Math. Struct. in Comp. Science (2008), *vol.* 18, *pp.* 373–409.
doi:10.1017/S0960129507006548 Printed in the United Kingdom

Maurer computers for pipelined instruction processing[†]

J. A. BERGSTRA[‡] and C. A. MIDDELBURG[§]

[‡]*Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, the Netherlands*
and
Department of Philosophy, Utrecht University, P.O. Box 80126, 3508 TC Utrecht, the Netherlands
Email: J.A.Bergstra@uva.nl
[§]*Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, the Netherlands*
Email: C.A.Middelburg@uva.nl

Received 13 June 2006; revised 27 June 2007

We model micro-architectures with non-pipelined instruction processing and pipelined instruction processing using Maurer machines, basic thread algebra and program algebra. We show that stored programs are executed as intended with these micro-architectures. We believe that this work provides a new mathematical approach to the modelling of micro-architectures and the verification of their correctness and the anticipated speed-up results.

1. Introduction

Pipelined instruction processing is a basic technique that is used in the design of micro-architectures (see, for example, Hennessy and Patterson (2003) or Sima (2004)). In this paper, we investigate the issue of dealing with pipelined instruction processing when modelling micro-architectures in a mathematically precise way. We model micro-architectures with non-pipelined instruction processing and pipelined instruction processing using Maurer machines, basic thread algebra and program algebra. Moreover, we show that stored programs are executed as intended with these micro-architectures.

Maurer machines are based on a model for computers proposed in Maurer (1966). Maurer's model for computers is quite different from other, better-known, models such as register machines, multi-stack machines and Turing machines (see, for example, Hopcroft *et al.* (2001)). The strength of Maurer's model is that it is close to real computers. The operations that can be performed on the state of a computer play a prominent part in the model. Basic thread algebra is a form of process algebra that was introduced in Bergstra and Loots (2002) under the name of basic polarised process algebra. It is a form of process

[†] The work presented in this paper was carried out as part of the GLANCE-project MICROGRIDS, which is funded by the Netherlands Organisation for Scientific Research (NWO).
[§] The second author was also at Eindhoven University of Technology, Department of Mathematics and Computer Science while part of the work presented in this paper was carried out.

algebra that is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called threads. Basic thread algebra is used in this paper to direct a Maurer machine in performing operations on its state. Program algebra was also introduced in Bergstra and Loots (2002). In program algebra, rather than considering the behaviour of deterministic sequential programs under execution, we consider the programs themselves. A program is viewed as an instruction sequence. The behaviour of a program is taken for a thread of the kind considered in basic thread algebra. With regard to the execution of stored programs on a Maurer machine, we take the line that the programs concerned are programs of the kind considered in program algebra.

To make it possible for threads to direct a Maurer machine in performing operations on its state, basic thread algebra must be extended, for each Maurer machine, with an operator for applying a thread to the Maurer machine from one of its states. Applying a thread to a Maurer machine amounts to generating a sequence of state changes according to the operations that the Maurer machine associates with the basic actions performed by the thread. Because a program is viewed as an instruction sequence in the setting of program algebra, the representation of programs in the memory of a Maurer machine becomes trivial.

Why did we choose to use Maurer machines, basic thread algebra and program algebra to model micro-architectures? First, other, better-known, models for computers, such as register machines, multi-stack machines and Turing machines, are too general for our purposes. Unlike Maurer's model for computers, those models have little in common with real computers. For example, a real computer has a memory, and the contents of all memory elements make up the state of the computer. Moreover, a real computer processes instructions, and the processing of an instruction results in changes of the contents of certain memory elements. The design of micro-architectures must deal with these aspects of real computers. Second, general process algebras, such as ACP (Bergstra and Klop 1984; Baeten and Weijland 1990), CCS (Milner 1980; Milner 1989) and CSP (Brookes *et al.* 1984; Hoare 1985), are also too general for our purposes. Basic thread algebra was designed as an algebra of deterministic sequential processes that interact with a machine. In Bergstra and Middelburg (2006b), we show that the processes considered in basic thread algebra can be viewed as processes that are definable over an extension of ACP with conditions introduced in Bergstra and Middelburg (2006a). However, it is quite awkward to describe and analyse processes of this kind using such a general process algebra. Third, there are two reasons for using program algebra:

1 the view that programs are instruction sequences fits in well with real computers, and
2 program behaviours are taken to be threads as considered in basic thread algebra.

We demonstrated the feasibility of the micro-architecture modelling approach taken in this paper in Bergstra and Middelburg (2007a). In the current paper, we make use of the experience gained in that feasibility study to model more advanced micro-architectures. As mentioned above, Maurer's model for computers is quite different from Turing's model. The latter model is part of the foundations of theoretical computer science, whereas the model used in our approach to model micro-architectures is relatively unknown. For that

reason, we have investigated the connections between the two models in Bergstra and Middelburg (2007b).

We treat the instruction set architecture for which the micro-architectures are modelled as a parameter that must fulfil a simple assumption: each instruction from the instruction set must be of a kind considered in program algebra. For example, program algebra considers test instructions and unconditional jump instructions, but not conditional jump instructions. Also, program algebra considers forward jump instructions, but not backward jump instructions. The effect of a conditional jump instruction can be mimicked by a test instruction and an unconditional jump instruction; and the effect of a backward jump instruction can be mimicked by a forward jump instruction since programs may be infinite instruction sequences in program algebra.

In pipelined instruction processing, conditional jump instructions have to be treated differently from unconditional jump instructions; but backward and forward jump instructions can be treated in the same way. In order to demonstrate the generality of our approach, we also look in this paper at the effect of extending program algebra with conditional jump instructions on non-pipelined and pipelined instruction processing. We also pay some attention to backward jump instructions.

We do not make the instruction set architecture for which micro-architectures are modelled explicit. In our modelling of a micro-architecture, we start from an arbitrary Maurer machine and enhance it. That Maurer machine determines the instruction set architecture for which a micro-architecture is modelled. However, there are some specific Maurer machines, called strict load/store Maurer instruction set architectures, for which the enhancement is primarily intended, and we will describe these Maurer machines in this paper.

We regard the work presented in this paper as one of the preparatory steps in developing a formal approach to design new micro-architectures, as part of a project investigating micro-threading (Bolychevsky *et al.* 1996; Jesshope and Luo 2000). This approach should allow for the verification of the correctness of new micro-architectures and their anticipated speed-up results. The work presented in this paper, as well as the preceding work presented in Bergstra and Middelburg (2007a), has convinced us that a special notation for the description of micro-architectures is desirable. However, we found that fixing an appropriate notation still requires some significant design decisions. We will return to this issue in Section 13.

The structure of this paper is as follows. First, we review Maurer computers (Section 2) and basic thread algebra (Section 3). Next, we extend basic thread algebra, for each Maurer machine, with the operator for applying a thread to the Maurer machine from one of its states (Section 4). Following this, we review program algebra (Section 5) and describe the way in which programs are represented in the memory of Maurer machines (Section 6). Then we model a micro-architecture with non-pipelined instruction processing (Section 7). After that, we model a variant of that micro-architecture with pipelined instruction processing (Sections 8 and 9). Following this, we look at the influence of the addition of conditional jump instructions (Section 10), and briefly discuss the addition of backward jump instructions (Section 11). Then, we describe strict load/store Maurer instruction set architectures (Section 12). Finally, we give some concluding remarks (Section 13).

2. Maurer computers

In this section, we briefly review Maurer computers, that is, computers as defined in Maurer (1966).

A *Maurer computer* C consists of

— a non-empty set M
— a set B with $card(B) \geqslant 2$
— a set $\mathscr{S}$ of functions $S : M \to B$
— a set $\mathscr{O}$ of functions $O : \mathscr{S} \to \mathscr{S}$

satisfying the following conditions:

— if $S_1, S_2 \in \mathscr{S}$, $M' \subseteq M$ and $S_3 : M \to B$ is such that $S_3(x) = S_1(x)$ if $x \in M'$ and $S_3(x) = S_2(x)$ if $x \notin M'$, then $S_3 \in \mathscr{S}$;
— if $S_1, S_2 \in \mathscr{S}$, then the set $\{x \in M \mid S_1(x) \neq S_2(x)\}$ is finite.

M is called the *memory*, B is called the *base set*, the members of $\mathscr{S}$ are called the *states*, and the members of $\mathscr{O}$ are called the *operations*. It is obvious that the first condition is satisfied if C is *complete*, that is, if $\mathscr{S}$ is the set of all functions $S : M \to B$, and that the second condition is satisfied if C is *finite*, that is, if M and B are finite sets.

Operations are called instructions in Maurer (1966). In the current paper, the term operation is used because of the confusion that would otherwise arise with the instructions from which program algebra programs are made up.

The memory of a Maurer computer consists of memory elements that have as content an element from the base set of the Maurer computer. The contents of all memory elements together make up a state of the Maurer computer. The operations of the Maurer computer transform states in certain ways and thus change the contents of certain memory elements. Thus, a Maurer computer has much in common with a real computer. The first condition on the states of a Maurer computer is a structural condition and the second one is a finite variability condition. We will return to these conditions, which are met by any real computer, after we have introduced the input and output regions of an operation.

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, and let $O : \mathscr{S} \to \mathscr{S}$. Then the *input region* of O, written $IR(O)$, and the *output region* of O, written $OR(O)$, are the subsets of M defined as follows[†]:

$$IR(O) = \{x \in M \mid \exists S_1, S_2 \in \mathscr{S} \cdot (\forall z \in M \setminus \{x\} \cdot S_1(z) = S_2(z) \ \&\ \exists y \in OR(O) \cdot O(S_1)(y) \neq O(S_2)(y))\},$$

$$OR(O) = \{x \in M \mid \exists S \in \mathscr{S} \cdot S(x) \neq O(S)(x)\}.$$

$OR(O)$ is the set of all memory elements that are possibly affected by O; and $IR(O)$ is the set of all memory elements that possibly affect elements of $OR(O)$ under O.

[†] We use the following precedence conventions in logical formulas. Operators bind stronger than predicate symbols, and predicate symbols bind stronger than logical connectives and quantifiers. Moreover, $\neg$ binds stronger than $\&$ and $\vee$, and $\&$ and $\vee$ bind stronger than $\Rightarrow$ and $\Leftrightarrow$. Quantifiers are given the smallest possible scope.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, let $S_1, S_2 \in \mathscr{S}$, and let $O \in \mathscr{O}$. Then $S_1 \restriction IR(O) = S_2 \restriction IR(O)$ implies $O(S_1) \restriction OR(O) = O(S_2) \restriction OR(O)$[†]. In other words, every operation transforms states that coincide on the input region of the operation to states that coincide on the output region of the operation. The second condition on the states of a Maurer computer is necessary for this fundamental property to hold. The first condition on the states of a Maurer computer could be relaxed somewhat.

