Building Parallel Distributed Models for Asynchronous Computer Architectures

G. Theodoropoulos J. V. Woods
Department of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, United Kingdom

Abstract. Recently, there has been a resurgence of interest in asynchronous design techniques. Asynchronous logic provides a solution to the clock-related timing problems of synchronous systems and can offer higher performance and lower power consumption. This paper presents an approach for modeling and simulating asynchronous computer architectures using occam as a description language.

1. Introduction

Conventional synchronous architectures use design techniques based on global clocking whereby all the functional units operate in lockstep under the control of a central clock. As VLSI technology advances and systems become larger, faster and more complex, timing problems become increasingly severe and account for more and more of the design and debugging expense. Increased clock speeds make on-chip clock skew significant and interchip skew a major problem. One solution to clock-related timing problems is to use asynchronous design techniques without any global synchronization signals to control the rate at which different elements operate [20]. Another potential advantage of asynchronous logic, which recently has led to a resurgence of interest in its use, is lower power consumption.

In order to investigate the potential of asynchronous logic for low power systems, the AMULET group (University of Manchester, UK) has designed and implemented a completely asynchronous version of the ARM RISC processor [8] as part of the ESPRIT OMI-MAP (Open Microprocessor systems Initiative-Microprocessor Architecture Project).

The absence of a global synchronization scheme introduces a problem unique to the asynchronous world, namely the problem of deadlocks. The concurrent, nondeterministic nature of asynchronous architectures makes their verification particularly difficult. In this case simulation can be an invaluable aid. It is extremely important to have a methodology that provides for the rapid production of simulation models and their fast execution so that possible deadlocks in the asynchronous design are detected at an early stage of the design process. This paper describes such a methodology.

2. Asynchronous logic

An asynchronous system may be designed as a set of functional modules each operating at its own rate and cooperating through communication. The communication protocol synchronizes the modules involved in the communication and allows data to be shared between them. There exist many different approaches for designing asynchronous systems [10]. *Delay-insensitive* designs make no assumptions about delays within the system; any gate or interconnection may take an arbitrary time to propagate a signal. *Speed-independent* systems are tolerable

Figure 2: The bundled data interface of micropipelines

to variations in gate speeds but assume instantaneous transmission along wires. Regarding the communication protocol, a system may use *dual rail encoding* or *data bundling*; in dual rail encoding each boolean is implemented by two wires, something that allows the value and the timing information to to be communicated for each data bit. Bundled data has one wire for each bit and a separate wire to indicate the timing. Furthermore an asynchronous circuit may use *level encoding*, where different logic values are represented by different voltages, or *transition signaling* (referred to as *event signaling*) where only changes ("events") in the level (and not the actual level) of signals are taken into account. Figure 1 shows some of the basic event processing blocks that can be used to build control circuits for transition signaling.

In his influential 1989 Turing Award lecture, Sutherland presented an asynchronous design approach called "Micropipelines" [22]. A micropipeline is a simple data processing pipeline whose stages operate asynchronously. This approach uses bundled data with an event-signaled handshake protocol. Figure 2 shows the interface between a sender and a receiver. The sender puts a data bundle on the data wires and then produces a transition on the Request wire; once the data has been used by the receiver, a transition is produced by the receiver on the Acknowledge wire. This sequence of events is presented in figure 3. Once this protocol sequence is enforced, a micropipeline is delay insensitive.

2.1. The AMULET1 processor

Following Sutherland's micropipelining approach, the AMULET group have designed and implemented an asynchronous version of the ARM processor, namely the AMULET1. Fig-

Figure 4: The AMULET1 interface

ure 4 shows the top level interface of the AMULET1. The processor produces output bundles consisting of memory addresses, output data and control information; in response, the memory sends bundles of input data back to the processor (the actual implementation is more complicated for there is an MMU unit to handle memory faults). The internal organisation of the processor is depicted in figure 5[8]; all the separate functional modules are implemented as micropipelines.

3. Hazards of asynchronous architectures

The concurrent nature of such systems along with the absence of a global clock for synchronization introduces a problem, common in asynchronous, parallel structures, namely the problem of *deadlocks*. Deadlocks are a high-level issue of the design, and occur when at least one functional module becomes indefinitely blocked as a result of a particular sequence of events in the system.

