Instruction Issue and Data Forwarding Mechanisms for Asynchronous Superscalar Processors

D.K. Arvind and R.D. Mullins†
Division of Informatics
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka@dcs.ed.ac.uk or Robert.Mullins@cl.cam.ac.uk

Abstract

An asynchronous design methodology offers potential advantages for architectures implemented in deep sub-micron technologies, such as low power dissipation and good electro-magnetic compatibility. This paper explores the impact of such a methodology on the architecture of superscalar processors. We examine in particular out-of-order instruction issue and data forwarding in the absence of global synchronisation. Three schemes are presented, and the performances of the resulting asynchronous superscalar architectures are compared to an equivalent synchronous one for a set of well-known benchmarks.

1. Introduction

The design of high-performance processors is becoming increasingly complex. This is due to a combination of factors such as their sheer size, architectural complexity, and the inherent difficulties of designing in deep sub-micron technologies [6], such as interconnect delays and limits on power dissipation and peak currents. The generation of global clock signals in the gigahertz frequency ranges is challenging, which is further compounded by requirements to gate clocks and support multiple on-chip clock domains.

An alternative and radical approach is to remove the clock altogether, and adopt an asynchronous design style [17]. At the architectural-level, there is no longer a need to distribute the clock globally, and the control structure can potentially exploit a range of computational and communication delays, without being limited to the worst-case timing behaviour. At the circuit-level, asynchrony attempts to minimise superfluous switching activity, which naturally benefits power consumption. The use of delay-insensitive interfaces and circuits leads to a modular approach to processor design which is free of complex timing requirements. This will become increasingly important as verification and design re-use become critical factors in design methodologies which are qualified by time-to-market considerations.

An asynchronous approach often requires novel design solutions to problems which have traditionally exploited global synchronisation. Naive solutions which operate in a pseudo-synchronous manner are often uninteresting, as they expose the full overhead of asynchronous handshaking and completion detection. A case-in-point is the handling of data dependencies in synchronous superscalar processors, viz. out-of-order instruction issue and data-forwarding, which rely on global synchronisation, and pose challenges in developing efficient asynchronous solutions.

†Robert Mullins is now with the Rainbow Group, University of Cambridge Computer Laboratory.

The following section provides a brief overview of the core operations in a typical synchronous superscalar processor. Section 3 develops the idea of an asynchronous version of the central instruction window which attempts to minimise synchronisations which degrade performance. Section 4 discusses a queued asynchronous architecture and develops data forwarding mechanisms for both this and the architecture described in Section 3. An alternative approach to dynamic scheduling and data forwarding which relies on a compile-time analysis of possible data forwarding opportunities, called *instruction compounding*, is described in Section 5. Finally, simulation results are presented comparing the performances of synchronous and the asynchronous architectures.

2 Superscalar Processors

Figure 1 illustrates a generic superscalar processor. The pipeline is capable of fetching and executing multiple instructions in each clock cycle, and is typically supported by branch prediction and speculative execution in order to maintain a high instruction bandwidth.

We first describe the operations which take place during instruction issue and data-forwarding, which will clarify the descriptions in the later sections. (See [13] for more details about the stages in such a pipeline) In the rest of the paper, we assume the presence of register renaming, and references to registers always imply physical ones, unless otherwise stated.

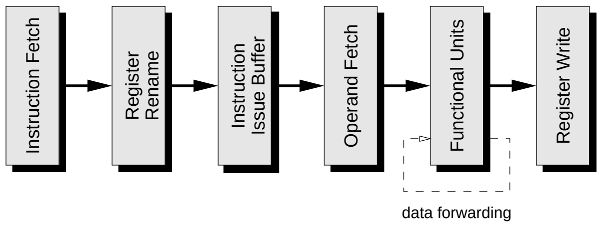

Figure 1. Synchronous Superscalar Pipeline

The central instruction window (CIW) or instruction issue buffer provides a limited dataflow capability, which allows instructions to be issued out of order, when both their operands and the appropriate execution resources are available. The basic operations performed within the window are listed below:

- **Write** New instructions are loaded into the window.
- **Wakeup** The status of the instruction's operands are updated, based on the lifetimes of issued instructions. It is useful to further decompose this phase into:
  - **Initialise** The operand's status is set on entering the window.
  - **Update** When instructions dependent on a particular result are issued, then all instructions which use the result are found and their status updated.
- **Selection** The ready instructions request to be issued. Instructions are selected for the available functional units.
- **Issue** A ready instruction is removed from the window and sent to the appropriate execution pipeline.

