PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

The Influence of VLSI Technology on Computer Architecture [and Discussion]

D. May and S. F. Reddaway

Phil. Trans. R. Soc. Lond. A 1988 **326**, 377-393
doi: 10.1098/rsta.1988.0094

Phil. Trans. R. Soc. Lond. A **326**, 377–393 (1988) [377]
Printed in Great Britain

The influence of VLSI technology on computer architecture

By D. May
INMOS Limited, 100 Azbec West, Almondsbury, Bristol BS12 4SQ, U.K.

Very-large-scale integration (VLSI) offers new opportunities in computer architecture. The cost of a processor has been reduced to that of a few thousand bytes of memory, with the result that parallel computers can be constructed as easily and economically as their sequential predecessors. In particular, a parallel computer constructed by replication of a standard computing element is well suited to the mass-production economics of the technology.

The emergence of the new parallel computers has stimulated the development of new programming languages and algorithms. One example is the OCCAM language which has been designed to enable applications to be expressed in a form suitable for execution on a variety of parallel architectures. Further developments in language and architecture will enable processing resources to be allocated and deallocated as freely as memory, giving rise to some hope that users of general-purpose parallel computers will be freed from the current need to design algorithms to suit specific architectures.

1. The technology

Current technology allows a high-performance computer to be implemented on a single chip. One example is the Inmos IMS T800 transputer (Homewood *et al.* 1987), which incorporates two processors (central processor and floating-point processor), 4 Kbytes of RAM (random access read–write memory) and a communications system on a single chip of about 1 cm^2. The floating-point performance of this device is over 1 megaflop (million floating point operations per second).

For the T800 and its some of its major functional blocks, the dimensions in square millimetres are:

T800, 93.0; central processor, 12.0; communications link, 2.5; floating point unit, 17.5; 4 Kbytes memory, 12.0; arithmetic logic unit, 0.6; 70 Kbits microcode, 4.5.

Semiconductor manufacturing processes evolve very rapidly, and within a year it will be possible to manufacture devices like the IMS T800 in less than half the area. In practice, however, the improvements in the manufacturing process are used to increase the capability of the device.

Like all VLSI devices, a transputer is ultimately assembled from connections in metal and polysilicon, transistor and 'contacts' which connect metal to polysilicon. The IMS T800 is manufactured with a complementary metal oxide silicon (CMOS) process similar to that used in the manufacture of 64 Kbit static RAMs. For the transputer manufacturing process, the dimensions of transistors and contacts in square micrometres are: transistor, 14; contact, 36. Metal tracks are 2.75 μm wide, polysilicon tracks 1.75 μm. However, they cannot be placed close together; buses are formed by placing metal (or polysilicon) tracks on a pitch of 6 μm. It can be seen from these figures that the majority of the silicon area is used for interconnect.

Transistors are the smallest components, for example, 13700 transistors occupy the same area as 1 mm of a 32-bit-wide bus. The cost of interconnection off the chip is large, a 'pad' to which a wire is bonded in the final stages of manufacturing occupies 0.125 mm^2 and so to transmit a 32-bit-wide bus off-chip requires 4 mm^2 of pads.

It can be seen from the above figures that interconnections consume a relatively large silicon area. Consequently, considerable effort is being expended to reduce the cost of interconnect in new manufacturing processes. This can be done with smaller geometries or by adding extra layers of interconnect (or both). There are now a number of processes with two metal interconnect levels, and development of three- and four-layer processes is in hand. New processes will offer substantially smaller geometries in both metal width and pitch. However, reducing the metal width increases its resistivity in proportion, and reducing the pitch increases the capacitance with the result that the interconnect becomes slower but the transistors become faster. The speed can, to some extent, be increased but only at the cost of more power consumption and larger signal drivers occupying more area.

2. VLSI architecture

VLSI technology has significant implications for computer architecture:

(1) processing is cheap; extra processor power should be used to save interconnect and to save memory;

(2) locality of operation is essential.

For example, the fact that the processor occupies the same space as a 4 Kbyte memory suggests that, provided that performance is maintained, it is worth using a processor wherever it will save more than 4 Kbytes of memory (for example, by compressing data or by computing values when needed instead of storing them). Similarly, it is often worth encoding data and transmitting it serially rather than transmitting it in parallel.

