CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Simulating asynchronous hardware on multiprocessor platforms: the case of AMULET1[‡]

Georgios K. Theodoropoulos[†]

School of Computer Science, The University of Birmingham, Birmingham B15 2TT, U.K.

SUMMARY

Synchronous VLSI design is approaching a critical point, with clock distribution becoming an increasingly costly and complicated issue and power consumption rapidly emerging as a major concern. Hence, recently, there has been a resurgence of interest in asynchronous digital design techniques which promise to liberate digital design from the inherent problems of synchronous systems. This activity has revealed a need for modelling and simulation techniques suitable for the asynchronous design style. The concurrent process algebra Communicating Sequential Processes (CSP) and its executable counterpart, occam, are increasingly advocated as particularly suitable for this purpose. This paper focuses on issues related to the execution of CSP/occam models of asynchronous hardware on multiprocessor machines, including I/O, monitoring and debugging, partition, mapping and load balancing. These issues are addressed in the context of occarm, an occam simulation model of the AMULET1 asynchronous microprocessor, however the solutions devised are more general and may be applied to other systems too.

KEY WORDS: Asynchronous hardware, distributed simulation, CSP, occam, monitoring, load balancing

1. INTRODUCTION

A digital system is typically designed as a collection of subsystems, each performing a different computation and communicating with its peers to exchange information. Before a

[†]E-mail: gkt@cs.bham.ac.uk
[‡]Updated version to address reviews of 10 January 2001. Sent to Professor Anthony J. G. Hey, University of Southampton, 5 February 2001. Original sent to Professor Anthony J. G. Hey to consider for publication in the Concurrency-Practice and Experience Journal, 18 October 2000.

communication transaction takes place, the subsystems involved need to synchronise, namely to wait for a common control state to be reached, which guarantees the validity of data exchanged.

In synchronous systems, the synchronisation of communicating subsystems is achieved by means of a global clock whose transitions define the points in time when communication transactions can take place. The operation of a synchronous system proceeds in lockstep, with the different subsystems being activated to perform their computations in a strict, predefined order [54]. Synchronous VLSI design however is approaching a critical point, with clock distribution becoming an increasingly costly and complicated issue and power consumption rapidly emerging as a major concern.

Another digital design philosophy allows subsystems to communicate only when it is necessary to exchange information. The operation of the system does not proceed in lockstep, but rather is *asynchronous*; each sub-system operates at its own rate synchronising with its peers only when it needs to exchange information. This synchronisation is not achieved by means of a global clock but rather, by the communication protocol employed. This protocol is typically in the form of local request and acknowledge signals which provide information regarding the validity of data signals.

Although asynchronous design techniques have been explored since, at least, the mid 1950s [57, 22, 25, 43], they have not hitherto been established as a major philosophy in digital design. This failure was mainly related to the difficulty to enforce specific orderings of operations and to deal with circuit hazards and dynamic states in an asynchronous, non-deterministic environment [40]. However, recently, there has been a resurgence of interest in asynchronous design techniques, due to the significant potential benefits that the elimination of global synchronisation may offer to issues such as clock distribution, power consumption, performance and modularity [36].

Various asynchronous digital design techniques have been developed, which are typically categorised by the *timing model* (namely, the assumptions made regarding the circuit and signal delay), the *signalling protocol* (namely, the sequence of events which must take place in a communication transaction between two elements), and the technique they employ for the *transfer of data* between two elements (namely, encoding the value of each bit transmitted during a communication transaction). In his influential 1988 Turing award lecture, Ivan Sutherland introduced *Micropipelines*, a new conceptual framework for designing asynchronous systems [71]. In depth surveys of existing asynchronous methodologies may be found in [13, 40, 10]. Additionally, the Asynchronous Online Logic Home Page maintained by the AMULET group at the University of Manchester provides continuous, up to date information regarding asynchronous systems research [5].

A number of asynchronous architectures have been developed [86] including one at CalTech [52], NSR [15] and Fred [62] at the University of Utah, STRiP at Stanford University [27], Sun's Counterflow pipeline processor [69], FAM [20] and TITAC [59] at Tokyo University and Institute of Technology respectively, Hades at the University of Hertfordshire [30] and Sharp's Data-Driven Media Processor [66]. The AMULET group at the University of Manchester have developed a series of asynchronous implementations of the ARM RISC processor using Sutherland's Micropipelines. The AMULET1 [87] microprocessor has been the first asynchronous implementation of a commercial instruction set; AMULET2e [37] has sought to improve performance via an improved design while AMULET3i [38] aims at embedded applications.

The quest for the exploitation of the potential advantages offered by asynchronous logic has revealed a need for modelling and simulation techniques, which would be appropriate for the asynchronous design style. Thus, the recent interest in asynchronous design has fuelled an intense research activity aiming to develop techniques appropriate for modelling and simulating asynchronous systems. I-Nets [55], Petri Nets (e.g. the Petrify tool [24]), Signal Transition Graphs (STGs) [21] (e.g. the Versify [81] and SIS [65] tools), State Transition Diagrams (e.g. the MEAT tools [26]), ST-V (the Self-Time Verilog developed by Cogency Technology Inc. [23]) and CCS (by Birtwistle et. al. at the University of Leeds [51]) are some of these tools and formalisms that have been employed in asynchronous logic design.

Communication Sequential Processes (CSP) [41], in particular, the concurrent process algebra developed by Tony Hoare for the specification of parallel systems, has been extensively advocated as a suitable and natural means for describing asynchronous behaviour: CSP supports a concurrent, process based, asynchronous, non-deterministic model of computation which exactly matches the structure and behaviour of hardware built using asynchronous logic. Furthermore, in CSP, the communication between different modules is by means of a point-to-point, synchronised and unbuffered channel. This behaviour directly reflects the interaction between subsystems in asynchronous hardware, where a sender and a receiver rendezvous before they physically exchange data via wires, which are memoryless media. Several asynchronous modelling approaches and systems have been developed which use CSP-based notations, including Martin's [53, 52], Hulgaard's [42] and Brunvand's [14] work, *trace theory* [29], *Delay-Insensitive algebra* [45], *Tangram* [79], *SHILPA* [39], *LARD* [31] and Balsa [9].

However most, if not all, work undertaken so far in this area, has placed emphasis on producing specifications to be used as input to silicon compilers for the automatic synthesis of asynchronous circuits. An important issue, which has been largely overlooked and neglected, and which is addressed for the first time by the research presented in this paper, is simulation, namely the execution of such concurrent CSP models on a computer system. Aiming to contribute to this area, and motivated by the increasing debate regarding the potential use of CSP, in the context of the AMULET work, a research project has investigated the suitability of *occam* [44], the executable counterpart of CSP, for the modelling and distributed simulation of complex asynchronous architectures [74]. The investigation targeted asynchronous systems that are based on Sutherland's Micropipelines using the AMULET1 microprocessor as a testbed, however the results may also be applied to any asynchronous design methodology. As part of this project, a generic modelling methodology has been developed [76]; timing and synchronisation issues arising from the parallel semantics of CSP and occam have been addressed [77] and *occarm*, an occam simulation model of the AMULET1 microprocessor has been developed [74]. Complementing this work, this paper focuses on issues related to the actual execution of parallel, CSP/occam simulation models of asynchronous hardware.

In a sequential environment, the execution and testing of a simulation model written in a conventional sequential language is a straightforward issue. This however does not apply to asynchronous parallel models. The extra dimension that concurrency introduces to distributed systems complicates the programming activity and imposes a number of issues which need to be addressed before any parallel program may be executed. These issues include monitoring,

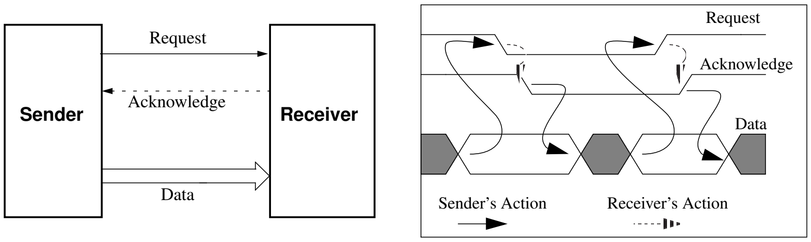


Figure 1. The Two-Phase Bundled Data Interface Protocol

debugging and terminating the processes of the system; for multiprocessor configurations, remote I/O, partitioning, mapping and load balancing issues need also to be addressed. This is particularly true for the development of occam systems where the aforementioned tasks are typically the responsibility of the application developer, and need to be explicitly addressed irrespective of whether the occam program is to executed on a single (internal concurrency) or multiple processors (parallelism in time). The paper discusses how the above issues have been addressed in the context of the occarm simulation model of AMULET1; however, the solutions devised may also be applied to the simulation of other asynchronous achitectures.

The rest of the paper is organised as follows: Section 2 provides a brief overview of Sutherland's Micropipelines. Section 3 provides a discussion of the role of parallel simulation in asynchronous VLSI design. Section 4 outlines a generic framework which to model asynchronous micropipelined systems using CSP/occam, while section 5 discusses the application of this framework for the development the occarm model of the AMULET1 microprocessor. Section 6 describes the multiprocessor platforms that hosts the simulation while sections 7 and 8 deal with monitoring and termination issues. Section 9 describes the simulation environment, section 10 discusses mapping and load balancing issues and section 11 deals with the validation of the model and presents performance results. Finally, section 12 epitomises the conclusions drawn.

2. MICROPIPELINES

Using Micropipelines, the asynchronous architecture is designed as a set of simple, data processing elastic pipelines, whose stages operate asynchronously and exchange data via a *two-phase bundled data* handshake synchronisation protocol (figure 1). Two-phase signalling recognises and responds to transitions of the voltage on a wire, regardless of whether the transition is rising or falling; a transition is referred to as an *event*.

Ivan Sutherland also proposed a set of event control blocks for the design of control circuits in micropipelined systems as well as event controlled storage elements to be used in such systems. The event control blocks include the *Muller-C*, *Select*, *Call*, *Toggle*, *Xor* and the *Arbiter* (figure

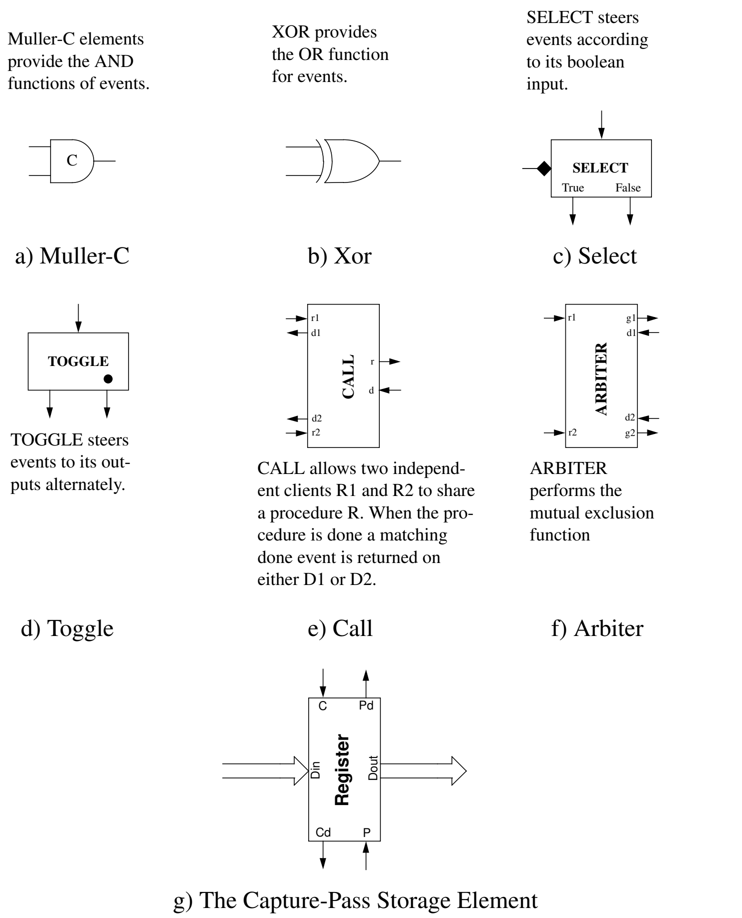


Figure 2. Event Processing Blocks and the Capture-Pass Storage Element

2). An event controlled storage element is the *Capture-Pass* latch, depicted in figure 2g. The latch is controlled by two control signals, namely *Capture* (*C*) and *Pass* (*P*). Initially the latch is in its *transparent* state, where the input is connected through to the output (i.e. *Din*=*Dout*). When an event is issued on the Capture wire the input-output connection is interrupted, the data is “latched”, and an event is issued on the *Cd* signal (*Capture done*) to indicate the change of state in the latch (i.e. from transparent to *opaque*); the latched data does not change with subsequent data input changes. When an event arrives on the Pass wire, the input is connected back through to the output, thus making the latch transparent again; this change is indicated

