Type-correct Forwarding in Asynchronous Processors

Lennart Beringer

Division of Informatics, University of Edinburgh,
King's Buildings, Edinburgh EH9 3JZ, UK
Tel.: ++44-131-6506015, Fax: ++44-131-6677209
lenb@dcs.ed.ac.uk

Abstract. This paper is concerned with type-checking assembly programs for asynchronous processors with parallel functional units and dynamic out-of-order execution. In particular, we are interested in processors which can forward results between instructions executing on the same or different functional units, based on corresponding tags in the instruction code. The architectural concepts lead to an interesting model of computation and effect the legality of programs. In order to execute correctly, programs have to use the forwarding mechanism in a coordinated way which avoids deadlock due to operand starvation and ensures that out-of-order execution behaves correctly. We show how a type system based on notation from linear logics allows to type-check straight-line code, based on a high-level view on the net-effect of instruction execution. Generalizing this, we enable forwarding across basic block boundaries and extend the opportunities to use forwarding over previous work in the asynchronous processor design community. To our knowledge, this is the first formal or informal analysis of programs for asynchronous processors with explicit forwarding.

1 Introduction

In contrast to synchronous circuits, asynchronous designs eliminate the global clock [1]. Envisioned advantages of this approach include low power consumption, low electro-magnetic interference and simpler timing constraints during the design. Instead of having gates switching at every clock tick, actions are event-triggered: a change of an output signal occurs because of a change of an input signal.
Several asynchronous processor architectures have been proposed: an asynchronous MIPS [9], the NSR/FRED architecture [11], asynchronous ARM processors (AMULETs) [4], and the Counterflow processor [12]. The shared architectural concept of these architectures is the Micropipeline [13]. It allows to lift the asynchrony from the circuit level to the architectural level, where components are composed in a pipeline and different stages are decoupled by buffering queues. Communication of data is based on receive/acknowledge signals. Several authors have observed that the basic pipeline can be generalised to a regime where different functional unit are located in parallel [9] [11]. Instructions can use different paths through a processor and instruction execution is triggered by the availability of operands. This enables dynamic out-of-order execution.
Another feature in some of these architectures is *result forwarding*. Whenever it is detected that an operand was produced by a closely preceding instruction, the data request is served by *forwarding* the value from the producing functional unit to the consumer, bypassing the register bank. Since register accesses are slow and power-consuming, the extensive usage of forwarding is expected to contribute to high speed and low energy consumption. It also increases the scalability by keeping the number of register read/write ports low. With the exception of a single special register R1 in FRED, [9] and [11] rely on hardware to identify forwarding opportunities at run-time. The SCALP architecture [3] proposed

explicit forwarding where this task is performed in the compiler. The instruction code communicates to the hardware if the result of an instruction should be forwarded.
In this paper, we generalise this concept to *operand queues* which are statically bound to functional units. The resulting computational model deviates from other abstract machines such as the Java Virtual Machine [8] and machines coming from the SECD tradition [7] [6]. Instead of using a *central synchronizing* operand *stack* it contains *several distributed queues*.
The way the forwarding mechanism is used has a crucial effect on how a program behaves. In fact, [3] observed that some programs are not suited for execution since they don't access operand queues in an admissible way. Consequently, a principled method is needed of saying when a program is guaranteed not to show any hazardous behaviour. Preferably, this check should be performed before execution time and capture forwarding in enough detail to ensure that a well-formed program can never go wrong. Ideally, such a method would allow several implementations of the processor model to be used, with slightly differing executional models. This would support the exploration of hardware design alternatives and ensure compatibility over successive incarnations of an architecture.
In this paper, we propose a *typing system* based on notation from linear logic to rule out programs which would show undesirable effects on most realistic implementations of the model. Later, we concentrate on two particular implementations: one where instructions are executed in program order and one where they are issued in program order to a set of distributed instruction queues and might thus execute out-of-order. In both cases, we consider failure by operand mismatch and deadlock due to starvation. These phenomena are classified as *programmer errors* in [3] and the hardware would either raise a resetting exception or crash if such an illegal program was to be executed. For the distributed model, we also consider functional non-determinism due to race hazards between instructions.

The experimental results given in [3] in terms of power consumption and the reduction of register usage were not as encouraging as the designer of SCALP expected. One explanation given was the inability to use forwarding sufficiently often. This was rooted in the requirements that forwarding tags had to be statically determined and instructions could only forward their result to one successor. By introducing a new instruction and extending the linear type system, we overcome these limitations, achieving *multiple* and *redirected forwarding*, also across basic block boundaries.

The rest of the paper is structured as follows. In Section 2, we outline characteristics of the SCALP architecture as far as they are necessary for our purpose and give motivating example programs which are either admissible or faulty. Section 3 introduces the basic method of associating linear types to instructions and demonstrates its ability to filter out unsafe programs. The initial instruction set is limited to straight line code within basic blocks. In Section 4, we show how to redirect forwarded values and consider branches, thus allowing more instructions to use the forwarding mechanism than [3] did. While all the restrictions on programs discussed in Sections 2 to 4 apply to several implementations of the architecture, Section 5 exemplifies how for a more specific model a more detailed analysis can be achieved. For this, the basic calculus is extended to capture additional program properties known at compile time. The paper finishes with a conclusion in Section 6.