In the general case, the sequence of events in an asynchronous architecture is nondeterministic. This is mainly due to the behaviour of the arbiters and the delay-insensitivity of the system. An arbiter will serve request events according to their arrival order. If two requests arrive at the same time, the choice will be nondeterministic; the arbiter can reach a *metastable* state in which case a decision will take an arbitrary, nondeterministic amount of time. The delay insensitivity of the hardware allows variable delays within the different functional elements, which will affect the order in which independent events arrive at the arbiter. The correct functionality of the asynchronous architecture should not depend on the ordering of independent streams of events in the system; a correct design should be deadlock free for all possible combinations of events.

Verifying that an asynchronous design is deadlock free is not a trivial issue [23]. Existing formal techniques are not mature enough to tackle systems of the complexity of computer architectures, although research is ongoing in this area [24].

Figure 5: The AMULET1 organization

In practice, it is generally possible to identify, and thus avoid, design desisions that are susceptible to deadlock [21]. However the complexity and the nondeterministic behaviour of the designs do not allow intuition to guarantee the correctness of a design. Simulation can be an invaluable aid for this problem. The approach is to run the simulation model of the architecture more than once, each time with a different set of delays in the component modules. Changing the internal delays of the functional elements changes the order in which events are produced. Consequently, the order in which events from different data streams arrive at the arbiters changes. Since delays dictate event orderings, following this approach the design can be tested for possible deadlocks. The degree of confidence that a design is deadlock free is proportional to the number of runs of the simulation model. The speed of simulation here plays a crucial role; a fast simulator, would allow the delay insensitivity of the architecture to be rigorously and extensively tested for a large number of possible combinations of events.

This technique requires a modeling approach that would allow the rapid production of executable models of the architecture at a high-level so that possible deadlocks are located at an early stage of the design process.

4. Simulation of asynchronous architectures

Assuming a correct implementation of the communication protocol, at the Register Transfer (or higher) level, an asynchronous architecture can be viewed as a network of concurrent, communicating modules. The communication is synchronous and unbuffered; a sender and a receiver must wait for each other to reach a common control state before they physically exchange data via wires, which are memoryless media. The modules are data-driven; each module will start computation when data is available on its input wires, and will signal when the result has been computed.

Several simulation techniques, both sequential [5] and parallel [7] have been developed.

Distributed, event driven, simulation provides a natural and efficient way for simulating asynchronous, concurrent, process-based systems [6]. Using such an approach the simulation model consists of a network (topologically identical to the physical system) of concurrent logical processes that communicate with each other by exchanging timestamped messages. This approach exploits the parallelism of the physical system allowing the concurrent execution of events at different points in simulated time, thus yielding high performance.

However, this technique introduces synchronization problems related to the correct modeling of time. The fundamental problem is guaranteeing that causality constrains are not violated. To achieve this, the simulation model must ensure that each logical process consumes and processes events in increasing timestamp order. Several mechanisms have been developed to address the problem of modeling time correctly. These mechanisms broadly fall into two categories: *conservative* [3]and *optimistic* [15].

These attempts to maintain absolute timing precision increase the complexity of the simulation model and reduce its performance. Indeed, the accurate time modeling would require the implementation of sophisticated mechanisms in the model, increasing thus the complexity of the modeling process. Furthermore, any synchronization would not allow any process to be temporally more advanced than the slowest independent process in the whole simulator.

The simulation approach proposed in this paper does not attempt to enforce timing accuracy. Using a distributed parallel language that supports synchronous unbuffered interprocess communication, an asynchronous architecture can be modeled as a set of concurrently executing processes. Time is not required for synchronization. The processes of the simulation model are entirely data-driven and self-scheduling and are synchronised by the communication protocol of the language employed in the same way that the communication protocol implemented in the architecture synchronizes the different functional modules. Each process will always consume event messages as soon as they become available, and it will always wait for subsequent messages if the messages it has generated has been successfully forwarded. However time is still needed to provide an indication of the performance of the design, and since no mechanism for correct timing modeling is employed, one should expect that the simulation model has no temporal resemblance to architecture. As explained in later sections this is not necessarily true.

The proposed approach is appropriate for making simulation models of the architecture at the Register Transfer (or higher) level. For the modeling of the architecture at lower levels of abstraction no assumptions should be made concerning the communication protocol of the design; instead, explicit modeling of the protocol to verify its correctness is required. In the asynchronous circuit, communication occurs over individual wires while message passing necessarily involves lower level communication for synchronization of the processes involved and therefore it is inappropriate to describe the behaviour of the system at this level. Message passing may still be used for the communication of the low-level processes (gates or event processing elements) but at this level, time is essential for the synchronization and the correct operation of the simulation model.