A pipelined processor includes a data-forwarding mechanism to permit dependent instructions to be executed in consecutive cycles. This mechanism enables data to be sent directly between functional units, bypassing the register file.

3 Transformation to an Asynchronous Design with CIW

The first asynchronous design is a gentle transformation of a typical synchronous central instruction window. We impose the following minimum requirement to avoid pseudo-synchronous operations: (1) we will not impose a strict order on the basic operations, as described in the previous section, which implement the dynamic scheduling mechanism, and (2) the operation of the functional units should not be synchronised by the scheduling mechanism. It should be possible to, issue to, and, handle results from, each FU independently.

Figure 2 shows a block diagram of the pipeline of such a design. An instruction in the CIW is ready for issue when all other instructions which produce its operands have been issued. An instructions initially obtains the current status of its operands (the *initialise* operation) from the register scoreboard (or more precisely, the status of the instructions which will produce its operands). The scoreboard operates in the following manner:

- An instruction at the rename stage (after renaming has taken place):
    1. sets the bit in the scoreboard corresponding to its destination register.
    2. reads the current status of the bits corresponding to its source operands.
    3. is written into the instruction window.
- An instructions leaving the instruction window:
    1. clears the bit corresponding to its destination register.

The status of instruction operands already in the instruction window are also updated as instructions leave the window. This is performed in the style of synchronous designs using content-addressable memory cells. The global CAM operation and the clearing of bits in the scoreboard together constitute the *update* phase.

In order to ensure that the status of the operands of each instruction is updated correctly, it is required that the *write* and *initialise* phases, and the *update* phase are not performed simultaneously. A single arbiter enforces mutual exclusion in this first design.

3.1 The Instruction Buffer

The CIW is itself implemented as a parallel FIFO [20, 19]. Reads and writes to each entry are controlled using head and tail pointers, which effectively implements a circular buffer. The fact that entries in this buffer do not move between memory elements simplifies our implementation of the instruction window. Other asynchronous FIFO implementations such as micropipelines [16], although offering other advantages, are problematic in this respect.

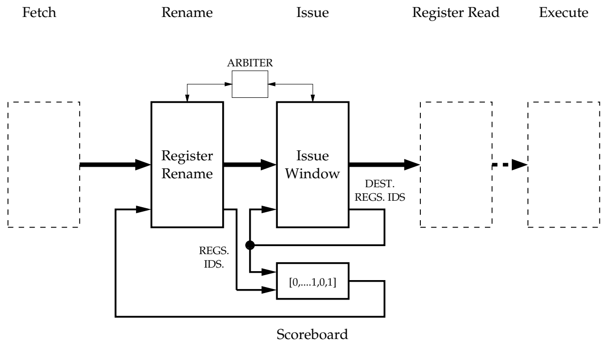

Figure 2. Asynchronous Central Instruction Window

3.2 The Update and Selection Processes

Instructions become ready for issue once the status bits associated with each of their operands have been reset, which may occur either initially when the scoreboard is read, or after subsequent update operations. At the point when an instruction is ready, it will arbitrate for issue by raising a ready signal from its entry in the instruction window.

In order to avoid the need for F, N-input arbiters (where F and N are the number of functional units and entries in the window, respectively) and to ensure that the selection of older, ready instructions can be prioritised, a static selection phase is created for each functional unit. Static inputs to the selection circuit are provided through the use of selection arbiters as shown in Figure 3.