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, let $O \in \mathscr{O}$, let $M' \subseteq OR(O)$, and let $M'' \subseteq IR(O)$. Then the *region affecting* M' *under* O, written $RA(M', O)$, and the *region affected by* M'' *under* O, written $AR(M'', O)$, are the subsets of M defined as follows:

$$RA(M', O) = \{x \in IR(O) \mid AR(\{x\}, O) \cap M' \neq \varnothing\}$$

$$AR(M'', O) = \{x \in OR(O) \mid \exists S_1, S_2 \in \mathscr{S} \cdot (\forall z \in IR(O) \setminus M'' \cdot S_1(z) = S_2(z) \ \&\ O(S_1)(x) \neq O(S_2)(x))\}.$$

$AR(M'', O)$ is the set of all elements of $OR(O)$ that are possibly affected by the elements of M'' under O; and $RA(M', O)$ is the set of all elements of $IR(O)$ that possibly affect elements of M' under O.

In Maurer (1966), Maurer gives many results about the relation between the input region and output region of operations, the composition of operations, the decomposition of operations and the existence of operations with specified input, output and affected regions: we summarise the main results in Bergstra and Middelburg (2007a). Recently, a revised and expanded version of Maurer (1966), which includes all the proofs, has appeared as Maurer (2006).

3. Basic thread algebra

In this section, we review BTA (Basic Thread Algebra), which is a form of process algebra that is tailored to describing the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there is a fixed but arbitrary set of *basic actions* $\mathscr{A}$ with $\mathsf{tau} \notin \mathscr{A}$. We write $\mathscr{A}_{\mathsf{tau}}$ for $\mathscr{A} \cup \{\mathsf{tau}\}$. BTA has the following constants and operators:

— the *deadlock* constant D;
— the *termination* constant S;
— for each $a \in \mathscr{A}_{\mathsf{tau}}$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where p is a term of BTA, abbreviates $p \trianglelefteq a \trianglerighteq p$.

The intuition is that each basic action performed by a thread is taken as a command to be processed by the execution environment of the thread. The processing of a command may involve a change of state of the execution environment. At completion of the processing of the command, the execution environment produces a reply value. This reply

[†] We use the notation $f \restriction D$, where f is a function and $D \subseteq \mathrm{dom}(f)$, for the function g with $\mathrm{dom}(g) = D$ such that for all $d \in \mathrm{dom}(g)$, $g(d) = f(d)$.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 1. *Axiom of* BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = x \trianglelefteq \mathsf{tau} \trianglerighteq x$	(T1)

Table 2. *Axioms for guarded recursion*

$\langle X\|E\rangle = \langle t_X\|E\rangle$	if $X = t_X \in E$	(RDP)
$E \Rightarrow X = \langle X\|E\rangle$	if $X \in \mathrm{V}(E)$	(RSP)

is either T or F, and is returned to the thread concerned. Let p and q be closed terms of BTA. Then $p \trianglelefteq a \trianglerighteq q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply) and proceed as q if the processing of a leads to the reply F (called a negative reply). The action tau plays a special role. Its execution will never change any state and always produces a positive reply.

BTA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

A *recursive specification* over BTA is a set of equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is a term of BTA that contains only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. Let t be a term of BTA containing a variable X. Then an occurrence of X in t is *guarded* if t has a subterm of the form $t' \trianglelefteq a \trianglerighteq t''$ containing this occurrence of X. A recursive specification E is *guarded* if all occurrences of variables on the right-hand sides of its equations are guarded or it can be rewritten as such a recursive specification using the equations of E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in Bergstra and Bethke (2003). A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

We extend BTA with guarded recursion by adding constants for solutions of guarded recursive specifications and axioms concerning these additional constants. For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we add a constant standing for the unique solution of E for X in the constants of BTA. The constant standing for the unique solution of E for X is denoted $\langle X|E\rangle$. Moreover, we use the following notation. Let t be a term of BTA and E be a guarded recursive specification. Then we write $\langle t|E\rangle$ for t with, for all $X \in \mathrm{V}(E)$, all occurrences of X in t replaced by $\langle X|E\rangle$. We now add the axioms for guarded recursion given in Table 2 to the axioms of BTA. In this table, X, t_X and E stand for an arbitrary variable, an arbitrary term of BTA and an arbitrary guarded recursive specification, respectively. Side conditions are added to restrict the variables, terms and guarded recursive specifications for which X, t_X and E stand. The additional axioms for guarded recursion are known as the recursive definition principle (RDP) and

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 3. *Approximation induction principle*

$\bigwedge_{n \geqslant 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	(AIP)

Table 4. *Axioms for projection operators*

$\pi_0(x) = \mathsf{D}$	(P0)
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	(P1)
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	(P2)
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	(P3)

the recursive specification principle (RSP). The equations $\langle X|E\rangle = \langle t_X|E\rangle$ for a fixed E express the fact that the constants $\langle X|E\rangle$ make up a solution of E. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express the fact that this solution is the only one.

We often write X for $\langle X|E\rangle$ if E is clear from the context. It should be borne in mind that, in such cases, we use X as a constant.

The projective limit characterisation of process equivalence on threads is based on the notion of a finite approximation of depth n. When these approximations are identical for two given threads for all n, the threads are considered to be identical. This is expressed by the infinitary conditional equation AIP (Approximation Induction Principle) given in Table 3. Here, following Bergstra and Bethke (2003), approximation of depth n is expressed in terms of a unary *projection* operator $\pi_n(\_)$. The projection operators are defined inductively by means of the axioms given in Table 4. In this table, a stands for an arbitrary member of $\mathscr{A}_{\mathsf{tau}}$. It happens that RSP follows from AIP.

The structural operational semantics of BTA and its extensions with guarded recursion and projection can be found in Bergstra and Middelburg (2005) and Bergstra and Middelburg (2007a).

From now on, we write $\mathscr{T}_{\mathsf{finrec}}$ for the set of all closed terms of BTA with guarded recursion in which no constants $\langle X|E\rangle$ for infinite E occur. We write $\mathscr{T}_{\mathsf{finrec}}(A)$, where $A \subseteq \mathscr{A}$, for the set of all closed terms from $\mathscr{T}_{\mathsf{finrec}}$ that contain only basic actions from A.

4. Applying threads to Maurer machines

In this section we introduce Maurer machines and add for each Maurer machine H a binary *apply* operator $\_ \bullet_H \_$ to BTA.

A *Maurer machine* is a tuple $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$, where $(M, B, \mathscr{S}, \mathscr{O})$ is a Maurer computer and:

— $A \subseteq \mathscr{A}$;

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 5. *Defining equations for apply operator*

$x \bullet_H \uparrow = \uparrow$	
$\mathsf{S} \bullet_H S = S$	
$\mathsf{D} \bullet_H S = \uparrow$	
$(\mathsf{tau} \circ x) \bullet_H S = x \bullet_H S$	
$(x \trianglelefteq a \trianglerighteq y) \bullet_H S = x \bullet_H O_a(S)$	if $O_a(S)(m_a) = \mathsf{T}$
$(x \trianglelefteq a \trianglerighteq y) \bullet_H S = y \bullet_H O_a(S)$	if $O_a(S)(m_a) = \mathsf{F}$

Table 6. *Rule for divergence*

$$\bigwedge_{n \geqslant 0} \pi_n(x) \bullet_H S = \uparrow \Rightarrow x \bullet_H S = \uparrow$$

— $[\![\_]\!] : A \to (\mathscr{O} \times M)$ is such that for all $S \in \mathscr{S}$ and $a \in A$, $S(\mathfrak{p}_2([\![a]\!])) \in \{\mathsf{T}, \mathsf{F}\}$[†].

The members of A are called the *basic actions* of H, and $[\![\_]\!]$ is called the *basic action interpretation function* of H. A and $[\![\_]\!]$ constitute the interface between the Maurer computer and its environment.

The apply operators associated with Maurer machines are related to the apply operators introduced in Bergstra and Ponse (2002). They allow for threads to transform states of the associated Maurer machine by means of its operations. Such state transformations produce either a state of the associated Maurer machine or the *undefined state* $\uparrow$. It is assumed that $\uparrow$ is not a state of any Maurer machine. We extend function restriction to $\uparrow$ by stipulating that $\uparrow \restriction M = \uparrow$ for any set M. The first operand of the apply operator $\_ \bullet_H \_$ associated with Maurer machine $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ must be a term from $\mathscr{T}_{\mathsf{finrec}}(A)$ and its second argument must be a state from $\mathscr{S} \cup \{\uparrow\}$.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine, $p \in \mathscr{T}_{\mathsf{finrec}}(A)$ and $S \in \mathscr{S}$. Then $p \bullet_H S$ is the state that results if all basic actions performed by thread p are processed by the Maurer machine H from initial state S. Moreover, let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the processing of a basic action a by H amounts to a state change according to the operation O_a. In the resulting state, the reply produced by H is contained in memory element m_a. If p is S, there will be no state change. If p is D, the result is $\uparrow$.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine and $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the apply operator $\_ \bullet_H \_$ is defined by the equations given in Table 5 and the rule given in Table 6. In these tables, a stands for an arbitrary member of A and S stands for an arbitrary member of $\mathscr{S}$.

[†] Let $A_1, \ldots, A_n$ be sets. Then the function from $A_1 \times \ldots \times A_n$ to A_i $(1 \leqslant i \leqslant n)$ that maps each $(a_1, \ldots, a_n) \in A_1 \times \ldots \times A_n$ to a_i is usually denoted by π_i. We write $\mathfrak{p}_i$ instead of π_i because of the confusion that would otherwise arise with the projection operator introduced in Section 3.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine, $p \in \mathscr{T}_{\mathsf{finrec}}(A)$ and $S \in \mathscr{S}$. Then p *converges* from S on H if there exists an $n \in \mathbb{N}$ such that $\pi_n(p) \bullet_H S \neq \uparrow$. We say that p *diverges* from S on H if p does not converge from S on H. The rule from Table 6 can be read as follows: if x diverges from S on H, then $x \bullet_H S$ equals $\uparrow$.

We introduce some auxiliary notions, which will be useful in proofs to come.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine and $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the *step relation* $\_ \vdash_H \_ \subseteq (\mathscr{T}_{\mathsf{finrec}}(A) \times \mathscr{S}) \times (\mathscr{T}_{\mathsf{finrec}}(A) \times \mathscr{S})$ is inductively defined as follows:

— if $p = \mathsf{tau} \circ p'$, then $(p, S) \vdash_H (p', S)$;
— if $O_a(S)(m_a) = \mathsf{T}$ and $p = p' \trianglelefteq a \trianglerighteq p''$, then $(p, S) \vdash_H (p', O_a(S))$;
— if $O_a(S)(m_a) = \mathsf{F}$ and $p = p' \trianglelefteq a \trianglerighteq p''$, then $(p, S) \vdash_H (p'', O_a(S))$.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine. Then a *full path* in $\_ \vdash_H \_$ is one of the following:

— a finite path $\langle (p_0, S_0), \ldots, (p_n, S_n) \rangle$ in $\_ \vdash_H \_$ such that there exists no $(p_{n+1}, S_{n+1}) \in \mathscr{T}_{\mathsf{finrec}}(A) \times \mathscr{S}$ with $(p_n, S_n) \vdash_H (p_{n+1}, S_{n+1})$;
— an infinite path $\langle (p_0, S_0), (p_1, S_1), \ldots \rangle$ in $\_ \vdash_H \_$.