The need for locality arises because of the high cost in area and delay of moving data between chips. However, locality is also becoming important even within a single chip. For the manufacturing process described above, a transistor can switch in about the same time as a signal will travel along 1 mm of metal. It is therefore important that the various functional units of a processor (registers, arithmetic logic units and control logic) are physically close together so that data can be moved rapidly between them. Perhaps more important is the need to distribute clock signals to synchronize the various components of the processor. As mentioned above, the effect of reducing the dimensions of the manufacturing process is to further decrease the signal speed in relation to the switching speed. The inevitable conclusion is that an architecture based on a network of simple computers operating independently and communicating data as needed is ideally matched to the technology. However, such a radical change of architecture cannot be hidden from the programmer; new languages and algorithms are needed to exploit such machines.

3. Occam

The occam programming language (Inmos 1988) enables an application to be described as a collection of processes which operate concurrently and communicate through channels. In such a description, each occam process describes the behaviour of one component of the implementation, and each channel describes a connection between components.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

The design of OCCAM allows the components and their connections to be implemented in many different ways. This allows the choice of implementation technique to be chosen to suit available technology, to optimize performance, or to minimize cost.

OCCAM has proved useful in many application areas. It can be efficiently implemented on almost any computer and is being used in applications ranging from single processor embedded control systems to concurrent supercomputers. OCCAM programs can also be directly compiled and implemented as specialized VLSI devices.

The main design objective of OCCAM was to provide a language which could be directly implemented by a network of processing elements, and could directly express concurrent algorithms. In many applications OCCAM is used as an assembly language; there is a one-to-one relation between OCCAM processes and processing elements, and between OCCAM channels and links between processing elements.

A further important objective in the design of OCCAM was to use the same concurrent programming techniques both for a single computer and for a network of computers. In practice, this meant that the choice of features in OCCAM was partly determined by the need for an efficient distributed implementation. Once this had been achieved, only simple modifications were needed to ensure an efficient implementation of concurrency on a single sequential computer. This approach to the design of OCCAM perhaps explains some of the differences between OCCAM and other 'concurrent' languages.

3.1. *Locality*

Almost every operation performed by a process involves access to a variable, and so it is desirable to provide each processing element with local memory in the same VLSI device.

The speed of communication between electronic devices is optimized by the use of one directional signal wires, each connecting only two devices. This provides local communication between pairs of devices.

OCCAM can express the locality of processing, in that each process has local variables; it can express locality of communication in that each channel connects only two processes.

3.2. *The OCCAM primitives*

OCCAM programs are built from three primitive processes:

`v := e`	assign expression *e* to variable *v*,
`c ! e`	output expression *e* to channel c,
`c ? v`	input variable *v* from channel c.

The primitive processes are combined to form constructs:

`SEQ`	sequence,
`IF`	conditional,
`WHILE`	loop,
`PAR`	parallel,
`ALT`	alternative.

A construct is itself a process, and may be used as a component of another construct; in other words, OCCAM is a hierarchical block-structured language.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential and conditional and loop constructs. The order of expression evaluation is unimportant, as there are no side effects and operators always yield a value.

Concurrent programs make use of channels, inputs and outputs, combined by parallel and alternative constructs.

The definition and use of occam procedures follows ALGOL-like scope rules, with channel, variable and value parameters. The body of an occam procedure may be any process, sequential or parallel.

A very simple example of an occam program is the buffer process below.

```
WHILE TRUE
  VAR ch:
  SEQ
    in ? ch
    out ! ch
```

Indentation is used to indicate program structure. The buffer consists of an endless loop, first setting the variable `ch` to a value from the channel `in`, and then outputting the value of `ch` to the channel `out`. The variable `ch` is declared by `VAR ch`.

3.3. *The parallel construct*

The components of a parallel construct may not share access to variables, and communicate only through channels. Each channel provides one-way communication between two components; one component may only output to the channel and the other may only input from it. These rules are checked by the compiler.

The parallel construct specifies that the component processes are 'executed together'. This means that the primitive components may be interleaved in any order. More formally,

```
PAR                SEQ
  SEQ       =        x := e
    x := e           PAR
    P                  P
  Q                    Q
```

so that the initial assignments of two concurrent processes may be executed in sequence until both processes start with an input or output. If one process starts with an input on channel `c`, and the other an output on the same channel `c`, communication takes place:

```
PAR                SEQ
  SEQ       =        x := e
    c ! e            PAR
    P                  P
  SEQ                  Q
    c ? x
    Q
```

The above rule states that communication can be thought of as a distributed assignment. Two examples of the parallel construct are shown below.

```
CHAN c:                VAR ch1:
PAR                    VAR ch2:
  WHILE TRUE           SEQ
    VAR ch:              in ? ch1
    SEQ                  WHILE TRUE
      in ? ch              SEQ
      c ! ch                 PAR
  WHILE TRUE                   in ? ch2
    VAR ch:                    out ! ch1
    SEQ                      PAR
      c ? ch                   in ? ch1
      out ! ch                 out ! ch2
```

The first consists of two concurrent versions of the previous example, joined by a channel to form a 'double buffer'. The second is perhaps a more conventional version. As 'black boxes', each with an input and an output channel, the behaviour of these two programs is identical; only their internals differ.

