument for separated memory in multiprocessors (for example Darlington and Reeve et al in [MDF*87]) is that the overhead of context-switching makes local cacheing impossible in a multiple-thread concurrent computation. I have shown that the cacheing of a small number of local contexts near to an ALU can optimise the use of a memory connection with latency-bandwidth product greater than 1. However there are implementation problems in the control of concurrent execution so that intermediate data does not swamp local caches: these will be discussed in more detail in Section 5.1.4 of the next chapter.

The complexity of the implementation problems to be overcome in cacheing multiple-thread symbolic computation perhaps explains why strategies of giving up altogether, either with no local cacheing as in ALICE, or no attempt to use synchronism or coherence, as in dataflow machines, have in the past seemed attractive.

In contrast with Gurd's work, the study of multiprocessor architecture which I have presented does not lead to any categorical classification, and especially not to a taxonomy. My concern is to compare and relate different design strategies in the hope of identifying (possibly superior) intermediate designs. In particular I have shown that the von Neumann / dataflow design dichotomy is not intrinsic, and hence that a range of intermediate designs should be considered. The multiproces-

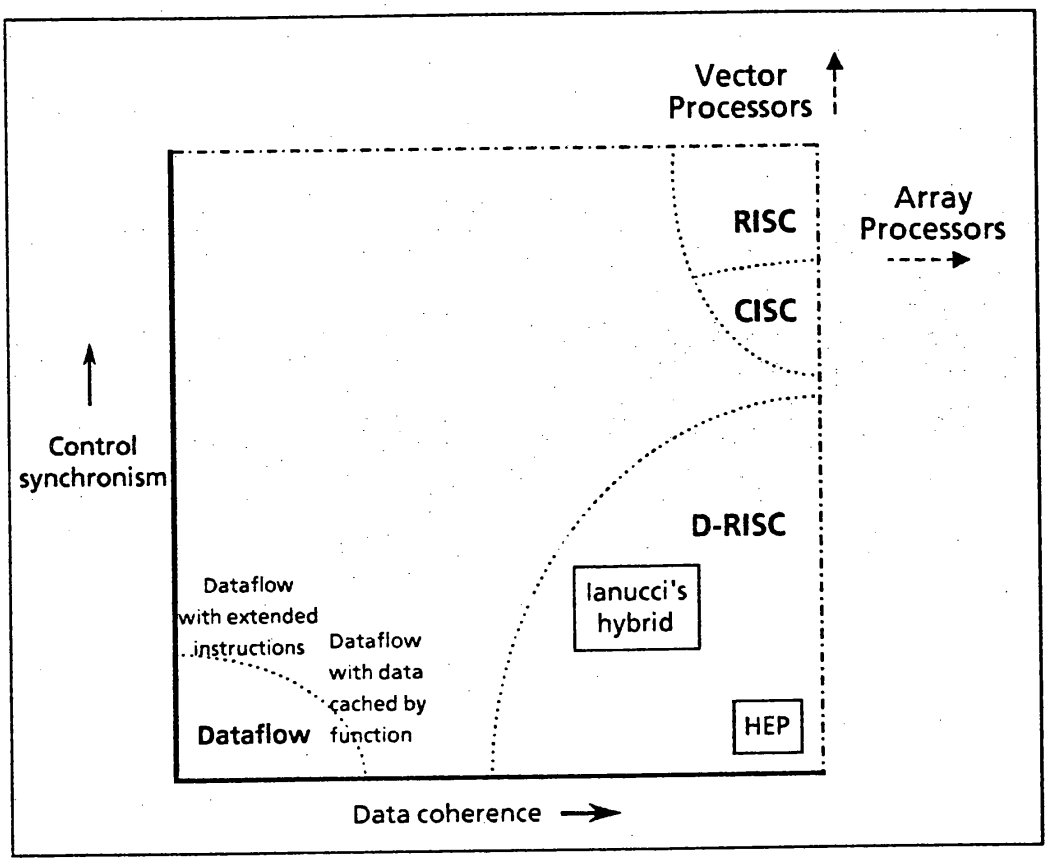


Figure 4.6: CPU Design Space

sor design space is viewed by me as parametrised by the factors discussed above: synchronism, coherence, cacheing, and then particularised by bandwidths and latencies of connections between concurrent components, with the corresponding latency-bandwidth products indicating the need for overlapped operations.

The design space of dataflow machines, von Neumann processors, and D-RISCs may be broadly classified according to these ideas by considering overall use of synchronism and coherence. This results in a picture with von Neumann and dataflow CPUs at opposite corners of a triangle of possible architectures for symbolic computation as in Figure 4.6. This picture suppresses much important detail. For example The HEP has a long pipeline with successive instructions from different (arbitrarily chosen) threads. It thus is synchronous at a coarser grain than a RISC with a shorter pipeline, however the dependence between successive instructions in the pipeline is less. Should this be classified as more or less synchronous than a RISC? A more careful analysis is required to distinguish these designs properly.

4.5.1 Scalability

In [AI86] Arvind and Ianucci suggest that von Neumann CPUs do not form a basis good for scalable multiprocessors because they cannot optimise for fast thread switching without reducing single thread performance. This is partially true, and

but argument in this chapter shows that it is possible to compromise between single and multiple thread performance. A D-RISC cache performs reasonably well in single thread computation, the difference between it and a RISC stack cache is the added cost of associative cacheing of each stack frame and the waste of space that mapping stack frames into blocks entails.

They make a further assertion that for von Neumann based multiprocessor systems there is an architecturally defined limit beyond which adding processors can never increase system performance. This section shows clearly that there is no fundamental difference between dataflow and von Neumann execution. The key question is whether it is worth trying to achieve some locality in a multiprocessor CPU. If this is true then a DRISC architecture is an efficient way of obtaining it.

General arguments about scalability are interesting. For a given program possible concurrency will be limited, furthermore static decomposition of the program that provides less concurrency may be more efficient. Thus particular programs impose a limit on useful concurrency, and this is often low. Now suppose that program size is scaled with the number of CPUs in the architecture, so that system loading remains constant and high. The cost of increasing the number of processors on system performance is very roughly a logarithmic increase in interprocessor latency. With faster technology this cost might increase faster, since the average distance between n objects of a given size must increase as $n^{\frac{1}{3}}$. A safe upper bound on the rate of this increase would therefore be $\log nn^{\frac{1}{3}}$.

Whatever this increase in latency its effect on system performance is to make necessary a larger number of concurrent thread to hide it. The number of threads is proportional to the latency, but the argument above shows that increasing the number of threads decreases bandwidth performance by roughly the square root of the increase. Thus CPU performance decreases by a factor of $\sqrt{\log n}$, furthermore the necessary program concurrency for optimal execution scales as $n\sqrt{\log n}$. Systems thus require a modest increase in loading per processor for optimal operation as they scale.

4.6 Summary and Related Work

By following an analysis of fundamental constraints which exist in any hardware design this chapter has shown that dataflow and von Neumann architectures form two extremes in a design space (the opposite corners of Figure 4.6) and suffer from complementary design problems. The intermediate space has not been investigated with many concrete designs. Only one such design exists, the Denelcor HEP ([Smi78]), and its limitations—very bad single thread performance and a hardware-limited number of threads—have not encouraged other architects to follow this path. However this chapter shows that the very large design space which remains to be explored will contain the balance between fundamental design constraints which is lacking in both von Neumann and dataflow designs.

Section 4.5 is central to this thesis, relating the ideas of low-level hardware design presented in Chapter 3 to multiprocessor design, and establishing connections

with Gurd's research work.

The next chapter will consider in more detail the implementation problems associated with running symbolic computation on any multiple-thread computer, and particularly on the new type of architecture. The complexity of these problems perhaps explains why progress towards more parallel architectures has been in the past so slow.

It will be seen that von Neumann-like data locality can be maintained in spite of the difficulties of implementing concurrent symbolic computation. What then can be concluded about D-RISC architectures? For concurrency from multiple independent threads of sequential computation D-RISC is becoming a serious contender as RISCs become faster and the problems in maintaining full RISC pipelines therefore greater. This application is independent of the results in the next chapter, and a D-RISC for such an application need not have the sophisticated fine-grain thread communication and synchronisation hardware otherwise necessary. However this design trades increased bandwidth with multiple threads for inferior single-thread bandwidth. Since multiple independent threads can be run on separate processors the possible D-RISC advantage here is one of performance for a given cost, not absolute performance.

For general purpose concurrent symbolic computation D-RISC has a clear advantage in its ability to cope with synchronisation latency, the case for large-scale concrete design investigation is clear. The design space to be investigated before the best designs for current technology can be established is very large, because the availability of fine-grain inter-thread concurrency allows a plethora of different ways of mixing asynchronous and synchronous execution.

In parallel work for his recent thesis ([Ian88]) Ianucci has proposed a concrete D-RISC design, starting from dataflow architecture experience. This work is very interesting and promises to be the first of many such concrete investigations into D-RISC design space [2].

[2] I found Ianucci's work at a very late stage in the revision of this thesis, and a thorough assessment of his design must therefore be a future development. It is extremely encouraging to find that work starting from a completely different practical design experience should lead to results which appear to be so similar to my own.

Chapter 5

Implementation Issues

The theory outlined so far examines some of the fundamental compromises inherent in any implementation of concurrent computation. This chapter builds on this work, investigating more specific implementation techniques. The next section considers the implementation of concurrent computation on a uniprocessor: this is a necessary first step to understanding the more complicated multiprocessor implementation problem.

The Section 5.3 describes a simple model of multiprocessor hardware which captures the important features of multiprocessor execution while abstracting from details of inter-processor communication design. The object of this thesis is to understand the relationship between concurrent computation and processor efficiency. Communication performance, like other details of low-level design, will be vital in the construction of any real design: however its consideration in detail here would be orthogonal to the aim of this investigation.

Section 5.4 describes specific issues that arise from attempts to optimise latency-limited computation (e.g. multiprocessor computation where there are always spare processors and computation time is determined by a critical path length).

5.1 Concurrent Implementation On Uniprocessors

5.1.1 Thread implementation techniques

Threads and vectors

In sequential computation execution is determined by a stack of frames (vectors of physical storage locations) and an instruction pointer indexing a sequential instruction stream. Instructions modify locations in the top frame, push a new frame, usually saving a return address, or pop a frame writing some result to the previous frame. Note that I suppose that all data access is local or heap, as for example will happen in a functional language implementation using supercombinators. The type of global communication provided by sequencers can be managed by passing a common heap data reference to all threads which call the sequencer.

In concurrent computation a thread may therefore be specified by a vector, together with an instruction pointer which specifies a set of modifications to locations in the vector. Each thread also has a reference which may be passed as a data object to other threads and read, returning a value. (The implementation of thread reference is discussed in the next section). The creation of new threads is analogous to the creation of new stack frames; certainly any stack based computation can be equivalently expressed using threads.

However where a thread creates just one child thread and then reads its value the full overhead of thread referencing is unnecessary, and the execution is identical to the sequential case. In what follows I will therefore assume that a thread corresponds to a stack (or linked list) of vectors in a heap, together with an instruction pointer. Of course when not executing on a TEU the instruction

pointer of a thread will itself be stored in some vector, probably the top vector in the stack. A thread may be identified for wakeup by a pointer to this top vector. In contrast on creation a thread is identified by its base vector which contains the initial specification for the thread's computation and will be used by thread references.

Thus the primitive objects in a concurrent system are thread bases, private vectors stacked on top of bases, and other non-modifiable vectors constructed during execution and garbage collected. Private vectors may easily be locally garbage collected. Finally a thread base must have available certain atomic read-write operations to interlock computation: a flag indicating that it is executing and other operations to interlock thread reading with thread finishing.

This initial description of threads shows the store and process management which must be supported by any hardware using threads, and which will be used as a base for subsequent analysis. Allocation of storage in vectors from a garbage collected heap is a prerequisite, although the sizes of allocated vectors may be limited by a particular implementation.

The hardware discussed in the previous section uses asynchronous concurrency to hide certain latencies by having available a small number of threads in TEUs for concurrent execution. The overheads of switching threads attached to a TEU must be relatively small, since threads in symbolic computation are typically short.

The latency associated with thread switching in a D-RISC is hidden by the execution of other threads. Together with other hidable CPU latencies its cost is seen only as an increase in the number of threads necessary to use all available CPU bandwidth. For optimal design this entails an increase in the number of TEUs, and perhaps a corresponding decrease in CPU bandwidth, but this is a much smaller loss of efficiency than that attributable to the original switch latency in a conventional design.

However, even in a D-RISC, thread-switching entails bandwidth overheads in the construction and scheduling of ready threads. Other overheads result from the reading of thread references, and queuing of waiting thread readers until the required data is available.

This section considers these overheads, and how they can be minimised. Throughout the following discussion it will be useful to compare threads in a concurrent implementation with function calls in a sequential implementation.

Ideally the overhead of thread management should be no greater than that of executing the corresponding function calls sequentially. It is instructive to see why this is not so. Created thread references must be returned immediately and may then, as first class data objects, be moved to different locations, copied, etc. At any time the reading of any instance of a thread reference must result in either the corresponding value, or a synchronisation wait.

Threads thus have very similar properties to consed storage, requiring a globally unique name which must be garbage collected some time after thread termination. A thread name can always be reclaimed after termination, but this is in general impossible without global writing to all existing thread references. Compare this with the corresponding reclamation of a stack frame on function

exit, where the function's value must be copied to only one place whose location is known to the terminating function.

The previous chapter discusses hardware on which single-thread data and control localisation can be used in the implementation of concurrent computation. Computation with threads is most like (applicative order) sequential computation when threads are executed in an order so that threads will have finished whenever they are read. The relationship between thread scheduling order and performance is central to an understanding of the implementation of concurrent computation: it will be analysed in detail in Section 5.1.4 below. I anticipate the results of this and suppose that it desirable to arrange that synchronisation waits are relatively rare.