Moreover, let $p \in \mathscr{T}_{\mathsf{finrec}}(A)$ and $S \in \mathscr{S}$. Then the *full path* of (p, S) on H is the unique full path in $\_ \vdash_H \_$ from (p, S). If p converges from S on H, the full path of (p, S) on H is called the *computation* of (p, S) on H and we write $\|(p, S)\|_H$ for the length of the computation of (p, S) on H.

It is easy to see that $(p_0, S_0) \vdash_H (p_1, S_1)$ only if $p_0 \bullet_H S_0 = p_1 \bullet_H S_1$, and that $\langle (p_0, S_0), \ldots, (p_n, S_n) \rangle$ is the computation of (p_0, S_0) on H only if $p_n = \mathsf{S}$ and $S_n = p_0 \bullet_H S_0$. It is also easy to see that, if p_0 converges from S_0 on H, then $\|(p_0, S_0)\|_H$ is the least $n \in \mathbb{N}$ such that $\pi_n(p_0) \bullet_H S_0 \neq \uparrow$.

In the definition of a Maurer machine, we could have taken a function $[\![\_]\!]$ that associates with each $a \in A$ a triple $(n_a, O_a, m_a) \in M \times \mathscr{O} \times M$ such that $S(n_a), S(m_a) \in \{\mathsf{T}, \mathsf{F}\}$ for all $S \in \mathscr{S}$. In that case, $S(n_a)$ would indicate whether basic action a is enabled in state S, that is, whether the processing of a is not blocked in state S. In this paper, we consider only threads that are behaviours of deterministic sequential programs under execution. For such behaviours, we are not interested in the possibility that some basic actions are not always enabled, so we assume that all basic actions of a Maurer machine are enabled in all states. Under this assumption, it is sufficient that the function $[\![\_]\!]$ associates with each $a \in A$ a pair $(O_a, m_a) \in \mathscr{O} \times M$ as in the definition given at the beginning of this section.

5. Program algebra

In this section, we review PGA (ProGram Algebra), an algebra of sequential programs based on the idea that sequential programs are, in essence, sequences of instructions. PGA provides a program notation for finite-state threads. A hierarchy of program notations that provide more and more sophisticated programming features are rooted in PGA (see Bergstra and Loots (2002)).

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

In PGA, it is assumed that there is a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions*. PGA has the following *primitive instructions*:

— for each $a \in \mathfrak{A}$, a *void basic instruction* a
— for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$
— for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$
— for each $k \in \mathbb{N}$, a *forward jump instruction* $\#k$
— a *termination instruction* !.

We write $\mathfrak{I}$ for the set of all primitive instructions.

The intuition is that the execution of a basic instruction a may modify a state and produces T or F at its completion. In the case of a positive test instruction $+a$, basic instruction a is executed and execution proceeds with the next primitive instruction if T is produced, otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. In the case where T is produced and there is not at least one subsequent primitive instruction and in the case where F is produced and there are not at least two subsequent primitive instructions, deadlock occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a void basic instruction a, the value produced is disregarded: execution always proceeds as if T is produced. The effect of a forward jump instruction $\#k$ is that execution proceeds with the kth next instruction of the program concerned. If k equals 0 or the kth next instruction does not exist, $\#k$ results in deadlock. The effect of the termination instruction ! is that execution terminates.

The thread extraction operator introduced below, together with the apply operators introduced in Section 4, allows us to associate operations of Maurer machines with basic instructions, and consequently with primitive instructions of PGA.

PGA has the following constants and operators:

— for each $u \in \mathfrak{I}$, an *instruction* constant u;
— the binary *concatenation* operator $\_ \,;\, \_$;
— the unary *repetition* operator $\_^{\omega}$.

Closed terms of PGA are considered to denote programs. The intuition is that a program is, in essence, a non-empty finite or infinite sequence of primitive instructions. These sequences are called *single pass instruction sequences* because PGA has been designed to enable single pass execution of instruction sequences: each instruction can be dropped after it has been executed. Programs are considered to be equal if they represent the same single pass instruction sequence. The axioms for instruction sequence equivalence are given in Table 7. In this table, n stands for an arbitrary natural number greater than 0. For each $n > 0$, the term X^n is defined by induction on n as follows: $X^1 = X$ and $X^{n+1} = X \,;\, X^n$. The *unfolding* equation $X^{\omega} = X \,;\, X^{\omega}$ is derivable. Each closed term of PGA is derivably equal to a term in *canonical form*, that is, a term of the form P or $P \,;\, Q^{\omega}$, where P and Q are closed terms of PGA that do not contain the repetition operator.

Each closed term of PGA is considered to denote a program of which the behaviour is a finite-state thread, taking the set $\mathfrak{A}$ of basic instructions for the set $\mathscr{A}$ of actions. The *thread extraction* operator $|\_|$ assigns a thread to each program. The thread extraction operator is defined by the equations given in Table 8 (for $a \in \mathfrak{A}$, $k \in \mathbb{N}$ and $u \in \mathfrak{I}$)

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 7. *Axioms of* PGA

$(X\,;Y)\,;Z = X\,;(Y\,;Z)$	(PGA1)
$(X^n)^\omega = X^\omega$	(PGA2)
$X^\omega\,;Y = X^\omega$	(PGA3)
$(X\,;Y)^\omega = X\,;(Y\,;X)^\omega$	(PGA4)

Table 8. *Defining equations for thread extraction operator*

$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert \#k\rvert = \mathsf{D}$
$\lvert a\,;X\rvert = a \circ \lvert X\rvert$	$\lvert \#0\,;X\rvert = \mathsf{D}$
$\lvert +a\rvert = a \circ \mathsf{D}$	$\lvert \#1\,;X\rvert = \lvert X\rvert$
$\lvert +a\,;X\rvert = \lvert X\rvert \trianglelefteq a \trianglerighteq \lvert \#2\,;X\rvert$	$\lvert \#k+2\,;u\rvert = \mathsf{D}$
$\lvert -a\rvert = a \circ \mathsf{D}$	$\lvert \#k+2\,;u\,;X\rvert = \lvert \#k+1\,;X\rvert$
$\lvert -a\,;X\rvert = \lvert \#2\,;X\rvert \trianglelefteq a \trianglerighteq \lvert X\rvert$	$\lvert !\rvert = \mathsf{S}$
	$\lvert !\,;X\rvert = \mathsf{S}$

and the rule given in Table 9. This rule is expressed in terms of the *structural congruence* predicate $\_ \cong \_$, which is defined by the formulas given in Table 10 (for $n, m, k \in \mathbb{N}$ and $u_1, \ldots, u_n, v_1, \ldots, v_{m+1} \in \mathfrak{I}$).

The equations given in Table 8 do not cover the case where there is a cyclic chain of forward jumps. Programs are structurally congruent if they are the same after removing all chains of forward jumps in favour of direct jumps. Because a cyclic chain of forward jumps corresponds to #0, the rule from Table 9 can be read as follows: if X starts with a cyclic chain of forward jumps, then $\lvert X\rvert$ equals D. It is easy to see that the thread extraction operator assigns the same thread to structurally congruent programs. Therefore, the rule from Table 9 can be replaced by the following generalisation: $X \cong Y \Rightarrow \lvert X\rvert = \lvert Y\rvert$.

Let E be a finite guarded recursive specification over BTA and P_X be a closed term of PGA for each $X \in \mathrm{V}(E)$. Let E' be the set of equations that results from replacing all occurrences of X in E by $\lvert P_X\rvert$ for each $X \in \mathrm{V}(E)$. If E' can be obtained by applications of axioms PGA1–PGA4, the defining equations for the thread extraction operator, and the rule for cyclic jump chains, then $\lvert P_X\rvert$ is the solution of E for X. Such a finite guarded recursive specification can always be found. Thus, the behaviour of each closed PGA term is a thread that is definable by a finite guarded recursive specification over BTA. Moreover, each finite guarded recursive specification over BTA can be translated to a

Table 9. *Rule for cyclic jump chains*

$$X \cong \#0\,;Y \;\Rightarrow\; \lvert X\rvert = \mathsf{D}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 10. *Defining formulas for structural congruence predicate*

$$\#n+1\,;u_1\,;\ldots;u_n\,;\#0 \cong \#0\,;u_1\,;\ldots;u_n\,;\#0$$
$$\#n+1\,;u_1\,;\ldots;u_n\,;\#m \cong \#m+n+1\,;u_1\,;\ldots;u_n\,;\#m$$
$$(\#n+k+1\,;u_1\,;\ldots;u_n)^\omega \cong (\#k\,;u_1\,;\ldots;u_n)^\omega$$
$$\#m+n+k+2\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega \cong \#n+k+1\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega$$
$$X \cong X$$
$$X_1 \cong Y_1 \;\&\; X_2 \cong Y_2 \Rightarrow X_1\,;X_2 \cong Y_1\,;Y_2 \;\&\; {X_1}^\omega \cong {Y_1}^\omega$$

PGA program of which the behaviour is the solution of the finite guarded recursive specification concerned.

Closed terms of PGA are loosely called PGA *programs*. PGA programs in which the repetition operator does not occur are called *finite* PGA programs. Henceforth, we write $\mathscr{P}_{\mathsf{fin}}$ for the set of all finite PGA programs. We write $\mathscr{P}_{\mathsf{fin}}(A)$, where $A \subseteq \mathfrak{A}$, for the set of all closed terms from $\mathscr{P}_{\mathsf{fin}}$ that contain only basic instructions from A.

In the remainder of this paper, with the exception of Section 11, we consider finite PGA programs only.

6. Stored programs

In this short section we describe how we can represent PGA programs in the memory of a Maurer machine.