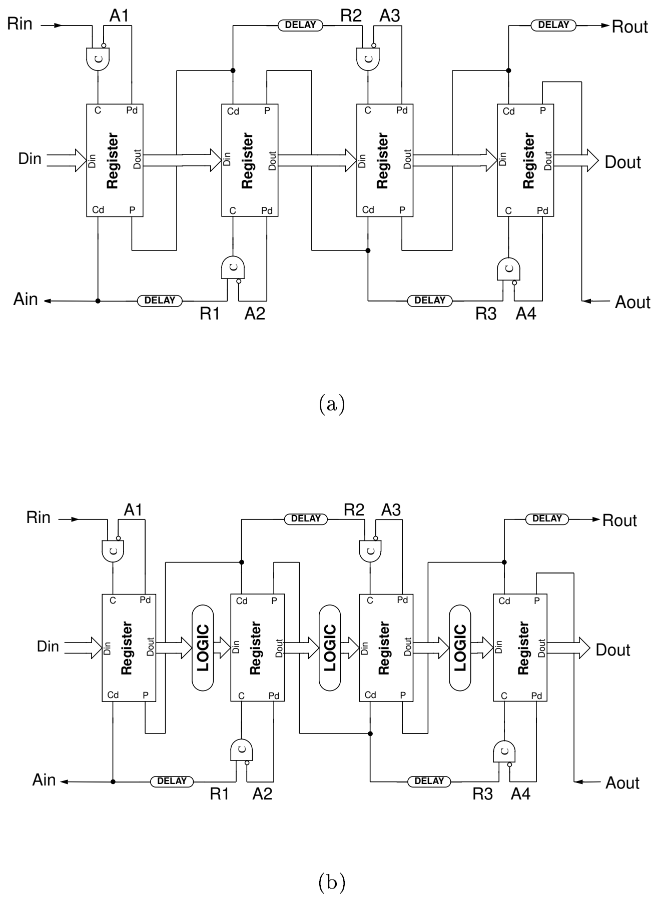


Figure 3. Micropipelines Without and With Processing Elements

by an event on the *Pd* (*Pass done*) signal. The Capture-Pass may repeat, with events arriving alternately on the C and P wires respectively.

The simplest micropipeline is a series of Capture-Pass registers connected together to form a FIFO structure as depicted in figure 3(a). A micropipeline may perform processing on the data, by interposing the necessary logic between adjacent register stages (figure 3(b)). A delay unit is typically used to slow down the request event and give the data enough time to arrive at the register before the request, thus guaranteeing the validity of data captured by the register (the *bundled data constraint*).

3. THE ROLE OF PARALLEL SIMULATION IN ASYNCHRONOUS HARDWARE DESIGN

Simulation modelling languages and tools for synchronous logic design have underpinned the development of ever more complex synchronous VLSI circuits. The simulation of digital systems in general, and computer architectures in particular, has long been categorized among the highly computation intensive applications. In the case of asynchronous systems, the role of simulation is even more crucial as their concurrent, non-deterministic behaviour makes any attempt to reason about their correctness and performance a very complicated task.

The concurrent behaviour of asynchronous systems renders them susceptible to deadlocks, a problem typical in parallel systems but not an issue in systems synchronised by a global clock and operating in lock step. Deadlock is a high-level issue of the design, and occurs when the system, as a result of a particular sequence of events, reaches a state wherein at least one sub-system becomes indefinitely blocked. Typically, the sequence of events in an asynchronous system is non-deterministic. This is due mainly to the behaviour of the arbiters which will typically service request events in arrival order; in the case of two requests arriving at the same time, the choice will be non-deterministic. Asynchronous logic allows variable delays within the different sub-systems, which will affect the order in which independent request events arrive at the arbiters of the system. The correct functionality of the asynchronous system should not depend on the ordering of independent streams of events; a correct design should be deadlock free for all possible combinations of events.

Verifying that a concurrent, asynchronous structure is deadlock free is a complex and difficult issue. Substantial research effort has been invested to develop formal methods which guarantee deadlock freedom [67, 64, 32]. However, existing formal techniques are not yet mature enough to tackle systems of the complexity of asynchronous computer architectures although research is ongoing in this area.

In practice, it is generally possible to identify, and thus avoid, certain design decisions that are susceptible to deadlock [61]. However the size, complexity and the non-deterministic behaviour of asynchronous hardware systems do not allow intuition to guarantee a deadlock free design.

Simulation can be an invaluable aid for this problem*. The approach is to run the simulation model of the system many times, each time with a different set of delays in the component sub-systems [36]. Changing the internal delays of the sub-systems, changes the order in which events are generated. Consequently, the order in which events from different data streams arrive at the arbiters also 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. In this context the speed

*A different approach would be to guarantee deadlock freedom by construction, namely, by applying certain rules during the design of the system [84]. However, the applicability of this approach for the design of asynchronous systems has not yet been investigated. Such an approach, which would attempt to exploit the characteristics of asynchronous hardware systems in order to establish design rules that guarantee deadlock freedom would be extremely important and would contribute enormously to asynchronous digital system design.

of simulation is crucial, as a fast simulator would allow the delay independence of the system with regard to deadlocks to be rigorously and extensively tested for a large number of possible combinations of events.

Furthermore, performance evaluation is a much more complex task in asynchronous systems than in their synchronous counterparts. In the latter, benchmark execution times are easy to interpret based on the number of clock cycles and the existence of a critical path. Delays in the critical path can determine the clock period while non-critical path delays have no effect on the performance of the system. In contrast, the temporal behaviour in asynchronous systems is much more difficult to understand and interpret as delay inter-dependencies are much more complex. Delays in one module may often be masked by occasional longer delays in another module, while the accumulation of delays through a chain-reaction in a non-deterministic concurrent environment may have a chaotic effect on system performance. The need to evaluate the asynchronous architecture for different sub-system delays further complicates the process rendering simulation speed a crucial element.

Hence, a parallel approach to simulation, could potentially achieve high simulation performance and contribute significantly in reducing the duration and cost of the design cycle. Indicatively, for the testing and evaluation of the AMULET1 design more than 4 million instruction cycles were simulated [61], a number which corresponds to several hours of simulation.

Asynchronous hardware systems are an excellent candidate for parallel simulation. The concurrent operation of the different subsystems of an asynchronous system, the inherent parallelism within each subsystem and the lack of any global synchronisation, are characteristics which support the concurrent execution of events in a simulation model. In his *flashback simulation* approach [72], Sutherland attempts to exploit these characteristics of asynchronous systems and allow "out-of-order" processing of events to increase simulation speed; however, his simulation retains its sequential nature, and is intended for execution on conventional von Neumann computers.

Assuming a correct implementation of the communication protocol, at the Register Transfer or higher levels, an asynchronous system may be viewed as a network of concurrent modules communicating via synchronous, unbuffered communication. The modules are data-driven; each module will start computation as soon as data is available on its input wires, and will signal when the result has been computed. Using occam, the asynchronous system may be modelled at the Register Transfer as a network of concurrent occam processes, topologically identical to the asynchronous system, with each occam process corresponding to a different functional module of the system, and communicating with its peers via timestamped messages. The asynchronous communication protocol is modelled implicitly in the semantics of the occam channel. This approach is analogous to the "Logical Process Paradigm", typically employed in distributed simulation modelling [35, 33].

Occam is primarily a general purpose parallel programming language. Thus, a specification developed using occam is automatically an executable simulation model of the asynchronous system. No extra simulation engine is required. This model, which is in the form of a network of occam processes, may be executed on a multiprocessor machine to achieve high performance. Furthermore, occam allows explicit description of parallel as well as sequential computation. This explicit control of concurrency which extends down to the command level, along with

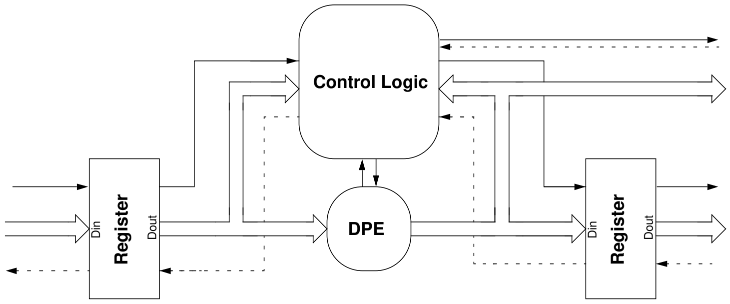


Figure 4. Micropipeline With Processing: A High Level View

its simple but powerful syntax and "send" and "receive" commands, makes occam ideal for describing digital systems (indeed, occam has been employed for modelling digital systems at various levels, e.g. [2]).

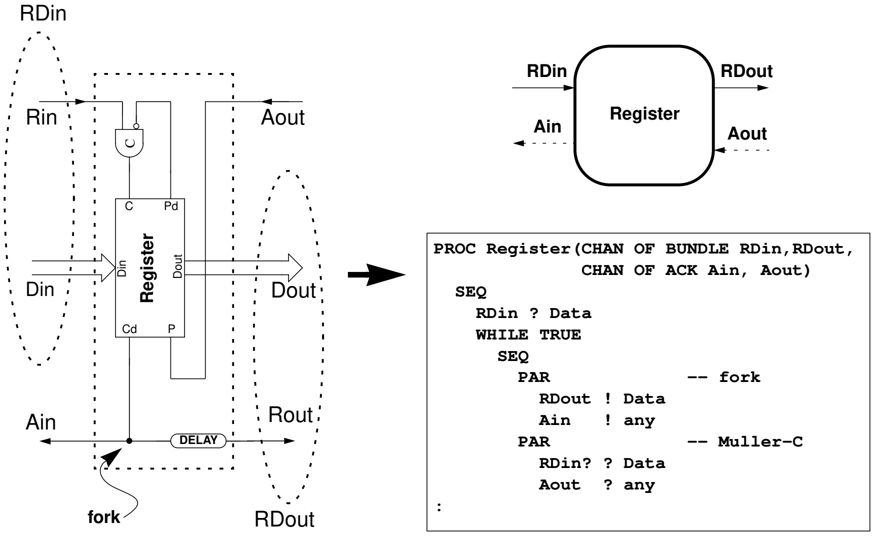
4. MODELLING MICROPIPELINES

At the Register Transfer Level, a Micropipeline with processing may be generally viewed as depicted in figure 4. The sending register outputs its contents, consisting of data and control bits, onto the data bus and produces a request event (request wires are indicated in the figure by solid lines, while acknowledge wires are denoted by dotted lines). The control bits, are used by the control logic to direct the request event to its correct destination, activating if necessary the data path elements (DPEs, e.g. ALUs, multipliers, shifters etc.) of the circuit. Data passes through the DPEs and propagates to the next stage. This general Micropipeline may be modelled by three occam processes, two for the registers and one for the control/data processing logic; the control logic and the DPE may be modelled as one process, with the DPE being a procedure called by the control process.

Figure 5 illustrates the register occam model. The model makes use of two PAR statements, one to model the Muller-C element and one to model the fork on the Ain/Rout wire. Two channels are used in each direction, one for the data/request bundle and one for the acknowledgement signal. The latter is required to keep the register processes tightly coupled and synchronised, as in a different case, the control process would act as a buffer introducing an extra pipeline stage in the model that does not exist in the physical system. In the case of pipelines without processing, the acknowledgement channel is not required.

A multi-stage Micropipeline may be modelled by means of a parallel replication of the register process.

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




```
PROC Register(CHAN OF BUNDLE RDin,RDout,
              CHAN OF ACK Ain, Aout)
  SEQ
    RDin ? Data
    WHILE TRUE
      SEQ
        PAR                -- fork
          RDout ! Data
          Ain   ! any
        PAR                -- Muller-C
          RDin? ? Data
          Aout  ? any
:
```

Figure 5. Micropipeline With Processing: The Register Model

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* (parallel execution) and *SEQ* (sequential execution) commands being used within each process to implement the partial ordering of events of the circuit (internal concurrency). The number of these processes depends on the degree of modularity and fidelity required in the simulation model.