CSP [11] provides a natural way for describing the concurrent, nondeterministic behaviour of asynchronous circuits and several CSP-like notations have been devised for this purpose [1]. Occam[1] [12], a CSP based language, is particularly suitable for implementing asynchronous architectural simulation models. It supports a distributed, process-based model of computation where message passing is done over fixed, synchronized and unbuffered channels, and is ideal for describing pipeline structures. It allows explicit description of parallel as well as sequential computation. This explicit control of concurrency which extends to the command

[1]Occam is a registered trademark of INMOS Group of Companies

Figure 7: Micropipeline with processing

level, along with its simple but powerful syntax and “send” and “receive” commands, makes occam ideal for describing digital systems.

5. The case of micropipelined architectures

Following the approach proposed in the previous section and using occam as a description language, the production of simulation models for micropipelined systems is straightforward. Figure 6 shows how the functional modules depicted in figure 2 are modeled assuming that they are simple registers. The Request and Acknowledge signals are used in the circuit to synchronise the two registers. In the model, these signals are actually part of the semantics of occam, therefore no extra channels are required for them. Since the circuit is delay insensitive, transmission and communication interference is not allowed to happen. The sender will never produce a second request until the previous request has been acknowledged. Furthermore, absence of computational interference guarantees that the receiver will always be ready to read an event on the request signal. The semantics of the **?** and **!** occam operators accurately model this behaviour. Figure 7a presents a general case of a micropipe with processing. The sending register outputs its contents (data and control) onto the data bus and produces a request event.

The control information is used by the control logic to direct the request event to its correct destination, activating if necessary the data processing elements (DPE) of the pipeline. Data passes through the DPEs and are either modified or propagated unchanged to the next stage.

This circuit at the Register Transfer level can be modeled as three processes, two for the registers and one for the control/data logic (figure7b). Since the data are bundled with the request they may be viewed as "following" the request signal. Therefore only one occam channel is required for the request/data bundle. The control logic and the DPE may be modeled as one process, with the DPE being a procedure called by the control process.

The control process in the simulation model decouples the registers, introducing a third pipeline stage. As the design of the AMULET1 has shown, the number of stages in a pipeline is directly relevant to the occurrence of deadlocks. If the simulation model is to describe the architecture accurately, the occam register processes should remain tightly coupled. To achieve this, an extra channel is required to model the acknowledge event; figure 7c presents the generic register occam model.

The control logic is inherently concurrent; different parts of the circuit operate concurrently while, within each part events take place in a deterministic sequential order, i.e. the control logic implements a partial ordering of events. The simulation model should have the same degree of concurrency as the physical circuit. The control logic may be implemented as a network of communicating processes, with the occam PAR and SEQ commands being used within each process to implement the partial ordering of events of the circuit. The number of these processes depends on the degree of modularity and fidelity required in the simulation model. Channels that are used to transfer between control processes data that are not part of a request/acknowledge communication protocol should be buffered to prevent deadlock situations in the simulation model.

Adopting a data-driven approach to model asynchronous architectures, it is essential to have a mechanism for describing the functionality and the nondeterministic behaviour of arbiters. The occam ALT construct, which is based on Dijkstra's guarded commands [4], provides for the nondeterministic choice of messages from different channels and therefore may effectively model the behaviour of an arbiter [13].

5.1. Timing issues

Each occam process of the simulation model has its own local clock variable which advances as the process consumes and processes timestamped messages. The local clocks advance at completely different and independent rates. Each process tries to consume messages as fast as possible and no synchronization mechanism is incorporated into the simulation model to ensure that time is modeled accurately. In distributed simulations, causality errors may occur if merge modules consume and process input messages from different channels in nonincreasing timestamp order. In a micropipelined architecture, micropipelines may be merged in one of the following ways:

- **Synchronous merge**. A functional module has to wait for all input data to become available before it starts its operation. This is the case when a **Muller-C** element is used for the corresponding request events. In the simulation model, the occam process has to wait for all input channels to "fire". The message with the greatest timestamp is used to advance the local clock variable of the process and therefore the causality principle is preserved.