Each entry in the window holds a single instruction and is capable of raising a ready signal for each functional unit. Once all the operands of a particular entry are available, then a request is made to raise the ready signal for the

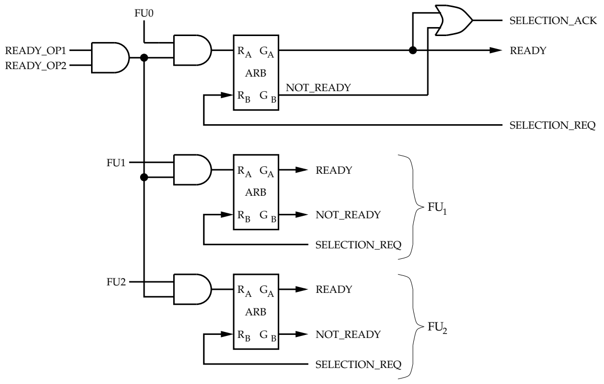

Figure 3. Selection Arbiters

target functional unit. In Figure 3, the signals *Ready_op1* and *Ready_op2* will be set if both operands are ready. This will result in a request to one of the selection arbiters corresponding to the target functional unit.

The selection and issue of instructions is outlined below. This process is performed concurrently for each functional unit.

- At least one of the ready signals is raised.
- The *selection_req* signal is asserted.
- All entries respond with a *selection_ack*. This is either a *ready* or *not_ready* signal (for the sake of clarity this is indicated for only the uppermost functional unit in the diagram). The inputs to the selection circuit are now guaranteed to be static and selection may take place.
- The selected instruction may be issued once the appropriate issue port is free (one port is associated with each execution pipeline).

The exact behaviour of the selection request signal is not specified here. A number of possibilities exist for controlling when the signal is to be asserted or deasserted. For example, we may wish to issue all ready instructions before deasserting the selection request signal, or deassert it after each issue in order maximise fairness. The point at which the selection request is raised is also undefined; one question is whether to make this dependent or independent of the state of the issue port. We do not elaborate further on these questions here as they depend on other architecture-specific parameters.

3.3 Reducing Synchronisation

The architecture as presented so far forces the write and initialise phases and update phase to be mutually exclusive. In practice, this is only required when each phase attempts to access the same entry in the scoreboard. Synchronisation may be reduced by providing an arbiter at each entry in the scoreboard, as shown in Figure 4.

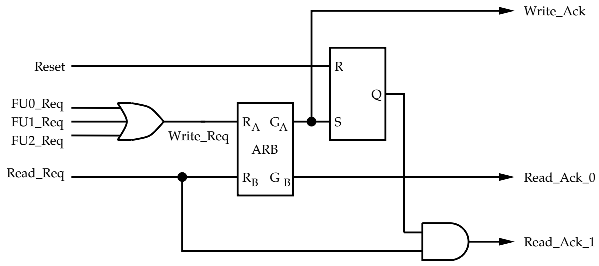

Figure 4. Arbitrated access to an individual scoreboard entry

Renaming guarantees that no two instructions with the same physical destination register may be simultaneously in flight. This ensures that no two functional units will ever attempt to access the same scoreboard entry at the same time. This enables a simple OR gate to generate the functional unit write request to the scoreboard entry. Access to the scoreboard is now arbitrated as before, but only enforces mutual exclusive operation, when both the rename stage and an update operation need to access the same entry.

Arbitration is only required within a particular entry, when it is possible for the entry to change its state from indicating a *not_ready* status to a *ready* one. No arbitration is required in the case when the entry is indicating a ready status (the RS flip-flop is set in the example).

Dependencies within a group of instructions which are fetched together may be determined without reference to the scoreboard during the rename process. This reduces the number of scoreboard reads and also enables the status of all the scoreboard entries associated with the destination registers of the instructions in the group to be reset simultaneously.

4 Data Forwarding in a Queued-based Architecture

Figure 5 shows an alternative asynchronous architecture with a limited form of dynamic scheduling. The monolithic instruction window has been replaced with a number of instruction queues. While this allows instructions at the head of each queue to be issued whenever dependencies permit, execution at a particular functional unit is always in order. The following section describes how such an architecture may be extended to support forwarding, and how a similar scheme may be used in general with the kind of central instruction window described previously.