It is assumed that a fixed but arbitrary finite set $\mathsf{M}_{\mathsf{prog}}$ and a fixed but arbitrary bijection $\mathsf{m}_{\mathsf{prog}} : [0, \mathit{card}(\mathsf{M}_{\mathsf{prog}}) - 1] \to \mathsf{M}_{\mathsf{prog}}$ have been given. $\mathsf{M}_{\mathsf{prog}}$ is called the *program memory*. We write $\mathit{size}(\mathsf{M}_{\mathsf{prog}})$ for $\mathit{card}(\mathsf{M}_{\mathsf{prog}})$. Let $n, n' \in [0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$ be such that $n \leqslant n'$. Then we write $\mathsf{M}_{\mathsf{prog}}[n]$ for $\mathsf{m}_{\mathsf{prog}}(n)$, and $\mathsf{M}_{\mathsf{prog}}[n, n']$ for $\{\mathsf{m}_{\mathsf{prog}}(k) \mid n \leqslant k \leqslant n'\}$.

The program memory is a memory of which the elements can be addressed by means of members of $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$. We write $\mathsf{MA}_{\mathsf{prog}}$ for $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$ and $\mathsf{MA}'_{\mathsf{prog}}$ for $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}})]$.

The program memory elements are meant to contain the primitive instructions that form part of a finite PGA program.

We write $\mathfrak{I}_{\mathsf{prog}}$ for $\mathfrak{I} \setminus \{\#k \mid k > \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1\}$. $\mathfrak{I}_{\mathsf{prog}}$ is the *program memory base set*. We write $\mathsf{S}_{\mathsf{prog}}$ for the set of all functions $S_{\mathsf{prog}} : \mathsf{M}_{\mathsf{prog}} \to \mathfrak{I}_{\mathsf{prog}}$.

Let $P = u_1\,;\ldots;u_n \in \mathscr{P}_{\mathsf{fin}}$ with $n \leqslant \mathit{size}(\mathsf{M}_{\mathsf{prog}})$. Then the *stored representation* of P, written $\mathsf{s}_{\mathsf{prog}}(P)$, is the unique function $s_{\mathsf{prog}} : \mathsf{M}_{\mathsf{prog}}[0, n-1] \to \mathfrak{I}_{\mathsf{prog}}$ such that for all $i \in [0, n-1]$, we have $s_{\mathsf{prog}}(\mathsf{M}_{\mathsf{prog}}[i]) = u_{i+1}$. We call $\mathsf{s}_{\mathsf{prog}}(P)$ a *stored program*.

Note that $\mathsf{s}_{\mathsf{prog}}(u_1\,;\ldots;u_n)$ is not defined if $n > \mathit{size}(\mathsf{M}_{\mathsf{prog}})$. The size of the program memory restricts the programs that can be stored.

7. Non-pipelined instruction processing

In this section we model a micro-architecture with non-pipelined instruction processing. We will not specify explicitly the instruction set architecture for which this

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

micro-architecture is modelled. We start from an arbitrary Maurer machine and enhance it. That Maurer machine determines the instruction set architecture for which a micro-architecture is modelled. However, there are some specific Maurer machines for which the enhancement is primarily intended, and these Maurer machines will be introduced in Section 12. From now on, when we write 'PGA instruction' we will mean 'primitive instruction of PGA'.

We enhance Maurer machines by extending the memory with a *program memory* ($\mathsf{M}_{\mathsf{prog}}$), a *program counter upper bound register* (pcbr), a *program counter* (pc), an *instruction register* (ir), a *decoded instruction type register* (ditr), a *basic action register* (bar), a *displacement register* (dr), an *executed instruction type register* (eitr), an *instruction reply register* (irr), a *fetch reply register* ($\mathsf{rr}_{\mathsf{fetch}}$), a *pre-process reply register* ($\mathsf{rr}_{\mathsf{prep}}$), an *execute reply register* ($\mathsf{rr}_{\mathsf{exec}}$) and a *post-process reply register* ($\mathsf{rr}_{\mathsf{postp}}$), and the operation set with a *fetch* operation (O_{fetch}), a *pre-process* operation (O_{prep}), an *execute* operation (O_{exec}) and a *post-process* operation (O_{postp}). Moreover, we replace the basic actions of the original Maurer machine by basic actions fetch, prep, exec and postp, with which the operations O_{fetch}, O_{prep}, O_{exec} and O_{postp} are associated. The resulting Maurer machines are called SP-NPL-enhancements. SP stands for stored program and NPL stands for non-pipelined instruction processing. In SP-NPL-enhancements of Maurer machines, the five *instruction types* bsc, ptst, ntst, fjmp and term are distinguished. These types correspond to the five kinds of PGA instructions introduced in Section 5. Henceforth, we write IT for the set $\{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{fjmp}, \mathsf{term}\}$. The memory elements pcbr, pc, ir, ditr, bar, dr, eitr and irr are used to communicate information between the execution handling operations O_{fetch}, O_{prep}, O_{exec} and O_{postp}. The memory elements $\mathsf{rr}_{\mathsf{fetch}}$, $\mathsf{rr}_{\mathsf{prep}}$, $\mathsf{rr}_{\mathsf{exec}}$ and $\mathsf{rr}_{\mathsf{postp}}$ are the reply registers of the execution handling operations O_{fetch}, O_{prep}, O_{exec} and O_{postp}, respectively. It is assumed that pcbr, pc, ir, ditr, bar, dr, eitr, irr, $\mathsf{rr}_{\mathsf{fetch}}$, $\mathsf{rr}_{\mathsf{prep}}$, $\mathsf{rr}_{\mathsf{exec}}$ and $\mathsf{rr}_{\mathsf{postp}}$ are pairwise different memory elements. From now on, we will write M'_{ip} for $\{\mathsf{pcbr}, \mathsf{pc}, \mathsf{ir}, \mathsf{ditr}, \mathsf{bar}, \mathsf{dr}, \mathsf{eitr}, \mathsf{irr}\}$ and M'_{rr} for $\{\mathsf{rr}_{\mathsf{fetch}}, \mathsf{rr}_{\mathsf{prep}}, \mathsf{rr}_{\mathsf{exec}}, \mathsf{rr}_{\mathsf{postp}}\}$. We assume that $\mathsf{M}_{\mathsf{prog}} \cap (M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}}) = \varnothing$. From now on, we will write $\mathbb{B}$ for the set $\{\mathsf{T}, \mathsf{F}\}$. After presenting a precise definition of an SP-NPL-enhancement, we will give further explanations of how an SP-NPL-enhancement operates.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine such that $M \cap (\mathsf{M}_{\mathsf{prog}} \cup M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}}) = \varnothing$ and $\mathsf{fetch}, \mathsf{prep}, \mathsf{exec}, \mathsf{postp} \notin A$, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the *SP-NPL-enhancement* of H is the Maurer machine $H' = (M', B', \mathscr{S}', \mathscr{O}', A', [\![\_]\!]')$ such that

$$
\begin{aligned}
M' &= M \cup \mathsf{M}_{\mathsf{prog}} \cup M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}},\\
B' &= B \cup \mathsf{MA}'_{\mathsf{prog}} \cup \mathfrak{I}_{\mathsf{prog}} \cup IT \cup A \cup \mathbb{B},\\
\mathscr{S}' &= \{S' : M' \to B' \mid\\
&\qquad S' \restriction M \in \mathscr{S} \ \ \&\ \ S' \restriction \mathsf{M}_{\mathsf{prog}} \in \mathsf{S}_{\mathsf{prog}} \ \ \&\ \ S'(\mathsf{pcbr}) \in \mathsf{MA}_{\mathsf{prog}} \ \ \&\\
&\qquad S'(\mathsf{pc}) \in \mathsf{MA}'_{\mathsf{prog}} \ \ \&\ \ S'(\mathsf{ir}) \in \mathfrak{I}_{\mathsf{prog}} \ \ \&\\
&\qquad S'(\mathsf{ditr}) \in IT \ \ \&\ \ S'(\mathsf{bar}) \in A \ \ \&\ \ S'(\mathsf{dr}) \in \mathsf{MA}_{\mathsf{prog}} \ \ \&\\
&\qquad S'(\mathsf{eitr}) \in IT \ \ \&\ \ S'(\mathsf{irr}) \in \mathbb{B} \ \ \&\\
&\qquad S'(\mathsf{rr}_{\mathsf{fetch}}) \in \mathbb{B} \ \ \&\ \ S'(\mathsf{rr}_{\mathsf{prep}}) \in \mathbb{B} \ \ \&\ \ S'(\mathsf{rr}_{\mathsf{exec}}) \in \mathbb{B} \ \ \&\ \ S'(\mathsf{rr}_{\mathsf{postp}}) \in \mathbb{B}\}
\end{aligned}
$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

$$\begin{aligned}
\mathcal{O}' = \{&O' : \mathcal{S}' \to \mathcal{S}' \mid \\
&\exists O \in \mathcal{O} \bullet \forall S' \in \mathcal{S}' \bullet \\
&\quad (O'(S') \restriction M = O(S' \restriction M) \ \ \&\ \ O'(S') \restriction (M' \setminus M) = S' \restriction (M' \setminus M))\} \\
&\cup \{O_{\mathsf{fetch}}, O_{\mathsf{prep}}, O_{\mathsf{exec}}, O_{\mathsf{postp}}\} \\
A' = \{&\mathsf{fetch}, \mathsf{prep}, \mathsf{exec}, \mathsf{postp}\} \\
[\![a]\!]' = (&O_a, \mathsf{rr}_a) \quad \text{for all } a \in A'.
\end{aligned}$$

O_{fetch} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$

$$\begin{aligned}
O_{\mathsf{fetch}}(S') \restriction M &= S' \restriction M \\
O_{\mathsf{fetch}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \\
O_{\mathsf{fetch}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S')(\mathsf{pc}) &= S'(\mathsf{pc}) + 1 && \text{if } S'(\mathsf{pc}) + 1 \leqslant S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S')(\mathsf{pc}) &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S')(\mathsf{ir}) &= S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) && \text{if } S'(\mathsf{pc}) \leqslant S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S')(\mathsf{ir}) &= \#0 && \text{if } S'(\mathsf{pc}) > S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} \\
O_{\mathsf{fetch}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\} \\
O_{\mathsf{fetch}}(S')(\mathsf{rr}_{\mathsf{fetch}}) &= \mathsf{T} && \text{if } S'(\mathsf{pc}) \leqslant S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S')(\mathsf{rr}_{\mathsf{fetch}}) &= \mathsf{F} && \text{if } S'(\mathsf{pc}) > S'(\mathsf{pcbr}) \\
O_{\mathsf{fetch}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{fetch}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{fetch}}\}).
\end{aligned}$$