- **Data dependent merge**. The functionality of the system dictates the order in which messages from different source processes should be consumed and processed. This sit-

Figure 8: Message timestamps

uation is implemented in hardware using a combination of a **select** and a **call** or **xor**. The process in this case behaves as a single input module, hence causality is not violated.

- **Arbitrated merge**. The order of arrival defines the order of consumption. If events from two micropipelines arrive at the same time, an arbitrary choice is made. In the circuit, **arbiters** are used to achieve this behaviour.

In the proposed simulation approach, an arbiter is modeled by the occam ALT command. The order in which the ALT construct will consume messages in the simulation model does not adhere strictly to the order in which events arrive at the corresponding arbiter in the physical circuit; it merely depends on the order in which the corresponding input occam channels are selected and not on the timestamps of simulated time that these messages carry. Therefore, in the general case, messages will be consumed by the ALT construct in a nonincreasing timestamp order and therefore the causality principle is violated.

This violation does not affect the correct functionality of the model; the very presence of an arbiter in the design implies that the order of consumption may be arbitrary. However it introduces an error in the simulated time and, consequently, in the performance evaluation of the simulated architecture. Nevertheless, the characteristics of the asynchronous architectures and the distributed nature of the simulation suggest that the inaccuracy in simulated time will be insignificant and therefore tolerable at this high level of simulation.

Each process will execute as soon as the data becomes available. The fundamental nature of the architectures, which are self regulating systems, will balance the throughput of the distributed processes preventing thus the local clocks from becoming too skewed. A similar approach adopted for the simulation of dataflow architectures has produced some encouraging results regarding the magnitude of the timing error [18]. Furthermore, the behaviour, the size and the cost of arbiters, make their use undesirable, therefore a typical design will contain a

very small number of such elements (the AMULET1 has only two).

Figure 8 describes the calculation of message timestamps by register and control processes. Data dependent (eg call) merge processes are treated as single input processes since only one of the input channels will be selected, while for synchronous merge processes, the input message with the greater timestamp is taken into account for the update of the clock. For arbitrated merge processes, the current clock is also taken into account so as to avoid preemptions on the output channel.

5.2. Delay Insensitivity

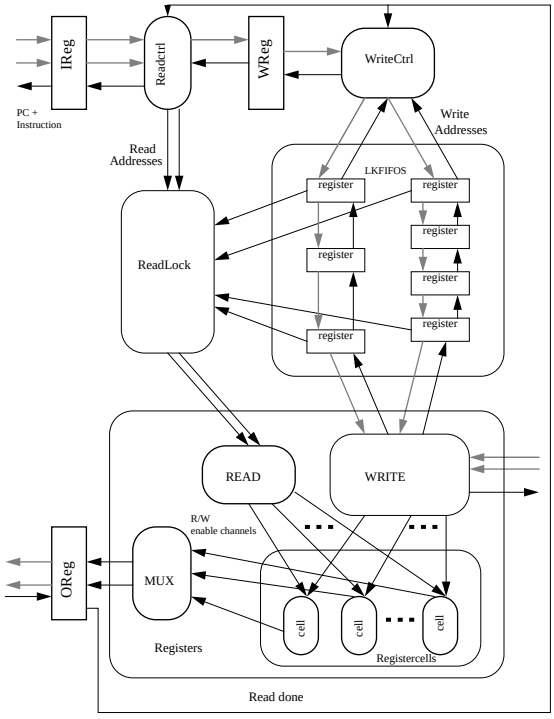
As explained in section 4, the asynchronous architecture can be tested for deadlocks by executing the simulator more than once, with a different event order each time. In a time driven simulation approach (sequential or parallel), where the simulated time is the synchronizing force, it is the actual delays (in simulated time) within the processes of the simulator that need to be modified in order to change the order in which events will occur in the simulation model. In the simulation approach proposed, the occam processes of the simulator are entirely data driven. The order in which the ALT construct, that models an arbiter, consumes messages is completely independent of the timestamps of the messages. Therefore changing the simulated time delays of the processes of the occam model would have no effect on the ordering of events in the simulation. The ordering of events can be changed by using the real execution time to affect the scheduling of the occam processes, i.e. the order in which the different processes are executed and produce messages. Occam supports a very simple mechanism for this, namely the **Timers**. Timers are special purpose channels that may be used to return the value of the local real time clock, or to delay the execution of the process until some time in the future. For example the statement *clock?time* (where clock is a Timer channel) returns the value of the local clock in the variable time, while the command *clock?AFTER T* will cause the process to be delayed until the value of the real-time clock is greater or equal to T. This technique does not allow full control of the process scheduling mechanism as the time that a process can be delayed is only approximate. Nevertheless, this approach provides a flexible mechanism for testing the delay insensitivity of the design. By using different benchmark programs, different paths of the design may be activated. By altering the order in which occam processes are executed for a particular benchmark, possible deadlock situations in the design can be located.