2 Program execution à la SCALP

The architectural model is a high-level simplification of the SCALP architecture which was introduced in [3] and developed within the AMULET project at Manchester.

A data path consists of functional units of different functionality (in our case **ALU**, **ADD**, **MUL**, **MEM**), which are located in parallel to each other. Instructions arrive as a code sequence at the dispatch/decode unit, which sends each instruction in program order to its functional unit.

In contrast to traditional processors, instructions receive operands not via registers but by picking them up from the head position of *operand queues* (*op-queues*)[1]. Correspondingly, results are not written to registers but *forwarded* to the (tail of) operand queues. The expected source queues for operands are part of the instruction code, as is the destination queue for its result. An operand queue can *receive* a value from instructions in any functional unit, but all instructions taking values out of an operand queue must execute on the same FU. We use the notation $\gamma(op)$ to denote the FU to which an operand queue op is bound. Example programs use the operand queues p, r, t, u, s and q with the association $\gamma(p) = \gamma(r) = \mathbf{ADD}$, $\gamma(t) = \gamma(u) = \mathbf{MUL}$ and $\gamma(s) = \gamma(q) = \mathbf{MEM}$. No operand queue exists for $\mathbf{ALU}$.

Table 1 shows the basic instruction set, and a mapping φ giving for each instruction the functional unit it will execute on. Each instruction consists of a unique identifying number n, an opcode and up to three operands. While the i in the code for **ldc** refers to an immediate value (*constant*) which is directly appended to its destination queue, all other instructions obtain their arguments through op-queues.

Operand queues are of arbitrary length. Reading a value removes it from the head of its queue while @ denotes appending.

Instructions **ld** and **st** mention the memory M which is a mapping from memory addresses to values. For simplicity, we assume that the domains of addresses and values are integers. $\mathbf{st}$ op_1 op_2 reads the head values v_1 from op_1 and v_2 from op_2 and stores v_1 at the location v_2 in M. $\mathbf{ld}$ op_1 op_2 also reads the head value v_1 from op_1 but interprets it as the memory address from which to read and sends the result to op_2.

Non-commutative instructions such as sub op_1 op_2 op_3 read operands in the order op_1, op_2.

Opcode n	Functional Unit $\varphi(n)$	Effect
[n]**ldc** i op	**ALU**	op @ $[i]$
[n]**add** op_1 op_2 op_3	**ADD**	op_3 @ $[op_1 + op_2]$
[n]**sub** op_1 op_2 op_3	**ADD**	op_3 @ $[op_1 - op_2]$
[n]**dec** op_1 op_2	**ADD**	op_2 @ $[op_1 - 1]$
[n]**mul** op_1 op_2 op_3	**MUL**	op_3 @ $[op_1 * op_2]$
[n]**st** op_1 op_2	**MEM**	$M(op_2) = op_1$
[n]**ld** op_1 op_2	**MEM**	op_2 @ $[M(op_1)]$

Table 1. Basic instruction set

[1] SCALP also contains a register bank, which is just another functional unit. We disregard registers completely and concentrate our analysis on forwarding.

Different implementations of this instruction set might be chosen. An initial sequential processor might issue one instruction at a time. In contrast, a superscalar implementation might decouple decoding from execution by introducing Tomasulo-like waiting stations (instruction queues) in front of functional units (see Figure 1). While instructions would still be decoded and issued in program order, the time at which

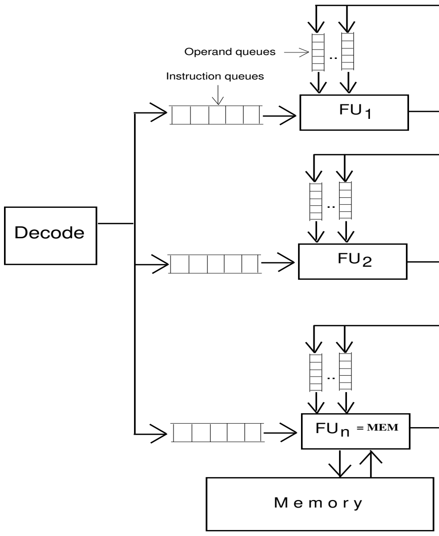

Fig. 1. Model with distributed instruction queues

an instruction actually executes would depend only on the availability of the operands and the corresponding functional unit.

Execution models impose restrictions on the formation of programs, illustrated by the following examples.

```
[1]ldc 5 p; [2]ldc 3 q; [3]ldc 7 r; [4]add p r s; [5]st s q    (1)
[1]ldc 5 p; [2]ldc 3 p; [3]ldc 7 r; [4]add p r s; [5]st s q    (2)
[1]ldc 5 p; [2]ldc 3 p; [3]ldc 7 r; [4]add p r s; [5]st s p    (3)
                         [1]ldc 5 p; [2]add p r q; [3]ldc 2 r    (4)
```

When program (1) enters the dispatch unit, the instructions are successively issued and executed on their respective functional units. Even in the super-scalar model, the instructions are forced to execute in program order: each instruction `[i]` waits for the completion of `[i-1]`, since it either queues behind it or needs its result. If, in contrast, `[2]` and `[3]` of program (1) are swapped, the resulting instruction sequence has two possible executions: instruction `[4]` can execute in parallel to (or even ahead of) the former `[2]`, since no dependency exists.