Adopting a data driven approach to model asynchronous systems, it is essential to have a mechanism for modelling the functionality and the nondeterministic behaviour of arbiters. The occam *ALT* construct provides for the non-deterministic choice of messages from different channels and therefore may effectively model the behaviour of an arbiter, however it introduces synchronisation problems. A detailed discussion of these problems is outside the scope of this paper and may be found in [74, 77]

5. MODELLING AMULET1: THE OCCARM MODEL

The modelling framework outlined above has been used to develop *occarm*, an occam simulation model of the AMULET1 microprocessor (the name of the model is derived from the combination the words **occ**am and **ARM**). Figures 6, 7 and 8 illustrate respectively the interface, the internal organisation and the physical layout of the 1.2 micron implementation of AMULET1. The processor comprises five major units, namely the *address interface*, the

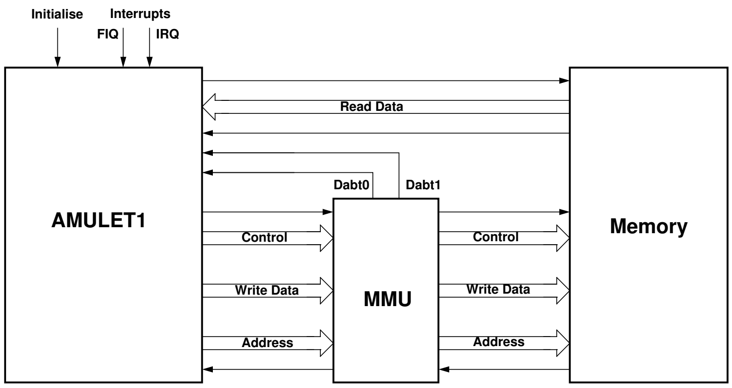


Figure 6. The AMULET1 Interface

data interface, the *execution unit*, the *register bank* and the *primary decode*. The execution unit consists of two major stages, namely *Decode2* (*Dec2-Ctr12*), which controls the operation of the shifter and multiplier units of the processor, and *Decode3* (*Dec3-Ctr13*), which controls the ALU. Detailed descriptions of the AMULET1 microprocessor are provided in [87, 61].

Occarm describes AMULET1 at the Register Transfer Level. It executes ARM6 machine code produced by a standard ARM compiler. Instructions enter the simulator as 32-bit integer numbers in hexadecimal format. Instruction decoding is performed by means of PLA models, which are implemented as two-dimensional arrays of Boolean values.

Occarm has been implemented as a hierarchy of occam processes, with each process modelling a different functional module of AMULET1. Its top-level process structure graph is depicted in figure 9, while figure 10 illustrates its internal structure (the reader is invited to relate these two figures with figures 7 and 8). *AddInt* and *DatInt* processes model AMULET1's address and data interface units respectively. The datapath is modelled by four processes, namely *Decode1*, *Decode2*, *Decode3* and *RegBank*. Decode1 describes the primary decode unit while Decode2 and Decode3 model the two major components of the execution unit of the processor. RegBank process incorporates the functionality of the register bank. *WrtCtrl* models the operation of AMULET1's write bus arbitration logic.

All the registers of AMULET1, have been modelled using the generic register model, with interprocess communication being performed using pairs of request/data (solid arrows in the figures) and acknowledgement (dotted arrows) channels. To illustrate the modelling of control logic, figures 11(a) and 11(b) depict the modelling of one of the control modules of the Address Interface (AddC).

For a detailed description of the structure and operation of occarm the reader is referred to [74].

Prepared using cpeauth.cls

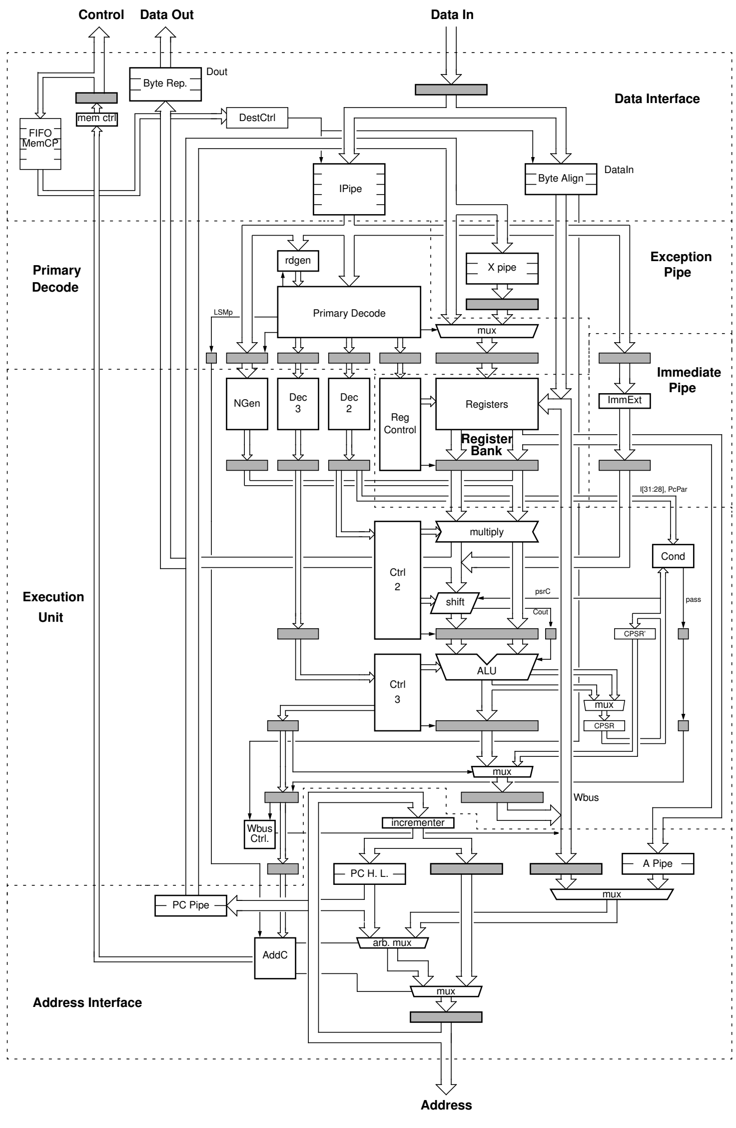


Figure 7. The AMULET1 Internal Organization

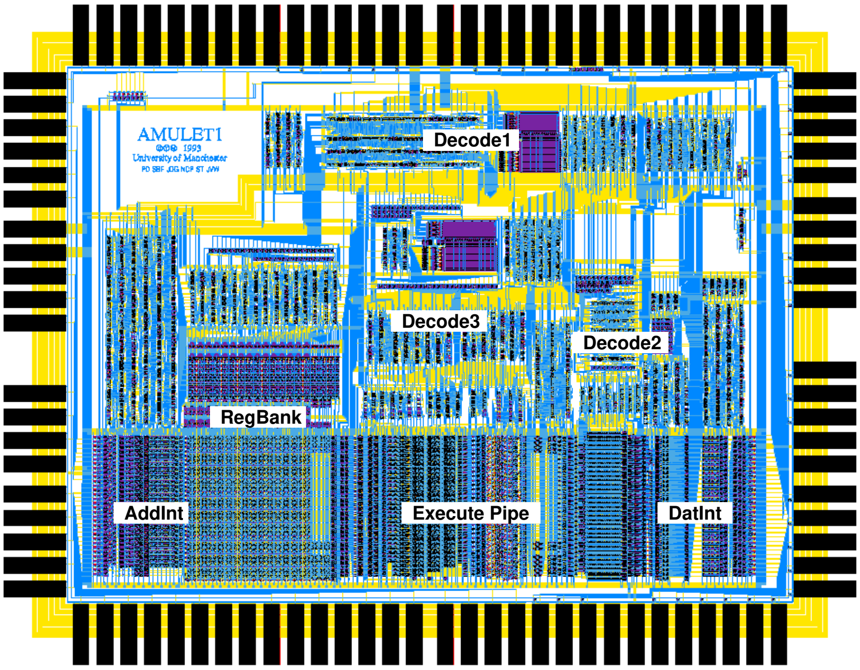


Figure 8. The AMULET1 Processor Physical Layout

6. THE SIMULATION HOST MACHINE: THE PARSIFAL T-RACK

The computer which hosts the occarm simulation model is the ParSiFal T-Rack [18], a reconfigurable, transputer based machine which has been developed at the University of Manchester as part of the *Parallel Simulation Facility* project under the U.K Alvey program. The T-Rack was developed to serve as a facility for the parallel simulation of computer architectures.

The basic architecture of the T-Rack is illustrated in figure 12. It comprises sixty four T800 transputers (*T0-T63*), each with one or two Megabytes of local dynamic memory. The transputers are housed on sixteen identical boards (four transputers per board). Each transputer communicates via four asynchronous bidirectional links numbered from 0 to 3. 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 which is built using twenty six INMOS C004 switch chips housed on two boards (*S1* and *S2*, also referred to as *near* and *far* boards respectively [58]). Connections within the switch networks may be defined using a software

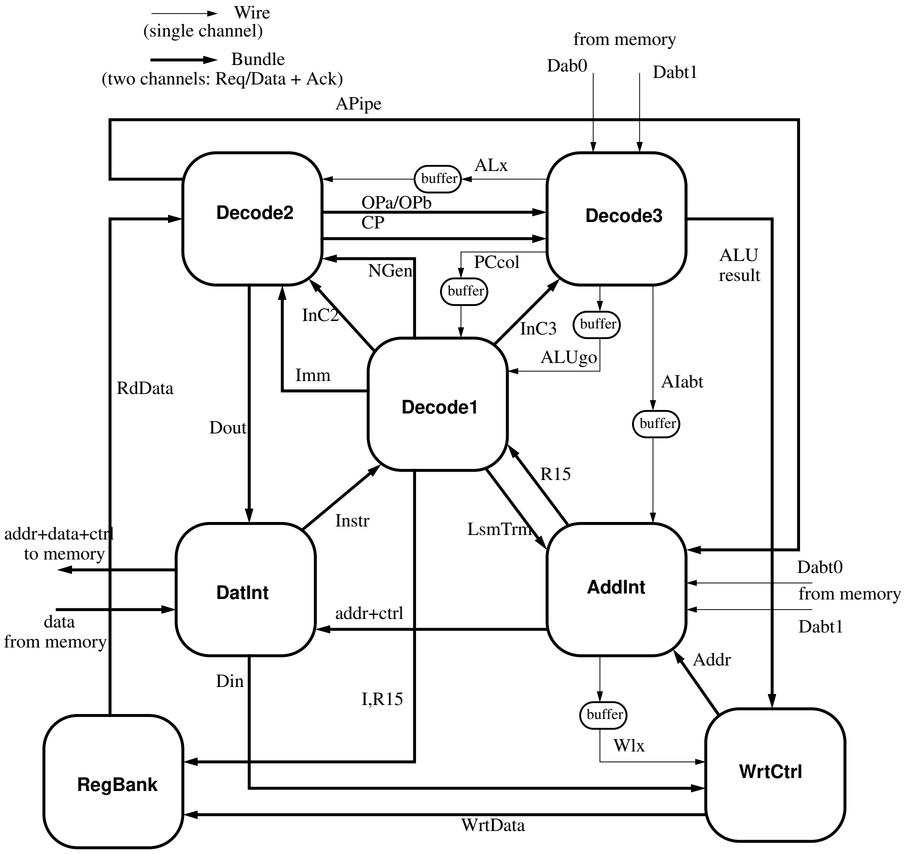


Figure 9. Occarm Top Level Process Graph

switch utility allowing the transputers to be connected to form the configuration required. The switches are statically set before the application is loaded, though dynamic reconfiguration is also possible. The crossbar switches also provide for the connection of transputer links to external devices. The T-Rack is hosted by a Sun workstation containing a *Tadpole Transputer Board* which acts as a "root" node for the T-Rack and is used for the downloading of code on to the rack and for I/O operations to and from the host machine. The Tadpole transputer forms part of the necklace (T-1, see figure 12). The control board is used for switch control functions and for system backplane monitoring which is achieved by means of a byte-wide monitoring bus. This bus provides an alternative route between the T-Rack and the outside world, as a terminal may be attached to the control board to display low level monitoring information [47]; the occarm simulator does not make use of this bus.