For a thread's value to be communicated to its references two different strategies may be adopted: **demand** and **supply transfer**. In demand transfer each reference is an indirection node to a location which will contain the thread's value when it finishes. Reading a reference either performs this indirection or results in a synchronisation wait until the thread has finished. Waiting threads are queued in a list and awakened when the thread finishes. In this system threads do not know where their references are.

In supply transfer a thread knows where its references are. Each thread keeps a list of all locations containing references to it, these are updated with the thread's value when it finishes. When a thread waits on a reference its instruction pointer replaces the reference in the thread's local data: this is found by the referenced thread when it finishes and used to supply a wake-up signal.

Supply transfer makes copying a reference an expensive operation, but has the great advantage that a thread's storage is released immediately when it finishes. It optimises sequential applicative order execution of code, where it corresponds exactly to the creation of a new stack frame and subsequent return of a result. However in a concurrent system it may be desirable to reclaim vectors containing references (which will never be read) before the corresponding thread finishes. This can't be done easily with supply transfer. It thus seems reasonable to use supply transfer only where it is known at compile-time that the references will be read.

Supply and demand transfer can be combined particularly simply: the first reference to a thread (whose location is known at thread creation time) can be supplied. Any copying of the reference will result in possible demand transfer of the thread's value to the copied references. A flag in the thread (more generally a reference count) indicates whether demanding copies of the reference exist and must be set or incremented when the reference is copied.

Supply transfer has one particular advantage over demand transfer: it facilitates implementation of tail recursion. Consider the case of a function f_c which constructs a list which is read concurrently by another function f_r. In a normal thread implementation this concurrency cannot be realised without decomposing f_c into a large number of separate threads. This is clearly wasteful of resources if f_r is the sole reader of these threads: the communication required is that of a single FIFO. Supply transfer provides this structure: the same thread can supply data

for each element in the constructed list sequentially. This issue will be discussed at further length in Section 5.3, where it will be shown to be useful, together with some form of compact list representation, in the efficient implementation of pipelines.

In general optimised reclamation of thread storage is a problem related to local garbage collection of heap, this is considered in the next section.

It can be seen from this section that efficient management of threads can result in complicated low-level primitives. Such complication has two distinct disadvantages:

- Unless it is contained by an appropriate and rigidly observed model of storage use it will be practically impossible to write correct low-level storage allocation and reclamation routines.
- The non-determinacy in thread finishing order necessary to maximise available concurrency results in conditional control-flow whenever threads are read.

In a general analysis of hardware design these two problems are of different importance. A complicated storage use strategy may necessitate rigorous verification of all system code, and compiler code generators. This is not a trivial task, but is certainly possible and does not alter performance.

In contrast conditional control-flow always reduces performance. However in a D-RISC executing bandwidth limited computation the latency associated with non-deterministic instruction fetch can always be hidden along with other latencies, and is therefore not as costly as in single-thread computation. Furthermore even without this hiding *exceptional* control flow may be implemented with little overhead to normal execution. In general it will be possible to arrange that bandwidth limited computation executes so that departures from applicative order termination of threads are rare: these may then become exceptional control flows. Nevertheless concurrent execution requires higher control instruction bandwidths than the corresponding deterministic evaluation order execution.

5.1.2 Local GC to speed up CPUs

This section explores various local garbage collection techniques, and their use in increasing CPU efficiency. The ideas behind this were outlined in Chapter 3.2—the reuse of physical locations on a stack must be replaced in concurrent computation by methods which can reclaim storage allocated in a tree, or even, more generally, a heap.

Types of Data

In this section I use the term **thread frame data** to refer to data statically allocated to locations within a vector on entry to a thread. This contrasts with **heap data** (e.g. consed store) which is dynamically allocated during thread execution

from a global heap. In a concurrent system frame vectors may themselves be allocated from a heap, but will not therefore be called heap storage.

How do the requirements of thread frame data collection differ from general heap garbage collection? The local locations used when executing a thread may be released immediately it finishes: if supply transfer is used then the local locations used to manage thread reference may also be released. In a similar manner, if all heap returned in the thread's value is copied to new locations then the heap locations used by the thread may be reclaimed on thread termination.

The difference between heap and thread frame locations is just this: it is usually the case that copying of heap is considered too great an expense to perform on function exit so heap locations are allocated from global storage and must be collected in some other way. Reference counts are such a method. It is interesting to compare reference count gc with function exit copying gc: one has the cost of copying live data on every function exit, the other the cost of scanning (and collecting) all dead data. Copying on certain function exits which are known to use much more heap than they return is thus cheaper than reference counting.

In a sequential system copying can lead to efficient local gc in which all store is allocated on a single stack, but it does not seem a good candidate for a parallel system where there is no deterministic total ordering of thread termination times.

However thread frame store accounts for most of the storage allocated during execution (in my simulations of compiled LISP I found that typically a frame is 6 locations and uses less than 1 cell of heap). Even when decomposed into threads, with intra-thread storage allocation on a stack, allocation thread frame storage predominates over heap. Whatever the merits of local garbage collection, collection of thread frame alone must thus be seriously considered since it costs much less than full local garbage collection and accounts for most of the storage allocated.

To collect or not to collect. Consider first a cache which is used only to hold newly constructed data. Figure 5.1 gives a diagram of such a cache wrapped around an ALU (which generates the new data) and identifies important bandwidths.

In this cache hit rate is critically dependent on data lifetime, which may be expressed by a function $f_l(T)$ defined to be the fraction of accesses which are of data which is older than T. Suppose that data is replaced on a least recently created basis. Then the cache will miss whenever an access occurs to a datum older than the cache persistence time, T_p, which is defined recursively from f_l

$$T_p = \frac{N}{f_{flush}} = \frac{N}{f_{cons} f_l(T_p)}.$$

If some fraction of data μ are not collected then $f_l > \mu$ so certainly $T_p < N/(f_{cons}\mu)$. This severely limits cache effectiveness in small caches of data with typically short lifetimes.

Is it reasonable to be concerned about cacheing of just newly consed data? In concurrent symbolic computation stack frames are consed rather than allocated

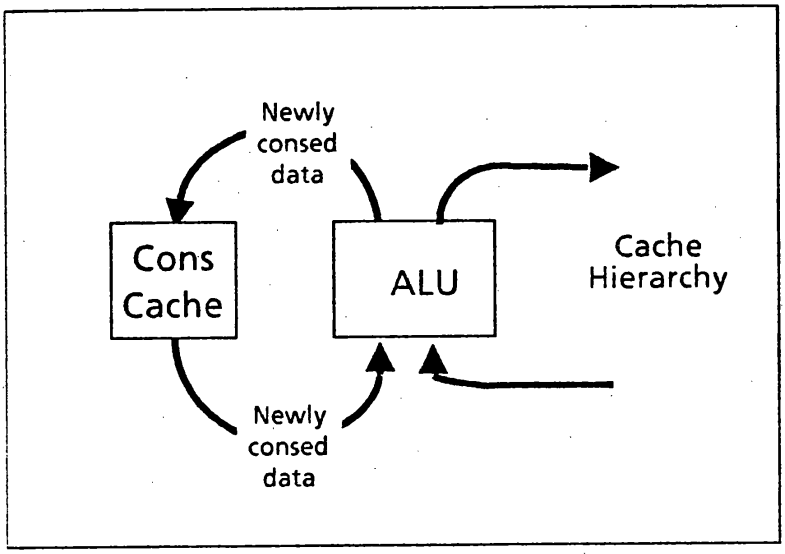


Figure 5.1: A cons cache

from a stack, and are mostly short-lived. Furthermore, as observed above, more than 6 out of 8 locations (75%) of storage allocated is for these frames and will therefore hit in a cons cache.

5.1.3 Data lifetimes in concurrent computation

The last section showed how data lifetime modulates cache effectiveness in a D-RISC CPU. This section examines the relationship between thread frame lifetime and scheduling.

Consider first the properties of conventional, sequential, thread execution. Here threads are identical to function calls, any thread creation results in an immediate call to the newly created thread, and thread frames may be allocated on a single stack and reclaimed immediately on thread return. This type of execution may be represented by a depth-first traverse of the TCT, see Figure 5.2(a).

The use of thread frames in this computation is predictable. At any time frame data for threads on the path from the TCT root to the currently executing thread is alive. The depth first traverse corresponds to data use which can be efficiently localised by a stack. Call this type of thread execution a **VN-thread**, by analogy with the use of this term on page 12.

In concurrent computation this picture is changed in two ways. Firstly there may be more than one thread executing—this could be represented by a number of simultaneous VN-threads, as in Figure 5.2(b). Secondly, concurrent execution may result in a thread which is part of a VN-thread waiting on the value of some other unfinished thread. This may be represented by a discontinuity in the TCT traverse as in Figure 5.2(c).

Call multiple VN-thread execution **VN-concurrency**, and waiting on a thread which is not local in the TCT **discontinuity**. VN-concurrency and discontinuity are particularly harmful when combined. During VN-concurrency without discon-

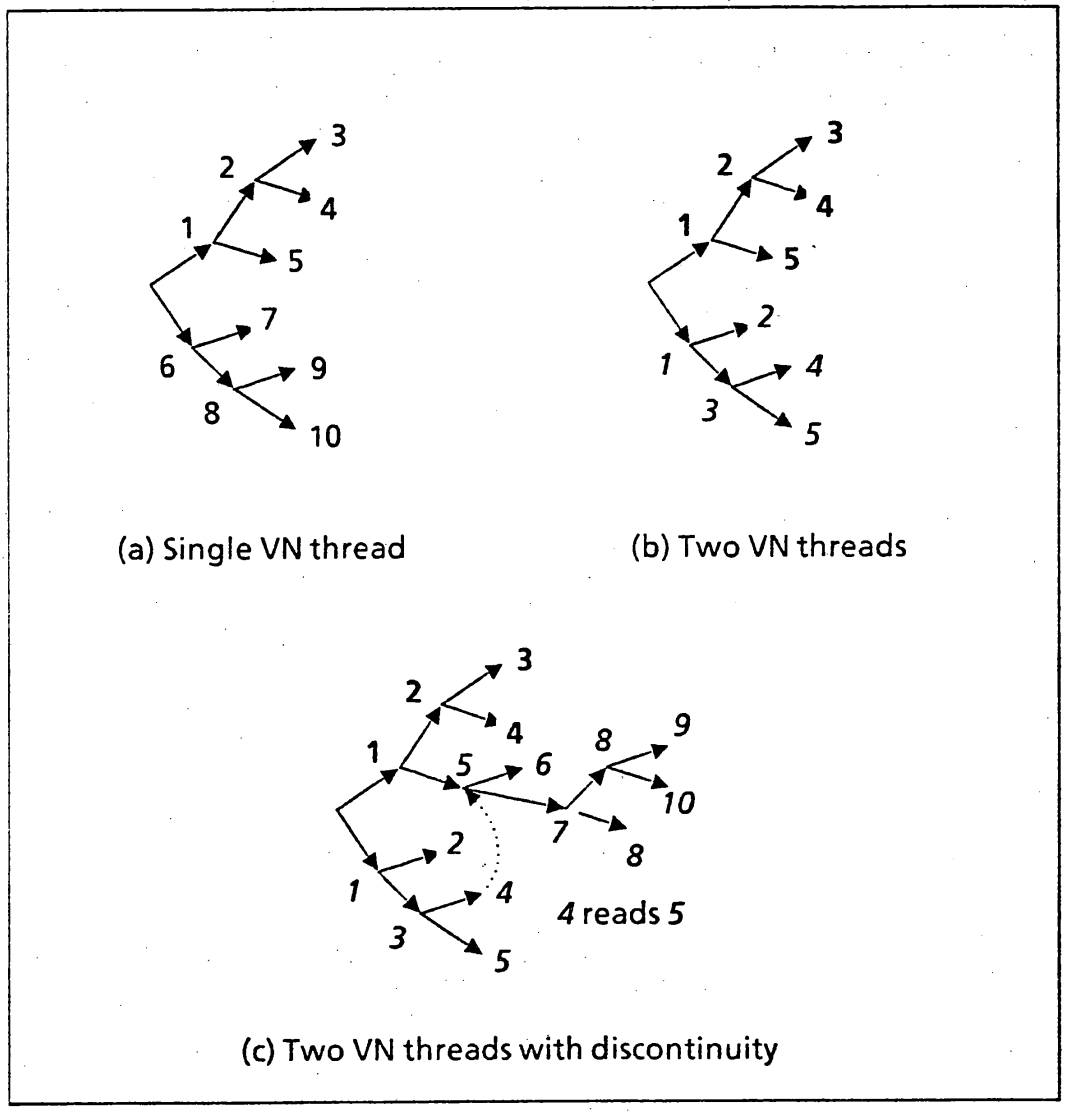


Figure 5.2: Different TCT traverses

tinuity the computation may proceed conventionally using the appropriate number of separate stacks. Equally, if execution is not VN-concurrent, discontinuity may be accommodated on a single stack by suspending the current thread and pushing the new thread onto the stack. This is exactly what happens in efficient lazy implementations of functional languages, for example TIM (see [FW87]). However multiple stacks in conjunction with thread suspension leads to **blocking**: a suspended thread may be awakened (by a thread on another stack) but unable to execute because it is trapped beneath some other unfinished thread.

This shows why concurrent computation needs a model of storage allocation which is more general than a set of stacks.

5.1.4 Scheduling and frame cache lifetimes

Different types of thread execution result in different frame cache lifetimes. The relationship between scheduling strategy, thread execution, and frame lifetime is investigated below.

VN-concurrency. Suppose that n VN-threads are independent of each other and executing in a D-RISC cache. This requires a storage of n times that needed for the same hit rate with a single VN-thread. However if the VN-threads are highly dependent, i.e. they join up and share a common root, the storage required will be little greater than that used by a single thread. Figure 5.3 illustrates this situation.