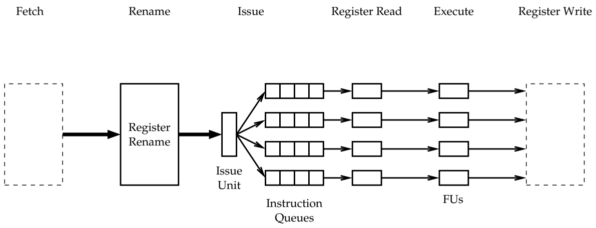

Figure 5. Queued Based Dynamic Scheduling

If it is possible to determine whether data can be obtained via forwarding at the instruction issue stage, then it would avoid the synchronisation necessary in traditional forwarding mechanisms, which effectively synchronises the operations of all the functional units. The HADES architecture [4] exploits a similar forwarding scheme, although in the absence of instruction queues or parallel register reads.

Recent results are stored at the output of each functional unit in a number of forwarding registers. The issue unit controls both the point at which the contents of these registers may be overwritten and when data is forwarded from these registers to other functional units. The issue unit maintains a record of the results which will be generated by the previous N instructions issued to each functional unit. If these results are required by subsequent instructions, then forwarding may be initiated by sending a forwarding request to the appropriate forwarding register. These requests together with commands to enable the contents of the forwarding registers to be overwritten, are queued in a forwarding register command buffer in each functional unit.

The existence of many forwarding sources introduces the problem that a number of outstanding forwarding requests may result in data being forwarded to a particular functional unit in a non-deterministic order. One solution to this problem is to provide a unique forwarded data input buffer, for each possible source, at each functional unit. We can, of course, guarantee that the data arriving from a particular functional unit will be in order, as forwarding requests are queued. The correct source is selected by the instruction during operand fetch, using information obtained during issue.

A block diagram illustrating a single execute pipeline incorporating data forwarding is shown in Figure 6.

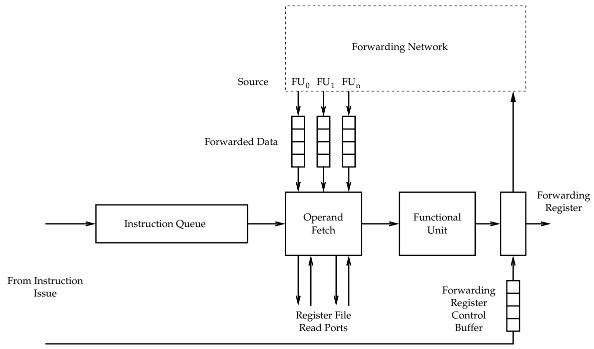

Figure 6. Execution Pipeline with Data Forwarding

A more general forwarding scheme is to enable the forwarding of results from the previous N instructions, irrespective of the functional unit they use. This requires that the individual forwarding registers be combined into a central forwarding queue. A similar structure, tailored for inclusion into the Amulet 3 is described in [9, 8]. The need to be able to forward results generated in a completely arbitrary order forces us to allocate entries in the forwarding queue during instruction decode. The queue must also be multi-ported for both reading (forwarding) and writing multiple results in parallel. This combination would probably preclude this type of design for the case when there are many functional units, each with its own input instruction buffer. Nevertheless, the queueing of forwarding requests and overwrite signals could be used in a manner similar to the distributed case described previously.

4.1 Data Forwarding and Central Instruction Windows

We now consider how data forwarding, as described in the previous sections, may be employed when instructions are issued in an arbitrary order (while respecting dependencies) from a central instruction window.

Consider the organisation shown in Figure 7, where just two functional units are represented. We now need to generate forwarding requests during the register read stage, as the order in which instructions will be executed is unknown prior to this. As in the previous description of a queue-based architecture, the source of data which may be forwarded is determined during issue. In the case of a central instruction window this information is appended to instructions in the window during the update phase. For example, if an instruction is issued to the memory unit, any operands requiring its result will record the memory unit as the source of data. Instruction operands which are initialised to be ready at the time the scoreboard is read will not generate forwarding requests. In reality such data will often already be available from the register file.

