Dealing with Time Modelling Problems in Parallel Models of 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. The research activity in this area has pointed to the need for suitable techniques for modelling and simulating asynchronous systems. The occam programming language allows the rapid development of parallel simulation models of asynchronous systems but its distributed nature introduces time modelling problems. This paper presents an approach for dealing with these problems.

1. Introduction

As VLSI technology advances and systems become larger, faster and more complex, timing problems in synchronous systems 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. In order to address those problems recently there has been a resurgence of interest in asynchronous design techniques which eliminate the need for global clocking. An asynchronous system may be designed as a set of functional modules each operating at its own rate and cooperating through communication. The synchronization of the functional modules is performed by means of the communication protocol which allows data to be shared between them.

There exist many different approaches for designing asynchronous systems [9]. Sutherland's "Micropipelines" [18] use bundled data with an event-signaled handshake protocol for synchronization (figure 1). The control circuits are implemented by means of a set of *event control blocks*, which include the *Muller-C, Select, Call, Toggle, Xor* and the *Arbiter* blocks (figure 2).

Following Sutherland's approach, the AMULET group at the University of Manchester have designed and implemented AMULET1, an asynchronous version of the ARM RISC processor (figure 3) [7] [8].

2. Modelling Asynchronous Architectures with Occam

The recent research activity in the area of asynchronous systems has pointed to the need for suitable techniques for modelling and simulating them [24]. Several notations and techniques have been suggested for this purpose [10]. CSP [11] in particular has attracted the interest of many researchers due to the strong relationship between its semantics (synchronous, unbuffered interprocess communication) and the behaviour of asynchronous systems [1] [15].

Figure 2: Event processing blocks

To support the design of AMULET1, a methodology for using occam [12] to build executable models of asynchronous architectures at the Register Transfer Level has been developed [20]. Using this methodology, an asynchronous architecture is modelled as a set of concurrent communicating occam processes. The processes are entirely data-driven, and self scheduled. Two channels are used for the synchronization of communicating processes, one for Request/data and one for the Acknowledgement signal. To describe the nondeterministic behaviour of arbiters, the occam ALT statement is used.

3. Timing Issues

The methodology described above allows the *spacial* characteristics of the asynchronous architecture to map naturally onto the simulation model. However, this does not hold for the *temporal* characteristics of the architecture, as the distributed nature of occam introduces a problem typical in distributed simulations, namely the problem of enforcing and maintaining strict temporal precision.

All physical systems obey the *causality principle* which defines the relationships between the various system states. More specifically, the causality principle requires that *the cause must always precede the effect in time*: state transitions that have some effect on some other transitions must occur before the latter, while state transitions that do not affect each other may take place in any order.

Thus, the causality principle imposes a partial ordering on the system's state transitions.

Figure 3: The AMULET1 Processor Physical Layout

This ordering of state transitions in the physical system also imposes an equivalent partial ordering on the corresponding events in the simulation model. In order to ensure that the simulation model faithfully and accurately reproduces the behaviour of the simulated physical system, the order in which logical processes receive, process and generate events must be the same as the order of the corresponding state transitions in the physical system.

In the physical system, causality is tracked using the physical time (real time clock). In a simulation environment, simulated time is the only means to maintain causality; each event (which models a state transition) is assigned a timestamp that denotes the point in the simulated time that the event occurs. In a sequential simulation model, wherein the physical time is modelled by a single global variable, causality is preserved through adherence to the rule that events are processed in nondecreasing timestamp order [5]; in a distributed setting however, in which concurrent processing of events in different simulated time is allowed, preservation of this fundamental monotonicity property associated with causality is not straightforward.

It has been shown (e.g. Lamport [14] and Misra [16]) that a distributed system consisting of processes which operate asynchronously and interact exclusively via timestamped messages, will adhere to the partial ordering imposed by causality constraints in the physical system, if each process consumes and processes events in nondecreasing timestamp order; this condition is referred to as the *local causality constraint* [5]. Thus, the problem of guaranteeing that a distributed simulation model implements exactly the same global causal precedence relationships of the physical system reduces to ensuring that each process obeys the local causality constraint and processes messages in nondecreasing timestamp order.