Several values can be entered into an operand queue before the first one is read. For example, the usage of r in (1) can be replaced by a second usage of p. This is admissible due to the buffering operand queues.
The first illegal program is (2) where the last instruction can never execute: no instruction sends a value to q. Consequently, we will require that the number of values entered into an operand queue and the number of reading attempts must match in a legal program.
Program (3) is illegal since it does not respect the association of op-queues to functional units. Both, the forth and the fifth instruction read from p but the former one executes on **ADD** and the latter one on **MEM**.
Finally, program (4) executes differently on the super-scalar model than on the sequential one. A sequential execution gets stuck after executing `[1]` since `[2]` misses its second operand. In a distributed execution, the third instruction can overtake the second one since they are issued to different instruction queues. It would thus show the same overall result as the program `[1];[3];[2]`. In this paper, we take the view that programs should only be called legal if they execute under at least the two models discussed. Therefore, we reject programs of the shape (4) despite the fact that the number of values entered into and read from operand queues match.

In summary, the following failures are harmful and have to be eliminated before execution time.

1. Instructions attempting to read from operand queues to which they have no access.
2. Programs entering more values into operand queues than are consumed.
3. Programs causing a deadlock since more operands are attempted to be read from the operand queues than are available.
4. Code sequences which deadlock under one execution model but execute under another one.

The following section introduces a type-based analysis which allows to rule out these errors at compile time for straight-line programs.

3 Linearly typed instructions and closed programs

The programs in the preceding section highlighted issues separating code sequences which execute correctly from faulty ones and thus motivated the need for a formal system distinguishing them from each other. In this section, we propose a method which is based on *typing*, i.e. on associating abstract properties to basic instructions and combining these statements to judgments about a program. In effect, a program will only pass the type-checker if it is free from errors according to the failure model under consideration: *well-typed programs can't go wrong*. We present a typing calculus based on *linear logic* [5] where the type of an instruction (and consequently of a program) denotes an abstraction of the effect its execution has on the operand queues.
The motivation for using *linear* logic lies in its ability to model resource sharing and contention at a level of detail which respects the multiplicities of items [15]. In our case, there will be one atom for each value in an operand queue, and a situation with two values should clearly be distinguished from a situation with only one value: in the first configuration, an instruction **add** *can* execute while in the second it *cannot*.
We associate a type of the form $S \multimap T$ to each instruction, where S describes the requirements necessary for executing the instruction and T the resulting configuration. Both S and T are snapshots of situations in the operand queues and respect the *number* of data items in each queue. However, they abstract from the particular values and the order of items in the queues.

Definition 1. *A* judgment *is a statement of the form $s : S \multimap T$ where s is a sequence of instructions $s_1; \ldots ; s_n$ and S and T are formal $\otimes$-products over the set of operand queues. We use $\mathbf{1}$ as the neutral element for $\otimes$.*

The following axiom rules allow to associate judgments to instructions of Table 1.

$$\frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{[n]ins\ op_1\ op_2\ op_3 :\ op_1 \otimes op_2 \multimap op_3}\ ins \in \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}\}$$

$$\frac{}{[n]\mathbf{ldc}\ i\ op :\ \mathbf{1} \multimap op} \qquad \frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{[n]\mathbf{st}\ op_1\ op_2 :\ op_1 \otimes op_2 \multimap \mathbf{1}}$$

$$\frac{\gamma(op_1) = \varphi(n)}{[n]ins\ op_1\ op_2 :\ op_1 \multimap op_2}\ ins \in \{\mathbf{dec}, \mathbf{ld}\}$$

These rules allow to equip an instruction `[n]` with a type $S \multimap T$ such that any configuration where there are S data items present in the operand queues allows `[n]` to execute, and the effect of execution is that S data items are removed and T items are added. Notice that both the order of items in the operand queues and the contained data are irrelevant.

Lemma 1. *The rules for simple instructions fulfill the following properties.*

1. *An instruction can pick up its operands only from operand queues to which its FU is connected.*
2. *An instruction can send its result to any operand queue.*

Example. *The following derivations show the type-correctness of instructions* `[3]`, `[4]` *and* `[5]` *of Program (1). (Instructions* `[1]` *and* `[2]` *are also well-typed, with derivations similar to the one for* `[3]`*.)*

$$\frac{}{\texttt{[3]ldc 7 r} : \mathbf{1} \multimap r}$$

$$\frac{\gamma(p) = \gamma(r) = \mathbf{ADD}}{\texttt{[4]add p r s} : p \otimes r \multimap s} \qquad \frac{\gamma(s) = \gamma(q) = \mathbf{MEM}}{\texttt{[5]st s q} : s \otimes q \multimap \mathbf{1}}$$

In contrast, the modified instructions `[4']add s r t` *and* `[5']st p t` *fail to type-check.*

For combining instructions to programs, we first define cancellation of operand queues in types to cater for the consumption of values produced by a code sequence by a following instruction.