1. Instructions are issued to the appropriate operand fetch stage.
2. Forwarding requests are made, if necessary, for each operand. Operands which cannot be obtained via forwarding are fetching from the register file.
3. An overwrite signal is sent to a forwarding register at the output of the instruction's functional unit. This invalidates the contents of one of the forwarding registers in preparation for a new result to be latched.

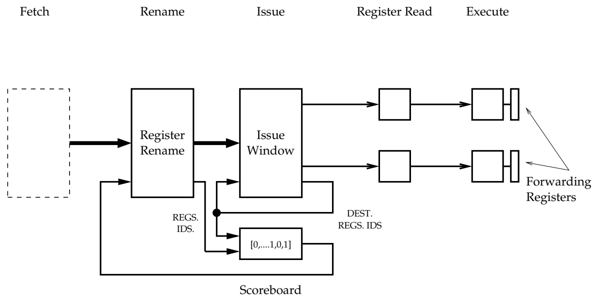

Figure 7. Asynchronous CIW Forwarding Problem

At this point it is clear that concurrent forwarding requests may be made from a number of operand fetch stages to a single forwarding register control buffer. One problem is that overwrite signals, invalidating the current contents of a forwarding register, may also be sent concurrently with forwarding requests for the current data.

This is solved by, firstly, arbitrating access to each forwarding register control buffer. Secondly, forwarding requests can no longer be guaranteed to be always successful. Forwarding requests may now occur too early - prior to the receipt of an overwrite signal preparing for the receipt of a particular result, or too late - after subsequent overwrite signals have effectively removed the data. To avoid incorrect data being forwarded, forwarding requests must now include the register identifier of the required data. A comparison in the forwarding register now determines if forwarding is possible, or if the forwarded operation must be cancelled, thereby forcing the value to be obtained via the register bank.

4.1.1 Source Counters

The cancellation of a large number of forwarding requests may result in a significant performance overhead. In order to reduce the number of surplus forwarding requests a mechanism may be introduced to cancel forwarding operations while instructions are still in the CIW.

A counter, which we will call the *source counter*, is associated with each operand in the CIW. This is initialised at the point when an update operation causes an operand entry to record a forwarding register as a source of data. Subsequent update operations from the same source now cause the counter to be decremented. When the counter's value is zero the source of the data is reset to the register file, a forwarding request will no longer be made.

5 Instruction Compounding

In the schemes discussed previously, forwarding has been initiated by the consumer. An alternative approach is to combine forwarding and dynamic scheduling, by allowing the producer to initiate a forwarding operation. This requires that dependencies be identified explicitly prior to issue. One way in which this may be achieved is to identify chains of dependent instructions at compile time. The following section gives an overview of such an approach, called instruction compounding - a more detailed description may be found in [1].

Instruction dependencies are identified by creating a number of instruction compounds during compilation, defined as groups of dependent instructions. A basic block is partitioned into compounds by grouping dependent instructions, the only constraint being that the resulting graph of compounds must be a DAG. Results may only be forwarded between successive instructions within a compound. The example in Figure 8 illustrates a possible compounding for the code fragment. Instructions 2, 3, and 4 are grouped together to form a compound, each instruction within the compound must be scheduled consecutively as shown. This allows membership of a particular compound to be indicated by a single compounding bit for each instruction.

```
(1) r5=mem[r2+4]
(2) r1=mem[r2]
(3) r2=r1*321
(4) r3=r2+r5
```

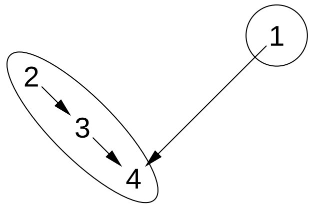

Figure 8. Example Compounding

The architecture consists of a number of independent instruction buffers and forwarded data result queues much like the queued-based architecture described in Section 4. A significant difference is that data-forwarding and instruction issue now exploit the explicit dependency information provided by compounding. An instruction, after it has been issued, is now able to make a request to forward its result to an instruction waiting in another instruction buffer. This partly implements both a forwarding and update operation without the need for the broadcasting of result identifiers. For those operands which may not be obtained via forwarding, initialisation of their status is provided by reference to a scoreboard much like the one described previously. Operations to read and update entries in the instruction buffers are queued upon issue.