Thus demand for use of frame storage in concurrent computation will relate to scheduling. The best scheduling strategy is one in which threads close together on the TCT are executed simultaneously. This could be achieved optimally by total ordering TCT nodes according to a breadth-first traverse and scheduling the least, according to this order, of the available ready threads whenever a thread finishes.

Discontinuity. The addition of discontinuity to VN-concurrent execution of threads in general increases the use of store by creating a supplementary path of frames in the cache leading to each waiting thread at a discontinuity. The data in these frames must be held in the cache but is separate from active thread execution until the waiting thread restarts. Bad scheduling strategies can lead to a large number of discontinuities, with corresponding loss of cache efficiency.

When is discontinuity likely to occur? Consider a single node in the TCT: this represents a parent thread with arcs from the node to each sub-thread created by the parent. Discontinuity may occur when a sub-thread is passed a reference to another sub-thread. The referencing thread may become discontinuous if it reads its reference before the referenced thread has terminated. Since any reference passed to a thread may be handed on to its sub-threads, and a thread's value may itself contain references to any part of the subtree of the TCT whose root is the thread, discontinuity may occur unless entire subtrees of the TCT are strictly sequenced.

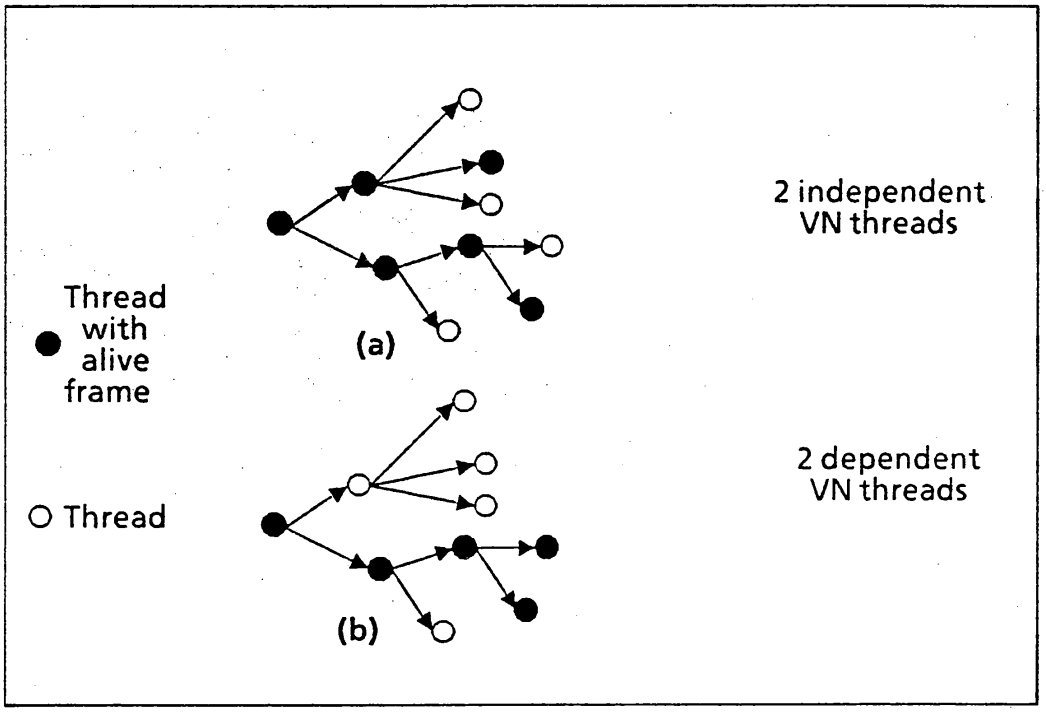


Figure 5.3: Cache use in VN-concurrency

Discontinuity must in general therefore be predicted from the program's DEG. Any data arc which crosses the TCT corresponds to a possible discontinuity unless its source thread is scheduled before its destination thread. Such a data dependence will be called a **pipe**, good scheduling must ensure that pipes do not lead to excessive discontinuity by scheduling sources before destinations.

The structure of symbolic computation is such that concurrent execution with pipes can often be expressed as communication between TCT subtrees through lists, the source subtree generates (from the head) a list which is passed to the destination subtree and read. The two subtrees may be seen as separate processes connected by a pipeline. A trivial example of this may be found in the concurrent pipelining of list processing functions, although here the maximum concurrency is limited statically to the number of functions pipelined, and will typically be less than this. More typically pipelines are composed with divide and conquer recursion so that a dynamic network of functions communicating through pipes results. One example of this is described in Section 5.5.3 below.

Scheduling strategies. On a uniprocessor scheduling may only approximate to an ideal strategy, and still result in good performance. The simplest strategies treat all newly created threads as separate entities in a stack or FIFO from which they are withdrawn for execution on demand. Waiting threads, when woken up, are similarly queued but given priority over any new thread.

LIFO scheduling corresponds exactly to depth-first traversal of the TCT, by a number of VN-threads, it is thus ideal for VN-concurrent computation. A problem

arises when discontinuity is introduced by pipes. If a and b are two threads with a discontinuously reading b then they must have some common ancestor p which created an ancestor of b before an ancestor of a. But of these two threads the one leading to b should be scheduled before the one leading to a, if possible, in order to remove this potential discontinuity. Thus sibling threads should be scheduled in FIFO order (at least when they contain references to each other). This modified scheduling ensures that in the absence of any demand for new threads execution will be both depth-first and continuous.

It is now possible to consider the implementation of scheduling strategies. The basic idea is create new threads dynamically only as required, holding specifications for new thread creation as a set of continuations, one for each parent with children still to be created.

Consider first a single VN-thread, scheduled as in the previous paragraph, call the (total) order in which this executes threads $>_{VN}$.

Thus $a >_{VN} b$ if any of

1. a is a descendant of b
2. a, b are siblings and $a > b$ in some sibling ordering as in the previous paragraph.
3. $a > b$ in the transitive closure of $>_{VN}$.

Optimal scheduling would ensure that the next thread to be executed is that which is earliest in this order total order and also created. Note that this may not be the earliest thread in the order, since a thread may not be created until its parent has been created. This scheduling results in execution which is depth-first and in which executing threads are as close as possible to each other in the TCT, so maximising cache sharing between the trails of concurrently executing threads.

Unfortunately it is not possible to implement this strategy efficiently: keeping track of the $>_{VN}$ order of every continuation is difficult since new continuations may be created at any point in this order. The best feasible strategy is very simple and consists of LIFO scheduling all continuations. Note that a continuation when executed will (if it contains more than one sibling) soon enter the first sibling, stacking a new continuation which will create subsequent siblings. The FIFO execution of siblings required to minimise discontinuity is ensured by static sequencing within continuations.

This discussion of optimal scheduling may be related to empirical work on the Manchester dataflow machine (see [RS87]) to control overall concurrency in a concurrently executing program. This motivation for selecting particular scheduling orders—to minimise total token storage—is different from the motivation considered here, minimisation of intermediate result persistency in caches. The empirical results presented in [RS87] lead to conclusions similar to those here which are however expressed in very different language. It is interesting that two apparently separate implementation issues should require similar implementation techniques. The similarity is that both require minimisation of intermediate result size, in one

case globally to satisfy global storage limitations, in the other locally to increase CPU cache hit rate.

5.2 Concurrency and Performance

The theory presented so far makes no conclusions about the performance of multiprocessor systems. It has not yet been shown that they are ever of higher performance than uniprocessors. The information that can be determined from this relates to the *nature* of performance in a multiprocessor system. Total ALU bandwidth

$$f_{SYS} = \sum_{\text{all ALUs}} f_{ALU}$$

is a function of n, the number of available threads. A naive model of multiprocessor performance would predict that for some f_1

$$f_{SYS} = f_1 \min(n, N)$$

where N is the number of ALUs in the multiprocessor. From this model one sees that performance is characterised by two parameters, f_1, the single thread performance, and Nf_1, the bandwidth limited performance.

The weight given to these two parameters must be determined for an individual application, without this information the question 'what is the best design for a multiprocessor' cannot be answered. One reason for stressing this pedantic point is that the programs that I consider typical of symbolic computation do not have very high concurrency and on a large multiprocessor are limited by periods of execution in which concurrency is extremely low.

This model of multiprocessor performance is however not accurate. The two optimisations discussed in Chapter 3.3 use high concurrency on a single ALU to increase its f_{ALU}. The way in which this changes the performance for a given program concurrency is complicated. Certainly either optimisation will require $n > N$ for maximum f_{SYS}. Bandwidth optimisation is likely not to increase correspondingly the program concurrency required for a given low f_{SYS}, latency hiding will. However if the latency to be hidden by concurrency is predominately inter-ALU bus latency then performance for a single thread of computation will be relatively higher.

A convenient way of describing system concurrency is with **loading**, L, where

$$L = \frac{\text{no. of ready}^1 \text{ threads}}{\text{no. of processors}}.$$

Execution may be classified by loading into three domains:

Latency limited. $L \ll 0.5$.

[1] A thread is ready when it may, but will not necessarily, be executing on a CPU.

Load balance limited. $L > 0.5$, but L is not a lot greater than 1. This domain is transitional; load balance between CPUs is important and difficult to maintain, hence the name. Latency remains a limitation in this domain until loading is high enough to hide all latencies.

Bandwidth limited computation. $L \gg 1$.

Thus comparing, or stating, the performance of different architectures is something which should be treated with a good deal of caution. The only easy parameter to compare is $\lim_{n \gg N} f_{SYS}$ (n, N as above), and this becomes less representative of likely application performance as the size of the multiprocessor scales up.

5.3 Multiprocessors

5.3.1 Data representation

I have been uncommittal about the way that heap data is represented in a multiprocessor. I have assumed that data is allocated for thread locals in vectors in a heap: in Chapter 4 I suggested that a frame cache with fixed size vectors be used, and heap data be either represented as compacted lists or accessed from outside the frame cache.

Simple data representation in symbolic computation uses tuples, most likely pairs, allocated from a heap. The issue examined in this section is when this representation should be replaced by one in which semantically separate tuples are encoded into blocks of sequential storage, in the process removing unnecessary internal pointers. The most effective such representation applies to lists, whose elements may be packed into sequential storage allocated in fixed size vectors. in [LH86] Kai Li and Hudak give a suitable allocation strategy which allows arbitrary compact list modification using where necessary indirection nodes when altering the structure of compact lists.

In a concurrent system data modification is relatively rare, it may reasonably assumed that lists once constructed will never be changed. However two important types of list construction may be identified: appending to the head of a list, or recursively constructing the tail of a list. This latter is particularly important in concurrent systems since it is the basic operation in a pipeline (see page 73).

I advocate a simple form of compact list representation in which linked lists of vectors represent lists of data which may be read or extended at the head and constructed at the tail. The reasons for wanting this in a multiprocessor are varied, and not primarily to do with more efficient use of store. Here is a list of factors all of which make compact list representation relatively more attractive in a D-RISC multiprocessor than a von Neumann uniprocessor. All of these apply equally to a D-RISC uniprocessor.

- Efficient global communication between ends of a pipeline.

- Low-level non-deterministic control flow may cost less on a D-RISC. This is just one example of the pervasive effect that optimising for concurrent execution in a D-RISC has on system design decisions. Low level latencies can be hidden so are not so pernicious as in the single thread case.

  It is not clear that any particular D-RISC design will hide code fetch latency in an efficient way, in the end the effectiveness of such a decision can only be evaluated in a complete design. However management of concurrent computation pushes design towards a tagged architecture with a rich conditional branch on tag structure. This in itself means that the hardware to support efficient list compaction is likely to cost less.

- Fixed size vector allocation may be forced by the hardware of a D-RISC frame cache, in which case the overhead of not representing heap compactly is large. This argument depends on the frame cache being of a size which makes cacheing of heap appropriate, and being of fixed size frames; neither of these conditions are necessarily true.

- Local garbage collection facilitates list compaction. This is a subtle and implementation-dependent point. We have seen in Section 5.1.2 that local garbage collection is desirable in a D-RISC to increase frame cache hit-rates. If this is implemented with reference counts, or special unique pointers, lists which are constructed disjointly but have no external references to their insides may be compacted by copying, reclaiming the old store. (This can happen automatically whenever such a structure is read, or perhaps as a background activity).

- Compact lists reduce sequential spines when mapping down lists. A map can be implemented by chaining down linked vectors, exporting each vector to a separate processor for mapping.

- Compact lists implement stacks efficiently. In a concurrent system stacks tend to be shorter and dynamically created, so static allocation of storage for stacks is difficult.

- Lists are used at a low level in concurrent systems for synchronisation and scheduling; these can always be compactly represented.

Global Data Transfer

In a multiprocessor inter-processor data transfer offers extra scope for data compaction. Because inter-processor bus latency is typically long compared with processor cycle time, and specialised hardware to encode and decode data for transmission across a bus lies physically on one edge of a CPU, and so is particularly cheap, it is reasonable to consider relatively complicated data compaction strategies in order to reduce bus bandwidth.

Global transfer of data may be also chunked in order either to reduce bus message overhead or global reading overhead. I have assumed that the values of

threads are transferred completely on reading, even when they are, excluding the values of embedded threads, large structures. This assumption is usually harmless: the nature of concurrency in symbolic computation means that typically structures which are read concurrently (and so should not be chunked and passed totally to every reading processor) are usually constructed concurrently, so that the sizes of thread values are small. Where this is not the case, or where conversely it is desirable to chunk for subsequent global reading a structure containing multiple threads, it is easy explicitly to change data representation. Thus the identification of thread and global data reference, tacitly made in this thesis for simplicity, may where required be broken.

Throughout this discussion of data I assume that heap data is nearly always used in a functional way so that copying or sharing may be determined at the implementation's convenience. In a multiprocessor more data will be copied than would be desirable in a uniprocessor.

Global data references make a suitable unit at which to preserve pointer comparison tests for structure equality. Also unique global reference pointers may be used to extend low-overhead cache local garbage collection across processors.