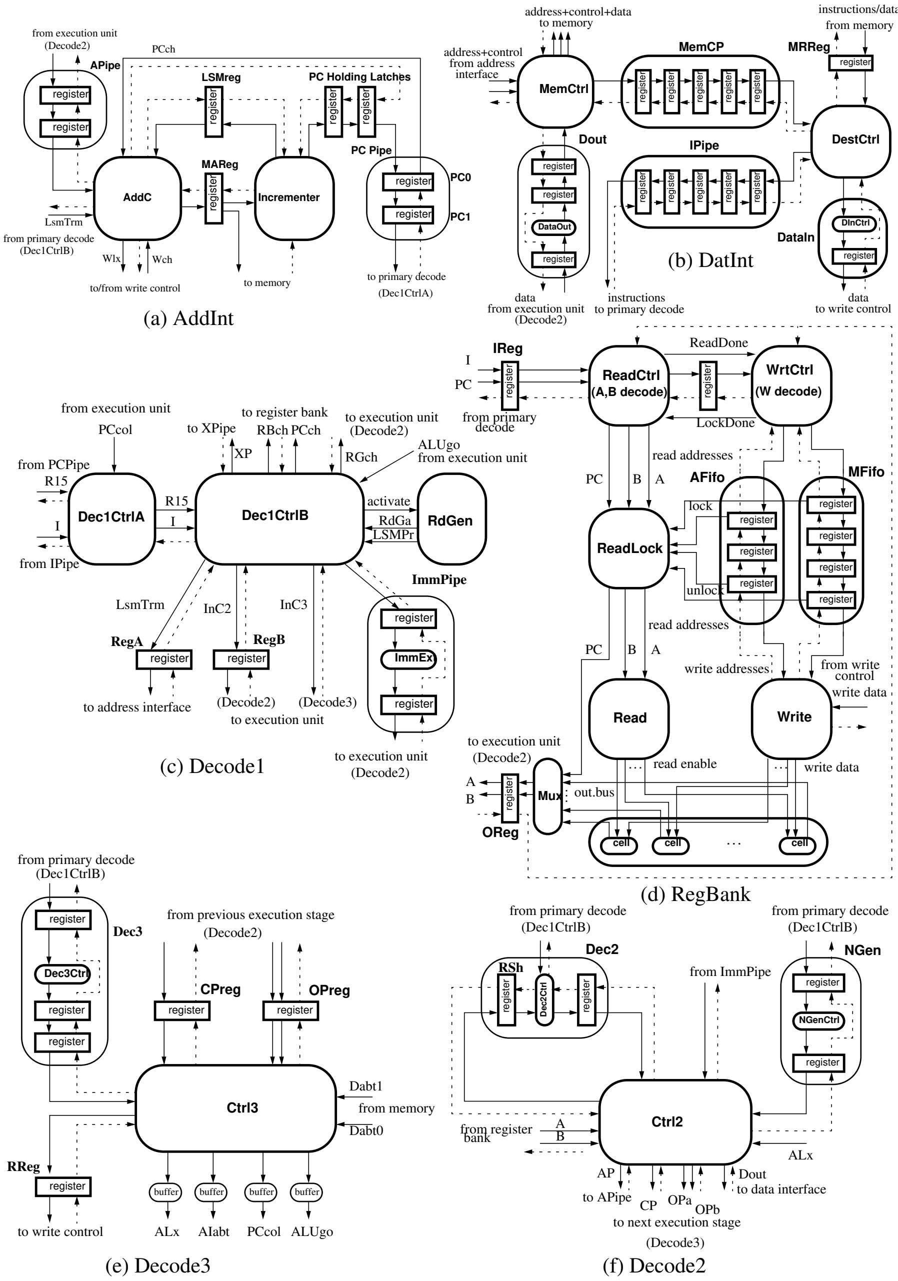
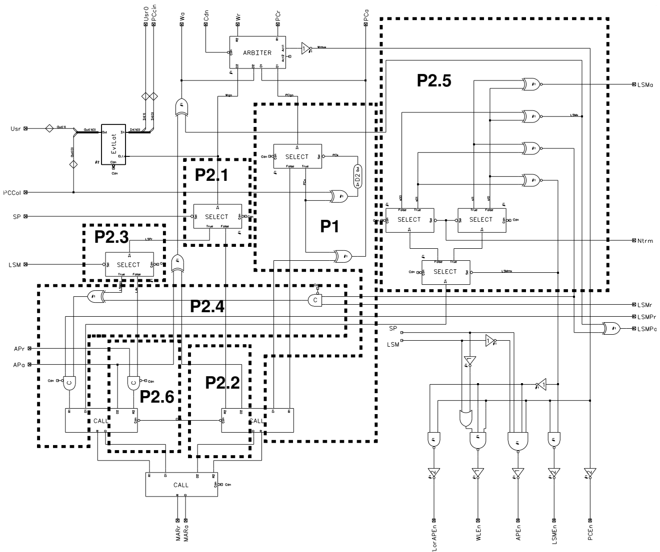


Figure 10. Occarm Internal Structure



(a)

```
SEQ
  ALT              --ARBITER
    PCr?
      P1           --PC loop
    Wr?
      P2.1         --IF
        FALSE
          SEQ
            P2.2   --data transfer
        TRUE
          SEQ
            P2.3  --IF
              TRUE
                SEQ
                  WHILE Ntrm=FALSE  -- LSM loop
                    SEQ
                      P2.4
                      P2.5
              FALSE
                SEQ
                  P2.6 --data transfer: Apipe
:
```

(b)

Figure 11. AddC: Control Circuit and Occam Process

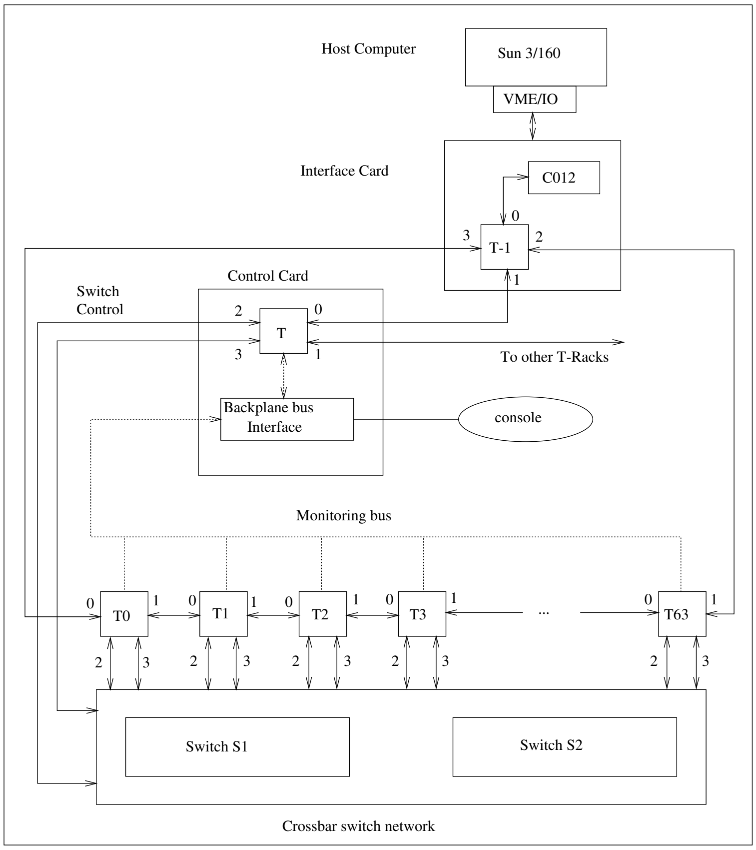


Figure 12. The T-Rack

The existence of the necklace in the T-Rack limits the set of processor graphs which may be implemented to those which possess a Hamiltonian Cycle. An occam program can execute on the T-Rack if the required processor network, either contains, or can be modified to contain a Hamiltonian Cycle. The route taken by the Hamiltonian Cycle through the network corresponds to the necklace of the T-Rack while the edges not on the Hamiltonian Cycle are mapped on switched links. The switching network implements a "split-link" switching policy, whereby link 2 outputs are connected to link 3 inputs via one switch board, while the connection between link 2 inputs and link 3 outputs is achieved via the other switch card. The split link switching mechanism provides a solution to the *Odd Cycle problem* which does not allow the construction of networks that possess an off-necklace closed path (cycle) which has an odd number of edges [58].

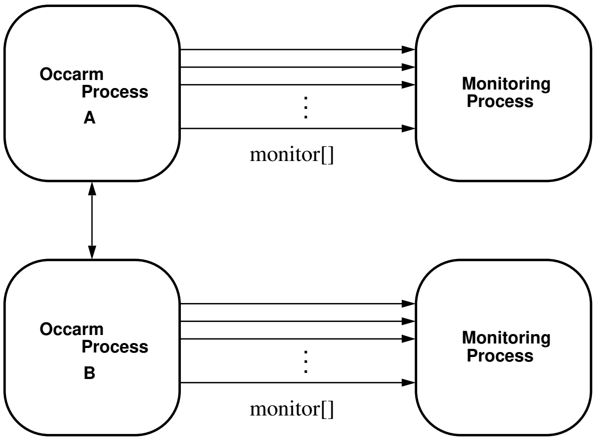


Figure 13. Collecting Event Traces in Occarm

7. MONITORING

Monitoring the runtime behaviour of the simulation model and collecting information regarding the characteristics of the simulated system is one of the main objectives of the simulation process. Monitoring is essential for the testing and performance evaluation of the simulated system as well as for the debugging of both the simulated system and the simulation model. The inherent properties of distributed asynchronous systems make monitoring a difficult and complicated issue for which sequential techniques are insufficient. The fundamental problem is the difficulty to deal with causality and obtain snapshots in a distributed environment [48, 19, 7]. To determine the system state, all the different local process as well as channel states need to be taken into account; the monitoring system should be able to correlate the histories of the different processes and put them in a global temporal perspective. The main issues which stem out of this fundamental problem and which an ideal distributed monitoring system should address include the multiple threads of control, intrusiveness, non-determinism and the need to cope with (i.e. generate, transport and analyse) a vast amount of monitoring data. For a detailed discussion of these issues the reader is referred to e.g. [46, 63].

Within the ParSiFal project, a number of experimental graphical tools were developed which illustrate or monitor different aspects of an occam parallel program (e.g. [70]); these however were not used in connection with occarm due to their limited capabilities and the need for portability of the occarm simulation environment. In occarm, monitoring is performed by means of a time-parallel network of occam monitoring processes (known as *reactive processes*), one for each of the top level processes (the *active processes*) of the occarm model, as depicted in figure 13. Clearly, as is typical in software monitoring, monitoring processes make use of the host machine's resources, thus imposing delays in the simulator's execution; if a reactive process is mapped on the same processor as its active process, the host machine will execute alternately simulation and monitoring code (e.g. in a time-sliced fashion). Delaying a process does not only have an impact on the performance of the system but it alters the event ordering in the distributed system in an arbitrary way, thus changing its behaviour (intrusiveness). In the context of asynchronous hardware such intrusiveness is even more crucial as, for instance, a delay introduced by a monitoring process may hide a deadlock situation which would otherwise occur. However, as explained in section 3, the multiple executions of the simulation each time with a different set of delays in the component sub-systems alleviates this problem.

Monitoring messages are issued to the monitoring system in an event driven fashion, namely the active processes detect and report to the reactive processes event records on their own initiative. To enable the model to detect the events to be reported, source-code instrumentation has been performed. Probes, in the form of a procedure call, have been inserted (manually) in the source occam code of the active processes. Each time the procedure which implements the probe is called, it constructs the corresponding monitoring message and sends it to the monitoring process. Since probes will be invoked in different parallel sections of the active process, several monitoring messages are issued simultaneously. Thus, for the communication between an active and the corresponding reactive process, a channel array is used (*monitor*, see figure 13); for the current implementation of occarm the total size of the monitoring channel array is fifty (50). The monitoring process acts as a multiplexor, employing an ALT construct to gather the messages issued by the corresponding active process on the channel array; in order to reduce the effects of the ALT bottleneck, the channel array is buffered to decouple the processes involved.

