Using Futurebus in a fifth-generation computer

Simon Peyton Jones describes his design approach to a parallel fifth-generation computer, emphasizing its bus-based architecture

Despite the bandwidth limitations of a bus, a design is presented for a parallel computer (GRIP) based on Futurebus, which limits bus bandwidth requirements by using intelligent memories. Such a machine offers higher performance than a uniprocessor and lower cost than a more extensible multiprocessor, as well as serving as a vehicle for research into parallel architectures.

microsystems **fifth generation** **graph reduction** **Futurebus**

In the future, computer programs will be executed by a large number of processors cooperating in a common task. The design and programming of such parallel machines is a major challenge, and is one of the principal objectives of the fifth-generation computer systems programmes which have sprung up in Japan, the USA and Europe.

The purpose of this paper is to present the architecture of a parallel fifth-generation computer called GRIP (graph reduction in parallel) based around a bus architecture. The main focus is on the hardware aspects of the design, and on the bus in particular, but the paper commences with a discussion of the challenges presented by a parallel architecture. This is followed by a brief overview of functional languages; this is the kind of language which GRIP is specifically designed to execute. Some of the options open to the designer of a parallel machine are discussed. Attention is then focussed onto GRIP and the bus chosen to implement it.

THE CHALLENGE OF PARALLELISM

Cooperation is expensive, yet it is the only way to get large tasks done quickly. This lesson is well illustrated by human organizations. Undoubtedly the most efficient way to get a task done is to assign a single individual to the task. There comes a time, however, when the sheer volume of work is more than a single individual can carry out in the required period of time, so he/she employs assistants to help. Inevitably the assistants must be told what to do and how to do it, and a proportion of the time of all concerned is spent in internal communication rather than in doing profitable work.

As the company grows, the overheads of internal communication can become very burdensome. The amount of internally generated information grows with the company, but each individual's capacity to digest this information remains fixed. The solution is to partition the work of the company in such a way as to reduce the amount of interaction required between workers, so that they can spend more of their time on profitable work and less on internal communication. This may be easy if the company is engaged in a number of essentially independent activities, but it can be very difficult if the company's activities are highly interrelated.

A primary challenge facing computer architects is the effective exploitation of parallelism. Raw processing power is now cheap, through replication of silicon, but mechanisms for connecting processors together so that they cooperate to achieve a common goal is very difficult. Inextricably connected with this challenge is the challenge of programming a parallel machine, and of partitioning the program in a way that minimizes communication.

In specific application areas it may be fairly easy to partition the problem so as to minimize internal communication. For example, in a multiuser Unix machine it is easy to assign a processor to each process awaiting execution. Less trivially, vector processors such as the Cray-1, or array processors such as the ICL DAP, have an arrangement of processing elements specifically adapted for the efficient execution of vector or array structured problems.

Programming vector or array processors is, however, a highly skilled and somewhat arcane art. In order fully to exploit the parallelism of the machine the programmer needs an intimate understanding of its workings and of the workings of the compiler. The investment required to produce such programs is very large — several of them represent 10 man years of work or more — and small program modifications risk destroying their finely balanced optimizations. Furthermore, such programs are often extremely complex, not because the task is complex, but in order to exploit the architecture most effectively.

Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK

0141-9331/86/02069-08 $03.00

An alternative approach is to have a number of processing elements connected together with some kind of network, each independently executing its own program: an MIMD (multiple instruction, multiple data) machine. Such a machine is relatively easy to build, but gives no clues about how best to program it. The problem of dividing the task up into concurrent subtasks, programming these subtasks in a sequential language and arranging the intertask communication is left entirely to the programmer. Even when the program is written it is hard to be sure that it is correct, and concurrency gives much scope for transient and irreproducible bugs which only occur under particular circumstances.

The challenge, then, is to produce a parallel programming *system*, including both architecture and a programming methodology, which

- is feasible to program; this is the overriding consideration
- is highly concurrent; this allows the developer to buy speed with raw processing power
- minimizes internal communication

FUNCTIONAL LANGUAGES AND GRAPH REDUCTION

Functional languages are fifth-generation programming languages which offer a powerful lever on programming parallel machines. The purpose of this section of the paper is to give a brief overview of where parallelism comes from, to set the rest of the paper in context. For full details of graph reduction, refer to Peyton Jones[1].