5.3.2 Thread export

In a multiprocessor threads must be *exported* in order to distribute computation. Thread references can thus become pointers to global memory, even after a thread has finished, and the uniprocessor model of thread implementation discussed in Section 5.1 must be augmented in two ways:

- Inter-processor thread export must be supported.
- Thread reading may result in global communication and or a wait on a thread on a distant processor.

Further consideration of this problem leads to a bewildering variety of possible export strategies. Threads may be exported on creation: **creation-export**, or on demand by an empty processor: **demand-export**. Thread export from fuller to less full processors may occur as a background task to distribute computation. In general these three strategies may be used singly or together, as dynamic conditions dictate.

It is clear that latency limited computation requires creation-export, since every newly created thread must be exported immediately to a waiting empty processor. Equally, in a highly bandwidth limited computation, demand-export is appropriate for reasons discussed in the next two sections.

Export policy can have important interactions with possible implementation optimisations. If the processor on which a thread will be executed is known at thread creation time then any references to the thread may contain this information, so that a subsequent exported reference to the thread may communicate directly with the thread, without indirecting through the thread's parent's processor.

Deterministic Thread Export

So far I have assumed that thread export is determined only by the desire to distribute computation so that all processors are occupied. A more intelligent export strategy would optimise to whom computation is exported in order to maximise cache hit rates or minimise communication bandwidth. The latitude to do this only exists when computation is bandwidth limited and so good thread distribution is easily obtained. Two cases of export to specific processors may be noted:

- Export destination is determined by function name: this allows static instruction storage to be localised, resulting in increased hit rates for a given code cache. In general it is very difficult to allocate code to processors statically while maintaining good distribution, but processors could preferentially pick up exports of functions whose code they have recently cached.

- Export destination is determined by thread data. The best example of this is when data is sparsely accessed from a large structure. The optimal access strategy may be, on reading a non-local thread, to export the entire reading thread to the processor containing the thread's value. Such an exported thread can be seen merely as an intelligent global read request, and its export can dramatically reduce communications bandwidth and latency for the read, especially when some structural locality is preserved. Even with every successive indirection strict on data in a different processor a data following structure access is twice as fast as a conventional one. Note that supply transfer provides an efficient implementation of such a data following thread (which would for example in LISP be something like CDADR).

  The problem with this is that the intermediate data read is not copied and so multiple accesses require repeated communications overhead. Furthermore a particular datum could create a sequential bottleneck to repeated concurrent access which would be eliminated if it were copied.

System efficiency can be affected by thread export and scheduling strategy in both obvious and subtle ways. These may be summarised under three headings: **distribution efficiency, thread locality** and **switching efficiency.** The first two of these are new parameters which have meaning only in a multiprocessor system. The last has however already been extensively discussed in the last chapter. In a D-RISC switching efficiency is related to cache hit rates and depends crucially on scheduling.

5.3.3 Distribution efficiency

Clearly it is bad if threads which are executable do not find their way to processors which are empty. This can be measured by a parameter which I call distribution efficiency and is determined as follows.

At any time let n = no of CPUs, a = no. of ready threads, e = no. of executing threads. The number of wasted CPUs in the system is then $w = \min(a-e, n-e)$.

Then the distribution efficiency in an executing system is $\frac{e}{w+e}$: this represents the reduction in speed due to bad thread distribution. The average distribution efficiency of a program will be 1 if bus latency is negligible and spare ready threads are exported to idle processors.

Distribution efficiency is a measure of performance in both latency and bandwidth limited computation, its average thus shows the total effect of distribution on performance.

Good thread distribution is relatively easy to manage in the extremes of latency limited and bandwidth limited computation. In the former case any thread can be exported to a random processor immediately on creation, the processor will probably be idle in which case thread distribution is optimal, although the corresponding distribution efficiency will not be 1 unless thread export time is 0. In the latter the average number of ready threads per processor is large and so processors which are running low have time to poll other processors for spare threads: again random polling is adequate.

These two methods of thread export—by supply and by demand—are efficient at different ends of system loading. In between, where loading is near 1, more complicated methods in which threads are exported from over-full processors to under-full processors are more efficient; however the gains from such a complicated strategy are minimal because most of the time computation is either Latency Limited or Bandwidth Limited.

5.3.4 Thread locality

In bandwidth limited computation, as the number of ready threads per CPU increases, so does the possibility of localising the execution of created threads. Call (sequential) length of computation still required in the execution of a thread and its descendants the thread's weight. During the evolution of a calculation the total weight of threads on each CPU decreases linearly with time unless it is changed by export moving weight from one CPU to another. The problem is that the run-time system knows only the number of threads on a CPU, not their weight, and so may have to equalise weights at any time in the computation: its task is to ensure if possible that no CPU ever has a total weight of 0, and that every CPU has at least one executable thread. The effectiveness of export in equalising weights depends on the loading of the system, L, since the average fractional change in weight after an export is $1/L$, however the necessity to export computation also depends on L. This analysis gives no insight into the actual variation of locality with loading: the argument in the next paragraph attempts to do this.

The ready number of threads on a CPU without export or import varies unpredictably according to a random walk. If the currently executing thread terminates or waits without issue then it decreases, if the executing thread creates more than one thread before terminating it increases. The rate at which these things happen relates to average thread creation or execution time and is roughly constant for a given calculation irrespective of system loading. It seems not unreasonable therefore to suppose that average time between CPUs being empty without export,

and hence the time between exports, is proportional to $\sqrt{L}$, as it would be if the weights of exported computation were fully random. This hypothesis is supported by my simulation results which export threads on demand and show increasing locality as loading increases.

This locality applies to thread export only and does not imply that data transfer is equally local. In fact typical heap reference in symbolic computation is depressingly non-local. The reason for this can be understood by considering the *distance* between threads. This is the defined to be the number of arcs separating the threads in their computation's TCT. The probability that a thread is executed on its parent's CPU will be called **thread locality**. If this is α the probability of local heap reading between threads separated by a distance d is α^d. Thus analysis of a computation's TCT gives information which, together with a dynamically determined measure of thread locality, shows what global bus bandwidth the computation will need.

In my simulated test programs TCT distance measurements showed clearly that most heap based computations have increasing average and median TCT distances between communicating threads as computation size increases. A certain proportion of reading is TCT local, the rest has distance which increases with program size.

Of the tests that I have examined only NFIB is truly local—it is clear that the distance of all reads is 1. This is true in general of computations which pass data from parents to children and back without skipping generations: such computations have very nice implementation properties and are natural candidates for truly hierarchical multiprocessor implementation.

In this dissertation I do not consider the possible optimisations available in hierarchical computation where data transfer is only allowed between parents and children; the TCT measurements on my test programs show that most symbolic computation is not hierarchical.

TCT distances provide a well defined measure of abstract computation locality which is independent of dynamic execution. Thus all thread export operations are of distance 1 and so (considered abstractly) local. The dynamic thread locality in a system then determines whether an operation of given abstract locality is actually likely to be local.

5.3.5 Switching efficiency

A simple measure of efficiency would be to assume that the overhead is inversely proportional to time between thread switches. This is only true in von Neumann processors where the cost of context switching is direct (from register flushing) rather than indirect from increased cache misses. Even in von Neumann machines cache coherency destruction can account for a large amount of the switching cost.

Suppose that the CPU has a D-RISC-like cache for local data. The cost of thread switching is then related to miss rates in this cache. Note however that in latency limited computation any local caches are underused, since CPUs are idle much of the time, and that in bandwidth limited computation thread switching

may be used to hide this cache miss latency.

Thus the real cost of switching, except in the rare case of load balanced computation, is an increased cache update and flush bandwidth rather than any latency. This bandwidth may be compared with the necessary inter-processor communications bandwidth for export and data transmission, if it is significantly smaller than these it may be ignored.

In sequential computation stack cache hit rates are high because average stack frame lifetime is low. In concurrent computation stack frame lifetime depends on the average number of threads which are being switched between on a CPU. Consider a typical divide and conquer concurrent program. Each thread creates some children, and then waits for them to terminate. The average length of this wait determines stack frame lifetime and hence cache efficiency. This wait is the parallel execution time of the computation, multiplied by the average number of threads which each CPU is executing.

Highly concurrent programs may have a very large system loading: does this necessarily mean that they will have low cache hit rates? The answer to this is no: providing that the right threads are selected for execution by each CPU and that thread export locality is maintained.

Suppose that only a fraction α of all threads that are created on a CPU are ever exported from it. Those that are not may be executed sequentially by the CPU, if ready threads are queued in a LIFO, so that the one most recently created is next executed, this results in a von Neumann-like depth first traverse of the TCT. Consider the progress of a newly imported computation: If it is shorter than $1/\alpha$ threads long it is likely to proceed without interrupting export and its execution time will be no longer than on a sequential machine. This means that stack frames waiting on short computations will be kept in a cache just as well as on a sequential machine.

This ignores the possibility of synchronisation waits in the executing computation. If the computation is D&C then these are only as long as two bus latencies because the data is assumed to come from threads which have finished. This will require thread switching between roughly:

$$1 + \frac{2t_{buslat}}{t_{synch}} = 1 + 2\lambda$$

threads. This is typically no more than 10, and possibly much smaller than this.

Modified LIFO scheduling, as discussed on page 74, means that synch waits will cause switches to threads that are local in the TCT whenever this is possible. If the number of threads between exports is larger than this number most thread switching will be harmless to stack lifetimes because it will be to a thread which is part of the same imported computation.

This argument works for D&C concurrency, where the only synch waits on unfinished threads are parents waiting on their children. Pipelines behave differently. A pipeline corresponds to two calculations that in a sequential machine would be executed separately, with the locals for the first calculation garbage collected before the second one begins. In a concurrent machine these locals coexist

on different CPUs.

In most of the test programs that I consider the average length of pipelines increases as the computation size does, so pipelines can be implemented with compact lists, transferring data (and so waking up a discontinuity) in chunks. Exactly this sort of coarsening of the grain of available concurrency is needed to avoid a high number of concurrent active threads on a processor resulting in bad cache behaviour. Where a pipeline is recognised as such the scheduling of its two ends can even be related to system loading so that whenever computation becomes bandwidth limited the pipeline is executed in sequential chunks, rather than with maximum concurrency.

The concern to schedule pipelines correctly in a multiprocessor system results in a thread which returns multiple values (as, for example, successive elements of a tail constructed list) which thus localises the construction end of the pipeline and results in a unique reference for reading the pipeline. One convenient way to implement this is as supply transfer of thread values to a compactly represented list which is then globally referenced.

5.3.6 Communication bus topology

The global behaviour of a multiprocessor is dependent on the way in which computation is exported. Of particular interest is the thread locality, α. Clearly, in latency limited computation $\alpha = 0$. In bandwidth computation with system loading L, given a minimal export strategy, I hypothesised in Section 5.3.4 above, that $\alpha \approx 1 - L^{-\frac{1}{2}}$. Certainly in simulations as L increases so α approaches 1. The relationship between α and communications bandwidth may be quantified:

$$BW_{comms} = BW_{exp}(1-\alpha) + BW_{read}\sum_{n=1}^{\infty}\rho_n(1-\alpha^n)$$

where ρ_n is the density of inter-thread reads of TCT distance n. This equation is an approximation, and ignores one important factor. I have assumed both in my simulations, and throughout this thesis, that thread values are cached on each processor once read. Consider the common case of a value which is read once by i separate threads. If $i > N$ (the number of processors) this cacheing will reduce communication bandwidths correspondingly. The overall effect of this, since typically $i \propto L$ whenever i is large, is a reduction in communication bandwidth above a loading threshold of order N^2.

This simple analysis, which gives some insight into the effect of loading on communication bandwidths for different types of programs, may be extended to a hierarchically structured multiprocessor. Each node of the structure may be treated in the same way as a globally communicating multiprocessor, with subnodes taking the place of processors. Typical values of α during bandwidth limited computation are larger for higher level nodes since the amount of computation in each node is larger. Export bandwidths localise nicely in a hierarchical structure, but distant read bandwidths do not. This means that some proportion of the system bandwidth remains increasing in proportion to the number of processors

communicating, which results in a bottleneck at the root of any hierarchy. It is easy to choose test programs with either very local or very non-local behaviour, so attempts to quantify these bandwidths are not particularly useful.

In a general multiprocessor storage hierarchy three types of bandwidth are significant:

- Interprocessor communications (BW_{comms} above)
- Static data (e. g. instruction fetch)
- Intermediate data

The last two of these may be reduced above any node in the hierarchy by increasing the nodes cache size, the first is irreducible.

When designing a multiprocessor it would be nice to be able to use empirical data on cache hit rates to balance cache sizes and communication bandwidths throughout the design. In practice the extreme variability of cache behaviour makes any reliance on cache hit rates, except most the conservative, potentially very damaging to performance.

Broadcasting Data

In a multiprocessor there are two ways in which busses which allow cheap one-to-many communication can be used to increase performance. Firstly thread values may have to be distributed globally. A good example of this is in the transposition of a $D*D$ matrix, if this is represented as a list of lists. An efficient and destructive sequential algorithm will transpose this structure without using storage in $O(D^2)$. A parallel algorithm using threads should manage the same job, with admittedly a higher cost for small matrices, in $O(D)$.

Closer inspection shows that there are problems. Any $O(D)$ computation must have separate threads for rows, to scan the original matrix, and for columns, to assemble the transposed matrix. Communication between these two sets must be through some locally non-deterministic means: this is provided within a thread world by the use of sequencers, discussed inChapter 2.

Each column is defined by a sequencer whose local store is a vector of length D. This is written with the required elements by the row threads, using an 'update n^{th} location' operation. Each column thread waits for its sequencer reference count to be zero and then reads the vector, constructing a list.

This algorithm requires one-to-D communication of rows in the matrix and also the names of the column sequencers: it is one where the communication cost changes from $O(D^2)$ to $O(D)$ by using broadcast.