3.4. *Synchronized communication*

Synchronized, unbuffered, communication greatly simplifies programming, and can be efficiently implemented. In fact, it corresponds directly to the conventions of self-timed signalling (Mead & Conway 1980). Unbuffered communication eliminates the need for message buffers and queues. Synchronized communication prevents accidental loss of data arising from programming errors. In an unsynchronized scheme, failure to acknowledge data often results in a program which is sensitive to scheduling and timing effects.

Synchronized communication requires that one process must wait for the other. However, a process which requires to continue processing while communicating can easily be written:

```
PAR
  c ! x
  P
```

3.5. *The alternative construct*

In OCCAM programs, it is sometimes necessary for a process to input from any one of several other concurrent processes. Consequently, OCCAM includes an alternative construct similar to that of CSP (Hoare 1978). As in CSP, each component of the alternative starts with a guard; an input, possibly accompanied by a boolean expression. From an implementation point of view, the alternative has the advantage that it can be implemented either 'busily' by a channel test or by a 'non-busy' scheme. The alternative enjoys a number of useful semantic properties more fully discussed in Roscoe & Hoare (1986); in particular, the formal relation between parallel and alternative is shown below:

```
                    ALT
                      c ? x
                        PAR
PAR                       P
  SEQ                     SEQ
    c ? x                   d ? y
    P          =            Q
  SEQ                 d ? y
    d ? y               PAR
    Q                     Q
                          SEQ
                            c ? x
                            P
```

This equivalence states that if two concurrent processes are both ready to input (communicate) on different channels, then either input (communication) may be performed first.

A simple example of the alternative is shown below; this is a 'multiplexor':

```
WHILE TRUE
  ALT
    in1 ? ch
      out ! ch
    in2 ? ch
      out ! ch
```

The multiplexor inputs from either channel `in1` or channel `in2`. It then outputs the value it has just input.

3.6. *Channels and hierarchical decomposition*

An important feature of occam is the ability to successively decompose a process into concurrent component processes. This is the main reason for the use of named communication channels in occam. Once a named channel is established between two processes, neither process need have any knowledge of the internal details of the other. Indeed, the internal structure of each process can change during execution of the program.

The parallel construct, together with named channels provides for decomposition of an application into a hierarchy of communicating processes, enabling occam to be applied to large-scale applications.

3.7. *Future developments*

The initial version of occam was a very simple language including only integer data types and one-dimensional arrays. In view of the need to provide a concurrent language suitable for a wide range of numerical applications, a derivative of occam has been developed, known as occam2. The occam2 language provides the concurrency features of occam together with the data types appropriate for numerical applications.

Current implementations of occam assume explicit control over the physical allocation of processes to processors and channels to interprocessor links. Allocation of physical resources is determined by the compiler. This allows an extremely efficient implementation of concurrency

and communication. However, it would be natural to remove these restrictions, especially if further hardware support could be used to make dynamic resource allocation fast. The result would be a language supporting concurrency and recursion, together with recursively defined data types. This would make OCCAM more suitable for concurrent symbolic application areas.

4. The transputer concept

VLSI technology allows a large number of identical devices to be manufactured cheaply. For this reason, it is attractive to implement an OCCAM program with a number of identical components, each programmed with the appropriate OCCAM process. A transputer (see, for example Homewood *et al.* 1987) is such a component.

A transputer is a single VLSI device with memory, processor and communications links for direct connection to other transputers. Concurrent systems can be constructed from a collection of transputers which operate concurrently and communicate through links.

The transputer can therefore be used as a building block for concurrent processing systems, which OCCAM as the associated design formalism.

4.1. *Transputer architecture*

An important property of VLSI technology is that communication between devices is very much slower than communication on the same device. In a computer, almost every operation that the processor performs involves the use of memory. A transputer therefore includes both processor and memory in the same integrated-circuit device.

In any system constructed from integrated circuit devices, much of the physical bulk arises from connections between devices. The size of the package for an integrated circuit is determined more by the number of connection pins than by the size of the device itself. In addition, connections between devices provided by paths on a circuit board consume a considerable amount of space.

The speed of communication between electronic devices is optimized by the use of one-directional signal wires, each connecting two devices. If many devices are connected by a shared bus, electrical problems of driving the bus require that the speed is reduced. Also, additional control logic and wiring is required to control sharing of the bus.