Definition 2. *For products S and T, the cancellation of S by T, denoted $\frac{S}{T}$ is $\frac{V}{U}$ if U and V have no common factor and $U \otimes S = V \otimes T$.*

Appending an instruction to a sequence then proceeds according to the following rule.

$$\frac{s : S_1 \multimap S_2 \qquad i : T_1 \multimap T_2}{s; i : S_1 \otimes U \multimap V \otimes T_2}\ \frac{V}{U} = \frac{S_2}{T_1}$$

As a derived rule, one can obtain similar rule for combining code sequences s and t.
Of particular interest is the case where a program is of type $\mathbf{1} \multimap R$, for some R. In such a situation, all input necessary for executing s is created by instructions in s. If it is even of type $\mathbf{1} \multimap \mathbf{1}$, it will not only require no external input but also deliver a clean state: all data items produced are later consumed.

Definition 3. *A code sequence s for which $s : \mathbf{1} \multimap \mathbf{1}$ can be derived is called a* (closed) program.

We briefly review the typability of the example programs of Section 2.

Example. *The following derivation shows that Program 1 is well-typed.*

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\texttt{[1]ldc 5 p} : \mathbf{1} \multimap p}
\quad
\dfrac{}{\texttt{[2]ldc 3 q} : \mathbf{1} \multimap q}
}{\texttt{[1];[2]} : \mathbf{1} \multimap p \otimes q}
\quad
\dfrac{}{\texttt{[3]ldc 7 r} : \mathbf{1} \multimap r}
}{\texttt{[1];[2];[3]} : \mathbf{1} \multimap p \otimes q \otimes r}
\quad
\dfrac{\gamma(p) = \gamma(r) = \mathbf{ADD}}{\texttt{[4]add p r s} : p \otimes r \multimap s}
}{\texttt{[1];[2];[3];[4]} : \mathbf{1} \multimap s \otimes q}
\quad
\dfrac{\gamma(s) = \gamma(q) = \mathbf{MEM}}{\texttt{[5]st s q} : s \otimes q \multimap 1}
}{\texttt{[1];[2];[3];[4];[5]} : \mathbf{1} \multimap \mathbf{1}}
$$

Similarly, the two modifications mentioned in Section 2 (swapping instructions `[2]` *and* `[3]` *and replacing r by p) are well-typed and have type $\mathbf{1} \multimap \mathbf{1}$.*
Program (2) is ill-typed since `[5]` $: s \otimes q \multimap \mathbf{1}$ *cannot be appended to* `[1];[2];[3];[4]` $: \mathbf{1} \multimap p \otimes s$.
Program (3) fails to type-check, since instruction `[5]` *cannot be typed: $\gamma(s) = \gamma(p) = \mathbf{MEM}$ is not fulfilled.*
Finally, program (4) again fails to type-check when one tries to append `[2]` *to* `[1]`.

More generally, closed programs never show any of the hazardous behaviors identified in Section 2.

Proposition 1. *Well-typed closed programs fulfill the following properties.*

1. *No instruction attempts to read from an operand queue to which its FU has no access.*
2. *As many values are added into each operand queue as are attempted to be read.*
3. *Since instructions are combined in program order, a code sequence can only have type $\mathbf{1} \multimap \mathbf{1}$ if each instruction finds its input T_1 in the output S_2 of its preceding instructions. This guarantees that no program can be executable under the distributed model and deadlock for sequential execution.*

This section introduced the basic method of linear typing to see whether a a program without branches or jumps makes a coordinated usage of the forwarding mechanism. The next section generalises this concept to programs spanning several basic blocks.

4 Beyond basic blocks

In [3], it is reported that for a number of benchmark programs roughly 64% of all instructions send their result to only one successor. However, using explicit forwarding does not result in an equally high reduction in register usage. The main reason given is that this percentage is a *dynamic* one, whereas the explicit forwarding tags are computed statically at compile time. The prime example mentioned in [3] is as follows, with the program on the left and a corresponding assembly code (using registers) on the right.

```
t = 0                         mov #0 R2    //R2 := 0
for i in 1 to 100 do      L1:      ...
                              ld [..] R1   //R1 := x[i]
    t = t + x[i]          [i] add R1 R2 R2 //R2 += R1              (5)
                                   ...
od                            brne L1      //conditional jump
a = t                     [j] st R2 M
```

The result of each *dynamic* execution of the $\mathbf{add}$ instruction `[i]` is indeed used by only one successor: either by instruction `[i]` itself (if the branch is taken) or by instruction `[j]`. However, since these instructions are allocated to different instruction queues, the compiler cannot set a static operand queue tag. The only solution is to stick to register communication altogether. In this section, we overcome this limitation and extend the typing calculus correspondingly.

Table 2 introduces three new instructions **fwd**, **brne** and **skip**. These are executed on a new functional unit **BU** (branch unit) to which new operand queues (in the examples: c and d) are connected. While

Opcode n	Functional Unit $\varphi(n)$	Effect
[n]**fwd** op_1 op_2	**BU**	op_2 @ $[op_1]$
[n]**brne** op m	**BU**	read from op and change flow control correspondingly
[n]**skip** op	**BU**	remove head item from op

Table 2. New instructions **fwd**, **brne** and **skip**

fwd op_1 op_2 reads a value from op_1 and enters it into op_2, **skip** op removes the head value from op. An instruction `[n]`**brne** op m branches to instruction `[m]` provided that the value read from op does not equal 0 and issues instruction `[n+1]` otherwise.