Out-of-order instruction issue, as in the case described in the previous section, forces some forwarding operations to be cancelled. In the case of instruction compounding this is necessary when the instruction, on receipt of a forwarding request, cannot issue immediately. This is necessary to both avoid deadlock and guarantee that forwarded data is consumed in the correct order.

6 Results

Results were obtained from a trace-driven event-based simulator running well-known benchmarks such as *cjpeg*, *compress*, *fgrep*, *gcc*, *go*, *perl* and *xlisp*. Around 1 million instructions were simulated for each benchmark, the size of which is at present limited by the complexity of the asynchronous models. Instruction and memory reference traces were collected using QPT2 [3]. The Dinero cache simulator [10] is integrated into the simulator to provide information concerning cache misses. A 64k, 2-way set associative data cache and perfect instruction cache is used in all the simulations.

Instruction compounds were selected at random using a greedy graph partitioning algorithm with a maximum compound length set to 10. Note that no optimisations were performed on the schedule for either the compounded instructions, or the queue-based asynchronous model. Perfect branch prediction and instruction fetch bandwidth are assumed. Models which perform out-of-order instruction issue, excluding the queue-based architecture, are capable of executing memory instructions out of order with the aid of speculative memory disambiguation, although store-to-load forwarding is not implemented at present. Key architectural parameters are listed in Table 1. In the case of the queued and compounded models additional adders (equal to the number of memory units) are incorporated to generate memory addresses, in the architectures with central instruction windows ALUs are shared between address calculations and ALU instructions.

Parameter	**Number**
No. of instrs. fetched per mem. cycle	4
Complex ALU	1
ALU	2
Memory Units	1 or 2
Logical Registers	32
Physical Registers	80
Total No. of Instruction buffers	64

Table 1. Architectural Parameters

Results are included for three main asynchronous processor configurations, labelled as: *ACIW* (Asynchronous Central Instruction Window) and *ACIWc* (ACIW with source counters), *Queued* (the queue-based architecture described in Section 4) and *Comp* (Instruction Compounding). Where appropriate, results have been obtained for architectures containing 1 or 2 forwarding registers (f), and 1 or 2 register read ports (r), per functional unit.

Parameter	**Delay (% Cycle Time)**
Memory Access	100
Register File (8 read)	100
Register File (4 read)	78
Scoreboard Access	35
4-way arbitration	20
Instruction Delays	
ALU (add/shift)	50
Logical	20
Set/Move/Clear	0
Load/Store	100

Table 2. Delay Parameters

A selection of delay values, given as percentages of the synchronous clock period, are given in Figure 2. The smaller buffers or queues in the asynchronous models are assumed to have a throughput of 4 elements per cycle. Wakeup and selection in the case of the asynchronous CIW takes the equivalent of a full clock cycle.

Figures 9 and 11 summarise the IPC obtained for each model, each result being the geometric mean of the IPCs for each benchmark. Figures 10 and 12 record the percentage of operands which were obtained through data-forwarding for each model. Also shown, as a percentage of the total forwarding requests made, is the number of requests which were denied or cancelled.

In the case of the asynchronous CIW the number of forwarding requests which must be cancelled is significantly reduced with an additional forwarding register. This increases the window of opportunity for obtaining data through forwarding. Increasing the number of forwarding registers beyond two produces a reduction in performance for both CIW and queued models, for the delay models used.