To provide maximum speed with minimal wiring, the transputer uses point-to-point serial communication links for direct connection to other transputers.

4.2. *Transputer systems*

A transputer system consists of a number of interconnected transputers, each executing an OCCAM process and communicating with other transputers. As a process executed by a transputer may itself consist of a number of concurrent processes the transputer has to support the OCCAM programming model internally. Within a transputer, concurrent processing is implemented by sharing the processor time between the concurrent processes.

The most effective implementation of simple programs by a programmable computer is provided by a sequential processor. Consequently, the transputer processor is fairly conventional, except that additional hardware and microcode support the OCCAM model of concurrent processing.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

4.3. *Sequential processing*

The design of the transputer processor exploits the availability of fast on-chip memory by having only a small number of registers; six registers are used in the execution of a sequential process. The small number of registers, together with the simplicity of the instruction set enables the processor to have relatively simple (and fast) data-paths and control logic.

4.4. *Support for concurrency*

The processor provides efficient support for the OCCAM model of concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. The processor does not need to support the dynamic allocation of storage as the OCCAM compiler is able to perform the allocation of space to concurrent processes.

At any time, a concurrent process may be either

active: being executed,
on a list waiting to be executed;
or inactive: ready to input,
ready to output,
waiting until a specified time.

The scheduler operates in such a way that inactive processes do not consume any processor time.

The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented with two registers, one of which points to the first process on the list, the other to the last.

A process is executed until it is unable to proceed because it is waiting to input or output, or waiting for the timer. Whenever a process is unable to proceed, its instruction pointer is saved in its workspace and the next process is taken from the list. Actual process switch times are very small as little state needs to be saved; it is not necessary to save the evaluation stack on rescheduling.

4.5. *Communications*

Communication between processes is achieved by means of channels. OCCAM communication is point-to-point, synchronized and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same transputer is implemented by a single word in memory; a channel between processes executing on different transputers is implemented by point-to-point links. The processor provides a number of operations to support message passing, the most important being 'input message' and 'output message'.

A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an input message or an output message instruction.

The input message and output message use the address of the channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both internal and external channels, allowing a process to be written and compiled without knowledge of where its channels are connected.

As in the OCCAM model, communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready.

4.6. *Internal channel communication*

At any time, an internal channel (a single word in memory) either holds the identity of a process, or holds the special value 'empty'. The channel is initialized to 'empty' before it is used.

When a message is passed by the channel, the identity of the first process to become ready is stored in the channel, and the processor starts to execute the next process from the scheduling list. When the second process to use the channel becomes ready, the message is copied, the waiting process is added to the scheduling list, and the channel reset is its initial state. It does not matter whether the inputting or the outputting process becomes ready first.

4.7. *External channel communication*

When a message is passed via an external channel, the processor delegates to an autonomous link interface the job of transferring the message and deschedules the process. When the message has been transferred, the link interface causes the processor to reschedule the waiting process. This allows the processor to continue the execution of other processes while the external message transfer is taking place.

4.8. *Intertransputer links*

To provide synchronized communication, each message must be acknowledged. Consequently, a link requires at least one signal wire in each direction.

A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one-directional signal lines, along which data is transmitted serially.

The two signal wires of the link can be used to provide two OCCAM channels, one in each direction. This requires a simple protocol. Each signal line carries data and control information.

The link protocol provides the synchronized communication of OCCAM. The use of a protocol providing for the transmission of an arbitrary sequence of bytes allows transputers of different wordlength to be connected.

Each message is transmitted as a sequence of single-byte communications, requiring only the presence of a single-byte buffer in the receiving transputer to ensure that no information is lost. Each byte is transmitted as a start bit, followed by a one bit, followed by the eight data bits, followed by a stop bit. After transmitting a data byte, the sender waits until an acknowledgement is received; this consists of a start bit followed by a zero bit. The acknowledgement signifies both that a process was able to receive the acknowledged byte, and that the receiving link is able to receive another byte. The sending link reschedules the sending process only after the acknowledgement for the final byte of the message has been received.

Data bytes and acknowledges are multiplexed down each signal line. An acknowledgement is transmitted as soon as reception of a data byte starts (if there is room to buffer another one). Consequently, transmission may be continuous, with no delays between data bytes.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

4.9. *Conclusions*

The transputer demonstrates that the concurrent processing features of OCCAM can be efficiently implemented by a small, simple and fast processor. The time taken for a process to start and terminate is about 1.5 μs; a small enough overhead to allow processes consisting of only a few statements. An interprocess communication has a fixed overhead of about 1.5 μs, and also requires time to transfer the message. Messages are transferred at up to 40 megabytes per second on-chip, and up to 1.5 megabytes per second through a link.