The corresponding axioms are as follows.

$$\frac{\gamma(op_1) = \varphi(n)}{[n]\mathbf{fwd}\ op_1\ op_2 :\ op_1 \multimap op_2} \qquad \frac{\gamma(op) = \varphi(n)}{[n]\mathbf{brne}\ op\ m : op \multimap \mathbf{1}} \qquad \frac{\gamma(op) = \varphi(n)}{[n]\mathbf{skip}\ op : op \multimap \mathbf{1}}$$

We also use variations of instructions which forward their result to *two* operand queues with typing rules such as the following.

$$\frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{[n]\mathbf{sub}\ op_1\ op_2\ (op_3, op_4) :\ op_1 \otimes op_2 \multimap op_3 \otimes op_4}$$

Instructions other than $\mathbf{brne}$ can be appended to a sequence s using the rule given in the previous section. $\mathbf{brne}$ needs special treatment, depending on whether $m \leq n$ or $m > n$ in `[n]`**brne** $op\ m$. If $m \leq n$, a sequence $s; [n]\mathbf{brne}\ op\ m$ can be typed if the loop `[m] ... [n-1]` is essentially of type $T \multimap T$ for some T which appears in the the state left by s. The reason for this is that the overall output type should be the same independent of the number of iterations taken (including 0). If we take into account that **brne** also consumes a data item, we arrive at the following rule.

$$\frac{s : T_1 \multimap T \otimes S \otimes op \quad [m]; \ldots ; [n-1] : T \multimap T \otimes op \quad [n]\mathbf{brne}\ op\ m :\ op \multimap \mathbf{1}}{s; [n]\mathbf{brne}\ op\ m : T_1 \multimap T \otimes S}\ s = t; [m] \ldots [n-1]$$

When defining the effect of a (conditional) jump to an instruction which occurs later in the program than the jump instruction (i.e. $[n]\mathbf{brne}\ op\ m$ with $m > n$), the following rule postpones the typing of

[n] until all instructions between [n] and [m] have been typed. Furthermore, the rule enforces that both paths leading to [m] (the direct jump and the path executing instructions [n+1];...;[m]) lead to the same configuration T, thus ensuring that the precondition for [m] and subsequent instructions will always be the same.

$$\frac{s : T_1 \multimap T \otimes op \qquad [n]\mathbf{brne}\ op\ m : op \multimap \mathbf{1} \qquad [n+1]\ldots[m-1] : T \multimap T}{s; [n]\mathbf{brne}\ op\ m; [n+1]\ldots[m-1] : T_1 \multimap T}$$

Both rules for branches require loops to be properly nested. For example, no code sequences of the form $[n]\mathbf{brne}\ op\ m; \ldots [n']\mathbf{brne}\ op'\ m'$ where $m' > m > n' > n$ are admitted.

Example. *The following program excerpt shows the interplay between* **fwd** *and* **brne** *for realizing dynamic forwarding as required by Program 5.*

```
[1]ldc 0 d        [5]add r p d       [9]fwd d q
[2]ldc 100 c      [6]fwd c r         [10]skip c
[3]fwd d r        [7]dec r (c,c)     [11]ldc ... s
[4]ld ... p       [8]brne c 3        [12]st s q
```
(6)

Concretely, this means that the addition instruction [i] add R1 R1 R2 *sends its result to c instead of* R2 *and the branching unit redirects it to either an operand queue connected to* **ADD** *or to one connected to* **MEM**. *This in effect replaces a register read/write pair in each iteration by two (sequential) forwardings.*

An alternative technique for the situation in Program (5) can be employed if there is a $\mathbf{skip}\ op$ instruction available for each functional unit. This corresponds to having a single instruction $\mathbf{skipFU}\ fu\ op$ where $fu \in \{\mathbf{ALU}, \mathbf{ADD}, \mathbf{MEM}, \mathbf{MUL}\}$ and the allocation to functional units and the typing rule are given as follows.

Opcode n	Functional Unit $\varphi(n)$	Effect
[n]**skipFU** *fu op*	*fu*	remove head item from op

$$\frac{\gamma(op) = fu}{[n]\mathbf{skip}\ fu\ op : op \multimap \mathbf{1}}$$

Table 3. New instruction **skipFU** and its typing

This can be used by letting [i] add R1 R2 R3 forward its result to two operand queues and eliminating in each branch the value which is not needed. In Example (6), this would mean that we had instructions [5]add r p (r,q), [3]skipFU MEM q and [9]skipFU ADD r. This effectively replaces the register accesses to R2 in each iteration by two forwardings, but contrary to the solution in Example (6) these can now proceed in parallel.

5 Execution Models

The preceding sections concentrated on enforcing safety conditions. Well-typed programs execute without running into situations from which processors could not recover. However, implementations differ in their fine-grained execution, and occasionally a more detailed analysis might be desired which is more closely tailored to a specific implementation. In most cases, such a detailed analysis requires an intimate understanding of the executional model and might draw on other program information available at compile time. As an example, we consider in this section a processor with instruction queues and super-scalar out-of-order execution as outlined earlier. Although a full formal treatment of the operational model is beyond the scope of this paper, we can partially analyse programs with respect to their distributed behaviour.