In the case of both the asynchronous CIW and compounding models some scope exists for

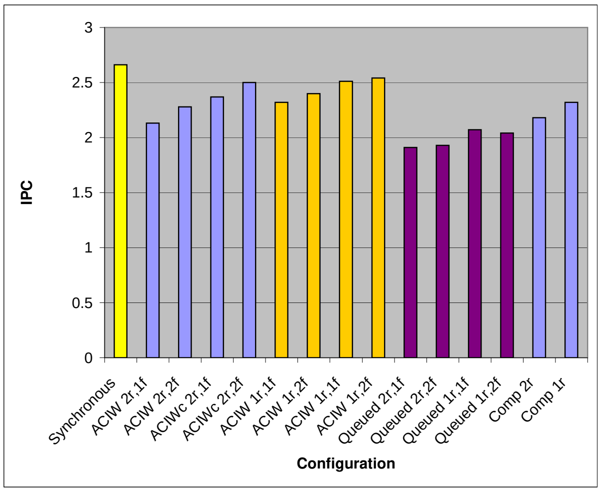

Figure 9. IPC for different processor models (single memory unit).

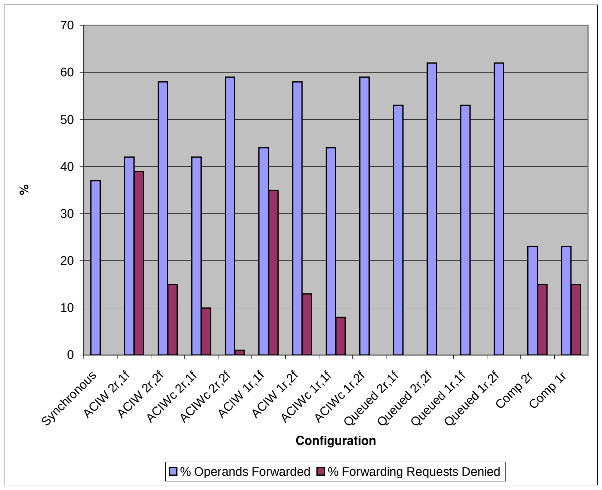

Figure 10. Percentage of operands forwarded and percentage of forwarding requests denied (single memory unit).

improving performance by reducing the number of these forwarding misses. Some attempts are already made to reduce misses in the current CIW design. For example, sources of forwarded data are reset when subsequent updates to the same instruction indicate a second operand will be obtained from the same FU. Results also indicate the potential of the counter-based scheme in reducing forwarding misses and increasing overall performance. Attempts to improve performance in the case of instruction compounding will centre on optimisations for compound selection and scheduling - which control both the forwarding of data and possible dynamic schedules.

Another factor influencing the performance of all of the asynchronous models is the reduction in the number of register file read ports. In all cases this improves performance due to a reduction in RF access time, which outweighs the penalty of serialising operand fetch in the relatively small number of cases where two or more registers must be fetched for a single instruction. The total number of operands forwarded differs significantly between the queued and compounding models; this is due to the need to restrict forwarding to between adjacent instructions and to within basic blocks. Compounding between basic blocks cannot be determined statically. The ability for all the models, including compounding, to outperform the queued model, even with a reduction in the levels of forwarding, is due to their more flexible dynamic scheduling schemes.

7 Related Work

The counterflow pipeline proposed in [15] consists of two pipelines flowing in opposite directions. Instructions in the instruction pipeline are able to inspect, and if necessary copy results from earlier instructions, as they flow past in the result pipeline. Unfortunately the orderings imposed upon data in the architecture may lead to local congestion and it has been suggested that from a purely performance viewpoint they do not compete with traditional architectures [2]. Attempts have also been made to exploit the counterflow pipeline structure to perform dynamic scheduling [18].

The rotary pipeline [12] again exploits a regular pipeline structure. Here results flow between a number of functional units organised into a ring. The availability of data on a particular bus is determined during issue and does not require

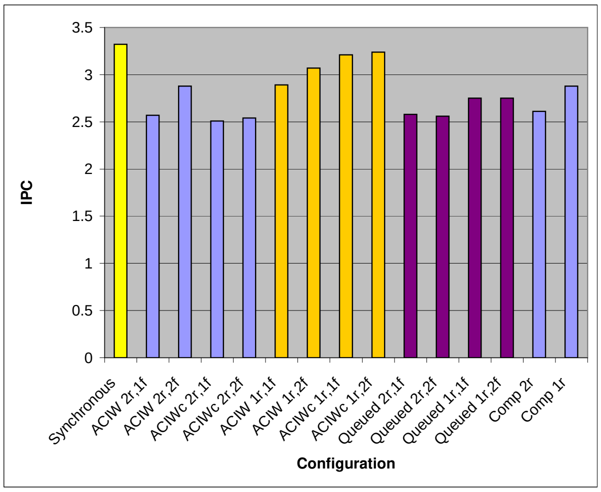

Figure 11. IPC for different processor models (two memory units).