O_{prep} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
O_{\mathsf{prep}}(S') \restriction M &= S' \restriction M \\
O_{\mathsf{prep}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \\
O_{\mathsf{prep}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \\
O_{\mathsf{prep}}(S') \restriction \{\mathsf{pc}, \mathsf{ir}\} &= S' \restriction \{\mathsf{pc}, \mathsf{ir}\} \\
O_{\mathsf{prep}}(S')(\mathsf{ditr}) &= \mathfrak{p}_1(dec(S')) \\
O_{\mathsf{prep}}(S')(\mathsf{bar}) &= \mathfrak{p}_2(dec(S')) \\
O_{\mathsf{prep}}(S')(\mathsf{dr}) &= \mathfrak{p}_3(dec(S')) \\
O_{\mathsf{prep}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\} \\
O_{\mathsf{prep}}(S')(\mathsf{rr}_{\mathsf{prep}}) &= \mathsf{T} \\
O_{\mathsf{prep}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{prep}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{prep}}\}),
\end{aligned}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

where $dec : \mathscr{S}' \to IT \times A \times \mathsf{MA}_{\mathsf{prog}}$ is defined by

$$\begin{aligned}
dec(S') &= (\mathsf{bsc}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = a\\
dec(S') &= (\mathsf{ptst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = +a\\
dec(S') &= (\mathsf{ntst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = -a\\
dec(S') &= (\mathsf{fjmp}, S'(\mathsf{bar}), k) && \text{if } S'(\mathsf{ir}) = \#k\\
dec(S') &= (\mathsf{term}, S'(\mathsf{bar}), S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = !.
\end{aligned}$$

O_{exec} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$

$$\begin{aligned}
O_{\mathsf{exec}}(S') \restriction M &= O_{S'(\mathsf{bar})}(S' \restriction M) && \text{if } opc(S')\\
O_{\mathsf{exec}}(S') \restriction M &= S' \restriction M && \text{if } \neg\, opc(S')\\
O_{\mathsf{exec}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}}\\
O_{\mathsf{exec}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr})\\
O_{\mathsf{exec}}(S') \restriction \{\mathsf{pc}, \mathsf{ir}\} &= S' \restriction \{\mathsf{pc}, \mathsf{ir}\}\\
O_{\mathsf{exec}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\}\\
O_{\mathsf{exec}}(S')(\mathsf{eitr}) &= S'(\mathsf{ditr})\\
O_{\mathsf{exec}}(S')(\mathsf{irr}) &= O_{S'(\mathsf{bar})}(S' \restriction M)(m_{S'(\mathsf{bar})}) && \text{if } opc(S')\\
O_{\mathsf{exec}}(S')(\mathsf{irr}) &= \mathsf{T} && \text{if } \neg\, opc(S')\\
O_{\mathsf{exec}}(S')(\mathsf{rr}_{\mathsf{exec}}) &= \mathsf{T}\\
O_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{exec}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{exec}}\}),
\end{aligned}$$

where $opc : \mathscr{S}' \to \mathbb{B}$ is defined by

$$opc(S') = \mathsf{T} \text{ iff } S'(\mathsf{ditr}) \in \{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}\}.$$

O_{postp} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$

$$\begin{aligned}
O_{\mathsf{postp}}(S') \restriction M &= S' \restriction M\\
O_{\mathsf{postp}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}}\\
O_{\mathsf{postp}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr})\\
O_{\mathsf{postp}}(S')(\mathsf{pc}) &= pcu(S')\\
O_{\mathsf{postp}}(S')(\mathsf{ir}) &= S'(\mathsf{ir})\\
O_{\mathsf{postp}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\}\\
O_{\mathsf{postp}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\}\\
O_{\mathsf{postp}}(S')(\mathsf{rr}_{\mathsf{postp}}) &= \mathsf{T} && \text{if } S'(\mathsf{eitr}) \neq \mathsf{term}\\
O_{\mathsf{postp}}(S')(\mathsf{rr}_{\mathsf{postp}}) &= \mathsf{F} && \text{if } S'(\mathsf{eitr}) = \mathsf{term}\\
O_{\mathsf{postp}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{postp}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{postp}}\}),
\end{aligned}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

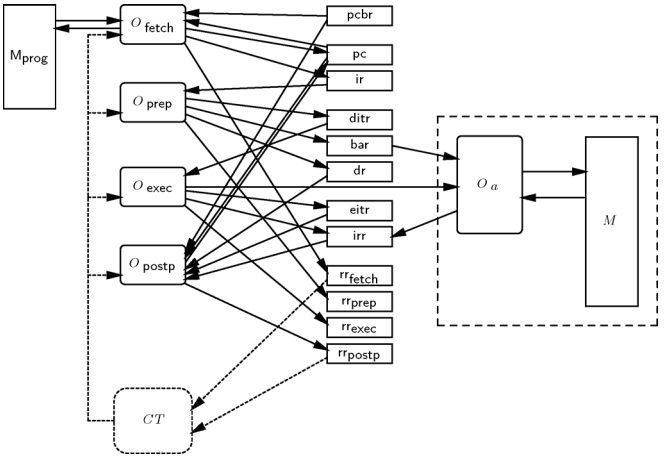

Fig. 1. Structure of an SP-NPL-enhancement

where $pcu : \mathscr{S}' \to \mathsf{MA}'_{\mathsf{prog}}$ is defined by

$$pcu(S') = S'(\mathsf{pc}) \quad \text{if } S'(\mathsf{eitr}) = \mathsf{bsc} \vee S'(\mathsf{eitr}) = \mathsf{ptst} \;\&\; S'(\mathsf{irr}) = \mathsf{T} \vee S'(\mathsf{eitr}) = \mathsf{ntst} \;\&\; S'(\mathsf{irr}) = \mathsf{F} \vee S'(\mathsf{eitr}) = \mathsf{term}$$

$$pcu(S') = S'(\mathsf{pc}) + 1 \quad \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \;\&\; S'(\mathsf{irr}) = \mathsf{F} \vee S'(\mathsf{eitr}) = \mathsf{ntst} \;\&\; S'(\mathsf{irr}) = \mathsf{T}) \;\& \; S'(\mathsf{pc}) + 1 \leqslant S'(\mathsf{pcbr})$$

$$pcu(S') = S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \quad \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \;\&\; S'(\mathsf{dr}) \neq 0 \;\& \; S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \leqslant S'(\mathsf{pcbr})$$

$$pcu(S') = S'(\mathsf{pcbr}) + 1 \quad \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \;\&\; S'(\mathsf{irr}) = \mathsf{F} \vee S'(\mathsf{eitr}) = \mathsf{ntst} \;\&\; S'(\mathsf{irr}) = \mathsf{T}) \;\& \; S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \vee S'(\mathsf{eitr}) = \mathsf{fjmp} \;\& \; (S'(\mathsf{dr}) = 0 \vee S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})).$$

Figure 1 shows the structure of an SP-NPL-enhancement.The program counter pc contains the address of the program memory element from which a PGA instruction is fetched next, unless its content is greater than the highest program address (contained in pcbr). Fetched PGA instructions are stored in ir. The program counter is incremented at every fetch. Pre-processing amounts to decoding the PGA instruction stored in ir: the type of that PGA instruction is stored in ditr, the basic action involved is stored in bar if it is not a jump or termination instruction, and the displacement is stored in dr if it

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

is a jump instruction. Execution does not deal with jump and termination instructions; they are dealt with by post-processing. Post-processing amounts to adjusting the program counter and recognising termination. The program counter is adjusted on a positive test instruction that has given a negative reply, a negative test instruction that has given a positive reply, and a jump instruction.

Essential information about the last fetched PGA instruction is forwarded from one execution handling operation to the next: from O_{fetch} to O_{prep} via ir, from O_{prep} to O_{exec} via ditr and either bar or dr, from O_{exec} to O_{postp} via eitr and irr. Moreover, each execution handling operation has its own reply register. All this fits in well with the pipelined variant of SP-NPL-enhancements we will introduce in Section 8.

Because the memory is extended with only finitely many memory elements, it is easy to check, using Maurer (1966, Proposition IV), that the SP-NPL-enhancement of a Maurer machine is indeed a Maurer machine. The same remark also applies to the SP-PL-enhancement of a Maurer machine introduced in Section 8.

Consider the guarded recursive specification over BTA given by the following equation:

$$CT = (\mathsf{prep} \circ \mathsf{exec} \circ (CT \trianglelefteq \mathsf{postp} \trianglerighteq \mathsf{S})) \trianglelefteq \mathsf{fetch} \trianglerighteq \mathsf{D}.$$

Let P be a finite PGA program. Then applying thread $|P|$ to a state of Maurer machine H has the same effect as applying the execution handling thread CT to the corresponding state of the SP-NPL-enhancement of H in which the program memory contains the stored representation of P. This is stated rigorously in the following theorem.

Theorem 1 (SP-NPL-enhancement). Let $H' = (M', B', \mathscr{S}', \mathscr{O}', A', [\![\_]\!]')$ be the SP-NPL-enhancement of $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$, let $P = u_1 ; \ldots ; u_n \in \mathscr{P}_{\mathsf{fin}}(A)$ be such that $n \leqslant \mathit{size}(\mathsf{M}_{\mathsf{prog}})$, and let $S'_0 \in \mathscr{S}'$ be such that $S'_0 \restriction \mathsf{M}_{\mathsf{prog}}[0, n-1] = \mathsf{s}_{\mathsf{prog}}(P)$, $S'_0(\mathsf{pcbr}) = n-1$ and $S'_0(\mathsf{pc}) = 0$. Then $|P| \bullet_H (S'_0 \restriction M) = (CT \bullet_{H'} S'_0) \restriction M$.

Proof. Let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$ and $(O_a, \mathsf{rr}_a) = [\![a]\!]'$ for all $a \in A'$. Then it is easy to see that, for all $S' \in \mathscr{S}'$ and $a \in A$ such that $S'(\mathsf{pc}) \leqslant S'(\mathsf{pcbr})$ and $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \in \{a, +a, -a\}$, we have

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S')))) \restriction M = O_a(S' \restriction M) \tag{1}$$

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S'))))(\mathsf{irr}) = O_a(S' \restriction M)(m_a), \tag{2}$$

and it is easy to see that, for all $S' \in \mathscr{S}'$ and $a \in A$ such that $S'(\mathsf{pc}) \leqslant S'(\mathsf{pcbr})$ and $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \notin \{a, +a, -a\}$, we have

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S')))) \restriction M = S' \restriction M. \tag{3}$$

Let (p'_i, S'_i) be the $(i+1)^{\text{th}}$ element in the full path of (CT, S'_0) on H'. Then it is easy to prove by induction on i that

$$\begin{aligned} p'_{4i+4} &= CT \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\ S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{T} \\ p'_{4i+4} &= \mathsf{S} \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\ S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{F} \\ p'_{4i+1} &= \mathsf{D} \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{F} \end{aligned} \tag{4}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

(if $4i + 4 < \|(CT, S'_0)\|_{H'}$ in case CT converges from S'_0 on H'). Let (p_i, S_i) be the $(i{+}1)^{\text{th}}$ element in the full path of $(|P|, S'_0 \upharpoonright M)$ on H, and let (p'_i, S'_i) be the $(i{+}1)^{\text{th}}$ element in the full path of (CT, S'_0) on H' of which the first component equals CT, S or D and the second component, say S', satisfies $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \neq \#k$ for all $k \in \mathsf{MA}_{\mathsf{prog}}$. Then, using (1), (2), (3) and (4), it is straightforward to prove by induction on i and case distinction on the structure of finite PGA programs that

$$p_i = |\mathsf{s}_{\mathsf{prog}}(P)(\mathsf{M}_{\mathsf{prog}}[S'_{4i}(\mathsf{pc})]) \,;\ldots;\, \mathsf{s}_{\mathsf{prog}}(P)(\mathsf{M}_{\mathsf{prog}}[n-1])|,$$
$$S_i = S'_{4i} \upharpoonright M$$

(if $i < \|(|P|, S'_0 \upharpoonright M)\|_H$ when $|P|$ converges from $S'_0 \upharpoonright M$ on H). From this, the theorem follows immediately. □

From now on, execution handling threads, like CT, will be called *power threads*.