The phenomenon we are interested in originates from an interference of a processor property (asynchrony visible at the programmer interface) and the forwarding discipline a program uses. In asynchronous systems, the latency of functional units when executing an instruction cannot be determined statically, since it might depend on environmental conditions (temperature) and (more drastically) on the operands. For example, a simple adder performs must faster on inputs where no carry bit develops in a lower bit which has to be sequentially promoted (ripple) to higher bits.
This can result in functional non-determinism of programs as in the following example.

```
[1]ldc 5 p; [2]ldc 3 r; [3]ldc 7 q; [4]add p r q; [5]st q q    (7)
```

Here, a race condition between instructions `[3]` and `[4]` develops because these instructions are functionally independent and execute on different functional units. Due to the unpredictable latency of execution, either one could finish first. Both instructions share the destination queue q for forwarding their result and thus the order of values as they arrive at q is not determined. Consequently, instruction `[5]` behaves unpredictably and the whole program is non-deterministic.
Although this feature might be utilised in some special circumstances, we expect that in most cases one would like to rule out non-deterministic programs before run-time, and in this section we modify our analysis accordingly.
The new type system is parametric in an acyclic relation $\sqsubseteq$ between instructions with the following property:

$$\text{if } n \sqsubseteq m, \text{ no execution exists where } m \text{ enters its functional unit before } n \text{ has finished execution.} \quad (8)$$

Since instructions allocated to the same functional unit execute in the order in which they are issued to the instruction queue, a suitable relation $\sqsubseteq$ would probably contain all pairs (n, m) where $\varphi(n) = \varphi(m)$ holds and n precedes m in the program order. More likely, $\sqsubseteq$ would also contain other pairs where the compiler is able to prove strict execution sequentiality, for example by using data-dependencies. Proofs that such relations fulfills property 8 involve a detailed formalisation of the execution model and are therefore not included in this paper.

We use the relation $\sqsubseteq$ by introducing *contexts* which store for each operand queue *op* the most recent instructions writing to *op*. Unless there is a branching instruction, there is in fact only one such instruction at any time. The rule for combining judgments of program fragments requires that instructions writing to a shared operand queue *op* be ordered by $\sqsubseteq$. This guarantees that the values arrive at *op* in a deterministic order, given by the order of their producers according to $\sqsubseteq$. If this in turn agrees with the order of reading instructions, an overall correct execution is obtained.
The following definitions modify the basic typing calculus accordingly.

Definition 4. *A* context Γ *is a set of statements of the form op* : N *where op is an operand queue name and* N *is a list of instruction IDs. A context is* valid *if it contains at most one entry for each op.*

Definition 5. *A* judgment *is a statement of the form $\Gamma \vdash s : S \multimap T$ where Γ is a valid context and s, S and T are as before. Again, we use $\mathbf{1}$ as the neutral element for $\otimes$.*

The axiom rules are modified by entering $op_i : [n]$ into the context for each op_i occurring on the right side of $\multimap$.

$$\frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{\{op_3 : [n]\} \vdash [n] ins\ op_1\ op_2\ op_3 :\ op_1 \otimes op_2 \multimap op_3}\ ins \in \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}\}$$

$$\frac{}{\{op : [n]\} \vdash [n]\mathbf{ldc}\ i\ op :\ \mathbf{1} \multimap op} \qquad \frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{\emptyset \vdash [n]\mathbf{st}\ op_1\ op_2 :\ op_1 \otimes op_2 \multimap \mathbf{1}}$$

$$\frac{\gamma(op_1) = \varphi(n)}{\{op_2 : [n]\} \vdash [n] ins\ op_1\ op_2 :\ op_1 \multimap op_2}\ ins \in \{\mathbf{dec}, \mathbf{ld}, \mathbf{fwd}\}$$

$$\frac{\gamma(op_1) = \gamma(op_2) = \varphi(n)}{\{op_3 : [n]\} \cup \{op_4 : [n]\} \vdash [n] ins\ op_1\ op_2\ (op_3, op_4) :\ (op_1 \otimes op_2) \multimap (op_3 \otimes op_4)}\ ins \in \{\mathbf{add}, \mathbf{sub}, \mathbf{mul}\}$$

$$\frac{\gamma(op) = \varphi(n)}{\emptyset \vdash [n] ins\ op\ (m) :\ op \multimap \mathbf{1}}\ ins \in \{\mathbf{brne}, \mathbf{skip}\}$$

Notice that all axioms generate valid contexts and that each entry in a context has in fact an instruction ID list of length one.
Non-symmetric merging of two contexts Δ and Γ is defined on the basis of $\sqsubseteq$.