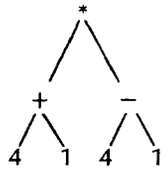
Programs written in a functional language can contain implicit parallelism. No new language constructs are required to write a parallel program, and the correctness of a program is no harder to establish than for a sequential functional program. This is in strong contrast with conventional procedural languages such as ADA, which require extra constructs to support parallelism, and where the parallelism has to be completely explicit.

To understand where the parallelism comes from, consider the following functional program

```
let f  x = (x + 1)* (x − 1)
in f  4
```

The 'let' defines a function *f* of a single argument *x*, which computes (x +1) * (x −1). The program executes by evaluating 'f 4', ie the function *f* applied to 4. We can think of the program as its parse tree

where the @ stands for function application. Applying *f* to 4 produces the expression (4 + 1) * (4 − 1), ie



We may now execute the addition and the subtraction simultaneously, giving

Finally the multiplication can be executed, to give the result

15

From this simple example it may be seen that

- Executing a functional program consists of evaluating an expression.
- A functional program has a natural representation as a tree or, more generally, a graph (in the sense of a network rather than a $y = mx + c$ graph).
- Evaluation proceeds by means of a sequence of simple steps, called reductions. Each reduction performs a local transformation of the graph (hence the term 'graph reduction').
- Reductions may safely take place simultaneously at different sites in the graph, since they cannot interfere with each other.
- All communication between processors performing concurrent reductions is implicit, mediated by the graph. No explicit communication between processors is required.
- Evaluation is complete when there are no further reducible expressions.

Graph reduction provides us with a simple and powerful execution model that can form the basis of a parallel implementation. The GRIP multiprocessor is designed to execute functional programs by performing graph reductions concurrently, exactly as described above. Despite the opportunities for parallelism offered by functional languages, only the ALICE project (at Imperial College, London, UK) has so far attempted a parallel implementation in hardware[2].

The smallest unit of concurrent computation is a single reduction, but this is a rather fine grain of parallelism, and there is a danger that the overheads of administering such small units will dominate the execution costs. To avoid this it is desirable (and possible) to choose certain reduction sequences to be the unit of concurrent computation, thus moving towards a coarser grain. The details of the generation, administration, execution and synchronization of concurrent computation are, however, beyond the scope of this paper. (The final chapter of Peyton Jones[1] could serve as a starting point for further reading.)

ARCHITECTURES FOR PARALLEL GRAPH REDUCTION

In this section we will discuss a range of possible architectures for a parallel graph reduction machine. This will set the scene for presentation of the GRIP architecture.

Almost any parallel reduction machine may be thought of as a variation of the scheme shown in Figure 1. The processing elements (PEs) are more or less conventional von Neumann processors, including some private memory.

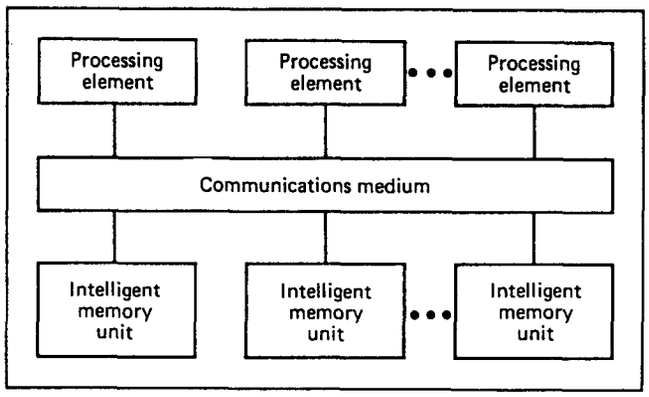


Figure 1. Physical structure of a parallel graph reduction machine

The intelligent memory units (IMUs) are the custodians of the graph, and may or may not also contain a processor.