In this example the broadcast can be inferred from the static structure of the program, and receiving threads primed to pick up the necessary data. Whenever the number of inter-thread reads is known to be larger than $(1-\alpha)N$ (α,N as above) it is even easier to broadcast the data globally. In general broadcast may be used in conjunction with a queue of requests on each receiving processor, satisfying multiple requests with one bus transfer. This may be compared with a data

cache, in which each processor holds previous data values rather than requests for values. Broadcast and normal cacheing however have good hit rates in different circumstances. Normal cacheing hits when identical requests on one processor are temporally local, broadcast when identical requests on multiple processors are temporally local. Furthermore broadcast bus bandwidth minimisation, by queuing requests, results in high bus latencies.

5.4 Latency Limited Computation

This section considers the dynamics of multiprocessor execution at low levels of system loading, so that total computation time is limited by latency along a critical path rather than by available CPU bandwidth.

Conventional CPU design has developed a set of understood implementation techniques which optimise computation in a single thread on one CPU. The available operation bandwidth is determined by critical latencies in operand fetch and execution, these are reduced by appropriate local cacheing of data. The last few chapters have shown that bandwidth limited computation can use locality in a similar way, albeit less efficiently.

Latency limited computation is intrinsically non-local and so is least similar to conventional execution. This profound difference is illustrated by an estimate of the cost of a computation. In either single thread or bandwidth limited execution this is proportional to the size (total no of nodes in) the DEG. In latency limited computation it is proportional to the DEG critical path length (CPL): in computation theory this corresponds to execution time on a non-deterministic machine.

The most important parameter determining latency limited performance is the ratio between single message bus latency and average time between synchronisation events (where a thread waits or finishes). I will call this ratio λ: if $\lambda \gg 1$ the length of computation is roughly proportional to the number of critical path bus latencies.

λ is itself determined by the way in which a computation is split into concurrent threads. More concurrency results in shorter thread length and longer λ. The test programs that I consider all have a 'normal' decomposition in which all the concurrency from calls to functions which are possibly long is available. It is impossible to increase thread length from this without losing large amounts of concurrency, conversely the export of all function calls results in no more than a small increase ($< 10\%$) in concurrency with a penalty of much smaller thread lengths.

Having made this static decomposition the average thread length between synchronisation events varies little between different test programs. This is because it relates to the minimum time in which both something sensible can be done and the computation can spawn a thread, e. g. mapping a function down a list. It is easy to devise programs whose inner loops are maps of long functions down lists, but these are not typical of symbolic computation.

Numeric floating point operations, not represented in my tests would on a

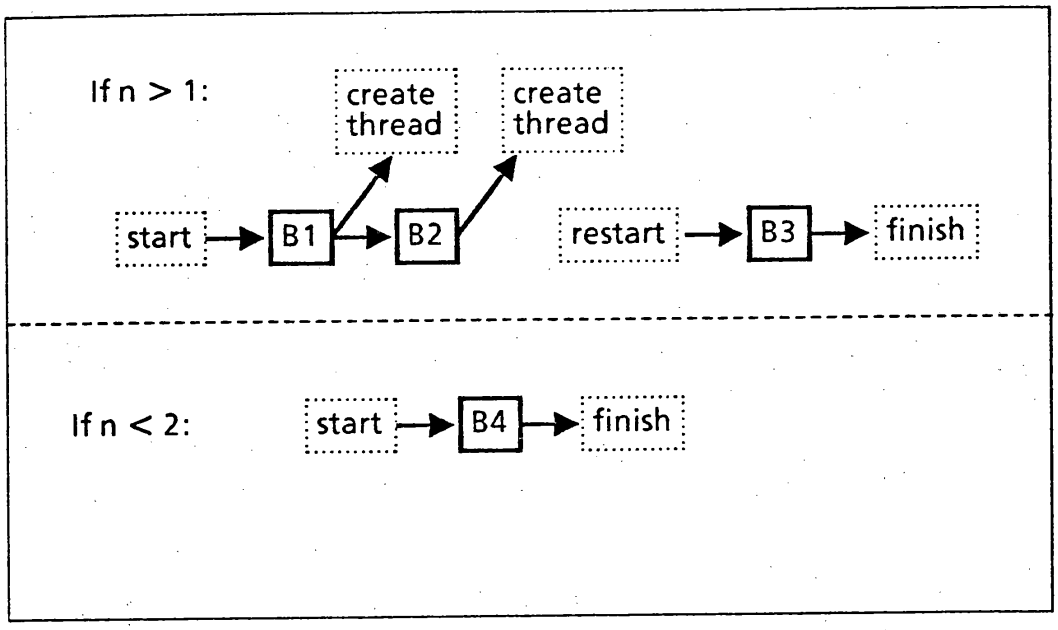


Figure 5.4: DEG of one function call of NFIB.

CPU without specialised numerical support result in much longer minimum thread lengths. However this is unrealistic: hardware for numerical computation would undoubtedly have such support, and in any case high precision arithmetic has itself a natural concurrent decomposition. The efficient concurrent implementation of integer arithmetic is too specialised to be considered here.

There is thus justification for making the pessimistic assumption that typical programs have thread lengths not much larger than the theoretical minimum; certainly if a system cannot efficiently support these its application domain will be very restricted. Having done this λ becomes a parameter determined by the hardware: its value can be established, for a particular technology, by looking at concrete designs.

5.4.1 Optimising NFIB

NFIB, defined by:

$$\text{NFIB n} = \text{If } n < 2 \text{ Then 1 Else NFIB}(n-1) + \text{NFIB}(n-2) + 1$$

is a particularly simple test program. It is certainly not typical of symbolic computation, however its simple structure illustrates clearly some general optimisation techniques. For the purposes of this analysis it is similar to many other simple divide and conquer programs, e.g. FIB.

Figure 5.4 shows the DEG of one function body of NFIB. This either returns after some computation, or spawns two children on which it then waits.

The entire execution dynamics of NFIB are contained in 5 execution times, representing activity between interesting events, and two critical latencies: one

from thread creation to its starting on a different CPU, one from thread finishing to wakeup of a waiting parent.

The critical path for NFIB(n) contains $n-1$ repetitions of:

from		to	time
start	→	create	$B1$ or $B1+B2$
create	→	start	L_{export}
finish	→	restart	L_{send}
restart	→	finish	$B3$
		And one of:	
start	→	finish	$B4$.

If $4 > n > 1$ the two children NFIB(n) are sterile threads of length $B4$, otherwise the NFIB(n-1) thread will be roughly 1.6 times longer than the NFIB(n-2) thread. This has an important implication: for NFIB functions with $n > 3$ the critical execution from start to creation will be $B1$ or $B1+B2$ according to whether NFIB(n-1) is created first or second: optimal compiling must ensure that it is first.

The total CPL for $n > 4$ is then: $n(B1 + B3 + L_{export} + L_{send}) + B4 + B2$ This contrasts with a total CPU time, significant in bandwidth limited computation, of NFIB(n)$(B1 + B2 + B3 + B4)/2$, since half of the calls to NFIB are sterile.

Thus optimal compiling for latency limited computation must minimise $B1$ and $B3$ at the expense of $B2$ and $B4$.

This decomposition of NFIB into threads results in 2 bus latencies per function body on the critical path. An important optimisation removes bus latency from the critical path by noting that NFIB(n-1) is nearly always longer than NFIB(n-2), and that creation of two child threads for NFIB is unnecessary. If the NFIB(n-1) computation is kept and executed by the parent the difference in (critical path) lengths of the two computations can absorb bus latency, resulting in a critical path which is purely function execution.

In general[2] whenever a computation forks it is good policy to keep one half of it with the parent, and if possible choose that which is known or guessed to be shorter. The bus latency associated with a computation fork is thus either $L_{export} + L_{send}$ or 0. The parameter L_{send} is composed of either one bus latency, if the request for the sent data is not critical, or two latencies. This must be qualified. In a system with a bus that allows reading of foreign memory in one operation there may be little difference between these two times. Furthermore if it can be established statically (as is the case here) that the result of a thread is needed by another thread *and the CPU on which this thread is or will be executing is known* then the thread can send its data directly to the requesting CPU without any data request.

[2]If one path from root to bottom of the TCT is never exported the resulting strand of computation committed to one processor is long. This can create a load balance problem as a divide-and conquer computation nears its finish and most threads terminate if the alive ones happen to be on the same processor. Although long threads mean less thread management cost *very* long threads can hurt latency limited computation.

NFIB illustrates an important fact in bandwidth limited concurrent implementation. Half of the calls to NFIB result in short sterile threads (though with the above optimisations this reduces to one quarter). In general CPU use is determined by the efficiency with which leaves of the TCT can be executed: there are usually a large number of these. This makes bus latency particularly significant.

5.4.2 Cloning

While a multiprocessor system is mostly idle it is natural to try to think of some use for the spare processing capacity. This section introduces an extraordinary way in which latency limited computation can be optimised by using spare processors to execute replicas of threads.

In [GLV84] Garcia-Molina et al propose an interesting scheme for managing very large address spaces on a uniprocessing system. Their idea is that a number of processors, each with part of the address space local to it, should execute the same program in lock-step synchronism, with the single processor which has access to memory at any time broadcasting data read from its memory to all the others. Whenever the sequence of data reads moves from one memory to another the corresponding delay is 1 bus latency, otherwise no global bus latency is incurred. This performance is superior to that obtained with a single processor performing (necessarily global) accesses to the whole memory.

I will now propose an extension of this technique to multiprocessors, in which lock-step execution is used to reduce global bus latencies encountered in export and reference. I call this process cloning and devote some time to discussing its properties. While it is has not been proven that cloning is a viable implementation technique it deserves consideration because it is a way of substantially speeding up latency-limited computation in a multiprocessor system with high λ. It is thus at least of theoretical interest.

The basic operation of cloning is the export of a single thread to a number of idle processors, which proceed simultaneously and identically with its execution. The copies of the exported thread will be called clones and will each lead to the same value.

The apparently redundant process of cloning leads to two implementation advantages:

Invisible Export. When copies of a thread need to create a thread its export can be accomplished without overhead by splitting the copies into two sets which execute the old and new threads respectively. Thus a number of individual exports have been replaced by a one multiple export creating the copies, which will be called clones. This reduces bus use if one to many communication is allowed, although bus bandwidth is hardly likely to be a limitation in very latency bound computation. More significantly the splitting of the cloned threads on thread creation introduces no export latency overhead.

Cache Prefetch. Every time a new calculation is exported to a previously idle processor it results in code and static data cache update overhead. If spare processors are occupied with copies of an executing function these cache updates happen simultaneously. This may reduce bus bandwidth if broadcasts are used, and reduces the contribution that cache miss latency makes to the critical path.

The second of these two advantages may be obtained without the full complication of cloning: idle processors could monitor a snoopy bus and update their caches automatically.

The implementation of cloning presents formidable complexity and proceeds as follows:

On exportable thread creation from an uncloned thread when the system is lightly loaded the child thread is cloned and exported to a number of idle processors.

On exportable thread creation from a cloned thread each clone uses a personal id to decide whether to execute the created thread or continue with the parent. This id could have been present in the cloning export, though this is not possible in a simultaneous broadcast. One possibility for export generating unique ids is to export in a binary tree, with each processor exporting to 2 others. This would work well with global network interconnection.

It is necessary for every clone to know what its copies will do, so that it knows when it is the only copy of a thread left, then it can itself create a new cloned export. One way of managing this is to clone a thread to 2^n processors, every subsequent thread creation halving the number of processors attached to the thread. This is wasteful of the original export unless the TCT is a binary tree, but it is easy to implement. Every processor can tell whether it is the only clone left by comparing its TCT distance from the initial clone export with n.

TCT shapes cannot usually be predicted in advance so cloning will be wasteful of processors, to clone through n exports with this method requires an initial export to 2^n processors.

Cloning can thus, at the cost of profligate processor use, reduce the number of export latencies in the CP arbitrarily. Multiple copies of threads can also be used to reduce the number of bus latencies when reading data. Whenever a critical datum is local to one of the copies it sends it to its clones: computation proceeds with a single bus latency whenever the sequence of data reads on the critical path changes to data on a different processor. In divide and conquer computation clones that have split into different threads can coalesce afterwards, so reducing to an absolute minimum the cost of inter-thread communication.

At any stage in cloning a global unclone signal must be able to release all copies except one of each thread: again this requires careful predetermined cooperation between clones.

In return for this complication cloning offers a possible reduction in export latency of n, using 2^n spare processors, and a reduction in reference latency which depends on the nature of inter-thread data reference and is therefore difficult to quantify.

Cloning techniques may be valuable in certain specialised applications though they are formidably complicated and have bandwidth use which is in general exponential on the possible gain. However this is bandwidth which is otherwise unused and so cloning may be viable in systems with long communication latency.

5.5 Analysis of Test Programs

In a book based on his experience with benchmarks of LISP systems, [Gab85], Gabriel discusses both the advantages and difficulties of detailed analysis of benchmarks. He suggests that the real constraints on performance of a concrete implementation are often found only by detailed study of puzzling benchmark times, from an attempt to understand these obscure figures comes an understanding of neglected and important implementation details.

In multiprocessor architectures the task of benchmarking is much more difficult because of the great variation in dynamic behaviour between different programs. This section discusses some easily identifiable properties of the programs which I simulated during my study of multiple thread implementation.

The first section below discusses the concurrent structure of symbolic computation and analyses the test programs that I use. The second section describes the simulation and analyses the results.

5.5.1 Concurrent computation.

As we saw in Chapter 2 the structure of a concurrent computation is represented by its DEG. In order for a DEG to be executed on hardware it is compiled into statically determined threads. The DEG then reduces to a set of threads related by creation dependence and data dependence. The creation dependence of threads in a computation may be represented by the computation's TCT, data dependence may be analysed with respect to this.

When examining the behaviour of particular programs three types of concurrency may be described particularly simply: maps, pipelines and divide and conquer programs.