If each process in the distributed model has a single input link (i.e. it receives messages from just one source process) then adherence to the local causality constraint is straightforward. Indeed, in single input processes, there is a direct correlation between order of arrival and order of consumption. Assuming that the timestamps in the input link are ordered in time then the output timestamps are guaranteed to be crescently monotonic. Thus on any particular output (and therefore input) link, messages will be issued in nondecreasing timestamp order.

However, most, if not all, practical simulation models include multiple input processes which receive messages from more than one source process; such processes are usually referred to as *merge* processes. In this case, the order in which messages arrive on the different

input links does not adhere strictly to the order in which the corresponding events occur in the physical system; it merely depends on the relative real time propagation delays of the messages in the distributed machine and not on the simulated time timestamps that these messages carry.

Therefore, in the general case, messages will be arriving at the merge processes of the model in a nonincreasing timestamp order. Consequently, immediate consumption and processing of messages by merge processes may result in violation of the local causality constraint; the processing of an out of order message (i.e. a message which according to its timestamp should have been processed in the simulated past) is referred to as a *preemption*.

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 situation 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 to the order in which events arrive at the corresponding arbiter in the physical circuit; it just depends on the order in which the corresponding input occam channels are selected. 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.

4. The Need for Accurate Time Modelling

The violation of the local causality constraint by the arbiter processes in the occam model 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, in distributed simulations time is not only a synchronizing agent but also a quantifier, which provides the means for the simulated system's performance evaluation; thus, errors in simulated time introduce inaccuracies in the evaluation of the simulated architecture. In [20] it was argued that the characteristics of the architectures being simulated will not allow the time error introduced by preemptions in arbiter processes to become significant. Recent results confirm this claim [22] [23]. Nevertheless, accuracy is still needed if the model is to serve as a tool for a more extensive and elaborate evaluation of the performance characteristics of the asynchronous architecture.

The requirement to test the architecture for potential deadlocks by modifying the delays in the system to achieve different event orderings [20] makes this necessity even more intense. Indeed, since the simulated time is not the synchronization agent in the simulator, different

event orderings may be achieved by using occam *Timers* to change the relative scheduling of the occam processes [20]. The major drawback of this approach is that small delays cannot guarantee the intended effects and behaviour in the model, as these delays are only approximate. Furthermore, real time delays have a direct effect on the performance of the simulator. Thus, large delays that would guarantee the planned process scheduling, would also affect the performance of the simulator.

Thus, it is extremely desirable to be able to develop accurate models of arbiter processes that obey the local causality constraint and avoid preemptions.

5. The Need for a New Synchronization Technique

Techniques that have been developed to address the preemption problem in distributed simulations and ensure that the local causality constraint is not violated, are traditionally classified into two broad categories, namely *conservative* and *optimistic* [6].

Conservative techniques [2] allow a logical process to accept and process an event only if it absolutely safe to do so, thus strictly avoiding the possibility of a preemption ever occurring. These techniques require that a merge process blocks until there is a message on each of its input links, and then selects and process the message with the smallest timestamp. However, deadlocks may occur if a message which is expected by a blocked process will eventually not be issued; two techniques have been devised to deal with that problem, namely *deadlock detection and correction* [4], whereby the simulation proceeds until it deadlocks and when deadlock is detected it is resolved, and *deadlock avoidance* [3], whereby timestamped Null messages are sent over the links of the model to enable merge processes decide if it is safe to process pending messages.

Optimistic approaches detect and recover from causality errors rather that strictly avoid them. The most important and influential optimistic mechanism is *Time Warp* [13]. In Time Warp, processes consume and process events as they arrive without first deciding whether such an action is safe or not. When a preemption is detected the process "rolls back" in simulated time undoing its illegal actions.

The rationale behind using occam for modelling asynchronous architectures, is the exploitation of the close relationship between the language semantics and the characteristics of the asynchronous system [20]. Once the occam simulation model is constructed, any attempt to introduce time accuracy into it should have as its prime objective to preserve this modelling philosophy.

Furthermore, any complexity that might be added to the model as a result of the synchronization protocol should be kept as low as possible. One of the purposes served by the occam model is to provide a description of the architecture's specification and operation; this information should not be obscured or hidden by the extra functionality which is related only to the accurate operation of the model and not to the simulated architecture per se.