Experience with OCCAM and the transputer has shown that many applications naturally decompose into a large number of fairly simple processes. Once an application has been described in OCCAM, a variety of implementations are possible. In particular, the use of OCCAM together with the transputer enables the designer to exploit the performance and economics of VLSI technology.

The transputer therefore has two important uses. Firstly it provides a new system 'building block', which enables OCCAM to be used as a design formalism. In this role, OCCAM serves both as a system-description language and a programming language. Secondly, OCCAM and the transputer can be used for prototyping highly concurrent systems in which the individual processes are ultimately intended to be implemented by dedicated hardware.

5. PARADIGMS FOR MASSIVE CONCURRENCY

Many uses of concurrency are in embedded applications where the programmer requires explicit control over the physical resources of the machine. However, in general-purpose concurrent computing, the programmer is primarily concerned with introducing concurrency in the interests of increased performance. Relatively simple program structures are adequate for many applications. Indeed, where large numbers of processors are to be used, simple regular structures are essential.

The major concern of the programmer is to express an application in a form in which

(1) there are enough processes to keep all of the processors busy;

(2) the amount of computation performed by a process each time it communicates is large enough to ensure that the communication cost is insignificant.

To do this, it may be necessary to hold copies of programs and data in every processing node; VLSI technology makes this attractive because of the high relative cost of communication.

Three important paradigms are:

(1) pipelines, in which data flows through a series of nodes, each node making a transformation of the incoming data;

(2) farms, in which a controller continually farms out tasks to a large number of identical processing nodes; as a node completes a task, it is given a new one;

(3) arrays, in which a problem is decomposed into a regular array and each component of the array is mapped onto a corresponding processor.

The emergence of these paradigms suggests that new and useful concurrent programming languages could be designed. One possibility is a scientific programming language incorporating constructs such as pipes, farms and arrays. Besides being relatively easy to use, such languages offer the possibility of automatic resource allocation and therefore the possibility of portable subroutine libraries.

A general-purpose concurrent computer must support such paradigms efficiently.

6. The concurrent computer

Several 'general-purpose' concurrent computers constructed as a network of sequential processors are now available or under development. At the present level of VLSI technology we can implement in the same area of silicon the following components of a computer: a 10 Mips (million instructions per second) processor; a 4 Kbytes of memory; a 10 megabyte per second communications system.

Consequently, using the same silicon area, we can construct a single 10 Mips (million instructions per second) processor with 4 megabytes of memory (a conventional sequential computer) or a 10000 Mips computer with 2 megabytes of memory. Both machines would require about 1000 VLSI devices, and so are quite small computers.

The problem is to choose the correct ratio of memory to processors and to construct a computer with many processing nodes each with a small amount of memory, interconnected in such a way that it can be applied to practical problems. For small numbers of processing nodes, reconfiguration switches can be used to avoid the need to chose a fixed configuration.

For large networks, reconfiguration is not very practical. Further, there is an increasing interest in programs which dynamically allocate processing resources. This makes a fixed configuration with software (or hardware) providing message routing through the processing node attractive. It is therefore to be expected that future hardware developments will provide message routing systems; one example is the torus routing chip (Dally 1986).

A number of different network structures have been proposed for concurrent computers: pipeline; array (1D, 2D, 3D, ...); hypercube; toroidal surface; shuffle.

These structures vary in three important respects: ability to extend; ability to implement on silicon (in two dimensions); cost of non-local communication.

Pipelines and simple (two dimensional) arrays can be easily implemented or extended. Arrays (and hypercubes) become progressively more difficult to implement as the dimension increases, with much space taken by connections which need to cross over. However, 1000 or so processing elements can be connected in this way.

One difficulty with the hypercube structure is that the number of links provided at each node must be at least the dimension of the hypercube. This means that a standard component (which has a fixed number of links) cannot be used to implement an arbitrarily extensible array. An alternative structure which avoids this problem is obtained by implementing each node of the hypercube with a ring of transputers; this structure is known as 'cube-connected cycles'.

The cost of non-local communication, which arises when two nodes need to communicate via intermediate nodes, varies widely. A one-dimensional array is obviously the worst. It may appear that a hypercube provides the best solution, and to understand why this may not be so, it is necessary to examine how a hypercube configuration is implemented in practice.