8. Pipelined instruction processing

In this section, we model a micro-architecture with pipelined instruction processing, which is a variant of the micro-architecture with non-pipelined instruction processing modelled in Section 7. In the latter micro-architecture, PGA instructions are processed after one another, whereas, in the micro-architecture modelled here, four PGA instructions can be simultaneously overlapped in processing. We again start from an arbitrary Maurer machine and enhance it.

We enhance Maurer machines by extending the memory as in the case of SP-NPL-enhancements and additionally with an *instruction skip flag* (isf), a *jump decoded flag* (jdf), a *jump processed flag* (jpf), a *pipeline status register* (plsr) and a *reply register* (rr), and the operation set with a *step* operation (O_{step}), a *pipeline control* operation (O_{plctr}) and a *halt* operation (O_{halt}). Moreover, we replace the basic actions of the original Maurer machine by basic actions step, plctr and halt with which the extra operations O_{step}, O_{plctr} and O_{halt} are associated. The resulting Maurer machines are called SP-PL-enhancements. SP again stands for stored program and PL stands for pipelined instruction processing. In SP-PL-enhancements of Maurer machines, the four *pipeline stages* $\mathsf{fetchst}$, prepst, execst and $\mathsf{postpst}$ are distinguished. Henceforth, we write PS for $\{\mathsf{fetchst}, \mathsf{prepst}, \mathsf{execst}, \mathsf{postpst}\}$. The memory elements isf, jdf, jpf and plsr are used to control the pipelined processing of PGA instructions and to produce a reply in rr at the completion of each step of the pipelined instruction processing. It is assumed that isf, jdf, jpf, plsr and rr are pairwise different memory elements. From now on, we will write M'_{plc} for $\{\mathsf{isf}, \mathsf{jdf}, \mathsf{jpf}, \mathsf{plsr}, \mathsf{rr}\}$. It is assumed that $(\mathsf{M}_{\mathsf{prog}} \cup M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}}) \cap M'_{\mathrm{plc}} = \varnothing$. We will give a more detailed explanation of how an SP-PL-enhancement operates after we have given a precise definition for it.

Let $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ be a Maurer machine such that $M \cap (\mathsf{M}_{\mathsf{prog}} \cup M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}} \cup M'_{\mathrm{plc}}) = \varnothing$ and $\mathsf{step}, \mathsf{plctr}, \mathsf{halt} \notin A$, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

SP-PL-enhancement of H is the Maurer machine $H' = (M', B', \mathscr{S}', \mathscr{O}', A', [\![\_]\!]')$ such that

$$
\begin{aligned}
M' &= M \cup \mathsf{M}_{\mathsf{prog}} \cup M'_{\mathrm{ip}} \cup M'_{\mathrm{rr}} \cup M'_{\mathrm{plc}} \\
B' &= B \cup \mathsf{MA}'_{\mathsf{prog}} \cup \mathfrak{I}_{\mathsf{prog}} \cup IT \cup A \cup \mathbb{B} \cup \mathscr{P}(PS) \\
\mathscr{S}' &= \{S' : M' \to B' \mid \\
&\qquad S' \restriction M \in \mathscr{S} \ \&\ S' \restriction \mathsf{M}_{\mathsf{prog}} \in \mathsf{S}_{\mathsf{prog}} \ \&\ S'(\mathsf{pcbr}) \in \mathsf{MA}_{\mathsf{prog}} \ \& \\
&\qquad S'(\mathsf{pc}) \in \mathsf{MA}'_{\mathsf{prog}} \ \&\ S'(\mathsf{ir}) \in \mathfrak{I}_{\mathsf{prog}} \ \& \\
&\qquad S'(\mathsf{ditr}) \in IT \ \&\ S'(\mathsf{bar}) \in A \ \&\ S'(\mathsf{dr}) \in \mathsf{MA}_{\mathsf{prog}} \ \& \\
&\qquad S'(\mathsf{eitr}) \in IT \ \&\ S'(\mathsf{irr}) \in \mathbb{B} \ \& \\
&\qquad S'(\mathsf{rr}_{\mathsf{fetch}}) \in \mathbb{B} \ \&\ S'(\mathsf{rr}_{\mathsf{prep}}) \in \mathbb{B} \ \&\ S'(\mathsf{rr}_{\mathsf{exec}}) \in \mathbb{B} \ \&\ S'(\mathsf{rr}_{\mathsf{postp}}) \in \mathbb{B} \ \& \\
&\qquad S'(\mathsf{jdf}) \in \mathbb{B} \ \&\ S'(\mathsf{isf}) \in \mathbb{B} \ \&\ S'(\mathsf{jpf}) \in \mathbb{B} \ \&\ S'(\mathsf{plsr}) \in \mathscr{P}(PS) \ \& \\
&\qquad S'(\mathsf{rr}) \in \mathbb{B}\} \\
\mathscr{O}' &= \{O' : \mathscr{S}' \to \mathscr{S}' \mid \\
&\qquad \exists O \in \mathscr{O} \cdot \forall S' \in \mathscr{S}' \cdot \\
&\qquad\quad (O'(S') \restriction M = O(S' \restriction M) \ \&\ O'(S') \restriction (M' \setminus M) = S' \restriction (M' \setminus M))\} \\
&\qquad \cup \{O_{\mathsf{step}}, O_{\mathsf{plctr}}, O_{\mathsf{halt}}\} \\
A' &= \{\mathsf{step}, \mathsf{plctr}, \mathsf{halt}\} \\
[\![a]\!]' &= (O_a, \mathsf{rr}) \quad \text{for all } a \in A',
\end{aligned}
$$

where:

— O_{step} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$:

$$O_{\mathsf{step}}(S') = O'_{\mathsf{fetch}}(O'_{\mathsf{prep}}(O'_{\mathsf{exec}}(O'_{\mathsf{postp}}(S')))),$$

where O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are suboperations defined by:

– O'_{fetch} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$,

$$
\begin{aligned}
O'_{\mathsf{fetch}}(S') &= S' && \text{if } \mathsf{fetchst} \notin S'(\mathsf{plsr}) \\
O'_{\mathsf{fetch}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{fetch}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{fetchst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{fetch}}(S') \restriction M'_{\mathrm{plc}} &= S' \restriction M'_{\mathrm{plc}} && \text{if } \mathsf{fetchst} \in S'(\mathsf{plsr}).
\end{aligned}
$$

– O'_{prep} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$,

$$
\begin{aligned}
O'_{\mathsf{prep}}(S') &= S' && \text{if } \mathsf{prepst} \notin S'(\mathsf{plsr}) \\
O'_{\mathsf{prep}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{prep}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{prep}}(S')(\mathsf{jdf}) &= jdc(S') && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{prep}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jdf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jdf}\}) && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}),
\end{aligned}
$$

where $jdc : \mathscr{S}' \to \mathbb{B}$ is the unique function from $\mathscr{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathscr{S}'$,

$$jdc(S') = \mathsf{T} \text{ iff } O_{\mathsf{prep}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{ditr}) \in \{\mathsf{fjmp}, \mathsf{term}\}.$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

– O'_{exec} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$,

$$\begin{aligned}
O'_{\mathsf{exec}}(S') &= S' && \text{if } \mathsf{execst} \notin S'(\mathsf{plsr}) \\
O'_{\mathsf{exec}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{exec}}(S')(\mathsf{isf}) &= \mathit{isc}(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}\}) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}),
\end{aligned}$$

where $\mathit{isc} : \mathscr{S}' \to \mathbb{B}$ is the unique function from $\mathscr{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathscr{S}'$,

$$\begin{aligned}
\mathit{isc}(S') = \mathsf{T} \text{ iff } & \\
& S'(\mathsf{ditr}) = \mathsf{ptst} \ \&\ O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{F} \vee \\
& S'(\mathsf{ditr}) = \mathsf{ntst} \ \&\ O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{T}.
\end{aligned}$$

– O'_{postp} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$,

$$\begin{aligned}
O'_{\mathsf{postp}}(S') &= S' && \text{if } \mathsf{postpst} \notin S'(\mathsf{plsr}) \\
O'_{\mathsf{postp}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O''_{\mathsf{postp}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{postp}}(S')(\mathsf{jpf}) &= \mathit{jpc}(S') && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}) \\
O'_{\mathsf{postp}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jpf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jpf}\}) && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}),
\end{aligned}$$

where $\mathit{jpc} : \mathscr{S}' \to \mathbb{B}$ is the unique function from $\mathscr{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathscr{S}'$,

$$\mathit{jpc}(S') = \mathsf{T} \text{ iff } S'(\mathsf{eitr}) = \mathsf{fjmp},$$

and O''_{postp} is defined as O_{postp} in the case of the SP-NPL-enhancement, except for the replacement of the auxiliary program counter update function pcu by the function pcu' defined by

$$\begin{aligned}
\mathit{pcu}'(S') &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{eitr}) \neq \mathsf{fjmp} \\
\mathit{pcu}'(S') &= S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \ \&\ S'(\mathsf{dr}) \neq 0 \ \& \\
& && \quad S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) \leqslant S'(\mathsf{pcbr}) \\
\mathit{pcu}'(S') &= S'(\mathsf{pcbr}) + 1 && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \ \& \\
& && \quad (S'(\mathsf{dr}) = 0 \vee \\
& && \quad\ S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})).
\end{aligned}$$