Definition 6. *For valid contexts Γ and Δ, $\Gamma \lhd \Delta$ is given by*

$$\Gamma \lhd\ \emptyset = \Gamma$$

$$\Gamma \lhd\ (\{op : N\} \cup \Delta) = \begin{cases} (\Gamma' \cup \{op : N\}) \lhd \Delta & \text{if } \Gamma = \Gamma' \cup \{op : M\} \text{ and } \forall m \in M, \forall n \in N, m \sqsubseteq n \\ \text{undefined} & \text{if } \Gamma = \Gamma' \cup \{op : M\} \text{ but } \exists m \in M, \exists n \in N, m \not\sqsubseteq n \\ (\Gamma \cup \{op : N\}) \lhd \Delta & \text{otherwise} \end{cases}$$

The modified rule for appending a (non-**brne**) instruction to a sequence is as follows.

$$\frac{\Gamma \vdash s : S_1 \multimap S_2 \qquad \Delta \vdash i : T_1 \multimap T_2 \qquad \Gamma \lhd \Delta \text{ defined}}{\Gamma \lhd \Delta \vdash s; i : S_1 \otimes U \multimap V \otimes T_2}\ \frac{V}{U} = \frac{S_2}{T_1}$$

While cancelling values left over by s against values consumed by i is left unchanged, the new side condition on $\Gamma \lhd \Delta$ ensures that if i writes is result to an op-queue which was also used in s, it has to do so strictly later.

The rules effect the typability of our example programs.

Example. *Programs (1) to (4) are still correctly classified: (1) is accepted (it is of type $\mathbf{1} \multimap \mathbf{1}$) (2) and (4) are rejected since they include an instruction which won't receive its operand, and program (3) is rejected since it does not respect φ. The legality of program (7) is dependent on $\sqsubseteq$: when appending $\{q : 4\} \vdash$ `[4]add p r q` $: p \otimes r \multimap q$ to $\{p : 1, r : 2, q : 3\} \vdash$ `[1];[2];[3]` $: \mathbf{1} \multimap p \otimes r \otimes q$, the condition $\{p : 1, r : 2, q : 3\} \lhd \{q : 4\}$*defined *succeeds only if* $3 \sqsubseteq 4$ *holds. Since we would not expect* $3 \sqsubseteq 4$ *to hold if the target architecture consists of the super-scalar model, the program is rightly ruled out.*

Again, appending a branching instruction requires special treatment, depending on the direction of the jump. In the case where $m \leq n$, we use the following rule.

$$\frac{\Gamma \vdash s : T_1 \multimap T \otimes S \otimes op \quad \Delta \vdash [m]; \ldots ; [n-1] : T \multimap T \otimes op \quad \emptyset \vdash [n]\mathbf{brne}\ op\ m :\ op \multimap \mathbf{1} \quad \Lambda \vdash [m]; \ldots ; [n-1]; [n]\mathbf{skip}\ op; [m'] \ldots ; [(n-1)'] : T \multimap T \otimes op}{\Gamma \vdash s; [n]\mathbf{brne}\ op\ m : T_1 \multimap T \otimes S}\ s = t; [m] \ldots [n-1]$$

Notice that the overall context is the one in which s was typed. Since all instructions inside the loop are contained in s due to $m \leq n$, a later instruction sharing an op-queue with an instruction in the loop is thus forced to follow sequentially. The new condition $\Lambda \vdash [m]; \ldots ; [n-1]; [n]\mathbf{skip}\ op; [m'] \ldots ; [(n-1)'] : T \multimap T \otimes op$ represents an unfolding of two loop iterations, where the primed instructions are the copies of the unprimed ones. Without this rule, loop-carried race conditions could occur if an early instruction in an iteration k forwards to the same operand queue as a late instruction in iteration $h < k$. If $\sqsubseteq$ contains orderings between unprimed and primed instructions, it suffices to consider two unfoldings, since the types of all copies of an instruction are identical[2].

Example. *The following (incomplete) code sequence shows that the precondition involving unfoldings is necessary.*

```
[1]ldc 1 p        [5]sub p p p
[2]ldc 2 t        [6]ldc 5 c
[3]ldc 3 u        [7]brne c 2
[4]mul t u p      [8] ...
```
(9)

Let us assume that `[1]` $\sqsubseteq$ `[2]` $\sqsubseteq$ `[4]` $\sqsubseteq$ `[5]` *hold, but* `[5]` $\sqsubseteq$ `[4']` *does not. (If* $\sqsubseteq$ *includes not only instruction queue orders but also data-dependencies, this assumption is realistic. For example,* `[5]` *consumes the values produced by* `[1]` *and* `[4]`*, so it cannot execute ahead of them if the execution respects data-dependencies). Without the new side condition, this code sequence type-checked:* $\Gamma = \{p : 5, t : 2, u : 3, c : 6\} \vdash$ `[1]-[7]` $: \mathbf{1} \multimap p$*. However, it shows a hazardous situation if the branch is taken: instruction* `[5]` *might not yet have finished execution when instruction* `[4]` *is executed for the second time. Both instructions send their result to p, so the order of values in that queue is undetermined. The side condition disallows Program 9, unless* `[5]` $\sqsubseteq$ `[4']` *holds.*

If a forward branch is encountered (i.e. $\mathbf{brne}\ op\ m$ with $m > n$), the following rule is used:

$$\frac{\Gamma \vdash s : T_1 \multimap T \otimes op \qquad \emptyset \vdash [n]\mathbf{brne}\ op\ m : op \multimap \mathbf{1} \qquad \Delta \vdash s; [n]\mathbf{skip}\ op; [n+1] \ldots [m-1] : T_1 \multimap T}{\Gamma @ \Delta \vdash s; [n]\mathbf{brne}\ op\ m; [n+1] \ldots [m-1] : T_1 \multimap T}$$

where $\Gamma @ \Delta$ appends for each op an entry in Δ to the corresponding entry in Γ. Again, appending the jump is postponed until instructions $[n+1] \ldots [m-1]$ have been typed. In contrast to the corresponding rule in Section 4, this is achieved by typing $s; [n]\mathbf{skip}\ op; [n+1] \ldots [m-1]$ rather than $[n+1] \ldots [m-1]$. The reason is that we must ensure that no interference is possible if the branch is not taken. Again, both paths lead to the same resulting configuration T.