Any attempt to employ any of the existing synchronization protocols would change the personality of the model, forcing it to depart from the modelling philosophy which provided its original basis. The Program Driven Synchronization Protocol described in this paper aims to meet the aforementioned requirements. This is a novel conservative protocol which is based on a combination of the exploitation of the characteristics of the simulated system and the employment of Null messages to achieve deadlock avoidance while maintaining the philosophy of the model virtually intact. It seeks to enable the development of accurate arbiter models involving only the processes required for this purpose. The processes of the model remain entirely data driven [21].

6. The Program Driven Synchronization Protocol (PDSP)

6.1. The Basis

Von Neumann computer architectures, synchronous or asynchronous, are deterministic systems: they accept as input instructions which they execute sequentially in a specific and predefined order.

Each instruction defines the steps that are required for its execution as well as the behaviour of each functional module of the architecture. Consequently, the kind and sequence of events that occur in the system are determined at any time by the executing instructions.

This ability to predict events in the architecture based on the information provided by the program under execution, forms the basis of the Program Driven Synchronization Protocol; by looking at the instructions being executed the arbiter processes of the simulation model can decide whether an event is expected on a particular input link and thus whether their blocking upon this link would result to a deadlock.

The key concept in the Program Driven philosophy is the "Instruction Lookahead Set" which is defined as follows:

Definition 1 *The* **Instruction Lookahead Set** *of a link* λ *is the set of instructions whose execution will potentially result to an event occurring on* λ*:*
$\mathbf{ILS}_\lambda = \{$*Instruction* **I***:* **I** *generates an event on link* $\lambda\}$.

An instruction **I** is referred to as an ILS_λ **instruction** if and only if $I \in ILS_\lambda$.

The Instruction Lookahead Set of any particular link in the system is directly defined by the architecture's specification and thus, may become available to the arbiter processes of the simulation model in advance. Based on the ILS of their input links, arbiter processes may directly make decisions regarding the potential arrival of messages, provided of course that they are also informed of the instructions being executed in the system.

6.2. The Rules

Based on the Instruction Lookahead Set defined above, the behaviour of arbiter processes with regard to message consumption may be specified as follows:

Rule 1 *An arbiter process* Π *is allowed to block and wait for an event on its input link* λ *during the execution of an instruction* **I** *if and only if* $\mathbf{I} \in \mathbf{ILS}_\lambda$.

The above rule ensures that arbiter processes block only for instructions that are likely to generate the corresponding events. However, depending on the status of the system, during the instruction's execution such an event might not occur; in this case Null messages are required otherwise the arbiter process will become blocked and the simulation model will deadlock. The following rule is concerned with the production of Null messages:

Rule 2 *A Null message will be sent to link* λ *of the arbiter process* Π *if and only if* Π *expects an event on* λ *based on the* $\mathbf{ILS}_\lambda$*, and for the current state of the system the event will not be generated.*

The two rules above, specify the behaviour of arbiter processes and their peers, when their interaction depends on the executed instructions. However, not all events in an asynchronous system occur in an instruction dependent fashion. Indeed, certain parts of the system may operate autonomously, irrespective of which instructions are being executed; the PC loop in the AMULET1 processor is an example of such an autonomous unit. In this case it is the state of the simulated system that dictates the behaviour of the arbiter process:

Figure 4: The PDSP Arbiter Process

Rule 3 *An arbiter process* Π *is allowed to block and wait for an event on its input link* λ *which fires in an instruction independent way, if and only if the state of the system guarantees that a message will be issued on* λ.

6.3. The PDSP Arbiter Process

The basic functionality of an arbiter process with regard to the Program Driven Synchronization Protocol is depicted in figure 4.

Upon receiving a message on one of its links (e.g. *msg1* message on *In1*) the arbiter invokes the *Select* process to determine whether the processing of this message would cause a preemption.

If there is a pending message *msg2* already received from the other input, then the message with the minimum timestamp is selected to be processed and forwarded to the arbiter process' output; if both timestamps have the same value, the selection is made in a random fashion to emulate the behaviour of the corresponding hardware arbiter.