In order to construct global snapshots of the system, the runtime traces of event records collected by the monitoring processes are transported to the host system, and are put in a temporal perspective off line, in a postmortem way. This is a typical approach followed in distributed monitoring, but is also a particular requirement for transputer-based systems, as only the root transputer of the network has access to the file system of the host machine. The monitoring data are generated in a copious volume and their transportation can have a significant negative impact on both the computational resources and the communication network of the distributed system. Occarm supports two different transport strategies, namely *immediate transport*, which transports the monitoring data as soon as they are generated and *store and unload*, whereby the data are stored in a buffer before they are transported. The latter makes use of a circular buffer to store only the recent history of the active process. The monitoring process flushes the contents of the history buffer when the simulation is complete or if no monitoring message arrives for a user-defined time interval (e.g. in the case of deadlock). The store and unload transport option is particularly useful as in most, if not all, cases the most recent history of the processes is sufficient to identify the cause of errors or deadlocks. Since the monitoring messages do not propagate further into the system, the communication overhead of the store and unload policy is minimal.

The collected traces encapsulate a high degree of parallelism implicitly contained in the event traces. In occarm, event traces arriving from the monitoring processes are distributed to separate trace files on the host system, one file for each of the processes of the occarm top level process graph. The existence of a separate trace file for each process provides a view of the parallelism of the system and thus facilitates the postmortem debugging and analysis task.

Prepared using cpeauth.cls

Process **process_name:** instruction **instruction_type** was executed at time **clock**

(a) Execution event trace

Process **process_name:** variable **variable_name** has the value **val** at time **clock**

(b) Data event trace

Process **process_name:** sending value **val** to process **proc** on channel **chan** at time **clock**

Process **process_name:** receiving value **val** from process **proc** on channel **chan** at time **clock**

(c) Parallelity event traces

Figure 14. Event Traces for Debugging

The simulation of an asynchronous architecture has two main objectives, namely the testing and debugging of the architecture and the evaluation of the architecture's performance.

7.1. Debugging

For the debugging of the architecture (as well as the simulation model) it is necessary to monitor both the flow of control and the flow of data in each of the different occam processes in the model; this is achieved by collecting traces regarding the *execution* and *data events* of the processes respectively (see figure 14a,b).

The concurrent nature of asynchronous hardware systems along with the absence of global synchronization, introduces a problem, common in asynchronous, parallel structures, namely *deadlocks*. Deadlock is a high-level issue of the design, and occurs when the system, as a result of a particular sequence of events, reaches a state wherein at least one sub-system becomes indefinitely blocked. In general, the sequence of events in an asynchronous system is non-deterministic. This is due mainly to the behaviour of the arbiters, which service request events in arrival order. If two requests arrive at the same time, the choice will be non-deterministic. Asynchronous logic allows variable delays within the different sub-systems, which will affect the order in which independent request events arrive at the arbiters of the system. The correct functionality of the asynchronous system should not depend on the ordering of independent streams of events; a correct design should be deadlock free for all possible combinations of events.

For the detection of deadlocks, it is essential to know the state of the channels in the system when the deadlock occurred. For this purpose, the *parallelity events*, which correspond to communication actions, need to be monitored (figure 14c). These will appear in pairs, one for the sending and one for the receiving process. The probe in the sending process code, is inserted before the output command while the receiving probe follows the input command. The absence of one parallelity event from a pair in the final trace indicates the occurrence of a deadlock.

7.2. Performance Evaluation

When analysing the performance of asynchronous pipelines, measures of special importance and interest are the *occupancy* as well as the *stall* and *idle* periods of the pipeline.

Pipeline Occupancy. The occupancy of a N stage pipeline is defined as the percentage of time the pipeline has 1, 2, ... , N elements in it.

In a synchronous pipeline, the clock frequency defines the period that any element stays in the pipeline; thus for the calculation of the occupancy only the entry (arrival) times of elements are required.

In asynchronous micropipelines however, the times that a particular data bundle enters and leaves the pipeline are arbitrary. Therefore, the calculation of occupancy in this case requires knowledge of both, the entry and exit times of bundles in the pipeline. A bundle enters a pipeline when the corresponding data is latched by the first register of the pipeline; thus the entry time is represented by the timestamp of the acknowledgement (Ain) signal issued by this register. Similarly, the exit time of a bundle is the value of the timestamp of the Acknowledgment signal to the last register of the pipeline.

The values of the two timestamps required for the calculation of the duration of a message's staying in the pipeline are not directly available, for they are possessed by different occam processes and occam does not support global variables. To overcome this problem a solution has been devised whereby request messages exiting 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.

In occarm, control processes maintain a set of *occupancy tables*, one for each of their input pipelines. The occupancy table is a circular buffer which contains the input and output timestamps of messages passing through the corresponding pipeline, thus providing a (postmortem) global view of the pipeline at any particular moment.

Each time the control process at the output side issues an acknowledgement message to a pipeline (i.e. each time a message exits the pipeline), it also invokes a probe procedure (the *calculate.occupancy()*) to calculate the current occupancy values for the pipeline; idle periods are also calculated at that point by the probe.

Contrary to the debugging traces which are sent immediately to the corresponding monitoring process, the type and quantity of monitoring values concerning the performance characteristics of the architecture permit active occarm processes to calculate and store them locally; this eliminates the extra communication overhead that their transport would impose. The stored values are unloaded by the occarm control processes upon their receiving the termination signals.

Stalls. An asynchronous pipeline will stall if the rate that request events are issued to the pipeline is greater than the rate that events propagate through the pipeline or the rate that events exit the pipeline (i.e. the rate that events are consumed and processed at the output side).

Stall situations refer to the input side of the pipeline. They may be detected by examining the delay between the sending of a Request event (Rin) to the pipeline and the issuing of the corresponding acknowledgement signal (Ain) by the first register of the pipeline: a stall situation has occurred if $timestamp(Rin) < timestamp(Ain)$. The duration of the stall is $timestamp(Ain) - timestamp(Rin)$; clearly the minimum stall period is equal to the propagation delay of the first register in the pipeline.

Monitoring information regarding pipeline stalls is collected by the occarm control processes at the input side and is unloaded upon detection of the termination signal.

8. TERMINATION

Detection of termination in a distributed parallel environment was brought to prominence in 1980 by Francez [34] and by Dijkstra and Scholten [28] and since then has constituted one of the basic problems in distributed computing. The issue of termination has also been examined for the special case of distributed discrete event simulation [50, 1]. The fundamental problem in detecting termination is the difficulty of constructing a global state of the distributed system. Several termination detection algorithms have been developed; these differ in the way they ensure correctness (e.g. liveness and safety) and the assumptions they make about the semantics and behaviour of the communication links in the distributed system (e.g. synchronous or asynchronous communication, FIFO or not, atomicity of communication actions etc.). Typically, termination mechanisms consist of a *termination detector*, superimposed on the distributed system, which either monitors the activity of the processes or uses a deadlock detection/breaking approach [16].

A simulation model of an asynchronous architecture has completed its operation, and thus must terminate, if it has executed all the instructions of a particular benchmark program. Within the ARM development environment used in the AMULET project, ARM programs notify their completion by writing a special End Of Program (EOP) character to a particular address in memory.

The above mechanism may be exploited for the termination of the occarm model too; upon receiving EOP, the memory process issues a *KILL* message which then propagates through the model, progressively killing the occam processes. A possible route for the KILL message which has been adopted for the termination of occarm is depicted in figure 15. The KILL signal enters occarm by means of the Acknowledgment message issued to the MemCtrl process by the memory for the EOP, and is then forwarded to Dec1CtrlA (through the IPipe) and to the Incrementer (on the corresponding Acknowledgment message) to terminate the datapath and address interface respectively. To cope with closed paths (loops) in the model and allow the KILL message to reach all processes in the loop, certain processes forward the KILL signal but do not terminate immediately; they continue their operation sinking subsequent messages until they receive the KILL for a second time. These processes include MemCtrl and the Incrementer, which sink PC values from the PC loop, DestCtrl, which sinks messages sent from memory before EOP, Dec1CtrlA, for PCcol signals and prefetched instructions, Dec1CtrlB, for ALUgo signals, Decode2, which sinks data from the register bank, and the Write process in the register bank, which sinks register addresses arriving from the lock fifos.

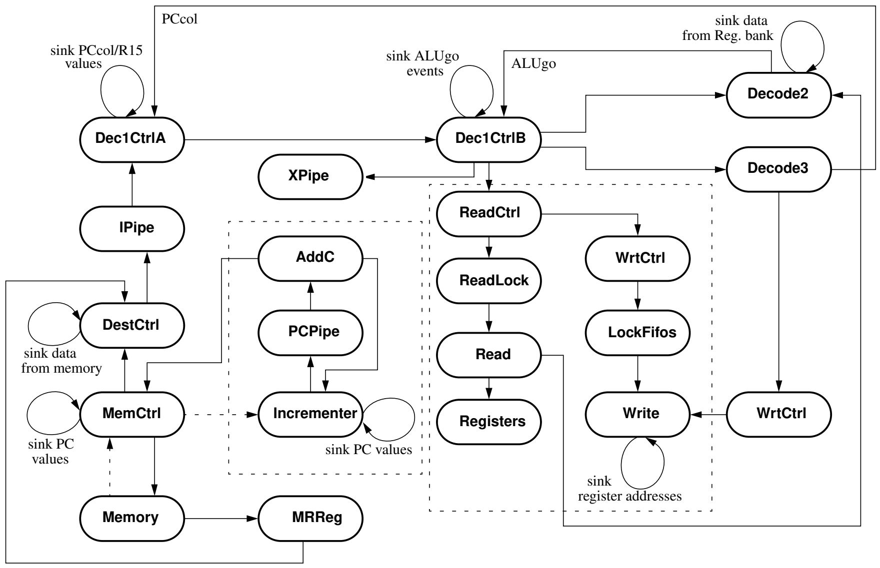


Figure 15. Terminating Occarm

9. THE SIMULATOR ENVIRONMENT

Occarm has been configured to execute on both, single and multiprocessor platforms. The single-transputer configuration is depicted in figure 16.

The Memory process models the memory control logic of the processor; the memory itself is implemented as a binary file to achieve compatibility with the existing ARM development environment. The operation of the simulator consists of reading instructions from the memory file, executing them and, possibly, writing results back into it; for compatibility reasons, messages regarding the correct operation of the simulated architecture produced by benchmark programs, are written to a separate text file, one character at a time.

Within the INMOS occam toolset environment, only one occam process may have access to the host machine's file system. In the occarm simulator environment, this role is served by the I/O process via which, all interactions with the outside world are performed. The I/O process employs an occam ALT construct to allow the multiplexing of system and monitoring messages arriving from the Memory and Monitoring processes respectively.

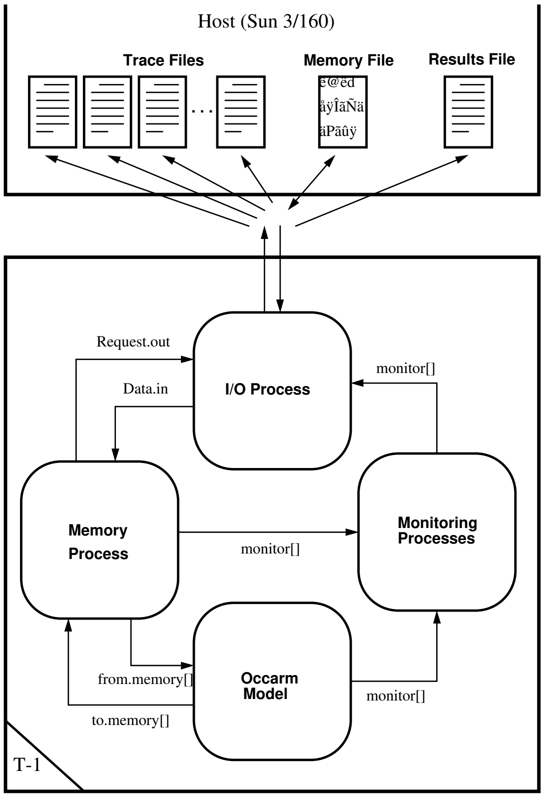


Figure 16. The Single transputer Environment of Occarm

10. MULTIPROCESSOR IMPLEMENTATION

One of the advantages of using occam as a specification language for asynchronous hardware, is the ability to exploit the inherent parallelism of the simulated architecture, and thus to achieve higher performance, by executing the simulation model on a multiprocessor machine. Mapping a parallel program onto a parallel machine is a fundamental problem in parallel processing and a detailed discussion would exceed the scope of this paper; in [60] Norman and Thanisch provide a comprehensive list of references concerning the subject. The mapping problem has also been investigated within the special context of distributed simulation, for both conservative (e.g. [12]) and optimistic (e.g. [17]) approaches.