This rather bland-looking diagram actually covers a huge spectrum of machine architectures. The two major axes along which variations are possible are: the intelligence of the IMUs; and the topology of the communications network.

Intelligence of the IMUs

The amount of intelligence contained in the IMUs has a radical effect on the architecture. The two extremes of the spectrum are as follows.

(1)The IMUs provide only the ability to perform read and write operations to the graph. All the intelligence required to perform reductions on the graph resides in the PEs. Such IMUs could be implemented with conventional memory boards, except that there would need to be some indivisible read-modify-write operation provided to assure mutual exclusion. This results in a classical tightly coupled system, where the graph is held in global memory and every access to the graph by a PE requires use of the communications medium.

(2)The IMUs each contain a sophisticated processor, sufficiently intelligent to perform graph reduction unaided. The PEs are now vestigial, since they have nothing left to do, and can be discarded altogether. This results in a collection of intelligent memories connected by a communications medium, which is a classical loosely coupled system (or distributed system). It is much cheaper for an IMU to access a graph node held in its own local memory than to use the communications medium to access remote nodes.

Between these two extremes there is a continuum of possible architectures. To move from one extreme to the other, imagine migrating functionality from the PEs into the IMUs. Moving along the spectrum, the IMUs become capable of more and more sophisticated functions. As will be seen later, GRIP occupies an intermediate point on the spectrum.

Communications medium

The purpose of the communications medium is to allow the PEs and IMUs to communicate with each other. A large amount of research has been done on optimal network topologies for various sorts of computer architectures[3,4], and this paper will do no more than lay out the major design issues, giving appropriate references. We will refer to a PE or an IMU as a box.

Before discussing possible network topologies, the characteristics sought after in the communications medium must first be established. Among these are

- cost — how much the network costs per box
- latency — how long it takes for a transaction to get from sender to recipient through the network
- bandwidth — how many transactions the network can handle simultaneously
- locality — whether or not it is essential that most transactions are local (ie between 'nearby' boxes)
- extensibility — whether the network can be expanded arbitrarily to accommodate more boxes

The sort of networks that are of interest break down into four main types, to be discussed according to the criteria given above.

Bus

A bus is the simplest form of interconnection; in a bus all the PEs and IMUs are connected to a common information highway (Figure 2a). Its cost is linear in the number of boxes, and its latency is roughly independent of the number of boxes. However, only one transaction can take place at a time over a bus, so its bandwidth is fixed and therefore places a fundamental limit on the extensibility of the bus communications medium. (The machine could, however, be extended by connecting together a number of independent buses through gateways, which would produce a hybrid of a bus system and a fully distributed system.) Locality is irrelevant since all boxes are equally accessible.

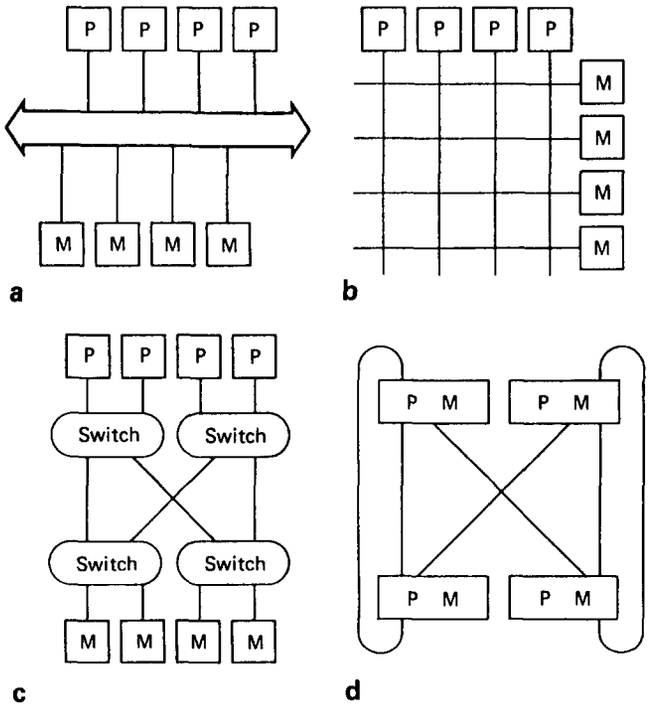


Figure 2. Types of interconnection: a, bus; b, fully connected network; c, partially connected network; d, fully distributed network

Fully connected networks

To avoid the bandwidth limitations of a bus, a crossbar switch (Figure 2b), in which simultaneous connections between any set of pairs of boxes are possible[5], may be considered.