[2] We assume that $\sqsubseteq$ includes at least the orders in the instruction queues.

Example. *The following initial sequence of a program shows the interplay between a forward branch and the context operations.*

$$\begin{array}{ll} \texttt{[1]ldc 2 c} & \texttt{[5]brne c 7} \\ \texttt{[2]ldc 1 p} & \texttt{[6]inc r r} \\ \texttt{[3]ldc 2 p} & \texttt{[7]add p p r} \\ \texttt{[4]ldc 3 r} & \texttt{[8] ...} \end{array} \tag{10}$$

Again, we assume a hypothetical $\sqsubseteq$ which includes the order of instructions in instruction queues and the data dependencies $1 \sqsubseteq 5$, $\{2,3\} \sqsubseteq 7$ and $4 \sqsubseteq 6$. In particular, $4 \not\sqsubseteq 7$. The attempted derivation reaches the following point:

$$\dfrac{\dfrac{\cdots}{\{c:[1],p:[3],r:[4]\} \vdash \texttt{[1];...;[4]} : \mathbf{1} \multimap p \otimes p \otimes r \otimes c} \quad \emptyset \vdash \texttt{[5]} : c \multimap \mathbf{1} \quad \dfrac{\cdots}{\{c:[1],p:[3],r:[6]\} \vdash \texttt{[1];...;[6]} : \mathbf{1} \multimap p \otimes p \otimes r}}{\{c:[1],p:[3],r:[4,6]\} \vdash \texttt{[1];...;[7]} : \mathbf{1} \multimap p \otimes p \otimes r}$$

When it is now attempted to add $\{r:[7]\} \vdash$ `[7]` $: p \otimes p \multimap r$, the rule for appending a simple instruction fails since $\{c:[1],p:[3],r:[4,6]\} \lhd \{r:[7]\}$ is not defined because $4 \sqsubseteq 7$ does not hold. Indeed, if the branch `[5]` *is taken in an execution, a race condition develops between* `[4]` *and* `[7]` *regarding r. In contrast, no hazard develops if the branch is not taken since* `[6]` *and* `[7]` *are allocated to the same instruction queue.*

6 Conclusion

In this paper, we proposed a type system to reason about assembly programs targeting asynchronous processors. Our target architecture and the criteria according to which we classified programs originated from practical work in the asynchronous hardware community. We generalised the typing to situations where forwarding was not possible in the original work and showed how non-deterministic behaviour can be eliminated. Alternative solutions to handle incorrect program involve hardware mechanisms like a more complex dispatch unit or a scoreboard to avoid race conditions. Since these alternatives are costly in terms of hardware complexity, run-time overhead and power consumption, we expect that compiler-based techniques such as ours might contribute to achieving the aims motivating asynchrony. More work is needed before this effect can be quantitatively evaluated.

The type system we introduced was based on linear logic, and types consisted of abstractions of program executions. We believe the usage of linear types to analyse data-flow in asynchronous processors to be novel. It is orthogonal to work done in the TAL project, where (linear) type systems have been employed to reason about other aspects of assembly languages, such as memory behaviour [2] [10]. This complementary nature might result in combined type systems for reasoning about data flow inside a processor as well as across the processor-memory boundary. We also expect that linear type systems for reasoning about the number of times a variable is used in a functional program [14] will be helpful for identifying opportunities for forwarding in assembly programs.

Acknowledgments

I would like to thank D K Arvind and Colin Stirling for proposing this line of research, guiding its progress, and in particular the latter for suggestions which helped to improve the presentation of this paper.

References

1. Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.
2. Karl Crary and Greg Morrisett. Type structure for low-level programming languages. *Lecture Notes in Computer Science*, 1644:40–??, 1999.
3. Philip B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, Department of Computer Science, University of Manchester, February 1996.
4. J.D. Garside, W.J. Bainbridge, A. Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, and J.V. Woods. Amulet3i - an asynchronous system-on-chip. In *Proceedings Async 2000*, pages 162–175. IEEE Computer Society Press, April 2000.
5. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 46:1–102, 1986.
6. Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
7. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.
9. 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.
10. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Lecture Notes in Computer Science*, 1473:28–??, 1998.
11. William F. Richardson and Erik Brunvand. Fred: An architecture for a self-timed decoupled computer. Technical Report UUCS-95-008, Department of Computer Science, University of Utah, May 1995.
12. Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI-TR-94-25, Sun Microsystems Research, April 1994.
13. Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. The 1988 Turing Award Lecture.
14. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.
15. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.