	Decode1	Decode2	Decode3	DatInt	AddInt	RegBank	WrtCtrl	Memory	I/O.Process
Decode1		*	*	*	*	*			
Decode2	*		*	*	*	*			
Decode3	*	*			*		*	*	
DatInt	*	*			*		*	*	
AddInt	*	*	*	*			*	*	
RegBank	*	*					*		
WrtCtrl			*	*	*	*			
Memory			*	*	*				*
I/O.Process								*	

: Merge into one link

Figure 17. Occarm Process Connectivity Table

The static nature of the occam language requires that the mapping of the occam process graph on the transputer network is specified in advance by the application developer, though a number of tools have been developed to automate various steps of this task(e.g. [11, 49, 73]). The mapping scheme should ideally take into account the following considerations:

- The limitations imposed by the four links of the transputer. The degree of each node in the top level process graph should not be greater that four, with an edge in the graph representing a bidirectional link (in other words, each node of the graph should have at most four neighbours).
- The limitations imposed by the interconnection network of the machine. Typically, transputer networks are not fully connected and therefore process graphs have to be transformed to match the underlying structure; the aim here is proximity, namely placing communicating processes as close to each other in the network as possible.
- The computation and communication load should be evenly balanced over the processors and the links of the system respectively.

Based on the above considerations, the first step for mapping occarm onto the T-Rack is the modification of the the top level process graph of occarm as depicted in figure 9 so that each node has at most four neighbours. To address this issue the Process Interconnection Table (*PIT*) depicted in figure 17 has been devised. The number of asterisks in a row of the table represents the number of neighbour processes of that particular process. Merging two columns together, effectively adds one more level of abstraction to the process hierarchy, assigning the corresponding processes to the same processor and forcing the two channels to share the same link.

Occarm Process	Node
Decode1 + Decode2	1
AddInt	2
Decode3 + DatInt	3
RegBank	4
WrtCtrl	5
Memory	6
I/O.Process	-1

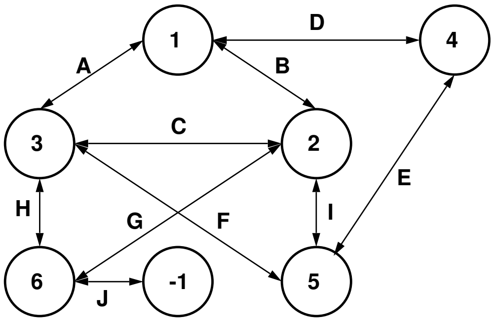


Figure 18. Modified Occarm Top Level Process Graph

Link	No. of Multiplexed Channels
A	13
B	9
C	4
D	6
E	3
F	6
G	2
H	8
I	3

Table I. Communication Load on Occarm Links

10.1. Balancing the Workload

The criterion adopted for the selection of the level of the occarm process hierarchy, each of whose process has at most four neighbours, is the maximization of processor utilization; namely, to occupy as many processors as possible.

Following this criterion the merges presented in figure 17 have been applied to occarm, deriving as a result the alternative graph of figure 18. This graph represents the lowest level in the process hierarchy which satisfies the four-link-per-transputer limitation; a possible alternative would be to merge columns 4 and 5, and 7 and 8, placing onto the same processors DatInt/AddInt and WrtCtrl/Memory respectively, instead of columns 3 and 4; this arrangement would require five, instead of seven, processors.

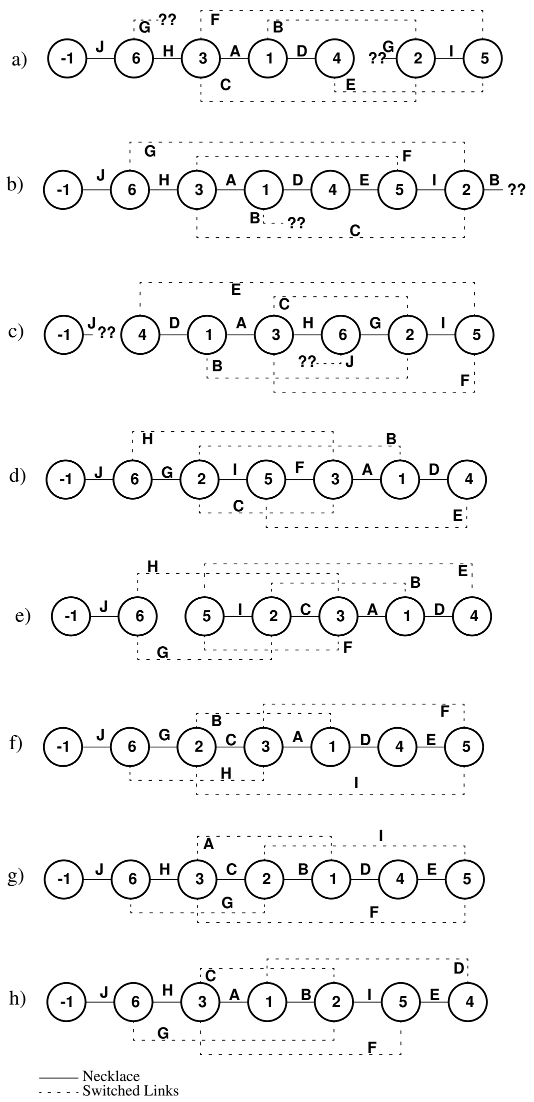


Figure 19. Occarm Graph Mappings

10.2. Balancing the Communication Load

The new top level occarm process graph possesses more than one Hamiltonean cycle, thus allowing an equivalent number of possible mappings on the T-Rack.

For the selection of the appropriate mapping, the criterion which has been followed is to balance the communication load. In the T-Rack the communication performance of a hardwired link is approximately double that of a switched link. Indeed, the performance of a hardwired unidirectional link is reported as being 1.72 Mbytes per second, with that of a switched link being 0.87 Mbytes per second; this is due to the increased latency of acknowledgment messages imposed by the C004 link switches [58]. Consequently, the objectives of the communication load balancing policy are to use as few switched links as possible, and to place onto hardwired links as many (multiplexed) channels as possible. The latter is based on the assumption that on average, during the execution of benchmark programs, all channels in the system will have similar traffic levels. The pipelined structure of asynchronous architectures provides a basis to this assumption; indeed instructions, as they execute, propagate through successive stages of the pipeline thus activating most, if not all, channels on their path. Thus in the initial stages of the design process, when no data regarding the behaviour of the simulated architecture is available (as was the case with the AMULET1, when occarm was developed), the above mapping criterion provides a reasonable option. Once a detailed performance analysis of the architecture has been performed, the mapping of the simulator may be altered accordingly; in this case, a possible criterion would be to map onto the fast links as many as possible of the channels which are part of the architecture's critical path.

The communication load on the links of the top level occarm process graph is given in table I. Figure 19 presents alternative mappings of the graph onto the T-Rack; $a \ldots c$ are examples of mappings which are not feasible due to the limited link connectivity of the T-Rack. The remaining mappings, namely $d \ldots h$, illustrate the different feasible ways to map occarm onto the T-Rack. From these, mapping h satisfies the communication balancing criteria specified above and therefore has been selected for occarm. This mapping uses the minimum possible number of switched links (namely 4) while allocating the maximum total number of channels onto the necklace (namely 36); mapping e makes use of 5 switched links while mappings d, f and g place onto the necklace 30, 28 and 30 channels respectively.

The selected mapping h, has also the advantage that the maximum number of channels from the Decode3-Memory path of AMULET1 (namely links F, H and I) are placed on the necklace. This is the path followed by the data transfer addresses and the corresponding abort signals during the execution of data transfer instructions in AMULET1. After sending the data transfer address to memory, Decode3 blocks until the corresponding abort signal is issued; therefore, in order to prevent stall and/or starvation phenomena in the simulation model, it is essential that Decode3 receives the abort message as soon as possible.

10.2.1. The Monitoring Path

To minimize the communication overhead imposed by the monitoring messages, the characteristics of the T-Rack may be exploited. As explained in section 9, the I/O process receives messages from both the Memory and the monitoring processes. Since the Tadpole transputer, where the I/O process resides, is connected to both ends of the necklace, the interaction with the Memory process may take place via one end of the necklace, with the monitoring messages following the other direction towards the other end of the necklace. This scheme is depicted in figure 20, where the complete mapping of occarm onto the T-Rack is

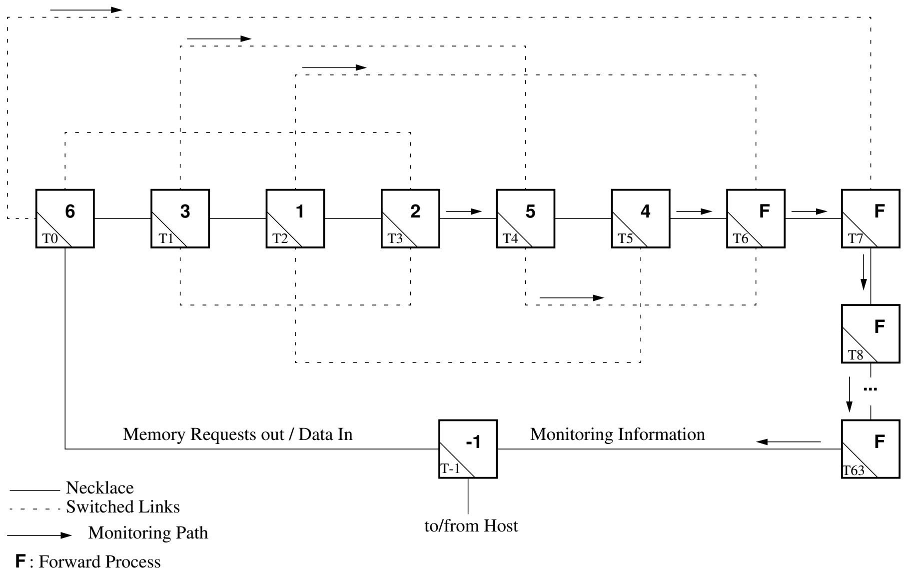


Figure 20. Mapping Occarm onto the T-Rack

presented; transputers T0-T5 host the simulation model, while transputers T6-T63 are simply used to forward monitoring information.

10.2.2. The Generic Simulator Node

Figure 21 depicts a generic node of the distributed configuration of the simulator. Typically this will include a number of active processes together with the corresponding monitoring modules. Extra multiplexing/demultiplexing processes are included to allow the sharing of the transputer links; to prevent deadlock situations (which for example might occur if one of the transputer links is blocked by a message destined for a particular process, while this process is blocked waiting for a message that may follow the former on the link), extra buffering has been incorporated into the demultiplexing modules (i.e. the distributor process). In practice, not all of the modules depicted in figure 21 will be included in a typical node.