For small numbers of boxes, this arrangement gives constant latency (although the latency becomes linear in the number of boxes for large systems, because the wire length becomes dominant), and no locality is required. Its bandwidth rises linearly with the number of boxes, but unfortunately the number of switch connections rises quadratically with the number of boxes, so its cost is prohibitive.

In theory crossbars are extensible, but their cost means that they are only practicable with rather small numbers of connections (eg 16×16[6, 7]).

Partially connected networks

Limited-access partially connected networks are a compromise between buses and crossbars (Figure 2c). They have formed a particularly active focus of research work in recent years[3, 5, 8]. They are normally multistage networks (ie a message may go through a number of switching elements on its way to its destination) based on various permutation networks (banyan, perfect shuffle, shuffle exchange, omega networks etc).

They offer: a cost of $O(N\log N)$, where N is the number of boxes; a latency of $O(\log N)$; and a bandwidth of $O(N)$. This is a useful compromise between the extremes of buses and crossbars. It may not be possible to establish simultaneous paths between all pairs of boxes requesting one, so in general the latency is not bounded. The probability distribution of latency drops off very sharply, however, and the chances of long latency are very low. They can be reduced further by adding redundant paths; this also gives the possibility of fault tolerance. Again, no locality is required, and these networks are completely extensible in a modular way.

Networks of this sort are used in many machines, such as the NYU Ultracomputer[9], Cedar[10], TRAC[11], the BBN Butterfly[12] and ALICE[13].

Fully distributed networks

The last alternative seems more radical, though it relates closely to the previous one. A fully distributed network provides communications paths from each box to only a limited number of neighbouring boxes (Figure 2d). A box may communicate with a non-neighbouring box only through an appropriate neighbour. The work cited in the previous section is also applicable here, since we can obtain a distributed network from a multistage network by replacing the switching elements with IMUs (which must now also handle the forwarding of messages destined for other boxes).

Some boxes are therefore 'closer' than other boxes, so the latency will be nonuniform; to obtain good performance from such networks it is essential that most transactions take place between nearby boxes. The average latency is $O(N\log N)$, but the constant of proportionality is often considerably greater than for multistage network switching elements. Though the latency is nonuniform, the network can be made isotropic (or homogeneous) in the sense that the network looks the same when viewed from any box (the network has no edges).

The cost of such a distributed network is linear in the number of boxes, since each box normally has a fixed number of links to neighbours. The bandwidth also rises linearly. The networks are, of course, readily extensible.

Burton and Sleep have long advocated such networks as a basis for a reduction machine[14, 15]. Recently a number of multiprocessors have become available — some commercially — based on cube-connected cycle networks, eg the CalTech Cosmic Cube[16] and the Intel Cube.

Summary

Table 1 summarizes our conclusions. It is easy to see the reason for the popularity of partially connected networks.

Locality

From Table 1 it is clear that if we could achieve locality then the way would be opened to cheap and extensible fully distributed communications networks. This is no mean feat, however.

The idea of locality is well established in conventional computer architecture. It is an observed property of most programs that they tend to reference data which has either been referenced in the recent past (temporal locality), or which is physically adjacent to recently referenced data (spatial locality)[17]. A conventional cache relies on locality of reference to hold the data in active use in fast memory close to the processor[18].

Functional programs are not so well behaved, since the physical adjacency of two cells bears no relation to their logical adjacency. Nevertheless, some sort of spatial locality is essential for a distributed loosely coupled network; otherwise the processors will spend all their time waiting to communicate rather than computing.

The analogy with a commercial company is again useful. The organization of the company is intended to enable workers to perform their tasks using mostly local communication — within an office, for example. An excessive proportion of nonlocal communication generally indicates an inefficiently organized company.