Figure 5: PDSP: Taking MLL into Account

If however, no pending message exists, but a positive prediction (based on the Instruction Lookahead) is made regarding its potential arrival, the arbiter process blocks and waits until this second message arrives. The arrival of this message provides the arbiter process with the information required to proceed its operation and enable *Select* to make a decision, namely the next timestamp on its other input link.

6.3.1. Improving PDSP Performance

The basic algorithm described in figure 4, enables arbiter processes to receive and process messages arriving on their input links in increasing timestamp order, always selecting the message with the smallest timestamp, thus guaranteeing the accurate, preemption-free operation of the simulation model.

However, it does not ensure that the concurrency of the simulated system is sufficiently exploited to increase the potential of the simulation model for high performance.

Indeed, as soon as it predicts that a message is expected on one of its input links In2, the arbiter process will stop accepting any messages arriving on its other input link In1 until the expected message on In1 arrives. As a consequence, all the processes that are part of the path that leads to In1 will block and wait, and the pipelines at the output side of the arbiter process will starve; during this time, large parts of the simulator will remain idle.

A solution to this problem is to provide arbiter processes with some indication as to when in the simulated future a message they expect will actually arrive. This information would enable them to consume a number of events occurring on In1 link before they block on In2 increasing thus the concurrency of the simulation model.

This information can be obtained by taking into account the propagation delays in the architecture being simulated. An event generated by an instruction will propagate through a number of pipeline stages before it reaches an arbiter. The path followed by the event is completely defined by its parent instruction; the latency of the path however at any particular time, depends on the number of elements in the micropipelines involved and thus, it is non-deterministic.

Consequently, it is not feasible to know in advance the exact time required for an event to propagate through a given micropipeline. However, there is a lower bound to this time,

Figure 6: Occarm Top Level Process Graph

namely the latency of the micropipeline when, at the moment of the event's entry, it is empty. Based on this observation, the Minimum Latency Lookahead, may be defined:

Definition 2 *The* **Minimum Latency Lookahead** *of a link* λ $\mathbf{MLL}_\lambda$*, is defined as the total propagation delay of the path leading to* λ*, when the pipelines of the path are empty:* $\mathbf{MLL}_\lambda = \sum d_i, d_i$ *= Propagation Delay of the i-th Pipeline Stage in the path.*

The Instruction Lookahead Set of a link informs the corresponding arbiter process *whether* a message should be expected on that link; the Minimum Latency Lookahead reveals *when* in the simulated future the expected message may arrive. Based on the Minimum Latency Lookahead, the following rule may be specified:

Rule 4 *An arbiter process* Π *will not process a message* μ_1 *received on its input link* λ_1 *but instead it will block and wait for a message* μ_2 *expected on its other input link* λ_2*, if and only if the timestamp of* μ_2 *as predicted by the* MLL_{λ_2} *is less than or equal to the timestamp of* μ_1.

Rule 4, combined with the ALT statement in the main loop of the PDSP arbiter process, will enable arbiter processes to process messages with appropriate timestamps as soon as they arrive.

Figure 5 depicts the *Select* process when the MLL is taken into account.

6.4. The Limitations

The Program Driven Approach is based on the exploitation of the Instruction Lookahead properties of the simulated architecture. Such an exploitation presupposes that arbiter processes

have knowledge as to which instructions are being executed. If this knowledge is not directly available, then an appropriate mechanism needs to be devised to provide arbiter processes with this information. Generally the functionality of the architecture being modelled will make the development of such a mechanism feasible. Otherwise the instruction lookahead properties of the system can not be exploited and the PDSP rules can not apply; in this case one of the conventional Deadlock Avoidance algorithms may be applied which however, would require a substantial modification of the model to deal with the regular flow of Null messages.

7. Applying PDSP to the Occam Simulation Model of AMULET1

In order to demonstrate its applicability, the Program Driven Synchronization Protocol was employed to develop models of the arbiter processes in *occarm*, the occam simulation model of AMULET1 which has been developed as part of the work of the AMULET group at the University of Manchester [19] [22].

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 6.

AddInt and *DatInt* processes model AMULET1's address and data interface 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 execution unit of the processor containing the Shifter/Multiplier and the ALU respectively. RegBank process incorporates the functionality of the register bank. *WrtCtrl* models the operation of AMULET1's write bus control logic.