6. OCCARM: An occam simulation model of the asynchronous ARM

Using the approach described in the previous sections, a simulation model of the AMULET1 processor has been developed. The model consists of a hierarchy of occam processes; it has the same degree of parallelism as the microprocessor and executes ARM machine code. Figure 9 shows the top level of the simulation model. The Address Interface operates autonomously and produces memory addresses starting from address 0; these addresses are synchronised within the Data Interface with output data from the Register Bank and are sent to memory. Input data from memory is directed to its appropriate destination depending on whether it is a data value or an instruction. The datapath control consists of three decode stages. Each stage includes a number of PLAs which generate the control signals required for the operation of the system; in the simulation model the PLAs are implemented as arrays of boolean values. Decode1 controls the operation of the Register Bank while Decode2 controls both the Shifter and the Multiplier(DPEs). Decode3 controls the operation of the ALU; this is where the condition flags are checked, possibly changing the program flow. Invalid prefetched instructions

Figure 9: OCCARM top level

following a branch are identified by comparing a parity flag in the ALU (the PCpar, changed every time a branch is taken) with the corresponding flag of the instruction. To make this mechanism more efficient an event is sent to Decode1 each time the parity changes so that instructions are discarded at the top of the pipeline. In the original design of Decode1 the timing characteristics of the processor ensured that the parity event would reach Decode1 before the instruction stream changes. However the nondeterministic nature of the occam model can not guarantee this; the PCpar channel may fire after the instruction stream changes thus discarding valid instructions. In this case, the modeled design of this particular piece of the architecture had to change to eliminate the time dependent behaviour and ensure asynchronous operation. The Register Bank has been a challenging part of the design; the concurrent, asynchronous nature of the processor introduces numerous operational problems: a) there may be multiple outstanding write operations, b) reads from registers with pending writes must be blocked and c) asynchronous reads and writes must interact correctly. A novel mechanism has been designed to resolve all these problems, namely the *lock fifo* [19] (figure 10a [8]). The lock fifo keeps decoded write register addresses, each address containing at most one "1" whose position indicates the locked register; by OR-ing a column the status of the corresponding register can be decided. It is clear that this mechanism is based on the global state of the lock fifo. The simulation model should effectively describe the asynchronous operation of the Register Bank at the same level of concurrency, and accurately model the behaviour of the locking mechanism. This is not straightforward as the distributed nature of occam does not allow processes to share global variables. The solution adopted for the implementation of the model is shown in figure 10b where the complete Register Bank model is depicted. An extra occam process (ReadLock) is used to hold information regarding the contents of the lock fifo and thus to achieve the register locking. Each time a write address enters the lock fifo, it is also sent to ReadLock process by the first process of the lock fifo; the process will not acknowledge back until the address has been received by the ReadLock. Similarly the last process of the lock fifo will signal the removal of its contents from the ReadLock once the write has been



b) Register Bank model

Figure 10: Register Bank

completed. This way, ReadLock always includes a local copy of the locked addresses and reads and writes can be performed concurrently and asynchronously when there is no interdependency. Since the simulation model is identical to the architecture in terms of concurrency, number of stages and interconnection pattern, the correct operation of the network of occam processes implies the correctness of the corresponding network of micropipelined stages. A deadlock in the simulation model directly points to a deadlock in the processor design.

6.1. Timing

The two arbiters of the system reside in the Write Control, where the data streams from the memory and the execution unit are merged, and in the Address Interface, where addresses from the prefetch unit(Incrementer/PC Pipe) are merged with addresses produced by the execution unit for branches or load/store operations. Figure 11 presents the model of the Address Interface (the merge for the Apipe and the LSMreg is completely data dependent and therefore deterministic). The Marreg register holds the address being sent to the memory. If the address is a PC value it is also circulated through the prefetch unit back to the AddC. The local clocks of the processes for the execution and prefetch units will become too skewed (and thus the timing error large) only if the ALT in the AddC process selects the same source channel a large number of times before accepting a message from the other one (assuming of course that both the corresponding units of the processor produce requests). However the self regulating nature of the design will prevent such a situation. The prefetch unit stops sending messages to the AddC when the PC Pipe becomes full. The PC Pipe acts as a *throttle* with its length (N=2) determining the maximum number of instructions that can be outstanding at any particular time (N+3). If the execution unit is stalled as a result of its output not having been read by the

Figure 12: Simulator structure

AddC, the PC Pipe will become full after issuing 5 instruction addresses, allowing the ALT in the AddC to select its other input channel.