Suppose the hypercube is being implemented using a two-dimensional interconnect such as metal tracks on silicon. When two order-n hypercubes are connected to form an order-$(n+1)$ hypercube, 2^n wires are needed to connect the corresponding nodes in the two order n cubes. These wires are not short; their length is the width of the order n cube, at least $2^{\frac{1}{2}n}\times$ the width of a single node. Consequently, the total area taken by the connections is of order $2^{\frac{1}{2}n}\times 2^n$. As the order, n, of the hypercube grows:

(1) the size of the hypercube interconnect grows faster than $2^{\frac{1}{2}n}\times 2^n$;

(2) the diameter, in terms of length of interconnect, grows faster than $2^{\frac{1}{2}n}$.

Of course, if the hypercube is implemented by using a three-dimensional interconnect the

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

results are slightly better, but the situation remains fundamentally unchanged; a large hypercube is a relatively expensive interconnect. This is an important conclusion as most of the interconnection networks used for concurrent computers are isomorphic to hypercubes.

It appears that arrays of low (two or three) dimension can provide lower latency communications with much smaller interconnects, even for very large numbers of processing nodes (1000–100000). This is primarily because no long wires are needed to implement them with a two-dimensional interconnect (Dally 1986).

7. Future developments

Existing transputers are designed to provide a very efficient implementation of concurrent processing and communication. Allocation of processors and memory is determined at compile-time to minimize runtime overheads. As a result, it is reasonable for a process to perform a simple operation, a few statements, between communications with other processes: static allocation allows fine grain parallelism to be used efficiently.

For many applications, it would be preferable to provide more dynamic allocation of processing and memory resources. This would, for example, enable concurrency to be used together with recursion to express algorithms which dynamically allocate and deallocate processors in the same way that a sequential program dynamically allocates and deallocates memory. This would raise the level of programming to the point at which it would be necessary to specify only where 'real' concurrency is to be used. Inevitably, the increased cost of the concurrency implementation would dictate that a process should perform a larger amount of processing between communications; dynamic allocation requires coarse grain parallelism.

Some of the overhead can be recovered by building more capabilities into the processing node. The most important of these is the provision of message through-routing. This will enable software structures to be mapped onto a fixed processor configuration such as a two- or three-dimensional array as described above.

7.1. *Technology developments*

The development of the next level of static RAM technology (256 Kbit) is well advanced, and by 1990 it is expected that 1 megabit static RAMs will be in production. By this time 4 megabit dynamic RAMs will be available.

This means that a transputer could be equipped with 100 Kbytes of on-chip RAM and a processor at least twice as fast as the present one. Of course there is also the possibility of considerably more on-chip processing and communication capability than is currently being achieved.

With scaling to smaller geometries comes an increase in speed. Present transputers achieve processor cycle times and memory cycle times of about 50 ns. The 1990 technology is expected to reduce this to about 20 ns. Notice that the switching time scales linearly, but the number of devices scales quadratically; another important reason for the increased use of concurrency.

7.2. *Memory*

The possibility of incorporating large memories opens up interesting architectural possibilities which are only just beginning to be explored. For example, an on-chip RAM can

provide very high data rates because it can be accessed in rows (for example, 256 bit 'words'). It is also expected that external (commodity) RAMs will provide for access to blocks of memory rather than just words. The access to successive (or in-block) locations will be limited by the rate at which data can be sent off chip, not by the speed of the memory itself.

The access time of an on-chip memory will continue to be about two to four times faster than the access time of an off-chip one. This means that sufficient on-chip memory should be provided for the frequently accessed process workspaces and stacks, and block transfer should be used to transfer data between on-chip and off-chip memory.

7.3. *Processing node architecture*

The possibility of wide (256 bit pages) access to the internal memory allows several processors to share the same (internal) memory. If each processor is equipped with a very small data and instruction cache, most accesses to the internal memory can take place in pages. This architecture would allow, for example, a communications processor to transfer application-specific data structures through communication links while the other processor(s) perform the computation associated with the node.

The processor itself can be provided with additional microcoded operation at very little cost. In practice, the major problem is to identify generally useful operations. However, support for dynamic storage allocation to permit recursive, concurrent programs and their associated user defined data types is a logical step.

7.4. *Communications architecture*

One important use of the extra silicon area available within a processing node is the provision of hardware for global message routing. As a message arrives (or is generated locally), its destination address is decoded and, if necessary, the message is forwarded via the appropriate communications link.

Two techniques can be used for message routing. In packet routing, a message is completely input before being output. This does not seem to be a very appropriate scheme; it introduces a significant delay through the node and also requires memory space within intermediate nodes to store the packets waiting to be output. A better scheme is 'wormhole' routing (Dally 1986) in which the address is input, the destination link is determined and the message forwarded as soon as the destination link is not busy.