The address interface is responsible for providing all the address information to memory. It operates as an autonomous unit, issuing sequential instruction addresses to maintain a steady flow of prefetched instructions to the processor. Instructions arriving from memory through the data interface, rendezvous with their associated R15 value (PC) extracted from the address interface (see figure 6) whereupon they enter the datapath for execution.

There are two cases wherein an instruction which has entered the processor will be invalidated and rejected and as a result, its execution will not eventually take place:

- The instruction fails its condition codes in Decode3. In ARM architecture, all instructions are conditionally executed. Their execution depends on the outcome of the comparison between the *Current Processor Status* (*CPS*) arithmetic flags and the condition field of the instruction word. In AMULET1, the test of the condition flags is performed in Decode3.

- The colour of the instruction does not match that of the processor. AMULET1 maintains a "colour" bit (at Decode3) which changes each time the instruction flow changes (e.g. due to a branch or an exception). Instructions are also "coloured" and if their colour does not match that of the processor at any particular moment they are discarded. To make this mechanism more efficient, the new colour is also sent to Decode1 (via the PCcol signal) to enable the rejection of invalid instructions before they enter the datapath. The invalidation and rejection of instructions may occur either in Decode1 or in Decode3; the choice depends on the exact point in time that the PCcol signal from Decode3 is detected by Decode1, and thus it is nondeterministic.

Occarm makes use of three arbiter processes, which are included in the address interface (AddInt), primary decode (Decode1) and the write control (WrtCtrl) respectively. The rest of the paper discusses how the principles introduced by PDSP have been employed to develop

Figure 7: The Address interface Model (AddInt)

an accurate, preemption-free model of the address interface; for a detailed description of the algorithms involved as well as of the use of PDSP to deal with the other two arbiter processes of occarm, the reader is referred to [22].

8. The Address Interface Arbiter

The Address Interface is depicted in figure 7. Address Interface employs arbitration (AddC process) to allow addresses arriving from the datapath on the Wch channel to break the *PC loop* (consisting of AddC, MAReg, Incrementer, PC Holding Latches, PC0 register of PC Pipe) and gain access of the MAReg.

8.1. Providing Instruction Lookahead Information

AddC is an example of an arbiter process which has no direct knowledge regarding the executing instructions. An address produced by AddC is sent to Memory following the path from AddC, through MAReg and DataInt. If it is an instruction address, the instruction message from memory enters the processor following the path from Memory through DataInt to Decode1. AddC is not in the path followed by the instruction and therefore has no direct information as to which instructions have entered the system; in order to apply PDSP a mechanism is required to provide AddC with this information and enable it to make decisions regarding the potential arrival of messages on its input links.

A neat and efficient solution is to take advantage of the hidden links in the above paths: the contra flow of the Acknowledgement messages. In the first path above, an address message produced by AddC will propagate to MARreg and to Memory and from there to DataInt; DatInt will generate an Acknowledgement message which will follow the opposite direction back to AddC. This Acknowledgement message can be used to carry the corresponding instruction to AddC; no communication overhead is generated as the Acknowledgement messages would be sent anyway.

8.2. The PCch Link

The PCch channel carries the Acknowledgement signal from the PC Pipe, which is issued each time the current circulating PC in the PC loop is latched by the first register of the PC Pipe. The operation of the PC loop is autonomous and independent from the operation of the rest of

Figure 8: The Address Interface - Datapath Loop

the processor. Thus, on the PCch channel there will be a continuous, instruction independent flow of messages.

The role of the PC Pipe is to provide the processor's datapath with the R15 values required for the execution of instructions as depicted in figure 8. If, for any reason, the datapath stalls, instructions will start to backlog and as a result, the PC Pipe will become full and will remain so for as long as the datapath stalls. During this period no further PC values will be allowed to enter the PC Pipe and thus no Acknowledgment signal will be issued on the PCch channel. The datapath may stall in the following cases, as illustrated in figure 9:

- If the datapath fills up; this will occur as a result of Decode3 and WrtCtrl processes waiting for the aborts and Wlx signals respectively.
- If an ILS_{Wch} instruction is followed by register read operations which refer to locked registers.
- If an ILS_{Wch} instruction is followed by instructions which activate the ALUgo signal.
- During the execution of load/store multiple instructions.