Similarly, even in the case of many consecutive load/store instructions, the AddC will not accept more than 5 consecutive messages from the execution unit as this is the maximum number of outstanding instructions allowed. Therefore, even in these worst case scenarios the timing error will be small and limited. In the general case the timing error will be even smaller, as the processes of the simulation model will be evenly scheduled and executed and thus the production of messages will be balanced.

Similar arguments apply for the arbiter in the Write Control process, but in this case the pipes of the Data Interface and Register Banks act as throttles.

7. Simulator environment

The interface to the OCCARM model is depicted in figure 12. The *I/O process* serves as the interface to the outside world since, within the INMOS occam toolset environment, only one occam process may have access to the host machine [14]. An extra *Monitoring process* is used to collect monitoring information provided by OCCARM.

7.1. Monitoring

To evaluate the performance of the architecture being modeled, values such as *occupancy*, *utilization* and *throughput* as well as *idle* and *overload* states of the pipelines in the design need to be measured. These values are calculated by the processes of the model and are sent to the Monitoring process over extra monitoring channels. Idle and overload situations of registers can be detected by comparing the timestamps of the incoming request and acknowledge messages: if $timestamp(R_{in}^{n}) > timestamp(Ack_{out}^{n-1})$, the register has been idle for the period $timestamp(R_{in}^{n}) - timestamp(Ack_{out}^{n-1})$. An overload situation has occured if $timestamp(R_{in}^{n}) < timestamp(Ack_{out}^{n-1})$.

The occupancy of a N stage pipeline indicates the proportion of time the pipeline has 0,1,2,..N values in it. Its calculation requires knowledge about both the input and output rates of the pipeline, which is not directly available since occam does not support global variables.To overcome this problem a solution has been devised whereby request messages entering the pipeline carry with them an extra timestamp denoting the time of their entry. Using this information, the calculation of the pipeline occupancy by the control process at the output side is straightforward.

For deadlock detection, it is essential to know the state of the processes of the model at the time when the deadlock occurred. One way to achieve this is to keep traces regarding the communication activity of the occam processes. In the OCCARM model, each process sends to the monitoring process messages regarding the success of its communication operations (e.g "waiting on channel **x**" or "received from channel **x**"). The detection of deadlocks using these messages is straightforward. In order to decide the cause of a deadlock, only the immediate past of the processes needs to be known. For this reason, circular buffers are used within the monitoring process to hold the most recent messages. If the monitoring process does not receive any messages for a certain period of time as a result of a deadlock, the contents of the buffers are flushed through the I/O process to a log file. An occam **variant** protocol allows different types of messages to be transmitted over the monitoring channels.

7.2. Host machine

The system used to host the simulation model is the ParSifal T-Rack [2], a reconfigurable 64-Transputer[2] machine, which has been developed at the University of Manchester to support the parallel simulation of computer architectures (figure 13). Two of the four links from each transputer of the T-Rack (*link0* and *link1*) are permanently hardwired to form a processor chain known as the *necklace*. The off-necklace links(*link2* and *link3*) may be connected by means of a crossbar switch, into a configuration which is appropriate for the code being executed. The crossbar switch is built using twenty six INMOS C004 switch chips housed on two boards (*S1* and *S2*).