Locality is a heuristic property of programs, and the best we can hope to do is to develop effective heuristics. This is at present an area more of speculation than experiment, though some simulations have been performed[19, 20].

The challenge of achieving locality in the execution of functional programs is at present an unsolved problem. Nevertheless, some researchers have chosen to confront the problem and build distributed machines requiring

Table 1. Comparison of interconnection networks

Type	Cost	Bandwidth	Latency	Locality essential?	Extensible?
Bus	N	1	1	No	No
Crossbar	N^2	N	N	No	Yes
Partially connected	$N\log N$	N	$\log N$	No	Yes
Distributed	N	N	$\log N$	Yes	Yes

locality[21, 22]. The ALICE project has chosen to use an extensible multistage partially connected network[13].

For the GRIP project we have chosen to avoid the problems of locality and the cost of a partially connected network by using a bus architecture. GRIP's architecture is discussed in more detail in the following section.

GRIP — A FUTUREBUS ARCHITECTURE

GRIP is intended to provide state-of-the-art performance at moderate cost by extracting maximum performance from a fast bus. This means that, within its performance range, GRIP should provide more power per unit cost than more extensible designs (such as ALICE). Our performance target for a fully populated GRIP is one million reductions per second.

We have chosen to implement the communications network using a fast bus, the IEEE P896 Futurebus. A bus offers a low-cost interconnection, but with the limitation that only one transaction can take place between a PE and an IMU at once, thus limiting concurrency. This places a fundamental limit on the parallelism achievable using GRIP, but it gives an extremely cost effective solution up to this limit.

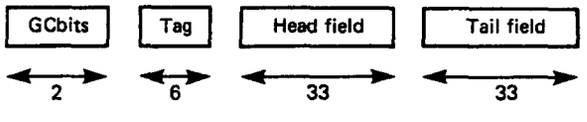
The decision to use a bus was a hard one to take because of the inherent inextensibility of the design. However, the use of a bus allows us to address one research issue — parallel reduction — at a time, rather than to try to solve several difficult problems (locality, in particular) at once. By using a bus we therefore expect to produce a machine of higher performance than a uniprocessor, and lower cost than an extensible multiprocessor.

In the case of GRIP we anticipate being able to integrate up to 120 PEs or so, on 30 boards, before running out of bus bandwidth and physical space.

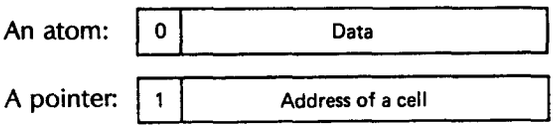
Intelligent memory units

GRIP's IMUs will consist of 1.25 Mbyte of RAM arranged in 40-bit words, with a fairly simple bit-slice microprogrammable processor on the front. Instead of supporting just read and write commands as normal memories do, the IMUs will support an instruction set of high-level operations, chosen to support parallel graph reduction. These operations are the unit of indivisibility supported by GRIP. (All concurrent machines must provide some indivisible operations to assure correct synchronization of parallel activities.) In addition the use of high-level operations reduces the bus bandwidth required to communicate with the IMUs.

The instruction set supported by the IMUs implements a view of memory as an unordered collection of cells. Each cell consists of a pair of 40-bit words, and is further structured as follows.



A cell contains two major fields, the head and tail. In addition, the tag field identifies the type of the cell, and the GC bits assist with garbage collection. The head and tail fields are structured as follows.



Each can be an atomic data item (identified by a zero in the top bit) or a pointer to another cell (identified by a one). Later versions of the IMUs will support variable-length cells.

In addition to managing the cell storage, the IMUs will also be responsible for managing task queues and certain parts of the garbage collection activity. The design of the IMUs will be one of the major technical hardware challenges of the project.

Processing elements

The PEs are autonomous units responsible for performing reductions on the graph held in the IMUs. They will be of straightforward design, based around a microprocessor (eg Transputer, MC68020, NS32332), and will include their own private memory which is inaccessible to the rest of the system. The processor within a PE executes a program held in local memory.