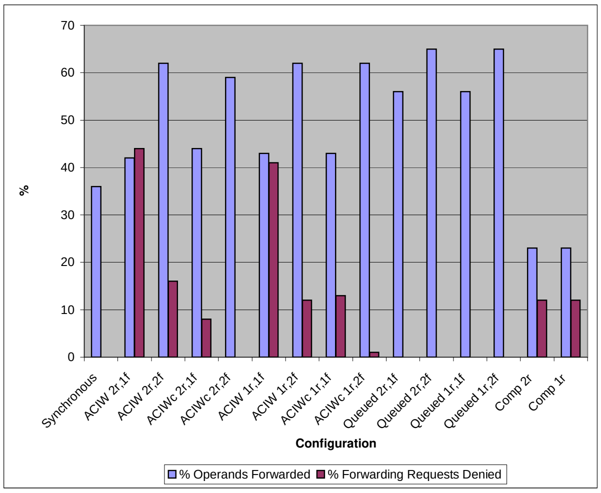

Figure 12. Percentage of operands forwarded and percentage of forwarding requests denied (two memory units).

the complex arbitration associated with the counterflow structure.

SCALP [5] was an attempt to develop a low-power superscalar processor, based on the ideas of explicit forwarding. The register bank is removed and instructions simply indicate the functional unit to which their result is to be sent. One problem of such an approach is that if results are required more than once, explicit duplicate operations must be performed. Problems also occur when the destination of a results cannot be determined at compile-time. A functional unit which is effectively a register bank is required to handle some communications.

Other implementations which have attempted to exploit asynchrony have used more traditional architectures [7, 14, 11]. These have provided promising power/performance results and have reported significant reductions in EMI.

8 Conclusions

This paper has presented three novel techniques for out-of-order instruction issue and data forwarding in the absence of global synchronisation. Their efficient implementation is important to the viability of asynchronous superscalar architectures. Results have been presented based on trace-driven simulations of detailed RTL models of the asynchronous architectures. Their performance is promising and there is scope for further improvement, in particular the compounding architecture could benefit from optimised compound selection and static scheduling.

Acknowledgements

The authors would like to acknowledge the support of the Institute for Computing Systems Architecture, Department of Informatics, University of Edinburgh. Robert Mullins would also like to acknowledge the Rainbow Group at the University of Cambridge Computer Laboratory for their kind support.

References

[1] D. K. Arvind and R. D. Mullins. A fully asynchronous superscalar processor. In *International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[2] Bill Coates, Jo Ebergen, Jon Lexau, Scott Fairbanks, Ian Jones, Alex Ridgway, David Harris, and Ivan Sutherland. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 161–172, April 1999.

[3] Mark D.Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood. WARTS Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/ larus/warts.html.

[4] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven. Hades - Towards the design of an asynchronous superscalar processor. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 200–209, London, UK, May 1995. IEEE Computer Society Press.

[5] Philip Brian Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, University of Manchester, 1995.

[6] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, pages 11–22, July 1999.

[7] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[8] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, April 1999.

[9] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 1997.

[10] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[11] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[12] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *IEE Proceedings, Computers and Digital Techniques*, 143(5):259–265, September 1996.

[13] Subbarao Palacharla, Norman P.Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical report TR-96-1328, University of Wisconsin-Madison, November 1996.

[14] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[15] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

[16] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[17] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[18] T. Werner and V.Akella. Counterflow pipeline-based dynamic instruction scheduling. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[19] Alexandre V. Yakovlev, Albert M. Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, Spring 1995.

[20] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev. Low-latency asynchronous FIFO buffers. In *Asynchronous Design Methodologies*, pages 24–31. IEEE Computer Society Press, May 1995.