— O_{plctr} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$

$$\begin{aligned}
O_{\mathsf{plctr}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= S' \restriction (M' \setminus M'_{\mathrm{plc}}) \\
O_{\mathsf{plctr}}(S')(\mathsf{jdf}) &= \mathsf{F} \\
O_{\mathsf{plctr}}(S')(\mathsf{isf}) &= \mathsf{F} \\
O_{\mathsf{plctr}}(S')(\mathsf{jpf}) &= \mathsf{F} \\
O_{\mathsf{plctr}}(S')(\mathsf{plsr}) &= \mathit{plsu}(S') \\
O_{\mathsf{plctr}}(S')(\mathsf{rr}) &= \mathit{ru}(S'),
\end{aligned}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

where $plsu : \mathscr{S}' \to \mathscr{P}(PS)$ is the unique function from $\mathscr{S}'$ to $\mathscr{P}(PS)$ such that for all $S' \in \mathscr{S}'$

$$\begin{aligned}
\mathsf{fetchst} \in plsu(S') \quad \text{iff} \quad & S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \ \& \\
& (\mathsf{fetchst} \in S'(\mathsf{plsr}) \ \ \& \ \ S'(\mathsf{jdf}) = \mathsf{F} \vee \\
& S'(\mathsf{isf}) = \mathsf{T} \vee S'(\mathsf{jpf}) = \mathsf{T}) \\
\mathsf{prepst} \in plsu(S') \quad \text{iff} \quad & S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \ \& \\
& (\mathsf{fetchst} \in S'(\mathsf{plsr}) \ \ \& \ \ S'(\mathsf{jdf}) = \mathsf{F} \vee \\
& S'(\mathsf{isf}) = \mathsf{T}) \\
\mathsf{execst} \in plsu(S') \quad \text{iff} \quad & \mathsf{prepst} \in S'(\mathsf{plsr}) \ \ \& \ \ S'(\mathsf{isf}) = \mathsf{F} \\
\mathsf{postpst} \in plsu(S') \quad \text{iff} \quad & \mathsf{execst} \in S'(\mathsf{plsr}),
\end{aligned}$$

and $ru : \mathscr{S}' \to \mathbb{B}$ is the unique function from $\mathscr{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathscr{S}'$,

$$ru(S') = \mathsf{T} \ \text{ iff } \ plsu(S') \neq \varnothing \ \ \& \ \ S'(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{T}.$$

— O_{halt} is the unique function from $\mathscr{S}'$ to $\mathscr{S}'$ such that for all $S' \in \mathscr{S}'$

$$\begin{aligned}
O_{\mathsf{halt}}(S') \restriction (M' \setminus \{\mathsf{rr}\}) &= S' \restriction (M' \setminus \{\mathsf{rr}\}) \\
O_{\mathsf{halt}}(S')(\mathsf{rr}) &= \mathsf{T} \qquad \text{if } S'(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{F} \\
O_{\mathsf{halt}}(S')(\mathsf{rr}) &= \mathsf{F} \qquad \text{if } S'(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{T}.
\end{aligned}$$

Figure 2 shows the structure of an SP-PL-enhancement. The suboperations O'_{fetch}, O'_{prep} and O'_{exec} of O_{step} either do not affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ or do affect these memory elements exactly in the way in which the operations O_{fetch}, O_{prep} and O_{exec} of the SP-NPL-enhancement of H would affect them. The suboperation O'_{postp} of O_{step} either does not affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ or affects them in a way that is similar to the way in which the operation O_{postp} of the SP-NPL-enhancement of H would affect them. The difference compared with O_{postp} is due to the different way in which skipping of a PGA instruction is accomplished in pipelined instruction processing.

The suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} correspond to the pipeline stages that a PGA instruction being processed passes through successively. When the suboperation corresponding to a stage other than the last one has handled a PGA instruction, the suboperation corresponding to the next stage is enabled to handle that PGA instruction in the next step, subject to the exceptions mentioned below. O'_{fetch}, the suboperation corresponding to the first stage, is always enabled to fetch a PGA instruction in the next step, subject to the exceptions mentioned below. The exceptions are:

— when O'_{prep} has decoded a jump or termination instruction, pipelined instruction processing is stalled beginning with the PGA instruction fetched in the same step;
— when O'_{exec} has executed either a positive test instruction with a negative reply as result or a negative test instruction with a positive reply as result, the PGA instruction fetched immediately after the test instruction is discarded and pipelined instruction processing is started again with the next step if the latter instruction is a jump or termination instruction;

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

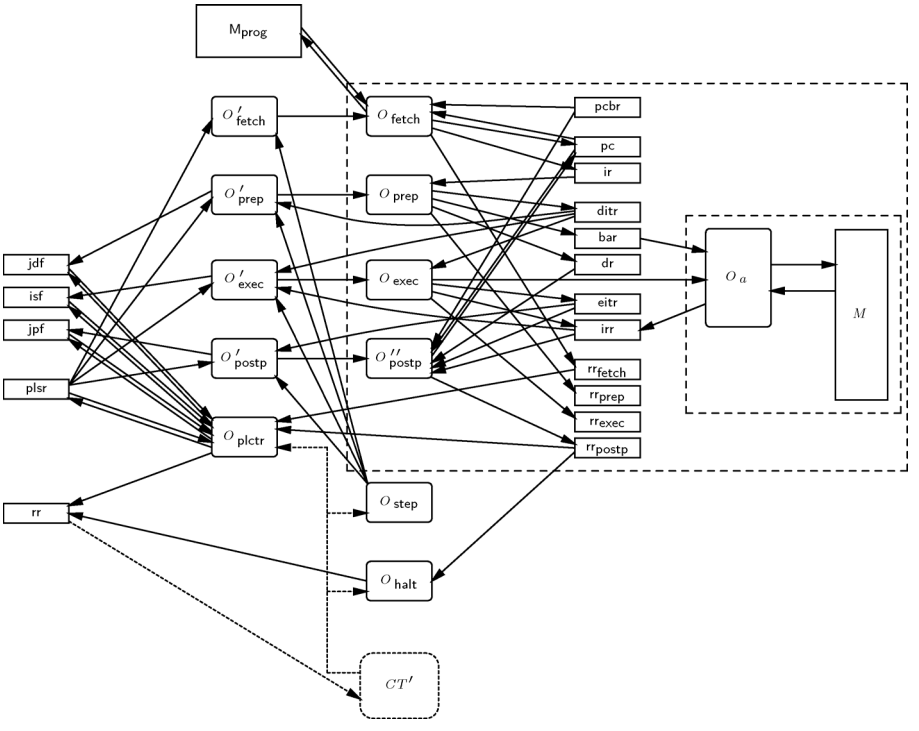

Fig. 2. Structure of an SP-PL-enhancement

— when O'_{postp} has adjusted the program counter on a jump instruction, the last fetched PGA instruction is discarded and pipelined instruction processing is started again with the next step.

Thus, the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are not all enabled to handle a PGA instruction in every step of the pipelined instruction processing. The content of the pipeline status register indicates which of the suboperations are enabled. Enabledness is controlled by the pipeline control operation O_{plctr}. This operation is intended to be performed immediately after O_{step}. It takes parts of the output of the suboperations of O_{step} to fix up the enabledness of these suboperations for the next step.

The idea is that in each step the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are performed in parallel. To justify the use of the term pipeline here, we have to show that the suboperations can actually be performed in parallel. We will return to this issue in Section 9.

Consider the guarded recursive specification over BTA given by the equation

$$CT' = \mathsf{step} \circ (CT' \trianglelefteq \mathsf{plctr} \trianglerighteq (\mathsf{S} \trianglelefteq \mathsf{halt} \trianglerighteq \mathsf{D})).$$

Let P be a finite PGA program. Then, applying thread $|P|$ to a state of the Maurer machine H has the same effect as applying power thread CT' to the corresponding state of the SP-PL-enhancement of H in which the program memory contains the stored representation of P. This is stated rigorously in the following theorem.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Theorem 2 (SP-PL-enhancement). Let $H' = (M', B', \mathscr{S}', \mathscr{O}', A', [\![\_]\!]')$ be the SP-PL-enhancement of $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$, let $P = u_1 ; \ldots ; u_n \in \mathscr{P}_{\mathsf{fin}}(A)$ be such that $n \leqslant size(\mathsf{M_{prog}})$, let $S'_0 \in \mathscr{S}'$ be such that $S'_0 \restriction \mathsf{M_{prog}}[0, n-1] = \mathsf{s_{prog}}(P)$, $S'_0(\mathsf{pcbr}) = n-1$, $S'_0(\mathsf{pc}) = 0$, $S'_0(\mathsf{rr_{fetch}}) = \mathsf{T}$, $S'_0(\mathsf{jdf}) = S'_0(\mathsf{isf}) = S'_0(\mathsf{jpf}) = \mathsf{F}$ and $S'_0(\mathsf{plsr}) = \{\mathsf{fetchst}\}$. Then $|P| \bullet_H (S'_0 \restriction M) = (CT' \bullet_{H'} S'_0) \restriction M$.

Proof. We prove that $(CT \bullet_{H''} (S'_0 \restriction (M' \setminus M'_{\mathsf{plc}}))) \restriction M = (CT' \bullet_{H'} S'_0) \restriction M$, where H'' is the SP-NPL-enhancement of H. From this and Theorem 1, the theorem will follow immediately.

We will use the following notation in the proof. For each $S' \in \mathscr{S}'$ and each $n > 0$, we define $cycle^n(S')$ by induction on n as follows: $cycle^1(S') = O_{\mathsf{plctr}}(O_{\mathsf{step}}(S'))$ and $cycle^{n+1}(S') = O_{\mathsf{plctr}}(O_{\mathsf{step}}(cycle^n(S')))$. For each $S' \in \mathscr{S}'$, we define $tip(S')$ as follows: $tip(S') \Leftrightarrow \mathsf{fetchst} \in S'(\mathsf{plsr})$ & $\mathsf{prepst} \in cycle^1(S')(\mathsf{plsr})$ & $\mathsf{execst} \in cycle^2(S')(\mathsf{plsr})$ & $\mathsf{postpst} \in cycle^3(S')(\mathsf{plsr})$. Thus, $tip(S')$ indicates that some instruction will be totally processed from state S'.