In order to avoid deadlock situations, it is essential that AddC be able to decide whether it should wait for yet another message from PCch or whether the PC Pipe has become full and thus no more messages will be sent on the PCch link (PDSP Rule 3). In order to do that, AddC needs to possess information regarding the possible invalidation of instructions that have entered the system. This information is provided by both Decode1 and Decode3 by means of extra messages sent via dedicated, buffered links that have been introduced in the occarm model.

Decode3 informs AddC of the possible changes in the value of the *Current Processor Status* that could result to the rejection of instructions, as illustrated in figure 10a.

The messages issued by Decode1 (figure 10b) aim to inform AddC of the exact time of arrival of the PCcol signal from Decode3, whereupon instructions may start being rejected in Decode1.

8.3. The Wch Link

The Instruction Lookahead Set of the Wch channel is:

Figure 9: Stalling of the Datapath

ILS$_{Wch}$=$\{$*B, BL, SWI, LDR, STR, LDM, STM, Data Processing with PC as Dest. Reg.*$\}$

Messages arriving on Wch channel are primarily sent to AddC from the datapath (through WrtCtrl) as a result of the execution of ILS_{Wch} instructions and carry either branch target or data transfer addresses. For data transfer operations whose destination register is R15, a second message will be sent over Wch, namely the new value of the Program Counter from memory. According to the PDSP algorithm (PDSP Rules 1 and 4), when AddC detects an ILS_{Wch} instruction it blocks until it receives the corresponding messages on Wch. If however the ILS_{Wch} instruction will not be executed or the memory fails to respond (i.e. an abort occurs), the expected messages will never be issued, thus leaving AddC blocked and causing the simulator to deadlock. There are two reasons why an instruction may not be executed, namely if its colour does not match that of the processor or if it fails its condition codes.

All the instructions whose execution may change the operating colour of the processor - either by explicitly writing a new value to the PC (i.e. the branch target address) or by causing an abort - belong to the Instruction Lookahead Set of the Wch channel. Thus, an ILS_{Wch} instruction will suffer a colour mismatch only if it follows another ILS_{Wch} which has changed the processor's colour. For instructions that explicitly change the PC, the new colour is provided to the AddC with the branch target address, making the decision as to whether an ILS_{Wch} instruction will be discarded straightforward. If however the colour changes due to an abort, AddC has no direct knowledge regarding this change; Decode3 will receive the abort signal from memory and will change the PCcol rejecting subsequent instructions. In this case a Null message must be sent by Decode3 to inform AddC of the occurrence of an abort and the colour change (PDSP Rule 2).

As described in the previous section, AddC is provided by Decode3 with all the CPS-related information required to predict the fate of subsequent instructions regarding their condition codes; this is performed via the dedicated link and lasts until a result is produced by Decode3 and forwarded to RReg as illustrated in figure 10a. Thus, for subsequent ILS_{Wch} instructions that fail their condition codes, Null messages are required to be sent by Decode3 to inform AddC of this event. Once a Null message is sent to AddC, no more messages of this kind will be issued for ILS_{Wch} instructions subsequently invalidated in Decode3, until a valid instruction is executed. This pattern will be followed until the next ILS_{Wch} to produce a result

Figure 10: AddC: The Instruction Dependent Generation of PDSP messages

Model	Elapsed Time (minutes)
$Occarm_{ALT}$	1.72
$Occarm_{PDSP}$	2.15
$Occarm_{PDSP-MLL}$	1.93

Benchmark: Dhrystone (1 loop)

Table 1: Performance of PDSP (Address Interface)

is encountered, whereupon the production of CPS-related messages to AddC will commence. This is illustrated in figure 10c where a complete picture of the interaction between Decode3 and AddC is provided.

9. Performance

The incorporation into occarm of a preemption-free model of the address interface based on the PDSP algorithms described in this paper, has resulted to a 20% decrease of the perfor-

mance of the simulator when no attempt is made to exploit MLL (i.e. using the algorithm of figure 4). Some preliminary experiments have indicated that by exploiting MLL a performance improvement of at least 10% can be achieved (table 1).