A subsequent version of GRIP might replace a collection of microprocessor-based PEs with a single bit-slice PE, taking up the same board area, but whose architecture was more closely matched to the task of graph reduction. This is the approach taken by the SKIM uniprocessor[23].

Integrating PEs and IMUs

The conceptual architecture calls for distinct PEs and IMUs to plug into the common bus. In fact we propose to use a single board which contains some PEs and an IMU, interfaced to the bus with a common bus interface processor (BIP). The PE and IMU are still logically quite distinct and would operate concurrently on different internal buses. However, transactions which the PE addresses to the IMU that happens to be on its board will be 'short circuited' through, bypassing the BIP altogether.

Such an amalgamation gives an obvious advantage in that only one board and one BIP need be designed, making replication easier. However, it appears to suffer from the disadvantage that the ratio between PEs and IMUs cannot be changed to 'mix and match'.

The argument to dispel this apparition is as follows. Standard Futurebus boards can be up to treble Eurocard size. This is far bigger than a PE would need — a PE consists only of a microprocessor, a couple of ROMs and some local RAM. It is also bigger than an IMU would need. A 40-chip memory array is the dominant space consumer, and this fits on less than a third of a treble Eurocard. In other words, we could fit several PEs and a respectable IMU on a single board. Other things being equal, this would be far preferable to having large amounts of empty and hence unproductive board area (on the PEs) plugged into the scarcest resource — the bus slots.

Many small IMUs are preferable to a few large ones, to gain maximum parallelism. Amalgamating an IMU with a

PE allows n IMUs in an n-slot bus, which is clearly the optimum situation. Amalgamation also allows maximization of the number of PEs, which is desirable because we want to have enough PEs to saturate the bus. PEs can always be depopulated to reduce bus loading (and cost). Looked at another way, putting a PE on the same board as an IMU allows up to $2n$ concurrent activities instead of only n (or more if there is more than one PE on a board).

As a corollary of amalgamation, it is now relatively easy to allow IMUs to become bus masters as well as slaves. This is useful for packet switching operations (discussed in the next subsection). Another spin off is that is should be possible to exploit to some degree the local access possible from a PE to its fellow IMU. No attempt would initially be made to exploit this locality.

In practice we hope to integrate four PEs and one IMU on each board.

Packet switching

Conventional microprocessor buses use circuit switching techniques. For instance, a memory read operation opens a circuit over the bus to memory that is held open during the entire read cycle, precluding any other unit from using the bus. During the memory access period the bus is inactive, and hence is used inefficiently.

Furthermore, at least some of the high-level memory operations supported by the IMUs will be considerably slower than even a memory read, worsening the inefficiency from circuit switching.

It is well known that packet switching can offer substantially better performance in cases where circuit switching makes inefficient use of the medium. In a packet switching regime a PE initiates a memory operation by writing a transaction request into a transaction buffer in the bus interface. The bus interface then acquires the bus (which might take some time), sends all pending transactions to the transaction buffers in the destination board(s) and relinquishes the bus. The request is then processed by the recipient IMU, which writes a reply transaction into its transaction buffer, whence it is returned to the initiating PE by the same mechanism.

This particular application of packet switching is sometimes known as split cycle bus operation. It makes more efficient use of the bus at the cost of increased overhead for each transaction, and increased latency. Because of these greater overheads we may want to circuit switch for some (fast) memory operations and packet switch for other (slow) ones.

Achieving an implementation of a packet switching protocol without imposing substantial latency on PE–IMU transactions is another of the major hardware challenges of the project.

IEEE P896 FUTUREBUS

Once the decision to use a bus was taken, we looked for a bus with the following characteristics

- high bandwidth, to alleviate the bottleneck as far as possible
- true multimaster capability, to reflect in the hardware the symmetry between the PEs; the arbitration time should be very short, to allow rapid changes of bus mastership
- the ability to support high-level PE–IMU transactions, not simply read and write
- the ability to support packet switched as well as circuit switched protocols

We chose to use the IEEE P896 Futurebus to meet these objectives, for the following reasons.