- **Maps** are threads which spawn a large number of independent subthreads, for example a thread which maps a function over a set. In functional languages sets are often implemented by lists, this imposes an inherent sequentiality on any corresponding map which is not important if the ratio of mapped function time to list node traversal time is greater than the length of the list.

- **Pipelines** are in contrast formed by two processes which communicate through a list which one of them constructs and the other concurrently examines. The concurrency of a pipeline is limited by the balance of its producer and consumer and is between one and two. The producer of a pipeline must be implemented by a number of threads in order to release elements of the

constructed list as soon as they are ready. This is a typical example of a situation where thread boundaries are needed in a computation which is single threaded and not in itself concurrent. The efficient implementation of pipelines must preserve locality between different threads.

- **Divide and conquer, or D&C,** programs have data dependency only from threads to their ancestors or descendents in the TCT. This dissallows the passing of a thread's name by its parent to one of its siblings. Programs which further restrict data dependence to between parents and children will be called **immediate divide and conquer**, and are particularly easy to implement because they map efficiently onto hierarchical communication topologies. Divide and conquer computation gets its name from the supposition that a parent splits its computational task into chunks which it hands to its children.

I will call data dependence which limits the possible order of execution of threads critical, since its corresponding DEG arc is critical. These three types of concurrency represent respectively the three possible types of critical communication between a thread and its siblings: none, intra-generation and inter-generation.

My use of functional languages as a basis for a computational model of concurrency, and the corresponding model of threads, requires that communication between siblings be only one way: two threads cannot be simultaneously created and passed each other's references.

The non-referentially transparent nature of code that can usefully use such cyclic pathways makes it unattractive in a concurrent system. Sequencers provide a form of two-way communication between siblings which does not in any way constrain execution order, and so need not be separately categorised here.

Annotations The concurrent structure of the code which I describe is exhibited by annotating those function calls which are to be exported as concurrent threads. All other code runs sequentially. Function calls are marked by prefixing characters to the name of the function whose call is to become a thread as follows:

`!.`	Create an exportable thread
`!$`	Tail recurse in a pipe with the current thread
`!|`	Create a new thread which is bound to its parent processor

5.5.2 Test programs

Some of the test programs which I examine here are chosen as exemplars of particular types of concurrent computation. However the polynomial arithmetic programs were extracted from a large LISP algebra system (REDUCE), with no alteration. They are as representative of 'typical' symbolic computation as any medium sized program is.

The language in which all code is written is a pure subset of LISP. Property list modification is not allowed, nor in general are non-local variables. However fluidly

Name	Code size (words)	Type	sequential time	parallel time
NFIB	30	D&C	$O(1.6^n)$	$O(n)$
LISTMUL	160	pipe, map	$O(n^2)$	$O(n)$
MERGESORT	110	D&C, pipe	$O(n \log n)$	$O(n)$
TREECRAWL	240	D&C, map	$O(n)$	$O(\log n)$
MUL	1500	D&C, map	$O(n^2)$	$O(n)$
MULV	1500	D&C	$O(n^2)$	$O(\log n)$
BOOLEVAL	670	D&C	$O(2^n)$	$O(n)$
BOOLENV	700	D&C	$O(2^n)$	$O(n)$

Figure 5.5: Test Program Summary

bound variables are used to pass parameters from parents to children, and globals are used used to parametrise functions as part of their static definition. These two exceptions allow the use of substantial chunks of REDUCE without modification. The source listings for these programs, annotated for concurrent execution, are given in Appendix B.

Figure 5.5 lists the tests and summarises their characteristics, a more detailed analysis of each is given below. The most notable feature of these tests is the way that even simple code can give rise to concurrent structure which is complex: the structure of mergesort is described in Section 5.5.3.

The first four tests are simple programs testing particular sorts of concurrent structure.

NFIB. Immediate (see above) binary divide and conquer program.

LISTMUL. Simple univariate non-sparse polynomial multiply using lists of coefficients to represent polynomials.

MERGESORT. Merge sort using binary splitting of lists, see Section 5.5.3.

TREECRAWL. A tree is created, reconstructed in mirror image by a traverse, and then traversed again. The two traverses each reference every node just once; each traverse is itself D&C, but the three parts of the program are pipelined.

The next four tests are more general. The first two are taken with no change except export annotation from an existing large algebra system; the last two breadth-first search to find the boolean variable sets that satisfy a proposition.

MUL. Multiply two univariate polynomials using a general multivariate spares representation polynomial multiply.

MULV. Multiply two multivariate polynomials.

```
Symbolic Procedure mergesort( l);
  If l AND cdr l
  Then  merge( !.mergesort !.halfof l, !.mergesort !.halfof cdr l)
  Else l;


Symbolic Procedure merge( l1, l2);
 If Null l1 Then l2
  Else If Null l2 Then l1
    Else
      Begin
        scalar x, y, z;
        x := car l1;
        y := car l2;
        return If orderp( x, y)
               Then x . !$merge( cdr l1, l2)
               Else y . !$merge( cdr l2, l1)
      End;


Symbolic Procedure halfof( l); l AND car l . (cdr l AND !$halfof cddr l);
```

Figure 5.6: Mergesort

BOOLENV. Breadth-first search a boolean expression using a list of variables so far assigned which is incrementally and non-destructively changed.

BOOLEVAL. Breadth first search a boolean expression re-evaluating it whenever a variable is bound.

Concurrency profiles for these test programs, running on idealised hardware with no bus latency and a large number of processors, can be found in the figures at the end of this chapter.

5.5.3 Conclusions from implementation of mergesort

The functional algorithm for Mergesort reproduced in Figure 5.6 has a concurrent structure which is particularly interesting.

The *mergesort* function creates a divide and conquer solution in which the input list is first subdivided into trivial lists by a binary tree of *halfof* threads, and then merged by a binary tree of *merge* threads. These two trees are created from the top downwards by mergesort: the depth of the two trees is determined dynamically and is approximately $\log_2$ of the size of the input list. Figure 5.7 illustrates this decomposition and shows the list dataflow through the two trees.

The thread decomposition of *halfof* is particularly problematic. A naive decomposition into threads without explicit implementation of pipes leads to a very large number of threads being generated. A solution is to make *halfof* a pipe thread with

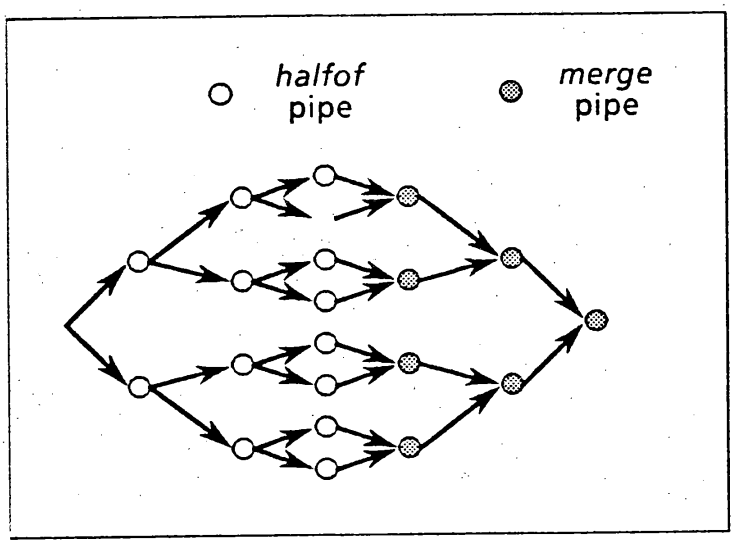


Figure 5.7: Concurrent structure of mergesort

a concurrently generated input and output list. This is signified in the source by a !$ annotation. In the implementation this results in a *cons* of the first element of the output list followed by a modified tail recursion. The trick is to perform this recursion without creating a new thread name. One way of doing this is by supply transfer of the thread value each to a succession of heap addresses, marking each such address as a system object which cannot be copied until it has been given a value by the appropriate tail recursion. Figure 5.8 illustrates the state of heap during two successive tail recursions of such a pipe implementation.

Use of pipes reduces the number of created threads in mergesort by a factor of $\approx \log_2 n$, where n is the size of the list to be sorted.

Another optimisation of mergesort is motivated by its tree structure, and the redundancy of having two separate copies of *halfof* to split each list into two. This operation could more efficiently be performed by a thread with one input pipe and two output pipes. Although this cannot be directly represented in a functional language it is a potential lower-level optimisation. Unfortunately it cannot directly be deduced from the source program without complicated analysis of *halfof*, and so this type of optimisation seems out of the reach of compilers at the moment.

I do not address the question of how threads with multiple output pipes should be represented in a source language.

5.6 Summary

This chapter contains two main results. The first, in Section 5.1.4, relates scheduling to resource use in symbolic computation, and proposes a strategy for minimising intermediate data lifetimes. This work relates to that of Ruggiera and Sargeant [RS87] on dataflow machines, where the same problem is explored empirically, and in the different framework of dataflow graph execution. My work for the first time establishes the relationship between von Neumann stack-based compiled language

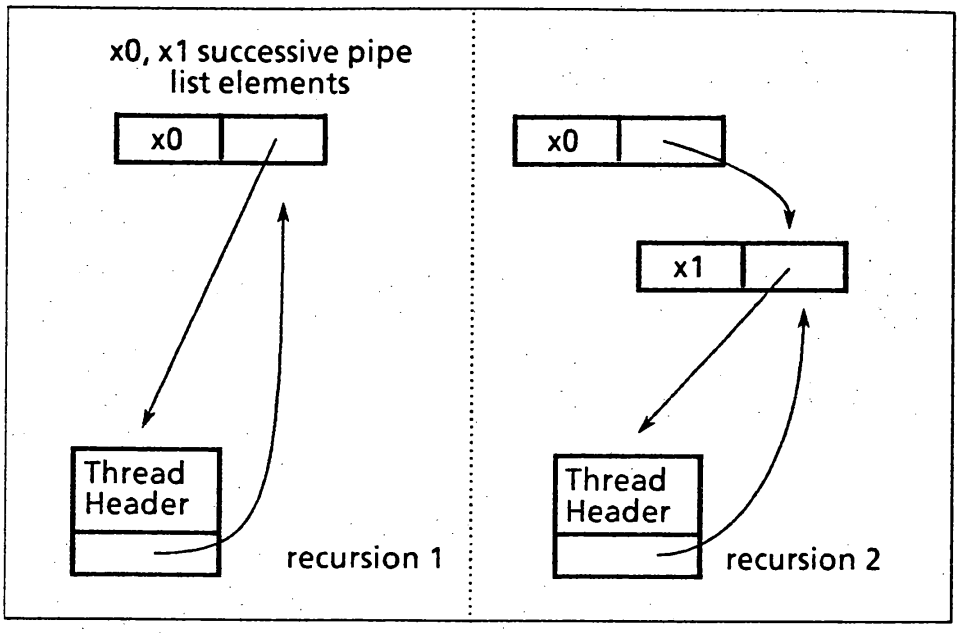


Figure 5.8: Implementation of pipe

execution and concurrent execution of threads either independently or with discontinuities.

Second, I analyse the problems of *latency limited* computation on a lightly loaded multiprocessor system. This has as far as I know received no attention from other researchers: I propose a new technique, called cloning, which trades unused processors in a lightly loaded multiprocessor system for decreased communication latency by executing multiple copies of threads. This is theoretically interesting, although its practical application is far from clear.

The other work on thread implementation parallels that of Halstead [Hal85] who has explored the same types of implementation problem using specifically imperative programming constructions, instead, as I have, of considering possible optimisations of functional programs.

This chapter is a compendium of implementation problems: it therefore raises more questions than it answers. Is compact list representation worthwhile in a multiprocessor system? Should system design pay especial attention to optimum latency limited as well as bandwidth limited computation? The first question that cannot be answered without much further work, the second requires an accurate specification of application and multiprocessor size.

The motivation behind this work, as with the rest of this dissertation, was to establish a framework within which multiprocessor CPU design for concurrent computation could reliably be explored. This chapter concentrates on problems which are intrinsic to any multiple thread implementation, and should therefore be carefully considered in any concrete design.

There remains much work in this area for future (concrete) implementations to address.