10. Conclusions

Asynchronous logic promises to provide solutions to problems such as clock skew, power efficiency, performance and modularity on VLSI design and is currently experiencing a resurgence of interest. Occam is particularly suitable for rapidly implementing distributed simulation models of asynchronous systems at the Register Transfer Level but its distributed nature introduces time modelling problems. This paper has described an approach for dealing with these problems, namely the Program Driven Synchronization Protocol. PDSP provides a general theoretical framework for the development of arbiter processes which eliminate preemptions and allow the accurate modelling of time in the simulation model; its philosophy is conservative aiming at deadlock avoidance. It is *program driven* in the sense that the behaviour of the processes in the model with regard to message consumption is determined by the instructions that are executed in the system at any particular moment.

The application of this approach depends on the exploitation of the instruction lookahead properties of the system. The application of the PDSP on the occarm simulation model has proven that such an exploitation is feasible, even for systems of the AMULET1's complexity.

Acknowledgements

For this work Georgios Theodoropoulos has been supported by the *Mpakalas Foundation*, Athens, Greece, under Grant No. 466/21121992.

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] Chandy, K. M., Misra, J., "Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs", IEEE Transactions on Sotware Engineering, 5, 5, September 1979, pp. 440-452.

[3] Chandy, K. M., Misra, J., "Deadlock Absence Proofs for Networks of Communicating Processes", Information Processing Letters, 9, 4, November 1979, pp. 185-189.

[4] Chandy, K. M., Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM, 24, 11, November 1981, pp. 198-205.

[5] Fujimoto R., "Applications: Distributed Simulation", in "Multicomputer Networks: Message-Based Parallel Processing", Reed, D. A., Fujimoto, R., MIT Press, 1988, chapter 6, pp. 240-267.

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

[7] Furber, S., Day, P., Garside, J., Paver, N.,Woods, J.V,"A Micropipelined ARM", Proceedings of the VLSI'93 Conference, September 1993.

[8] Furber, S., Day, P., Garside, J., Paver, N.,Woods, J.V, "AMULET1: A Micropipelined ARM", Invited Paper, COMPCON'94.

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

[10] Hauck, S., "Asynchronous Design Methodologies: An Overview". Technical Report UW-CSE-93-05-07. Department of Computer Science, University of Washington, April 1993.

[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] Jefferson, D., Sowizral, H., “Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control”, Technical Report N-1906-AF, RAND Corporation, December 1982.

[14] Lamport, L., “Time, Clocks and the Ordering of Events in Distributed Systems” Communications of the ACM, 21, 7, July 1978, pp. 558-565.

[15] Martin, A.J. 1986. “Compiling Communicating Processes into Delay-Insensitive VLSI Circuits”, Distributed Computing Vol 1, Number 4, April 1986, pp 226-234.

[16] Misra, J., “Distributed Discrete-Event Simulation”, ACM Computing Surveys, 18, 1, March 1986, pp. 39-65.

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

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

[19] Theodoropoulos, G., “An occam model of the AMULET1”, In Proceedings of the AMULET Modelling Workshop, Windermere, Cumbria, England, July 1984.

[20] Theodoropoulos, G., Woods J.V., “Building Parallel Distributed Models for Asynchronous Computer Architectures”, Proceedings of the World Transputer Congress 1994, Como, September 1994, pp 285-301.

[21] Theodoropoulos, G., Woods J.V., “Distributed Simulation of Asynchronous Computer Architectures: The Program Driven Conservative Approach”, Proceedings of the European Simulation Symposium 1994, Volume 2, Istanbul, Turkey, October 1994, pp 230-234.

[22] Theodoropoulos, G., “A Clever Title”, Ph.D Thesis, Department of Computer Science, University of Manchester, to be submitted.

[23] Theodoropoulos, G., Woods J.V.,“ Analysing the Timing Error in Distributed Simulations of Asynchronous Computer Architectures”, submitted to the Eurosim Congress ’95, TU Vienna, September 11-15, Vienna, Austria.

[24] Udding, J. T., “Formal Models”, Proceedings of the Workshop on the Design and Implementation of Asynchronous Circuits, Amsterdam, November 1991, pp 12-16.