Another possibility is the provision of hardware support for global memory addressing, even in a distributed memory multiprocessor. Non-local addresses can be decoded in hardware, and read and write operations translated into messages to remote processing nodes. These are routed in the same way as normal messages. (It might seem that the provision of a global address space removes the need for process communication. However, process communication cannot be performed efficiently with only memory-access operations; it must be possible to cause a process to be scheduled when a message arrives.)

The result of these developments is that the programmer will not need to adapt his programs to a particular system structure, nor will reconfiguration be needed. However, it will still be important to preserve locality in mapping algorithms onto the processor array. This is an important area for future language and compiler development.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

7.5. *Monitoring*

As the technology allows several processors to be combined onto the same chip, it is inevitable that some processors will have no direct connections to any pins; their behaviour will not be observable from outside the chip. Consequently, facilities for testing and monitoring a processing node will be built into it, and the most convenient way to do this is to include a communications link specifically for the purpose.

The monitoring of a large multiprocessor is a computationally intensive task. Monitoring information generated at many nodes in the system must be processed and useful information extracted in real time. By using a standard communication link on each node for this purpose a multiprocessor monitoring array similar to but smaller than the processing array can be used. This monitoring array can be programmed in exacty the same way as the processing array.

7.6. *The input–output system*

Future computer input and output will require specialized concurrent processors. These will perform operations such as image analysis, graphics, animation, speech and public key encryption.

These functions can be implemented as either fixed configurations of standard components such as transputers, or as specialized VLSI systems.

The construction of tools for the design and programming of systems constructed from standard components and specialized silicon devices is the subject of current research. Recent work has shown that a communicating process language such as OCCAM is suitable as a source language for silicon compilation (May *et al.* 1987). This approach allows a wide range of concurrent algorithms to be implemented directly in VLSI, and allows reliable migration of algorithms from software into specialized hardware as performance requirements increase. It also allows a wide range of program development and verification tools to be used for VLSI designs.

These tools are essential to realize the potential of the technology: a complete concurrent computer including input and output on a single chip.

8. Evolution of conventional architectures

In view of the development of the technology, certain changes in conventional architecture seem inevitable.

8.1. *Sequential architectures*

Firstly, manufacturers of conventional microprocessors will have no difficulty in incorporating large (for example, 64 Kbyte) cache memories on the same chip as the processor. There will be a strong incentive to do so in view of the difference in speed between internal and external memory access.

It seems likely that virtual memory will continue to be used: there will continue to be a significant difference in speed between the static RAMs, dynamic RAMs, discs, etc. Consequently, it seems likely that conventional microprocessor systems will combine a processor, cache and memory manager on a single VLSI chip, with external main memory provided by high-speed static or dynamic RAMs and secondary memory provided by dynamic RAMs or discs. As time

progresses, the high-speed on-chip caches will become so large as to remove the need for anything except secondary memory external to the device. As the transfer of data between secondary and primary memory involves blocks rather than randomly accessed words, the external memory interfaces can be optimized to this purpose; they can become more serial. This reduces pin-count and cost on both memory devices and processors.

Processor designs will continue to become more ambitious. Even the 'reduced instruction set' computers (RISC) now in design involve complex pipelining that uses large areas of silicon to produce small gains in performance improvement. Processors will normally include floating-point arithmetic and even pipelined vector floating point. As noted above, on-chip memory can provide data at the high speeds needed to support pipelined operations.

Inevitably, as an entire sequential computer will be easily implementable on a single chip, sequential architectures will evolve into concurrent architectures. This is already happening, with machines consisting of a small number of processors (up to 20) sharing a common bus to which a global memory is connected. Contention for access to the global memory is alleviated in one of two ways:

(1) by providing local cache memory on each processor and ensuring cache consistency (at considerable hardware cost);

(2) by providing local memory and using the global memory only for (explicit) message passing (this appears to the user rather like a distributed memory multiprocessor).

The presence of a global address space makes these machines appear easier to program than a truly distributed memory multiprocessor (such as a hypercube or transputer-based machine). However, in practice it is necessary to observe very similar principles, ensuring local operation wherever possible.

Communication involves synchronization to tell the receiving processor that data has arrived or to tell the sending processor that data has been taken. This cannot be efficiently implemented by a shared memory alone; an additional interconnect between the processors is needed. It is also necessary to incorporate microcode and hardware support for process scheduling and communication; without it the overheads of concurrent processing are excessive.

