マイクロコンピュータ 研資料 33－1
計算機アーキテクチャ 研資料 55－1
（1984. 11. 5）

CONCURRENT SILICON SYSTEMS

Prot. Iann M. Barron, Chief Strategic Officer, INMOS INTERNATIONAL PLC.

CONCURRENT SILICON SYSTEMS

The subject of my lecture is a wafer of silicon. Not just the wafer, but the potential that it offers for building more powerful computing and analytical devices.

A wafer of silicon four inches in diameter contains some 256 64K dynamic RAM chips; it is a masterpiece of modern technology and a minor work of art. On that wafer - which would cost less than $500 to manufacture - there are some 40 million transistors, which represents two megabytes of random access memory or 500 16 bit microprocessors. Of those 40 million transistors, some 39999800 are perfect and there are 200 imperfections or thereabout. The strategy in manufacturing has been to cut up the wafer into small sections to avoid these imperfections, making available just small chips that we see in current equipment.

Clearly with the development of technology and the exploitation of redundancy we can look forward to using not merely a small area of the chip but the whole wafer.

If we look at the amount of memory that we can get on a single chip compared to the amount of processing, we see that to a first order of approximation 1 MIP occupies about the same amount of silicon area as one kilobyte of memory. Now that is a very interesting ratio because if we look at the existing computers we see a very different ratio between processing and memory. It is conventionally regarded that processing is expensive and memory is cheap; the result is that we have computers with many megabytes of memory attached to single processors. A simple examination of silicon suggests that this is no longer the right relationship.

VLSI complexity creates a challenge for us. There are a vast number of devices available on a wafer of silicon. We need to devise ways to structure systems to exploit this capability. This means that we must create tools to exploit concurrency and to manage complexity.

Looking at the capability of devices over the past 25 odd years that silicon has been processed, we see an exponential growth in the number of devices available in an integrated circuit. Complexity has increased by an order of magnitude every five years and there is a remarkably good fit to an exponential curve.

That increased capability has come about from three factors. The linear dimensions of the artifact put onto silicon have been reducing in size by about an order of magnitude every fifteen years. Of course, since there are two linear dimensions in the design, that factor counted twice. The third factor is that the area of a silicon chip has itself been increasing, again at an order of magniture every fifteen years.

If I look at the use of components in conventional systems, and I have taken the example of computers, we see again an exponential growth in the performance of the systems. The trend line to the mid 1980's indicates that our super computers will have a computing capability of the order of 100 to 1,000 million instructions per second and that the mainframe computers in standard use will have a computing capability of the order of 10 to 100 million instructions per second. We also see that the humble microprocessor, a simple device which is becoming ubiquitous, itself has the potential computing capacity of 1 to 10 million instructions per second. The performance of the humble microprocessor will be no less than the performance of the super computers of the mid 1960's.

Look forward towards the things that we might like to do in the future, such as speech recognition of a substantial vocabulary, or recognition of images in a complex pattern, or accessing very large databases of information. The computing requirements become large. Those examples were not chosen at random. They are the targets that have been set by you Japanese in your Fifth Generation Programme to be achieved by 1990.

While it may not be possible to describe the algorithms that will be used to achieve these ends, it is possible to make estimates of the computing power that will be required. Such problems are estimated to require between 10 and 1000 Gips, here a GIP is 1000 million instructions per second.

It is reasonable to take the view that, just as it has been said in the past that there is an infinite market for bits of information for storage, so there will be an infinite demand for MIPS of information for processing.

But if we look at those targets in relation to the trend line for the fastest computers, there will be a serious mismatch if we follow the techniques that we have been using so far. There is no possibility of providing the kind of computing power that is estimated to be required. This means that if we want to address problems of the order of magnitude cited, we must adopt some new strategy.

Indeed, the development of silicon capability calls for a need for architectural change. We can increase the complexity of a system by a factor of 10 every five years but only increase its performance by a factor of 10 every fifteen years. This immediately suggests that the architecture of systems ought to be restructured in order to exploit the complexity that is available.