- It is the fastest bus available to us. Special bus drivers and a novel backplane design ensure that no 'bus settling time' delays are required.
- It provides an ingenious fully distributed fair arbitration scheme with rapid settling time. Furthermore, arbitration can take place concurrently with data transfer, and so will often be complete by the time the bus is relinquished.
- Its protocols are powerful enough to support a mixture of packet and circuit switching. These protocols are tightly defined and debugged, thus saving the designers a lot of work.
- It can accommodate packet switched as well as circuit switched protocols. In addition, it offers broadcast and broadcall protocols, which turn out to be useful for system synchronization.
- It defines standard racks, backplanes and drivers. Due to careful design the drivers are relatively low powered, and eight of them are available in a single 0.3 in (7.6 mm) package; this gives a high packing density, and allows the drivers to be close to the connector pins.

Overview of Futurebus

This subsection comprises a brief overview of the Futurebus, seen from the standpoint of our machine. For more detail, a special issue of *IEEE Micro*[24] contains several articles about the bus design.

P896 is an asynchronous bus, using a 32-bit multiplexed address and data path. It incorporates a fair distributed bus arbiter for allocating bus access. When transferring multiple data items, it is capable of particularly fast operation by using a two-cycle handshake.

The P896 bus consists of: the address/data bus (34 bit wide); the command bus (5 bit wide, from master to slave); the status bus (3 bit wide, from slave to master); and the arbitration bus (11 bit wide — 7 bit device number, 3 bit + 1 bit control). We expect that a fully loaded bus will be capable of a transfer cycle time of 60–80 ns. Most transactions would transfer data as well as address, thus taking at least two cycles.

Synchronous *vs* asynchronous bus

Certain conventional wisdom has it that a synchronous bus is faster than an asynchronous one, since there are no timing interfaces. A timing interface manages the interaction between two systems running with different clocks, and is a great time waster, since the transmitting side of the interface has to wait for the receiving side's clock edge. Timing interfaces are also sources of latch metastability problems.

In practice, however, the fastest convenient clock speed for the bus is most unlikely to be the same as that

for the processors, or the IMUs. This means that synchronous systems often have two timing interfaces. This makes an asynchronous system, which has only one timing interface, a faster bet if it is properly designed.

Asynchronous systems can also exploit actual-case gate and bus propagation times, rather than being tied to the worst case as synchronous systems are.

In order fully to exploit these speed advantages we plan to use an asynchronous finite state machine to control the transfer of data between transaction buffers across the bus.

Cache consistency protocols

Considerable effort has been expended in the Futurebus specification to develop protocols for maintaining multiple consistent caches in each processor, to avoid excessive bus use. These protocols rely on each cache manager 'snooping' on the bus to spot when a data item it holds has been altered by another processor.

Curiously, perhaps, we do not intend to use any of this work in GRIP. There are two reasons why it is not useful to us. First, it is closely tied to a simple read/write memory operation model, and would have difficulty in accommodating the more sophisticated instruction set supported by our IMUs.

Second, once a piece of graph is fully evaluated it can never alter. Hence it can be freely cached by any PE. Conversely, before it is fully evaluated only one PE should be given access to it, since otherwise several PEs would try to evaluate the same graph. Hence the situation in which several PEs might write to a location never arises.

CONCLUSIONS AND PROJECT STATUS

The use of a fast bus has enabled us to exploit a price/performance 'window' in parallel machine design. GRIP will be faster than a uniprocessor, and cheaper than a more extensible multiprocessor. More importantly, it will enable us to address many of the issues of parallel system design without requiring solution of all the problems at once.

The GRIP project is funded by the UK's Alvey directorate as a collaborative project between University College London (UCL), ICL, High Level Hardware Ltd and Research Software Ltd. Three full-time research assistants form the main team based at UCL, and work began in the late autumn of 1985.

We expect to have a working prototype after two years, and to spend the third year of the project tuning the design.