8.2. *Array processing*

Many applications can make use of a very large number of simple procesing elements operating in parallel. One example is image processing (or low-level vision). These applications have been implemented by SIMD (single instruction multiple data) machines such as the DAP (Distributed Array Processor).

The SIMD machine was conceived when a control unit was large and expensive in relation to an ALU (arithmetic logic unit) and a few registers, and when the speed of the physical interconnect between control unit and ALU did not limit performance. It consists of a large synchronous array of arithmetic logic units controlled by a single microprogram controller which synchronously broadcasts instructions through the array.

In VLSI technology, a control unit and a program store is small; no more than the connection points needed to receive the wide instructions from the control unit. Furthermore, current SIMD machines are becoming limited in speed by the time taken to broadcast instructions through a large processor array. This problem is getting worse.

It may be better, therefore, to use a MIMD (multiple instruction multiple data) machine based

on a very simple processing node. Such a machine could use the same architecture and interconnect as the more general-purpose MIMD processing array. This would allow fine-grain SIMD style processing to be combined with more general-purpose MIMD processing.

8.3. *Vector processing*

To have significant benefits a vector processor must be pipelined; this makes it much bigger than a scalar processor. It must have:

(1) two operand access paths;
(2) two normalizers for source operands;
(3) additional floating point ALU for scalar product;
(4) further normalizer for result;
(5) more complex control logic.

One vector processor will therefore take the same silicon area as several scalar ones. Several scalar processors sharing the same store are more flexible than a single vector processor, and so a collection of scalar processors is preferable to a vector processor if both have the same performance and use the same area. However, it is likely that there is a trade-off of flexibility against efficiency. This issue is not well undertood and requires further investigation.

9. Conclusions

A general-purpose concurrent computer will contain a processing array with many (between 10 and 1000000) similar processing elements. These will incorporate automatic message routing providing low-latency communication between all the processing elements. The elements will be connected in a fixed network.

The general-purpose processing array will be surrounded by input and output systems also containing many processing elements. These will be more specialized, consisting of general-purpose devices connected in specialized configurations or, alternatively, of specialized devices.

The concept of communicating processes can provide an architectural basis for both general-purpose and specialized parts of the computer. Languages based on communicating processes, such as OCCAM, serve as the system language for these concurrent computers. Experience gained from the application of these machines will form the basis for higher-level user languages.

References

Dally, W. J. 1986 *A VLSI architecture for concurrent data structures*. California Institute of Technology.
Hoare, C. A. R. 1978 Communicating sequential processes. *Communs ACM* **21** (8), 666.
Inmos Limited 1988 *Occam2 reference manual*. New York: Prentice-Hall.
May, D., Shepherd, R. & Keane, C. 1987 *Communicating process architecture, in future parallel computers* (ed. P. Trealeaven & M. Vannesch). New York: Springer-Verlag.
Homewood, M., May, D., Shepherd, D. & Shepherd, R. 1987 The IMS T800 transputer. *IEEE Micro*. **7** (5), 10–26.
Mead, C. A. & Conway, L. A. 1980 *Introduction to VLSI Systems*. New York: Addison Wesley.
Roscoe, A. W. & Hoare, C. A. R. 1986 The laws of Occam Programming. Programming Research Group, Oxford University.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Discussion

S. F. Reddaway (3 *Woodforde Close*, *Ashwell*, *Baldock*, *Hertfordshire*). I would like to respond to Dr May's critical comments on SIMD architecture in three ways.

(1) The best way to judge a machine is by results. There are many impressive demonstrations that can be done on the SIMD machine.

(2) The instruction time of that machine is 100 nsec, rather than the 500 or 1000 nsec mentioned by Dr May.

(3) If MIMD hardware is used to do SIMD processing the overheads are greater than when SIMD hardware is used. These are in terms of hardware cost, software time (for example dealing with MIMD communication) and awkward programming if the language used does not support array operations.

D. May. I agree that the best way to measure a machine is by results. However, I am primarily interested in the most cost-effective way to achieve the results in a given technology. The question to be resolved, therefore is: can the operations supported by a SIMD machine be implemented in less silicon area with a MIMD machine? (In making this comparison I would expect both machines to be programmed in a language with array operations.)

The additional hardware cost referred to by Dr Reddaway is the cost of *local* control and synchronization logic: this cost is small and is decreasing relative to the cost of broadcasting high-speed *global* control and synchronization signals.

The additional software time referred to by Dr Reddaway is the time to perform *local* synchronization: this time is decreasing relative to the time taken to perform *global* synchronization.

Consequently, I believe that the answer to the above question is (or will soon be) *yes*.