Clearly we would like to use more components for a given system and this means that we must devise ways of exploiting concurrency, so our problems can be solved with various parts being done simultaneously. But when we look at computers we immediately run into a problem that the computer architectures that we use today are inherently sequential. Indeed the whole basis of computing science has been to take a problem and to analyse it into a sequence of steps which the computer then solves. It has been observed by Carver Mead of Caltech that sequential systems will not be adequate for the future and he suggests there are an additional four orders of magnitude of computational capability available through concurrent systems. There may only be one thing wrong with that statement. He may have under estimated the potential.

Concurrency in computers can be exploited in a number of different ways. We can try to hide the use of concurrency and this is the approach that has been adopted largely up to the present. The simple way of hiding concurrency is to use pipelining in which the various elements of a programme are processed stage by stage in order to get a faster computational rate.

The second method of attempting to hide concurrency is to use vectorisation. There are problems - particularly large numeric problems-which involve numerical operations not on single values but on vectors, and as a result these operations can be carried out concurrently on all the elements of the vector. The techniques that are used are to take existing programs and to identify this potential for concurrency and then to build a computer and write a compiler to exploit the characteristic.

A third possibility is to use the concept of dataflow. In a conventional computer the instructions are performed one after another in a sequence defined by the programmer. In a dataflow machine the sequence of instructions is determined not in a preordained fashion but by the availability of the data relating to the instruction, so that instructions are executed in whatever order the computer thinks best, given the availability of data.

Regrettably all of these techniques suffer from a basic problem that the original programs being used were built and designed for sequential computers. The alogrithms used also tend to be sequential in nature and not to have

as much concurrency as perhaps alternative solutions to the problem could have. So that while we may get a computer to exploit the concurrency that is available in programs as they stand, this is unlikely to generate a substantial improvement in capability.

The next thing that one could do is to make the concurrency available exlicitly to the programmer, so that he controlled and decided what computations were to be done where in the computing system. We do not at the moment have mamy programming languages that can express these concepts effectively. At INMOS we have been working on the development of a concurrent programming language called occam which provides precisely this kind of explicit capability. The work on occam has been done by David May in conjunction with Professor Tony Hoare from Oxford University and is based upon earlier work by Tony Hoare on communicating sequential processes.

Now there is a third way in which we can exploit concurrency. There are some problems which have the potential for considerable implicit concurrency. The natural examples are the declarative languages which are being proposed for fifth generation computing systems. Such languages have neither the concept of sequence nor concurrency in them, and in creating an evaluation of a problem in such a language the compiler can create a concurrent solution equally as well as a sequential solution. Indeed since the solution is basically to create a search through a tree in the solution space, that search can well be carried out concurrently.

One of the key questions one must ask is at what level should concurrency be put into computer systems. I have already mentioned the idea of data flow. At this level of concurrency an attempt is made to exploit some operations in a computer in parallel - the basic additions and subtractions and multiplications. The difficulty with doing this is that there is a very substantial overhead associated with implementing this concurrency and it does appear that the overhead is high in relationship to the improved performance that is obtained.

Another alternative is to attempt to create concurrency at the program level by getting large scale programs to interact. Concurrent systems that are built at the moment using languages like Ada are operating at this level.

However, INMOS have come to the conclusion that if one wishes to exploit silicon effectively, there is an intermediate level which is characterised in computer programs by being at the procedural level, where the procedure is probably an object which has between 10 and 100 conventional computer instructions.

We need to build a capability to exploit the idea of concurrent computing at this level. In the past circuits have been designed at one of two levels of abstraction. At the lowest level they have been built from transistors using analog signalling techniques to communicate between the devices, using a set of design rules based upon conventional electronics. If we move up a level in complexity to build systems with say between 10 and 100 devices as a basic building block, then we can use the logic gate. A logic gate, of course, is a specialised circuit which admits only a binary signal as an input, rather than a general analog signal, and generates a binary signal as its output.

Now the significant thing about the logic gate is that there is a calculus available, Boolean algebra, for describing the behaviour of connections of such gates. At the moment all systems are designed basically using that concept. That was fine when one had a relatively small number of components in an electronic system. However you will remember that our wafer of silicon offers the potential of 40 million devices and that is far too many to manage in terms of this kind of design process. So we would like a rather larger building block from which to create our systems.