11. VALIDATION

Simulation model verification and validation is a complicated albeit important issue and has received considerable attention [8]. The validation of occarm has been performed by comparing

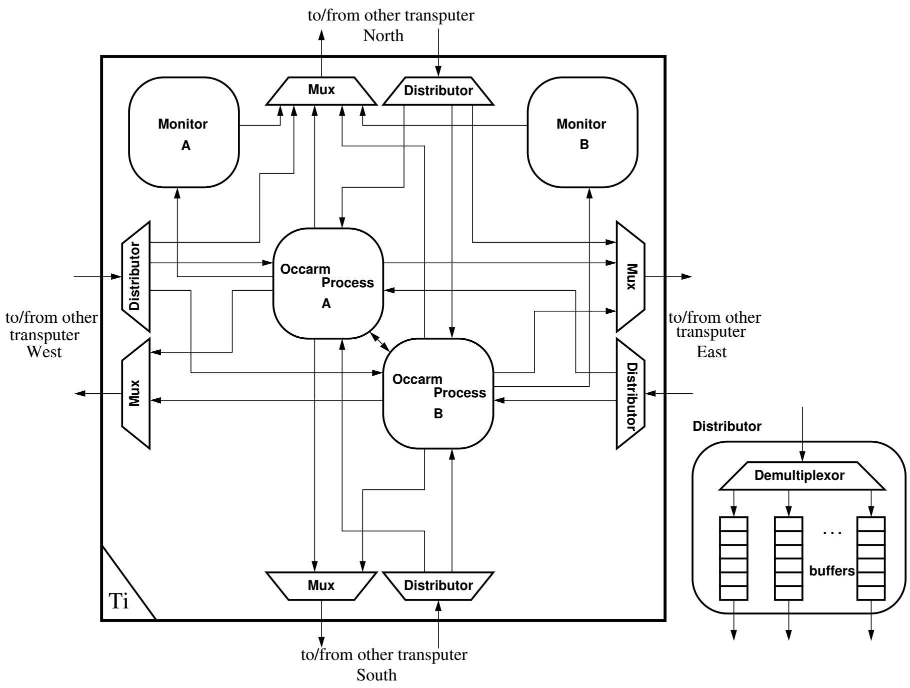


Figure 21. The Generic Simulator Node

results produced by occarm with those produced by a conventional sequential discrete event simulator of AMULET1 written in Asim, ARM Ltd in-house simulation language, and concerned the functional correctness, the timing accuracy and the performance of occarm. As benchmarks, the *ARM validation programs* [6] as well as *Dhrystone* [83] have been used. The former are “toy” benchmarks which invoke different instruction types to test different parts of the design. Dhrystone is a synthetic benchmark which has traditionally been used for the evaluation of computer architectures. Like all synthetic benchmarks, Dhrystone tries to match the average behaviour (i.e. the average frequency of operations and operands) of a large set of real programs and thus, the results obtained may be considered representative of the average behaviour of occarm too.

The functional correctness of occarm has been verified by (meta-)executing the complete set of the ARM validation programs. The accuracy of timing has been tested by obtaining and comparing simulated time based values that are used for the performance evaluation of AMULET1 (and asynchronous architectures in general), namely the Dhrystone number, as well as the occupancy and stall periods of the pipelines. These results have been reported in [74, 77].

With regard to performance, occarm and the Asim model are not directly comparable, since each describes the AMULET1 architecture at a different level, and the supporting machines (namely the T-Rack and a SPARC workstation) are intrinsically very different. Nevertheless, a comparative examination of the performance of the two models may provide an indication of the impact that the use of the occam model may have to the duration (and cost) of the design cycle. Tables II and III present the performance results achieved for the different occarm configuration and transport policies used. The results are compared with those achieved by the Asim simulator executing on an IPX Sun workstation. The figures represent mean values derived from ten simulation runs. The elapsed time for the Asim model has been obtained by taking into account the *user* and *sys* values provided by the Bourne shell "time" command.

Within the single transputer configuration (i.e. on a single 20MHz, T414 transputer), occarm requires on average 1.72 minutes to execute one Dhrystone loop when no monitoring traces are generated (this corresponds to the execution of 20 ARM machine instructions per second); this time is longer than that required by the Asim model by a factor of 1.16. This is a reasonable and expected performance, for the execution of the occarm processes on a single transputer is performed not in a parallel but rather in a time sharing fashion and the large number of parallel processes in the model (approximately 120 in the current implementation of occarm) make the context switching overhead in the transputer significant.

Table III presents the performance of occarm for both, its single and multiple transputer configurations and for the different policies employed for the transportation of monitoring data.

In the multi-trunsputer configuration, when no monitoring traces are generated, the distribution of occarm on to the seven transputers of the T-Rack yields a speedup of 1.69.

The "store and unload" transport policy allows a speedup of 2.26 to be achieved since, in this mode of operation, the performance of occarm on a single transputer drops by a factor of 2.45 compared to 1.83 in the multi transputer implementation. This difference in the performance drop may be attributed to the fact that the activation of the monitoring processes severely increases the frequency and, consequently, the overhead of context switching on the single transputer. The distribution of the monitoring processes onto multiple transputers alleviates this phenomenon as the context switching overhead is also distributed.

When the "immediate transport" policy is employed, the performance of both the single and multiple transputer configurations of occarm drops dramatically by 5.67 and 7.07 respectively, allowing a speedup of only 1.35. This behaviour may be attributed to the operation of the I/O process, which acts as a multiplexor for messages arriving from both the Memory and the Monitoring processes. This introduces a major bottleneck in the system, which imposes the ultimate limit on the performance of the simulator. The large number of monitoring messages generated by the "immediate transport" policy occupy a large proportion of I/O process activity, thus reducing the rate at which instructions and data are supplied to the model; as a consequence, the processes of the model remain idle for substantial periods.

The speedups achieved by the distribution of occarm onto the multiple transputers of the T-Rack may be considered acceptable and reasonable but not satisfactory. The poor speedup achieved may be attributed to a number of factors related to the characteristics of both the simulated architecture and the machine that hosts the simulator.

Model	Elapsed Time (minutes)
Occarm(Single)	1.72
Asim	1.48

Benchmark: Dhrystone (1 loop)

Table II. Asim versus Occarm (Single Transputer Implementation)

Transport Policy	Elapsed Time (minutes)		Speedup
	Occarm (single)	Occarm (multi)	
Tracing Off	1.72	1.02	1.69
Store and Unload	4.22	1.87	2.26
Immediate Transport	9.75	7.21	1.35

Benchmark: Dhrystone (1 loop)

Table III. Performance of Occarm

- Amdahl's law [4] specifies that the maximum possible speedup depends on the inherent parallelism of the executed system which may potentially be exploited. In the case of AMULET1, the requirement for instruction compatibility with the synchronous ARM, has resulted in an asynchronous design with a very complex pipeline structure and, indicatively, limited parallelism. The performance of AMULET1 itself is 70% of the performance of the synchronous ARM. The complexity of the AMULET1 architecture makes an analysis of the inherent parallelism of the design a complicated task which, as yet, has not been undertaken†.
- Asynchronous architectures are communication bound systems and therefore the efficiency of the communication system is crucial. The complex irregular interconnection pattern of AMULET1's functional modules and the extra multiplexing/demultiplexing processes required to cope with the connectivity constrains of the Transputer and the T-Rack introduce major bottlenecks in the system which severely reduce the communication efficiency.
- The transputer technology used, the only available to execute occam at the time of the experiments, is indeed dated, with poor performance characteristics.

†The estimation of the maximum possible speedup of a distributed simulation is an active area of research. Techniques which have been suggested for this purpose include the employment of a critical path analysis of a trace from a given simulation [68], and the treatment of the process graph as a queueing network model [82].

As explained in section 7, most of the time the simulation model will operate under the "store and unload" transport policy which permits the maximum speedup.

12. EPILOGUE

Asynchronous logic is being viewed as an increasingly viable alternative digital design approach which promises to liberate VLSI systems from clock skew problems, offer the potential for low power and high performance and encourage a modular design philosophy which makes incremental technological migration a much easier task. The advent of easily available custom ASIC technology in the form of VLSI or FPGAs has greatly facilitated the implementation of asynchronous circuits. However, asynchronous logic has to overcome several obstacles before it is established as a mainstream digital design style. One such obstacle is the lack of modelling languages and simulation techniques suitable for asynchronous design. Fundamentally, conventional, sequential, synchronous hardware description languages are not suitable for describing concurrent non-deterministic asynchronous behaviour. Modelling and simulation, being at the heart of digital system design, may perform a catalytic role in the quest for the realisation of the potential offered by asynchronous logic. Hence, the recurrence of interest in asynchronous design has been accompanied by intense research activity aimed at developing notations and techniques appropriate for modelling and simulating asynchronous systems.

The concurrent, asynchronous, process-based model of computation of CSP, with the support for non-deterministic behaviour, and the point-to-point, synchronous and unbuffered inter-process communication are particularly suitable for describing the concurrent, non-deterministic behaviour of asynchronous hardware systems and provide a natural and convenient means for the rapid construction of asynchronous hardware models. Hence, CSP has long and extensively been advocated as potential notation for the description of asynchronous hardware and various CSP-based and notations have already been employed for this purpose. However the work undertaken so far in this area, has placed emphasis on producing specifications to be used as input to silicon compilers for the automatic synthesis of asynchronous circuits; the use of these specifications as simulation models has received very little, if any, attention. Occam forms a practical realisation of CSP, and, consequently, it maintains the strong relationship with regard to communication and computation between CSP and asynchronous systems. An asynchronous architectural specification expressed in occam, is automatically an executable model, which may executed on a processor network.

However, this is a complicated endeavour as the distributed semantics of CSP and occam impose a number of issues that need to be resolved if the model is to serve as a simulation tool rather that just a simple textual description. These issues include debugging, monitoring and terminating the simulation; for distributed, multi-processor implementations, mapping and load balancing issues also need to be considered. This paper has presented a number of techniques and mechanisms have been developed to address all the aforementioned issues in the context of the occam model of the AMULET1 microprocessor.

Although the principal initial motive for using CSP and occam is the need to capture and model the concurrent, asynchronous, non-deterministic behaviour of asynchronous hardware, the exploitation of the inherent parallelism and the execution of the CSP/occam models on multiprocessor platforms can potentially achieve high simulation performance and can contribute in reducing the duration and cost of the design cycle. The poor speedups achieved by the multiprocessor implementation of the occarm model have been attributed to the particular characteristics of the testbed architecture and the transputer technology used. Other existing asynchronous architectures are generally characterised by a higher degree of parallelism and more regular interconnection patterns than AMULET1. The major deficiency of occam has been its close association with the transputer technology. The current availability of occam for non-transputer distributed platforms, including shared-memory multiprocessors, SMP systems and the Web [80], as well as the introduction of the CSP/occam model into other languages such as Java and C (e.g. JavaPP [85] and CCSP [56]), will alleviate the deficiencies of the transputer technology and will allow the potential for high performance to be realised. An alternative approach, would be to separate modelling from simulation: the former would provide a CSP based, occam-like notation for the specification of asynchronous hardware while the latter would utilise a generic distributed simulation kernel which would be optimised for the simulation of asynchronous hardware, incorporating the ideas discussed in this paper. Work in this direction has already commenced [78].

REFERENCES

1. M. Abrams, V. Sanjeevan, D. S. Richardson, Termination and Output Measure Generation in Parallel Simulations, *Journal of Parallel and Distributed Computing* 1993;18:454-472.
2. F. A. Almeida, P. H. Welch, A Parallel Emulator for a Multiprocessor Dataflow Machine, *Proceedings of the World Transputer Congress 1994*, Como, 1994; pp. 259-272.
3. J. M. Alonso, et al., Conservative Parallel Discrete Event Simulation in a Transputer-Based Multicomputer, *Proceedings of the World Transputer Congress 1993*, Aachen, 1993; pp. 636-650.
4. G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proceedings AFIPS 1967 Sping Joint Computer Conference*, Atlantic City, April 1967; pp. 483-485.
5. The AMULET Group, World Wide Web Home Page, URL: http://www.cs.man.ac.uk/amulet/index.html
6. ARM Ltd, World Wide Web Home Page, URL: http://www.arm.com
7. O. Babaoglou, , K. Marzullo, Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, *Technical Report UBLCS-93-1*, Laboratory for Computer Science, University of Bologna, January 1993.
8. O. Balci, Principles of Simulation Model Validation Verification and Testing, *Transactions of the Society for Computer Simulation International*1997; 14(1):3-12.
9. The Balsa Asynchronous Synthesis System, URL: http://www.cs.man.ac.uk/amulet/projects/balsa/
10. G. Birtwistle, A. Davis, eds., *Asynchronous Digital Circuit Design*, Springer Verlang, 1995.
11. J. E. Boillat, An Analysis and Reconfiguration Tool for Mapping Parallel Programs onto Transputer Networks, *Proceedings of the 7th Occam Users Group*, September 1987; pp. 186-194.
12. A. Boukerche, C. Tropper, A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations, *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94)*, SCS, July 1994; pp. 164-172.
13. J. A. Brozowski, C-J. H. Seger, *Asynchronous Circuits*. Springer Verlang, 1995.
14. E. Brunvand, M. Starkey, An Integrated Environment for the Design and Simulation of Self Timed Systems, in *Proceedings of VLSI 1991* 1991; pp. 4a.2.1-4a.3.1.
15. E. Brunvand, The NSR Processor, in *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, 1993; pp. 428-435.
16. J. Brzezinski, J. M. Helary, M. Raynal, Distributed Termination Detection: General Model and Algorithms, *Technical Report 1964*, INRIA, March 1993.