Concurrency Profiles

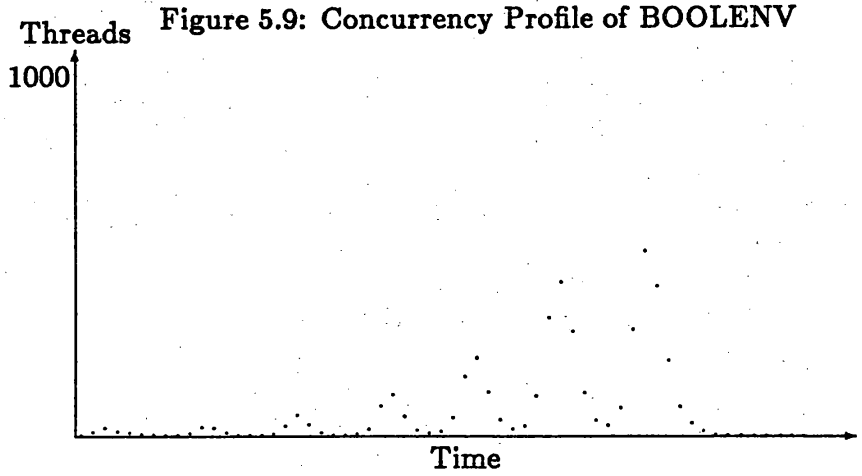


Figure 5.9: Concurrency Profile of BOOLENV

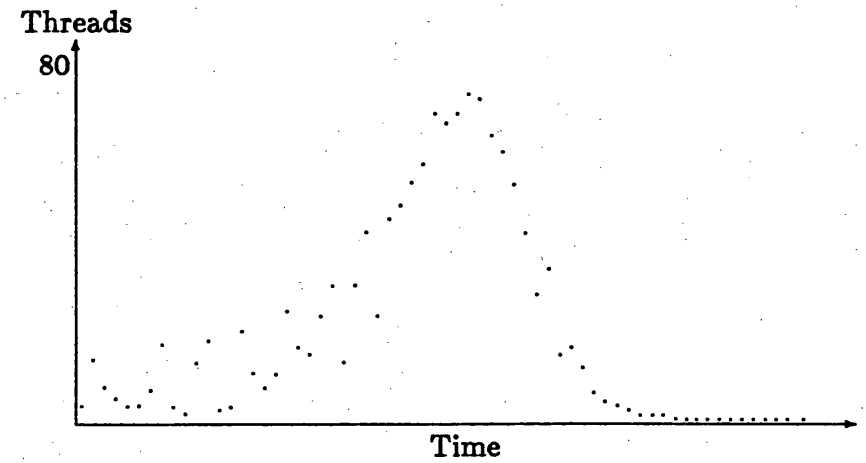


Figure 5.10: Concurrency Profile of BOOLEVAL

Figure 5.11: Concurrency Profile of MERGESORT

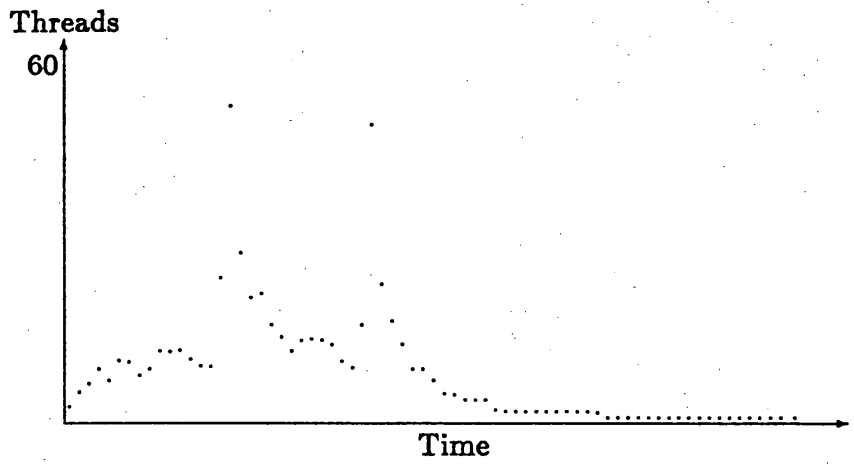


Figure 5.12: Concurrency Profile of TREECRAWL

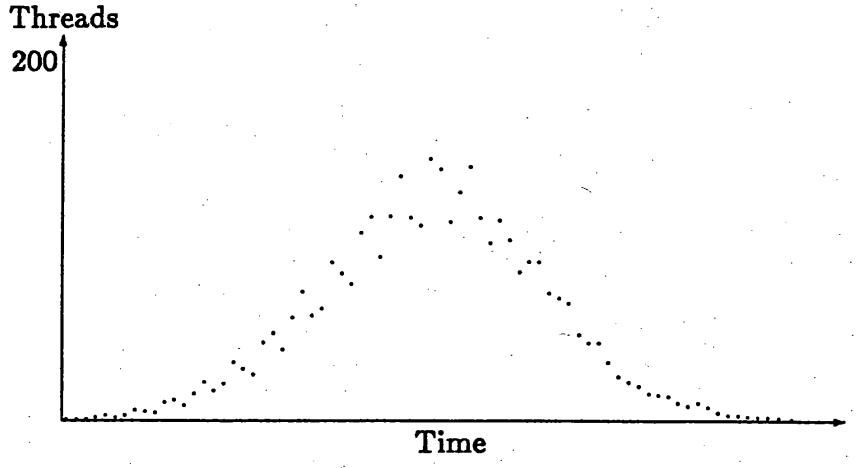


Figure 5.13: Concurrency Profile of LISTMUL

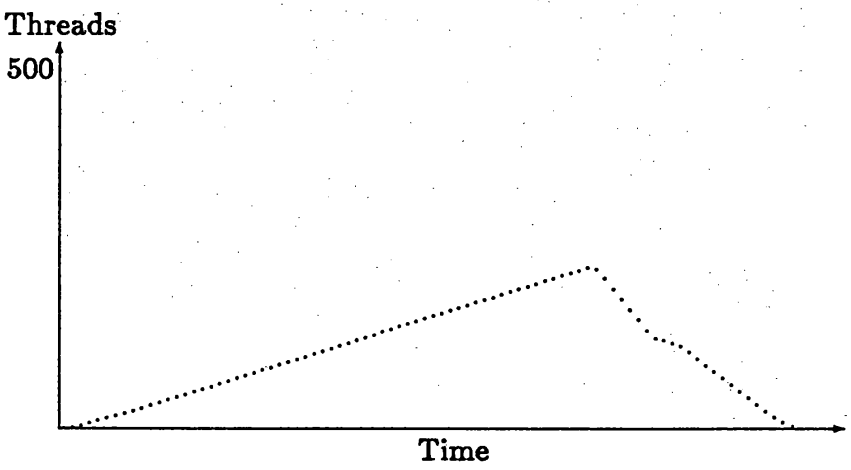


Figure 5.14: Concurrency Profile of MULV

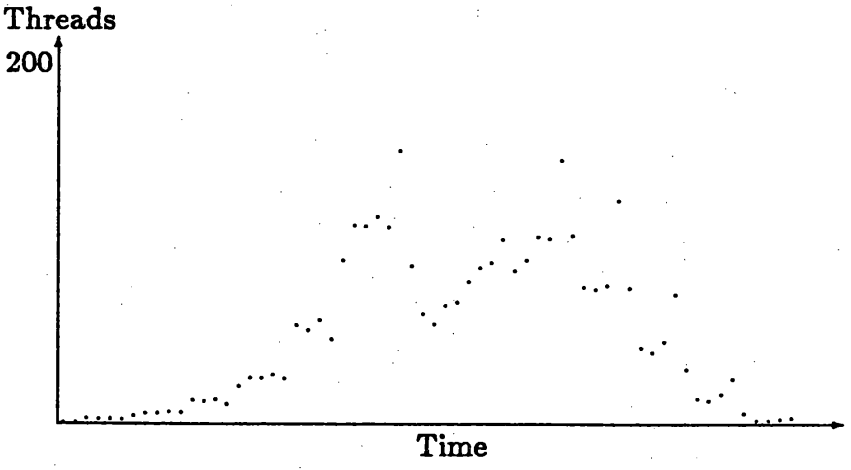


Figure 5.15: Concurrency Profile of MUL

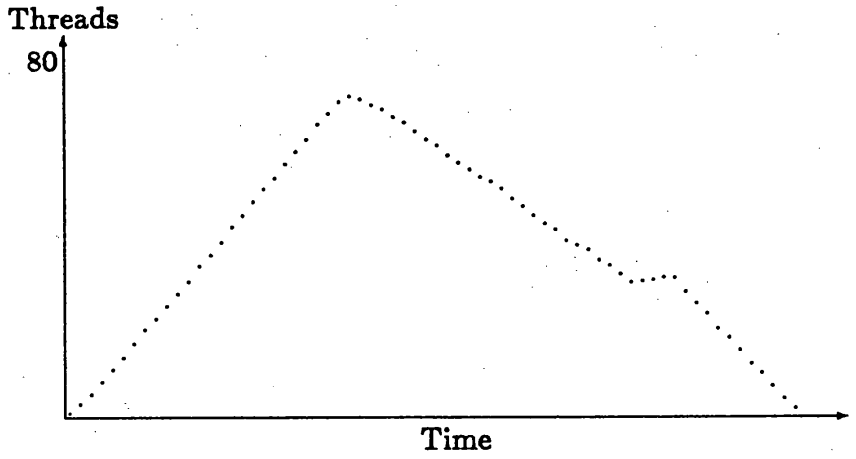
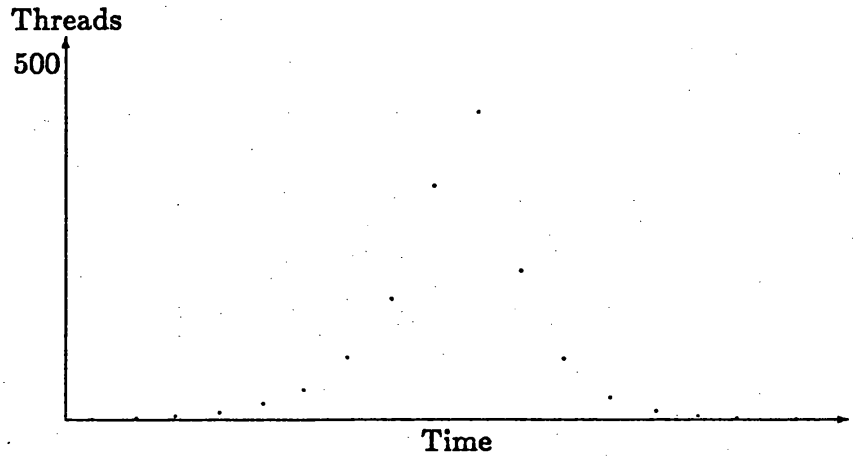


Figure 5.16: Concurrency Profile of NFIB



Chapter 6

Conclusions and Directions for Further Research

This dissertation raises many questions about concrete D-RISC design and implementation which are nowhere answered. My motivation for this work was indeed as a preliminary feasibility study for a multiprocessor CPU design. When I started this work I was thinking in terms of a conventional multiprocessor design with specialised support for memory management, and achieving performance through very high bandwidth low latency bus technology, as for example GRIP [PCSH87].

The work was driven by a dissatisfaction with current understanding of the constraints which govern multiprocessor CPU performance, and the investigation of D-RISCs was a result of this.

There are thus two parallel strands of argument in this dissertation. One is the theoretical analysis of computation starting with the definition of sequencers in Chapter 2, and ending with the discussion of possible processor architectures in Section 4.5. The other is the study of practical implementation problems in Chapter 5. Both of these strands were developed by me independently of other work, both are now being investigated by other researchers.

The study of latency-limited computation in Section 5.4 is a contribution to the understanding of multiprocessor execution dynamics with I believe no related work. The work on thread implementation has some common ground with Halstead's work on the Multilisp language [Hal85], although his investigation does not extend to radically different hardware design, or intermediate data cache hit rates. Interestingly, I discovered after completing my work that Halstead had suggested in [Hal81] that thread-switching be used in multiprocessors to hide global communication latency.

The work on CPU architecture has as far as I know no related work except, recently, for the parallel work carried out by Ianucci [Ian88], whose recent doctoral thesis is a proposal for a particular hybrid von Neumann / dataflow design based on experience with dataflow machines. It is very exciting to find other research on this and encouraging that D-RISC architectures should be proposed by somone starting from dataflow design technology and concepts. Ianucci is concerned with the advocacy of one particular architecture rather than an investigation of the

fundamental constraints on any concurrent achitecture: his work is thus complementary to mine.

The most important result of my work comes from the interconnection of these two research areas: implementation and architecture design. The work on D-RISC cache hit rates shows why symbolic concurrent computation has been such a thorny implementation problem, but also that careful attention to details of scheduling in a D-RISC design can ameliorate this.

The power of the theory presented in Chapters 2 and 3 derives from its minimalist assumptions about implementation techniques. The two single-thread operations which are commonly regarded as primitive—instruction execution in a fixed order and data location update—are shown to be potential optimisations of a more fundamental computational model of non-destructive function evaluation and atomic sequencer operations. This is the simplest starting point for a discussion of CPU design: if we were not exposed to concepts related to sequential programming languages we would naturally consider direct DEG execution on a tagged dataflow machine to be simple, and von Neumann single-thread execution to be an extremely bizarre and highly complicated model of computation.

The future research which this work most clearly provokes is a concrete D-RISC design. The remarks in Section 4.3.5 on the importance of new memory design in a D-RISC architecture imply optimal D-RISC design is an extremely large project. Nevertheless there are many smaller investigations which would be valuable.

The freedom to use synchronous and coherent orginisation in many different ways in a D-RISC CPU makes the structure of D-RISC design space extremely mysterious. I feel that this remains the fundamental problem for computer designers: in a concurrent system what combination of synchronism and coherence leads to highest performance? The comparison of bandwidths and latencies of connections between cooperating units may offer some more quantitative insight here, however the work in this thesis marks the extent of investigation possible for me without more specific consideration of concrete designs. I hope that future communication with Ianucci and other researchers at MIT will lead to further application of the theory developed here.