Simon Peyton Jones is a lecturer in computer science at University College London, UK. After graduating from Trinity College, Cambridge, UK, he worked in industrial computing for two years before taking up his present post. His main research interest is in functional programming languages and their implementation, and he is currently leading a team in an Alvey-funded project to design and build a high-performance parallel graph reduction machine called GRIP. He is writing a book[1] about the implementation of functional languages using graph reduction, due for publication in 1986.

REFERENCES

1 **Peyton Jones, S L** *The implementation of functional languages* Prentice Hall, Englewood Cliffs, NJ, USA (to be published)

2 **Darlington, J and Reeve, M** 'ALICE — a multiprocessor reduction machine for the parallel evaluation of applicative languages' *Proc. ACM Conf. Functional Programming Languages and Computer Architecture, NH, USA* (October 1981) pp 65–75

3 **Feng, T** 'A survey of interconnection networks' *IEEE Comput.* (December 1981) pp 12–27

4 **Thurber, K J and Masson, G M** *Distributed processor communication architecture* Gower, Farnborough, UK (1979)

5 **Broomell, G and Heath, J R** 'Classification categories and historical development of circuit switching topologies' *ACM Comput. Surv.* Vol 15 No 2 (June 1983) pp 95–134

6 **Wulf, W and Bell, C** 'C.mmmp — multi-miniprocessor' *AFIPS Proc. (FJCC)* Vol 41 No 2 (1972) pp 765–777

7 **Farmwald, P M** 'The S-1 Mark IIa supercomputer' in **Kowalik** (Ed.) *High speed computations* Springer Verlag, Berlin, FRG (1984)

8 **Siegel, H J** 'Interconnection networks for SIMD machines' *IEEE Comput.* (June 1979) pp 57–65

9 **Gottlieb, A *et al*** 'The NYU Ultracomputer — designing a MIMD shared memory parallel computer' *IEEE Trans. Comput.* Vol 32 No 2 (February 1983) pp 175–189

10 **Gajski, D *et al*** 'Cedar' *Proc. Compcon* (1984) pp 306–309

11 **Brown, J C** 'TRAC — an environment for parallel computing' *Proc. Compcon* (1984) pp 294–298

12 *Butterfly parallel processor overview* Bolt Beranek Newman (BBN) Laboratories, USA (June 1985)

13 **Cripps, M D and Field, A J** *An asynchronous structure-independent switching system with system-level fault tolerance* Dept of Computer Science, Imperial College, London, UK (1983)

14 **Burton, F W and Sleep, R** 'The zero assignment parallel processor (ZAPP) project' *Proc. Symp. Functional Languages and Computer Architecture, Goteborg, Sweden* (June 1981)

15 **Burton, F W and Sleep, R** 'Executing functional programs on a virtual tree of processors' *Proc. ACM Symp. Functional Languages and Computer Architecture, Portsmouth, UK* (October 1981)

16 **Seitz, C L** 'The cosmic cube' *CACM* Vol 28 No 1 (January 1985) pp 22–33

17 **Denning, P J** 'On modelling program behaviour' *Proc. Spring Joint Computer Conf. 40* AFIPS Press, USA (1972) pp 937–944

18 **Smith, A J** 'Cache memories' *ACM Comput. Surv.* Vol 14 No 3 (September 1982) pp 473–530
19 **Keller, R M and Lin, F C H** 'Simulated performance of a reduction based multiprocessor' *IEEE Comput.* Vol 17 No 7 (July 1984) pp 70–82
20 **Hudak, P and Goldberg, B** 'Distributed execution of functional programs using serial combinators' *IEEE Trans. Comput.* (September 1985)
21 **Hudak, P** *Functional programming on multiprocessor architectures — a survey of research in progress* Dept of Computer Science, Yale University, Newhaven, CT, USA (November 1985)
22 **Keller, R M** 'Rediflow architecture prospectus' *TR UUCS-85-105* Dept of Computer Science, University of Utah, USA (August 1985)
23 **Clarke, T J W, Gladstone, P J S, Maclean, C D and Norman A C** 'SKIM — the S K I reduction machine' *Proc. ACM Lisp Conf., Stanford, CA, USA* (1980)
24 Special issue on bus standards *IEEE Micro* Vol 4 No 4 (August 1984)