17. C. Burdorf, J.Marti, Load Balancing Strategies for Time Warp on Multi-User Workstations, *The Computer Journal* 1993;36(2):168-176.
18. P.C. Capon, J. R. Gurd, A. E. Knowles, ParSiFal: A Parallel Simulation Facility, in*Proceedings of IEE Colloquium: The Transputer: Applications and Case Studies*, IEE Digest,91, May 1986.
19. K. M. Chandy, L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems* 1985; 3(1):63-75.
20. K. R. Cho, K. Okura, K. Asada, Design of a 32-bit Fully Asynchronous Microprocessor (FAM), in *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, Washington D.C., 1992; pp. 1500-1503.
21. T. A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications*, Ph.D Thesis (MIT/LCS/TR-393), M.I.T., June 1987.
22. W. A. Clark, C. E. Molnar, Macromodular Computer Systems, R. W. Stacy, B.D, Waxman, eds., *Biomedical Research*, Academic Press, 1974, Chapter 3.
23. Cogency Technology Inc., World Wide Web Home Page, URL: http://www.cogency.com
24. J. Cortadella, et. al., Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, *IEICE Transactions on Information and Systems* 1997, E80-D(3):315-325.
25. A. Davis, The Architecture and System Method for DDM1: A recursively structures data-driven Machine, in *Proceedings of the 5th Annual Symposium on Computer Architecture*, Palo Alto, CA, 1978; pp. 210-215.
26. A. Davis, S. M. Nowick, Synthesizing Asynchronous Circuits: Practice and Experience, in [10], pp. 104-150.
27. M. E. Dean, STRiP: A Self-Timed RISC Processor, *Technical Report CSL-TR-92-543*, Computer Systems Laboratory, Stanford University, July 1992.
28. E. W. Dijkstra, C. S. Scholten, Termination Detection for Diffusing Computations, *Information Processing Letters* 1980, 11(1).
29. D. L. Dill, Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits, *ACM Distinguished Dissertations*, MIT Press, 1989.
30. C. J. Elston, et al., Hades - Towards the Design of an Asynchronous Superscalar Processor, in *Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, London, 1995; pp. 200-209.
31. P.B. Endecott, S.B. Furber, Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language, in *Proceedings of the 12th European Simulation Multiconference*, Society for Computer Simulation International, Manchester, 1998; pp. 39-43.
32. FDR User Manual (available via anonymous ftp at ftp.comlab.ox.ac.uk), Formal Systems Europe, 3 Alfred Street, Oxford, 1993.
33. A. Ferscha, S. K.Tripathi, Parallel and Distributed Simulation if Discrete Event Systems, *Technical Report CS.TR.3336*, University of Maryland, August 1994.
34. N. Francez, Distributed Termination, *ACM TOPLAS* 1980; 2(1):42-55.
35. R. Fujimoto, Parallel Discrete Event Simulation, *Communications of the ACM* 1990;33(10):31-53.
36. S. B. Furber, Computing Without Clocks, In [10], pp. 211-262.
37. S. B. Furber, et. al., AMULET2e: An Asynchronous Embedded Controller, in *Proceedings of Async '97 Conference*, IEEE Computer Society Press, 1997; pp. 290-299.
38. J. D. Garside, et. al., AMULET3 Revealed, in *Proceedings of Async'99 Conference*, IEEE Computer Society Press, 1997; pp. 51-59.
39. G. Gopalakrishnan, V. Akella, Specification, Simulation, and Synthesis of Self-Timed Circuits, in *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1993; pp. 399-408.
40. S. Hauck, Asynchronous Design Methodologies: An Overview, *Technical Report UW-CSE-93-05-07*, University of Washington, April 1993.
41. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.
42. H., Hulgaard, S. M. Burns, Bounded Delay Timing Analysis of a Class of CSP Programs with Choice, in *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994.
43. R. N. Ibbett, P. C. Capon, The Development of the MU-5 Computer System, *Communications of the ACM* 1978;21(1):13-24.
44. *Occam 2 Reference Manual*, Inmos, Prentice Hall International, 1988.
45. M. B. Josephs, J. T. Udding, Delay-Insensitive Circuits: An Algebraic Approach to their Design, in *Lecture Notes in Computer Science*, Vol. 458, 1990; pp. 342-366.
46. J. Joyce, et al., Monitoring Distributed Systems, *ACM Transactions on Computer Systems*1987;5(2):121-150.
47. A. E. Knowles, M. S. Illiev, Monitoring Facilities on the ParSiFal T-Rack, in *Proceedings of the ConPar'88*, Cambridge University Press, 1988.

48. L. Lamport, Time, Clocks and the Ordering of Events in Distributed Systems *Communications of the ACM* 1978;21(7):558-565.
49. F. C. M Lau, K. M. Shea, Mapping a Process Network onto a Processor Network, in *Occam and the Transputer-Research and Applications*, Editor C. Askew, IOS, 1988; pp. 91-100.
50. Y. B. Lin, On Terminating a Distributed Discrete Event Simulation, *Journal of Parallel and Distributed Computing* 1993;19:364-371.
51. Y. Liu, J. Aldwinckle, K. Stevens, and G. Birtwistle, Designing Parallel Specifications in CCS, in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.
52. A. J. Martin, et al., Design of an Asynchronous Microprocessor, in *Proceedings of the Decennial Caltech Conference on VLSI: Advanced Research in VLSI*, 1989; pp. 351-373.
53. A. J. Martin, Synthesis of Asynchronous VLSI Circuits, J.Staunstrup, editor, *Formal Methods for VLSI Design*, North Holland, 1990.
54. C. A. Mead, L. A. Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980.
55. C. E. Molnar, T-P. Fang, Synthesis of Reliable Speed-Independent Circuit Modules: I. General Method for Specification of Module-Environment Interaction and Derivation of a Circuit Realisation, *Technical Report 297*, Computer Systems Laboratory, Institute for Biomedical Computing, Washington University, St. Louis, 1983.
56. J.Moores, CCSP - a Portable CSP-based Run-time System Supporting C and occam, in *Architectures, Languages and Techniques for Concurrent Systems* ,B.M.Cook, editor, Concurrent Systems Engineering series, Vol. 57, WoTUG, IOS Press, Amsterdam, the Netherlands, 1999; pp. 147-168.
57. D. E. Muller, W. S. Bartky, A Theory of Asynchronous Circuits, *Digital Computer laboratory 75*, University of Illinois, November 1956.
58. A. D. Murta, *Support for Transputer Based Program Development via Run Time Link Reconfiguration*, Ph.D Thesis, Department of Computer Science, University of Manchester, 1991.
59. T. Nanya, et al., TITAC: Design of a Quasi-delay-Insensitive Microprocessor, *IEEE Design and Test of Computers* 1994; 11(2):50-63.
60. M. G. Norman, P. Thanisch, Models of Machines and Computation for Mapping in Multicomputers, *ACM Computing Surveys* 1993; 25(3):263-302.
61. N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, Ph.D Thesis, Department of Computer Science, University of Manchester, 1994.
62. W. F. Richardson, E. Brunvand, Fred: An Architecture for a Self-Timed Decoupled Computer, *Technical Report UUCS-95-008*, University of Utah, May 1995. Available at: ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z
63. M. Riek, B. Tourancheau, X. F. Vigouroux, Monitoring of Distributed Memory Multicomputer Programs, *Technical Report UT-CS-93-204*, University of Tenessee, October 1993.
64. A. W. Roscoe, N. Dathi, The Pursuit of Deadlock Freedom, *Technical Monograph PRG-57*, Programming Research Group, Computing Laboratory, Oxford University, November 1986.
65. E. M. Sentovich, et. al., SIS: A System for Sequential Circuit Synthesis, *Technical Report UCB/ERL M92/41*, U.C. Berkeley, May 1992.
66. Sharp's Data-Driven Media Processor, World Wide Web Home Page, URL: http://www.sharpsdi.com/DDMPhtmlpages/DDMPmain.html
67. J. Sifakis, 'Deadlocks and Livelocks in Transition Systems, *Lecture Notes in Computer Science*, 88, 1980, pp. 587-599.
68. T. K. Som, B. A. Cota, R. G. Sargent , On Analysing Events to Estimate the Possible Speedup of Parallel Discrete Event Simulation, in *Proceedings of the 1989 Winter Simulation Conference*, December 1989; pp. 729-737.
69. R. F. Sproull, I. E. Sutherland, C. E. Molnar, The Counterflow Pipeline Processor Architecture, *IEEE Design and Test of Computers* 1994; 11(3):48-59.
70. M. Stephenson, O. Boudillet, A Graphical tool for the Modeling and Manipulation of Occam Software and Transputer Hardware Topologies, in *Occam and the Transputer-Research and Applications*, Editor C. Askew, IOS, 1988; pp. 139-144.
71. I. E. Sutherland Micropipelines, *Communications of the ACM* 1989;32(1):720-738.
72. I. E. Sutherland, Flashback Simulation, *Research Report SunLab 93:0285*, Sun Microsystems Laboratories, Inc., August 1993.
73. G. Theodoropoulos, A. West, Graphical Configuration of Transputer Systems: The Graphical Configuration Assistant, *Proceedings of the 1994 Transputer Research and Applications Conference (NATUG-7)*, Athens, Georgia, October 1994.

74. G. Theodoropoulos, *Strategies for the Modelling and Simulation of Asynchronous Computer Architectures*, Ph.D Thesis, Department of Computer Science, University of Manchester, 1995. Available at: ftp://ftp.cs.man.ac.uk/pub/amulet/theses/theo95-phd.ps.Z
75. G. Theodoropoulos, J. V., Woods, Occam: An Asynchronous Hardware Description Language?, in *Proceedings of the 23rd IEEE Euromicro Conference on New Frontiers of Information Technology*, Budapest, Hungary, September 1997.
76. G. Theodoropoulos, Modelling and Distributed Simulation of Asynchronous Hardware, *Simulation Practice and Theory Journal* 2000, 7:741-767.
77. G. Theodoropoulos, Distributed Simulation of Asynchronous Hardware: The program Driven Synchronisation Protocol, *Journal of Parallel and Distributed Computing*, Special issue on Distributed Simulation, editor C. Tropper, to appear.
78. G. Theodoropoulos, Towards a Framework for the Distributed Simulation of Asynchronous Hardware, *5th United Kingdom Simulation Society Conference*, UKSim 2001, Emmanuel College, Cambridge, United Kingdom, 28-30 March 2001, to appear.
79. C. H. Van Berkel, J. Kessels, M. Roncken, R. Saeijs, F. Schalij, The VLSI-Programming Language Tangram and its Translation into Handshake Circuits, in *Proceedings of EDAC*, 1991; pp. 384-389.
80. K.Vella and P.H.Welch, CSP/occam on Shared Memory Multiprocessor Workstations, in *Architectures, Languages and Techniques for Concurrent Systems*, B.M.Cook, editor, Concurrent Systems Engineering series, Vol. 57, WoTUG, IOS Press, Amsterdam, the Netherlands, 1999; pp.87-119.
81. *VERSIFY Release 2.0*, Department d'Arquitectura de Computadors, Universitat Politcnica de Catalunya, Barcelona, Spain, November 1998, World Wide Web Home Page, URL: http://www.ac.upc.es/vlsi/versify/
82. D. B. Wagner, E. D. Lazowska, Parallel Simulation of Queueing Networks: Limitations and Potentials, *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Berkeley, USA, May 1989; pp. 146-155.
83. R. P. Weicker, Dhrystone, A Synthetic Systems Programming Benchmark, *Communications of the ACM* 1984;27(10):1013-1030.
84. P. H. Welch, G. Justo, C. Willock, High-Level Paradigms for Deadlock-Free High-Performance Systems, *Proceedings of the World Transputer Congress 1993*, September 1993, pp. 981-1004.
85. P. H. Welch, G. S. Stiles, G. H. Hilderink, A. P. Bakkers, CSP for Java : Multithreading for All, in *Architectures, Languages and Techniques for Concurrent Systems*, B.M.Cook, editor, Concurrent Systems Engineering series, Vol. 57, WoTUG, IOS Press, Amsterdam, the Netherlands, 1999; pp.112-120.
86. T. Werner, A. Venkatesh, Asynchronous Processor Survey, *IEEE Computer* 1997;30(11):67-76.
87. J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple, AMULET1: An Asynchronous ARM Microprocessor, *IEEE Transactions on Computers* 1997; 46(4):385-398.