[2]Transputer is a registered trademark of INMOS Group of Companies

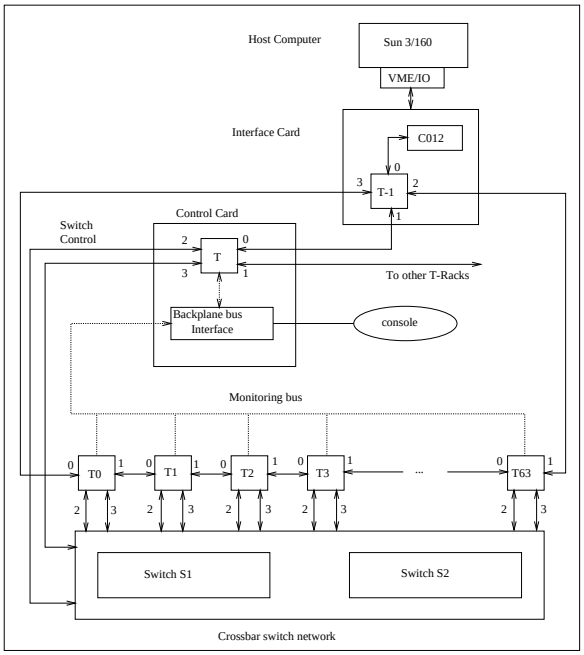


Figure 13: The T-Rack

8. Distributed Implementation

For the distribution of the OCCARM model onto the transputers of the T-Rack a number of issues need to be considered. The four links of the transputer limit the maximum number of processes with which any process at the top level of an occam program can communicate. To make the mapping of the OCCARM onto the transputers of the T-Rack feasible, the top level of the OCCARM model as depicted in figure 9 should be modified so that each process has at most four neighbours. Figure 14b depicts the modified top level process structure graph of the simulator (as derived from the table of figure 14a) where every process has at most four neighbors; the traffic on each edge of the graph(i.e. number of messages) is also presented.

8.1. Load Balancing

Since the top level process structure graph of the simulator is fixed and, in this particular case, unique, the load balancing criteria which were used for the mapping of the simulator onto the T-Rack in order to achieve high performance are:

- maximum processor utilization: occupy as many processors as possible.
- balanced communications: In the T-Rack, the hardwired links of the necklace are twice as fast as the links connected through the crossbar switch. Therefore channels with high traffic or channels that are part of the critical path of the asynchronous architecture (e.g channel "**H**" in figure 14b, via which the abort signals are sent from memory) should be mapped onto hardwired links.

Following these criteria and taking into account the T-Rack restrictions regarding the fixed links of the necklace and the limited connectivity of the switched links [17], the mapping depicted in figure 15 can be obtained.

Figure 14: OCCARM Process Interconnections

8.2. Monitoring path

To collect monitoring information within the distributed environment, the monitoring process of figure 12 has been replicated so that a copy of it resides on each of the transputers used. Extra multiplexing/demultiplexing processes have been introduced to allow the sharing of the transputer links. The monitoring information impose an extra communication burden to the simulator. To minimise the communication bottlenecks due to the monitoring messages, the characteristics of the T-Rack may be exploited. Since the *tadpole* transputer where the I/O process resides is connected to both ends of the necklace, the I/O operations of the OCCARM can be performed via one end of the necklace while the monitoring messages may follow the other direction towards the other end of the necklace. Figure 16 presents this scheme.

8.3. Performance

OCCARM executes an average of 20 ARM machine instructions per second when it runs on a single transputer of the T-Rack. This number is only slightly greater than that of an equivalent sequential simulator written in ASIM, the ARM's in-house simulation language, and executing on a SPARC[3] processor. This is an expected performance, for the execution of the parallel processes on a *single* Transputer is actually sequential and the large number of processes in the model make the context switching overhead in the Transputer significant.

The distribution of OCCARM onto the seven Transputers of the T-Rack yields an average speedup of 1.7. This figure may be attributed to a number of factors related to the characteristics of both, the simulated architecture and the machine that hosts the simulator. The requirement for instruction compatibility with the synchrononous ARM, has resulted in an asynchronous design with very low parallelism (and thus limited parallelism to be exploited in accordance with Amdahl's law) and very complex modularity; the performance of AMULET1

[3]SPARC is a registered trademark of Sun Microsystems, Incorporated

Figure 16: Mapping of the Simulator onto the T-Rack

is indeed lower than that of the synchrononous ARM by a factor of 0.4 [9]. Asynchronous architectures are communication bound systems, therefore the efficiency of the communication system is crucial; the complex irregural interconnection pattern of AMULET1's functional modules and the extra multiplexing/demultiplexing processes required due to the connectivity constrains of the Transputer and the T-Rack severely reduce communication efficiency. Currently a number of asynchronous architectures are under development with high degree of parallelism and regural interconnection patterns [16]. These characteristics coupled with the communication efficiency of the T9000 Transputer will allow the development of high performance distributed occam models.