The INMOS proposal is that building block should be a simple computer, which we call a transputer, which has formalised communication so that the communications between transputers can be regarded as communication of the information. Then one needs a set of design rules for interconnecting such systems, and these can be created in terms of our occam calculus, which can also be regarded as a programming language.

A transputer is a simple formalised computer. It contains processing memory and formalised communication. It is designed to be a programmable component and it will execute a concurrent programming language. Communications between one transputer and another will use the communications model in the language and the transputer will behave like a process as defined within the language.

The basic concept is the idea of a process. A process can be regarded as a black box with input signals and output signals. The behaviour of a process is only visable through the signals it receives and transmits. It may have an internal structure which can consist of more processes with the same properties.

If we consider at the programming language that we require, we can extend the ideas in conventional computing languages or introduce a number of constructors. The first of these constructors, for sequence, indicates that we will have our processes carried out one after another - which of course

is just the way that a conventional computer operates. The parallel constructor on the other hand says that a set of processes will be carried out together and the constructor will not be completed until all of the processes have been completed.

Within the language we retain the concept of expression evaluation as a primitive process where we evaluate an expression and set the result of that to a variable equal to the result of the evaluation. We have added further concepts. The first concept is the idea of output to a communication channel and the second matching concept is one of input from a communication channel. Communication is synchronized.

Imagine two pocesses that are executing and are going to communicate on a channel. Communication will only occur when the first process is ready to output and the second process is ready to input. Of course, one of the two processes will become ready first and must wait until the second process is ready before the communication occurs.

The final basic concept we need to add is the concept of an alternative construct where we allow input not from a single channel but from a set of channels, and the input that is selected is the first one that is available. What we have done is to add to conventional computing languages those primitives that actually exist in a computer, like input and output and interrupt, so that a language of this sort actually matches much more exactly the way that real computers work.

The system model we are adopting is to say that concurrency and communications are represented explicitly in the language. A program is represented in terms of concurrent processes which communicate using point to point communication channels. Clearly one obvious implementation of a program in occam is to take a network of computers, or transputers, each one representing a parallel process with the connections between them mirroring the communication channels in the language. We can also execute the same program on a single transputer by time sharing the concurrent processes. And, of course, since we can move between these two extremes we can find any point between the limit. We can select a subset of processed and put them onto one transputer and another subset of processes and put them onto a second transputer so that we can actually choose - after a program is written - to map the problem onto a computing network with a varying amount of concurrency.

There is an interesting analogy with an approach that already exists in computing, where a problem is mapped onto a real memory system of a particular size - the concept of virtual memory. It has proved very useful. Now we have the opportunity of virtual processing.

INMOS has created a calculus for designing systems out of processes, where one can implement a process either as program or as a logic configuration. A transputer, in this context, is a special process which can emulate any other process when given the description of that process as an occam program. So we have created yet another degree of flexibility. An occam program can be implemented either in hardware or in software, or we can take some of the processes and put them into hardware and leave others in software. One can choose any kind of performance tradeoff that is required.

(The lecture will contain some examples in more detail at this point)

So where does this get us to? If we go back to our wafer of silicon, we can get onto that something like 256 transputers, giving us a megabyte of memory and 2.5 giga instructions per second. The capability of a concurrent computing system of this sort can achieve the targets that are required for Fifth Generation Computing applications, using a collection of transputers where the number of elements is perhaps 1,000 to 10,000. That might seem a lot, but if you think how many memory chips there are in a conventional computer you suddenly realise that putting 10,000 transputers together creates a system which is no larger than a conventional computer. So I think we have an exciting time ahead of us.

We can build computers which will provide us with a power a step function above the computing power we have had available. These computers will work effectively not only with numerical calculations but with declarative languages, so they can be used directly for Fifth Generation Applications. But to do it we have to introduce a new type of computing component, the transputer, which can be interconnected to build these powerful systems. (end)