Appendix A

Glossary

Appendix B

Test Program Listings

MULTF

The test programs MUL and MULV use the same algebraic polynomial multiply routine (MULTF below) with different data. Mul multiplies two single-variate polynomials, multv multiplies two multivariate polynomials. The corresponding recursive structure is different.

```
SYMBOLIC PROCEDURE MULTF(U,V);
   %U and V are standard forms.
   %Value is standard form for U*V;
   BEGIN SCALAR NCMP,X,Y;
    A:  IF NULL U OR NULL V THEN RETURN NIL
         ELSE IF ONEP U THEN RETURN V
         ELSE IF ONEP V THEN RETURN U
         ELSE IF DOMAINP U THEN RETURN MULTD(U,V)
         ELSE IF DOMAINP V THEN RETURN MULTD(V,U)
         ELSE IF NOT(!*EXP OR NCMP!* OR WTL!* OR X)
          THEN <<U := MKPROD U; V := MKPROD V; X := T; GO TO A>>;
        X := MVAR U;
        Y := MVAR V;
        IF (NCMP := NONCOMP Y) AND NONCOMP X THEN RETURN MULTFNC(U,V)
         ELSE IF X EQ Y
          THEN <<X := !.MKSPM(X,LDEG U+LDEG V);
                 Y := !.ADDF(!.MULTF(!*T2F LT U,RED V),!.MULTF(RED U,V));
                 RETURN IF NULL X OR NULL(U := MULTF(LC U,LC V)) THEN Y
                   ELSE IF X=1 THEN ADDF(U,Y)
                   ELSE IF NULL !*MCD THEN ADDF(!*T2F(X .* U),Y)
                   ELSE X .* U .+ Y>>
         ELSE IF ORDOP(X,Y) OR NCMP AND NONCOMF LC U
          THEN <<Y := !.MULTF(RED U,V);
                 X := MULTF(LC U,V);
                 RETURN IF NULL X THEN Y ELSE LPOW U .* X .+ Y>>;
        Y := !.MULTF(U,RED V);
        X := MULTF(U,LC V);
        RETURN IF NULL X THEN Y ELSE LPOW V .* X .+ Y
   END;
```

```
SYMBOLIC PROCEDURE MULTD(U,V);
   %U is a domain element, V a standard form.
   %Value is standard form for U*V;
   IF NULL V THEN NIL
    ELSE IF DOMAINP V THEN MULTDM(U,V)
    Else If domainp lc v Then lpow v .* multdm( u, lc v) .+ !$multd( u, red v)
    ELSE LPOW V .* !.MULTD(U,LC V) .+ !$MULTD(U,RED V);


SYMBOLIC PROCEDURE ADDF(U,V);
   %U and V are standard forms. Value is standard form for U+V;
   IF NULL U THEN V
    ELSE IF NULL V THEN U
    ELSE IF DOMAINP U THEN ADDD(U,V)
    ELSE IF DOMAINP V THEN ADDD(V,U)
    ELSE IF PEQ(LPOW U,LPOW V)
       THEN (LAMBDA (X,Y); IF NULL X THEN Y ELSE LPOW U .* X .+ Y)
               (!.ADDF(LC U,LC V),!.ADDF(RED U,RED V))
    ELSE IF ORDPP(LPOW U,LPOW V) THEN LT U .+ !$ADDF(RED U,V)
    ELSE LT V .+ !$ADDF(U,RED V);

SYMBOLIC PROCEDURE ADDD(U,V);
   %U is a domain element, V a standard form.
   %Value is a standard form for U+V;
   IF NULL V THEN U
    ELSE IF DOMAINP V THEN ADDDM(U,V)
    ELSE LT V .+ !$ADDD(U,RED V);
```

POLYMUL

```
Symbolic Procedure polymul( a, b);
  a AND b AND
  ((car a * car b) . polyplus( 0 . !.polymul(cdr a, cdr b),
            !.polyplus( !.pdmul( car b, cdr a), !.pdmul( car a, cdr b))));

Symbolic Procedure pdmul( d, p);
  p AND ((d * car p) . !$pdmul( d, cdr p));

Symbolic Procedure polyplus( a, b);
  If Null a Then b
  Else If Null b Then a
  Else (car a + car b) . !$polyplus( cdr a, cdr b);
```

NFIB

```
Symbolic Procedure nfib( n);
  If n < 2 Then 1 Else 1 + !|nfib( n - 1) + !.nfib( n - 2);
}
```

MERGESORT

```
Symbolic Procedure mergesort( l);
  If l AND cdr l
  Then  merge( !.mergesort !.halfof l, !.mergesort !.halfof cdr l)
  Else l;


Symbolic Procedure merge( l1, l2);
 If Null l1 Then l2
  Else If Null l2 Then l1
    Else
      Begin
        scalar x, y, z;
        x := car l1;
        y := car l2;
        return If orderp( x, y)
               Then x . !$merge( cdr l1, l2)
               Else y . !$merge( cdr l2, l1)
      End;


Symbolic Procedure halfof( l); l AND car l . (cdr l AND !$halfof cddr l);
```

TREECRAWL

```
Symbolic Procedure treecrawl( arity, size);
  inspect_tree !.reverse_tree !.make_tree(arity, size, 0);

Symbolic Procedure make_tree( arity, size, level);
Begin
  Scalar l,x;
  If size <= arity
  Then << For n = 1 : size Do l := level . l;
          return l
       >>
  Else << For n = arity step -1 until 2
          Do << x := mrandom iquotient( (size * 2), n) >>;
          l := !.make_tree( arity, x, level + 1) . l;
          size := size - x;
          return( !$make_tree( arity, size, level + 1) . l)
       >>
End;


Smacro Procedure mapcar1( l, f);
Begin
  Scalar ww11, ww12;
  ww11 := l;
```

```
  While wwl1
  Do << wwl2 := f car wwl1 . wwl2;
        wwl1 := cdr wwl1
     >>;
  return wwl2
End;

Symbolic Procedure mrandom( n);
%returns a pseudo-random number in range 0 to n
Begin
  Scalar r;
  r := iquotient( random(), 1001);
  return iremainder( r, n+1)
End;



Symbolic Procedure reverse_tree tree;
  mapcar1( tree, !.reverse_tree1);

Symbolic Procedure reverse_tree1 x;
  If atom x Then x Else reverse_tree x;


Symbolic Procedure reverse1 x;
Begin
  Scalar l;
  While x Do << l := car x . l; x := cdr x >>;
  return l
End;

Symbolic Procedure inspect_tree tree;
Begin
  Scalar n;
  If atom tree Then return( tree OR 0)
  Else << tree := mapcar1(tree, !.inspect_tree);
          n := 0;
          While tree Do << n := max(car tree, n); tree := cdr tree >>;
          return n
       >>
End;
```

BOOLENV

```
Symbolic Procedure boolenv( exp, vb, v);
Begin
  Scalar x, exp1, exp2;
  If Not b_eval( exp, vb) Then return nil;
  If Not v Then return list vb;
  If exp EQ T Then return b_perms( v, vb);
  x := car v;
  v := cdr v;
```

```
  exp1 := !.boolenv( exp, (x . T) . vb, v);
  exp2 := !.boolenv( exp, (x . Nil) . vb, v);
  return b_join( exp1, exp2)
End;


Symbolic Procedure b_join( a, b);
  If a AND b
  Then rappend( a, b)
  Else a OR b;


Symbolic Procedure b_eval( exp, vb);
Begin
  Scalar x, y, z, l, l1;
  If numberp exp
  Then If x := atsoc( exp, vb)
       Then return cdr x
       Else return exp
  Else If atom exp Then return exp
  Else << y := car exp;
          l1 := exp;
          While ( l1 := cdr l1)
          Do l := !.b_bind( car l1, vb) . l;
          If y EQ 'not
          Then If Not (z := car l)
                  OR z EQ T
               Then return not z
               Else return list( 'not, z)
          Else If y EQ 'or
          Then << While l
                  Do If (z := car l) EQ T
                     Then << l := nil; l1 := T >>
                     Else << If z Then l1 := z . l1;
                             l := cdr l
                          >>;
                  If l1 EQ T OR Not l1 Then return l1
                  Else return( exp)
               >>
          Else If y EQ 'and
          Then << While l
                  Do If Not (z := car l)
                     Then << l := nil; l1 := 'F >>
                     Else << If z NEQ T Then l1 := z . l1;
                             l := cdr l
                          >>;
                  If l1 EQ 'F OR Not l1 Then return not l1
                  Else return( exp)
               >>
          Else errp_t("Bad expression")
       >>
End;
```

BOOLEVAL

```
Symbolic Procedure booleval( exp, vb, v);
Begin
  Scalar x, exp1, exp2;
  If Not exp Then return nil;
  If Not v Then return list vb;
  If exp EQ T Then return b_perms( v, vb);
  x := car v;
  v := cdr v;
  exp1 := !.booleval( b_bind( exp, list(x . T)), (x . T) . vb, v);
  exp2 := !.booleval( b_bind( exp, list(x . Nil)), (x . Nil) . vb, v);
  return b_join( exp1, exp2)
End;



Symbolic Procedure b_perms( v, vb);
Begin
  Scalar x;
  If Not v Then return list vb
  Else << x := car v;
          v := cdr v;
          return rappend( !.b_perms( v, (x . T) . vb),
                          !.b_perms( v, (x . Nil) . vb))
       >>
End;


Symbolic Procedure rappend( l1, l2);
 << While l1
    Do << l2 := car l1 . l2;
          l1 := cdr l1
       >>;
    l2
 >>;



Symbolic Procedure b_bind( exp, vb);
Begin
  Scalar x, y, z, l;
  If numberp exp
  Then If x := atsoc( exp, vb)
       Then return cdr x
       Else return exp
  Else If atom exp Then return exp
  Else << y := car exp;
          While ( exp := cdr exp)
          Do l := !.b_bind( car exp, vb) . l;
          If y EQ 'not
          Then If Not (z := car l)
                  OR z EQ T
               Then return not z
```

```
            Else return list( 'not, z)
        Else If y EQ 'or
        Then << While 1
                Do If (z := car 1) EQ T
                   Then << 1 := nil; exp := T >>
                   Else << If z Then exp := z . exp;
                           1 := cdr 1
                        >>;
                   If exp EQ T OR Not exp Then return exp
                   Else return( y . exp)
             >>
        Else If y EQ 'and
        Then << While 1
                Do If Not (z := car 1)
                   Then << 1 := nil; exp := 'F >>
                   Else << If z NEQ T Then exp := z . exp;
                           1 := cdr 1
                        >>;
                   If exp EQ 'F OR Not exp Then return not exp
                   Else return( y . exp)
             >>
        Else errp_t("Bad expression")
    >>
End;
```

Bibliography

[AC86] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews Of Computer Science*, Annual Reviews, Inc, 1986.

[AH88] G. S. Almasi and S. L. Harvey. Rp3. In *Design and Application of Parallel Digital Processors*, April 1988.

[AI86] Arvind and R. A. Ianucci. *Two Fundamental Issues in Multiprocessing.* Computation Structures Group Memo 226-5, M.I.T., 1986.

[amd88] *Am29000 Streamlined Instruction Processor, User's manual.* American Micro Devices, 1988.

[arm87] *VL86C010 RISC Family Data Manual.* 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[CJMN80] T. J. W. Clarke, Gladstone P. J., C. D. Maclean, and A. C. Norman. Skim - the s, k, i reduction machine. In *Proc. 1980 ACM LISP conference*, 1980.

[Cla84] T. J. W. Clarke. Hardware support for fast combinator reduction. July 1984. Diploma Dissertation, Univ. Cambridge.

[Coh81] Jaques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3), 1981.

[Den80] J. B. Dennis. Dataflow architectures. *Computing*, November 1980.

[FW87] J. Fairburn and S. Wray. Tim: a simple abstract machine to execute supercombinators. In *Proc. Functional Programming Languages and Computer Architectures 1987, Oregon, USA.*, pages 34–45, Springer Verlag, 1987.

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems. Research Reports and Notes*, MIT Press, 1985.

[GLV84] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdes. Massive memory machine. *IEEE Trans. Comp*, C-33(5), May 1984.

[Gur88] J. R. Gurd. A taxonomy of parallel computer architectures. In *Design and Application of Parallel Digital Processors*, April 1988.

[Hal81] R. Halstead. Architecture of a myriaprocessor. In *IEEE COMPCON*, pages 299–302, Feb 1981.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent computation. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[Hen82] P. Henderson. *Purely Functional Operating Systems*, pages 177–189. Cambridge University Press, 1982.

[Hoa] C.A.R. Hoare. *OCCAM Programming Manual.* INMOS Limited. Published by Prentice-Hall.

[Hoa85] C. A. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[HS85] P. Hudak and L. Smith. *Para-functional Programming – a Paradigm for Programming Multiprocessor Systems.* Technical Report, Comp. Sci. Dept., Yale University, 1985.

[Ian88] Robert Alan Ianucci. *A Dataflow / von Neumann Hybrid Architecture.* Technical Report MIT/LCS/TR-418, MIT, May 1988.

[JAF88] J.A.Fisher. Replacing hardware that thinks (especially about parallelism) with a very smart compiler. In *Design and Application of Parallel Digital Processors*, April 1988.

[JHP79] J.H.Patel. Processor-memory interconnections for multiprocessors. In *6th Annual Symposium on Computer Architecture*, 1979.

[Kat85] M. G. H. Katavenis. *Reduced Instruction Set Computer Architectures For VLSI. ACM Doctoral Dissertations*, MIT Press, 1985.

[LH86] Kai Li and P. Hudak. A new list compaction method. *Software—Practice and Experience*, 16(2):145–163, Feb 1986.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[mc888] *MC88000 Data Book.* 1988.

[MDF*87] M.D.Cripps, J. Darlington, A.J. Field, P. G. Harrison, and M. J. Reeve. *The Design and Implementation of ALICE: A Parallel Graph Reduction Machine*, pages 300–326. IEEE Computer Press, 1987.

[MP86] L. F. Monteiro and F. C. Pereira. A sheaf theoretic model of concurrency. In *Proc. Logic in Computer Science Symposium, 1960, Mass.*, June 1986.

[PCSH87] S. L. Peyton-Jones, C. Clack, J. Salkild, and M. Hardie. A high performance architecture for parallel graph reduction. In *Proc. Functional Programming Languages and Computer Architectures 1987, Oregon, USA.*, pages 98–112, Springer Verlag, 1987.

[RS87] C. A. Ruggiero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In *Proc. Functional Programming Languages and Compuater Architecture 87*, Springer-Verlag, 1987.

[Sab88] Gary W. Sabot. *The Paralation Model: Architecture Independent Parallel Programming.* The MIT Press, 1988.

[SCN84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Proc. 1984 ACM LISP Conference*, 1984.

[Smi78] B. J. Smith. A pipelined, shared resource, mimd computer. In *Proc. International Conference on Parallel Processing*, 1978.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.

[SUN] *Sun 4 Technical Documentation.*

[TS83] Y. Tamir and C. Sequin. Strategies for managing the register file in risc. *IEEE Transactions on Computers*, November 1983.

[Ung84] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *SIGPLAN notices (Proc ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, 19(2), 1984.

[Wil] Roger Wilson. Private communication.

[WW87] Paul Watson and Ian Watson. Evaluating fuctional programs on the flagship machine. In *Proc. Functional Programming Languages and Computer Architectures 1987, Oregon, USA.*, pages 81–97, Springer Verlag, 1987.