9. Conclusions

This paper has described an approach for building parallel distributed models for asynchronous computer architectures. This approach exploits the parallelism inherent in the asynchronous design as well as the close relation between the semantics of the occam programming language and the structure and behaviour of asynchronous systems to allow the rapid development of distributed simulation models.

Occam can describe asynchronous circuits at a fairly low level and can provide guidance for the realization of the design (e.g. an IF statement will correspond to a SELECT block, a PAR of input commands will be implemented using a Muller-C block etc).

Furthermore, the parallel, distributed nature of occam forces the designer to think in "asynchronous terms" and to perceive its design as indeed a distributed, asynchronous structure where a global state does not exist.

References

[1] Brunvand, E. and Sproull, R., “Translating Concurrent Communicating Programs into delay-Insensitive Circuits”, Technical Report CMU-CS-89-126, Carnegie Mellon University, April 1989.

[2] Capon, P.C., “ParSiFal: A Parallel Simulation Facility”, IEE Colloquium: The Transputer: Applications and Case Studies, IEE Digest, 1986/91.

[3] Chandy, K. and Misra, J., “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”, Communications of the ACM, Vol. 24, Number 4, April 1981, pp 198-206.

[4] Dijkstra, E., “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”, Communications of the ACM, Vol. 18, Number 8, August 1975, pp 453-457.

[5] Evans, J., “Structures of Discrete Event Simulation”, John Wiley and Sons, 1988.

[6] Fujimoto, R., “Parallel Discrete Event Simulation”, Communications of the ACM, Vol. 33, Number 10, October 1990, pp 31-53.

[7] Fujimoto, R. and Nicol, D., “State of the Art in Parallel Simulation”, Proceedings of the 1992 Winter Simulation Conference, pp 246-254.

[8] Furber, S., Day, P., Garside, J., Paver, N.,Woods, J.V,“A Micropipelined ARM”, Proceedings of the VLSI’93 Conference, September 1993.

[9] Furber, S., Day, P., Garside, J., Paver, N.,Woods, J.V, ”AMULET1: A Micropipelined ARM”, Invited Paper, COMPCON’94.

[10] Gopalakrishnan, G. and Jain, P., “Some Recent Asynchronous System Design Methodologies”, Technical Report UU-CS-TR-90-016, University of Utah, October 1990.

[11] Hoare, C.A.R., “Communicating Sequential Processes”, Communications of the ACM, Vol. 21, Number 8, August 1978, pp 666-677.

[12] Inmos Ltd, “Occam Programming Manual”, Prentice Hall, 1984.

[13] Inmos Ltd, “Transputer Reference Manual”, Inmos Limited, 650 Aztec West, Almondsbury, Bristol BS12 4SD, England.

[14] Inmos Ltd,“Occam 2 Toolset User Manual”, Inmos Limited, 650 Aztec West, Almondsbury, Bristol BS12 4SD, England.

[15] Jefferson, D., “Fast Concurrent Simulation using the Time Warp Mechanism”, Proceedings of the Conference on Distributed Simulation”, Society for Computer Simulation, January 1985.

[16] Jones, I. W., ”The Sproull Pipeline Processor (Sun Microsystems)”, Seminar, Department of Computer Science, University of Manchester, January 1994.

[17] Murta, A., “Tools for the Automated Configuration of a Transputer Network”, MSc Thesis, University of Manchester, October 1987.

[18] Neto, G.A., “Distributed Simulation Using Relaxed Timing”, Technical Report UMCS-91-2-1, University of Manchester, 1991.

[19] Paver, N., Day, P., Furber, S., Garside, J., Woods, J.V, “Register Locking in an Asynchronous Microprocessor”, IEEE International Conferenence on Computer Design, 1992.

[20] Mead, C. and Conway, L., “Introduction to VLSI Circuits”, Addison Wesley, 1980, Chapter 7, pp 218-254.

[21] Paver, N., “The design and Implementation of an Asynchronous Microprocessor”, Ph.D Thesis, Department of Computer Science, University of Manchester, 1994.

[22] Sutherland, I., “Micropipelines”, Communications of the ACM, Vol. 32, Number 6, June 1989, pp 720-738.

[23] Welch, P., Justo, G. and Willcock, C., “High Level Paradigms for Deadlock-Free High Performance Systems”, Transputer Application and Systems 1993, Proceedings of the 1993 World Transputer Congress, IOS Press, pp 981-1004.

[24] AMULET Modelling Workshop, Windermere, Cumbria, England, July 1984.