Analysis of input and output regions yields three potential sources of interference between the suboperations of O_{step}: $OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$, $OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$ and $IR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$. It is easy to see, by stalling pipelined instruction processing when O'_{prep} has decoded a jump instruction, that interference does not really happen: O'_{fetch} does not change any memory element if O'_{postp} has changed pc in the same step, and O'_{postp} does not change any memory element if O'_{fetch} has changed pc in the previous step. Because of this, it is not difficult to see that for all $S' \in \mathscr{S}'$,

$$tip(S') \Rightarrow cycle^4(S') \restriction M = O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S' \restriction (M' \setminus M'_{\mathsf{plc}}))))) \restriction M. \tag{5}$$

We have that $tip(S'_0)$ holds. Moreover, tip is preserved by the total processing of an instruction if there is a next instruction to be processed:

— If $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) = a$ and $S'(\mathsf{pc}) + 1 \leqslant S'(\mathsf{pcbr})$, then

$$tip(S') \Rightarrow tip(cycle^1(S')).$$

— If $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) \in \{+a, -a\}$, $cycle^3(S')(\mathsf{isf}) = \mathsf{F}$ and $S'(\mathsf{pc}) + 1 \leqslant S'(\mathsf{pcbr})$, then

$$tip(S') \Rightarrow tip(cycle^1(S')).$$

— If $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) \in \{+a, -a\}$, $cycle^3(S')(\mathsf{isf}) = \mathsf{T}$ and $S'(\mathsf{pc}) + 2 \leqslant S'(\mathsf{pcbr})$, then

$$tip(S') \Rightarrow tip(cycle^2(S')).$$

— If $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) = \#k$ and $S'(\mathsf{pc}) + k \leqslant S'(\mathsf{pcbr})$, then

$$tip(S') \Rightarrow tip(cycle^4(S')).$$

Let (p_i, S_i) be the $(i+1)^{\text{th}}$ element in the full path of $(CT, S'_0 \restriction (M' \setminus M'_{\mathsf{plc}}))$ on H''. Then it is easy to prove by induction on i that

$$\begin{array}{ll} p_{4i+4} = CT & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \ \ \& \ \ S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{T} \\ p_{4i+4} = \mathsf{S} & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \ \ \& \ \ S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{F} \\ p_{4i+1} = \mathsf{D} & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{F} \end{array} \tag{6}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 11. *Pipelined instruction processing of* $a\,;+b\,;\#3\,;c\,;\#2\,;d\,;!$

	1	2	3	4	5	6	7	8	9	10	11	12
a	fetch	prep	exec	postp								
$+b$		fetch	prep	exec	postp							
$\#3$			fetch	prep								
c				fetch	prep	exec	postp					
$\#2$					fetch	prep	exec	postp				
d						fetch						
!									fetch	prep	exec	postp
										fetch		

(if $4i+4 < \|(CT, S'_0 \upharpoonright (M' \setminus M'_{\mathrm{plc}}))\|_{H''}$ when CT converges from $S'_0 \upharpoonright (M' \setminus M'_{\mathrm{plc}})$ on H''). Let (p'_i, S'_i) be the $(i{+}1)^{\mathrm{th}}$ element in the full path of (CT', S'_0) on H'. Then it is easy to prove by induction on i that

$$\begin{aligned} p'_{4i+4} &= CT' \quad \text{if } tip(S_{4i}) \ \&\ S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\ S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{T} \\ p'_{4i+4} &= \mathsf{S} \quad \text{if } tip(S_{4i}) \ \&\ S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\ S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{F} \\ p'_{4i+1} &= \mathsf{D} \quad \text{if } tip(S_{4i}) \ \&\ S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{F} \end{aligned} \tag{7}$$

(if $4i+4 < \|(CT', S'_0)\|_{H'}$ when CT' converges from S'_0 on H'). Let (p_i, S_i) be the $(i{+}1)^{\mathrm{th}}$ element in the full path of $(CT, S'_0 \upharpoonright (M' \setminus M'_{\mathrm{plc}}))$ on H'' of which the first component equals CT, S or D, and let (p'_i, S'_i) be the $(i{+}1)^{\mathrm{th}}$ element in the full path of (CT', S'_0) on H' of which the first component equals CT', S or D and the second component, say S', satisfies $tip(S')$ if the first component equals CT'. Then, using (5), (6), (7) and the preservation properties of tip, it is straightforward to prove by induction on i and case distinction on the kinds of primitive instructions of PGA that

$$(p_i = CT \Leftrightarrow p'_i = CT') \ \&\ (p_i = \mathsf{S} \Leftrightarrow p'_i = \mathsf{S}) \ \&\ (p_i = \mathsf{D} \Leftrightarrow p'_i = \mathsf{D}),$$
$$S_i \upharpoonright (M' \setminus M'_{\mathrm{plc}}) = S'_i \upharpoonright (M' \setminus M'_{\mathrm{plc}})$$

(if $i < \|(CT, S'_0 \upharpoonright (M' \setminus M'_{\mathrm{plc}}))\|_{H''}$ when CT converges from $S'_0 \upharpoonright (M' \setminus M'_{\mathrm{plc}})$ on H''). The theorem then follows immediately. □

Example (Pipelined instruction processing). Table 11 shows the pipelined instruction processing of the PGA program $a\,;+b\,;\#3\,;c\,;\#2\,;d\,;!$. It is assumed that the execution of $+b$ results in a negative reply. We see that the pipelined instruction processing of this PGA program is stalled three times: after the jump instruction #3 has been decoded in step 4, after the jump instruction #2 has been decoded in step 6, and after the termination instruction ! has been decoded in step 10. Because the execution of the positive test instruction $+b$ has produced a negative reply in step 4, the next instruction in the pipeline, that is, the jump instruction #3, is not executed and post-processed in later steps. Pipelined instruction processing is started again from step 5, because there is no longer a jump instruction in the pipeline. The jump instruction #2 passes all

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Table 12. *Pipelined instruction processing of* $a\,;+b\,;c\,;\#3\,;d\,;e$

	1	2	3	4	5	6	7	8
a	fetch	prep	exec	postp				
$+b$		fetch	prep	exec	postp			
c			fetch	prep				
$\#3$				fetch	prep	exec	postp	
d					fetch			
e								
								fetch

four pipeline stages before pipelined instruction processing is started again from step 9. Moreover, because the jump is actually taken, the prematurely fetched instruction d is discarded when pipelined instruction processing is started again. The attempt to fetch another instruction prematurely in step 10 does not succeed because the last instruction of the PGA program was fetched in step 9. Instruction processing stops after step 12, because in that step the termination instruction is recognised.

Table 12 shows the pipelined instruction processing of the program $a\,;+b\,;c\,;\#3\,;d\,;e$. It is assumed that the execution of $+b$ results in a negative reply. We see that the pipelined instruction processing of this PGA program is stalled once: after the jump instruction $\#3$ has been decoded in step 5. Because the execution of the positive test instruction $+b$ has produced a negative reply in step 4, the next instruction in the pipeline, that is, the void basic instruction c, is not executed and post-processed in later steps. The jump instruction $\#3$ passes all four pipeline stages before pipelined instruction processing is started again from step 8. Moreover, because the jump is actually taken, the prematurely fetched instruction d is discarded when pipelined instruction processing is started again. The attempt to fetch another instruction in step 8 does not succeed because the jump instruction $\#3$ has brought the program counter beyond the last instruction of the PGA program. Instruction processing stops after step 8, because fetching fails in that step while there is no other instruction in the pipeline. This situation corresponds to a programming error, such as a jump out of the program, as a result of which further instruction processing is blocked.

With pipelined instruction processing, execution of the first example program takes 12 steps and execution of the second example program takes 8 steps. With non-pipelined instruction processing, these would take 20 steps and 13 steps, respectively. However, there will be no real gain unless O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} can be performed in parallel.

9. Parallel composability

In this section, we justify the use of the term pipeline in Section 8 by showing that the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} can actually be performed in parallel.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

In the case under consideration, performing a number of operations in parallel amounts to accomplishing the state transformations going with the different operations simultaneously. It should be borne in mind that accomplishing them simultaneously and accomplishing them in arbitrary order do not always yield the same result.

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, $O \in \mathscr{O}$ and $O_1, O_2 : \mathscr{S} \to \mathscr{S}$ be such that $O_2(O_1(S)) = O(S)$ for all $S \in \mathscr{S}$. Then O is parallel composable of O_1 and O_2 if the following conditions are fulfilled:

— O_1 is *consistent* with O_2: if O_1 and O_2 affect the same memory element, then they affect that memory element in the same way;

— O_1 is *transparent* to O_2: if O_1 affects a memory element, then that memory element does not affect any memory element under O_2.

More precisely, O is *parallel composable* of O_1 and O_2 if and only if $O_1 \mathsf{con} O_2 \ \&\ O_1 \mathsf{tra} O_2$, where con and tra are defined as follows:

— $O_1 \mathsf{con} O_2$ iff

$$\forall m \in OR(O_1) \cap OR(O_2), S \in \mathscr{S} \cdot (O_1(S)(m) \neq S(m) \ \&\ O_2(S)(m) \neq S(m) \Rightarrow O_1(S)(m) = O_2(S)(m))$$

— $O_1 \mathsf{tra} O_2$ iff

$$\forall m \in OR(O_1) \cap IR(O_2), S \in \mathscr{S} \cdot (O_1(S)(m) \neq S(m) \Rightarrow \neg(\exists S' \in \mathscr{S} \cdot (\forall m' \in M \setminus \{m\} \cdot O_1(S)(m') = S'(m') \ \&\ \exists m'' \in OR(O_2) \cdot O_2(O_1(S))(m'') \neq O_2(S')(m''))))).$$

Sufficient conditions for $O_1 \mathsf{con} O_2$ and $O_1 \mathsf{tra} O_2$ to hold are $OR(O_1) \cap OR(O_2) = \varnothing$ and $OR(O_1) \cap IR(O_2) = \varnothing$, respectively.

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, $O \in \mathscr{O}$, and $O_1, O_2 : \mathscr{S} \to \mathscr{S}$ be such that $O_2(O_1(S)) = O(S)$ for all $S \in \mathscr{S}$. Then O_1 and O_2 are commutative if $O_2(O_1(S)) = O_1(O_2(S))$ for all $S \in \mathscr{S}$. Note that O_1 and O_2 do not have to be commutative in order for O to be parallel composable of O_1 and O_2; and O does not have to be parallel composable of O_1 and O_2 in order for O_1 and O_2 to be commutative. In other words, parallel composability does not imply commutativity, or the other way round.

Parallel composability generalises easily to n operations (for $n \geqslant 2$).

Let $(M, B, \mathscr{S}, \mathscr{O})$ be a Maurer computer, let $O \in \mathscr{O}$, and let $O_1, \ldots, O_n : \mathscr{S} \to \mathscr{S}$ be such that $O_n(\ldots O_1(S)\ldots) = O(S)$ for all $S \in \mathscr{S}$. Then O is parallel composable of $O_1, \ldots, O_n$ if and only if $\bigwedge_{1 \leqslant i < n} \bigwedge_{i < j \leqslant n} (O_i \mathsf{con} O_j \ \&\ O_i \mathsf{tra} O_j)$.

The suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} from Section 8 can be performed in parallel. This is stated rigorously in the following theorem.

Theorem 3 (Parallel composability). Take the SP-PL-enhancement of a Maurer machine H as in Section 8. Then O_{step} is parallel composable of O'_{postp}, O'_{exec}, O'_{prep} and O'_{fetch}.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Proof. The following follow immediately from the definitions:

$$\begin{aligned}
OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{exec}}) &= \varnothing & OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{exec}}) &= \varnothing\\
OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{prep}}) &= \varnothing & OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{prep}}) &= \varnothing\\
OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) &= \{\mathsf{pc}\} & OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{fetch}}) &= \{\mathsf{pc}\}\\
OR(O'_{\mathsf{exec}}) \cap OR(O'_{\mathsf{prep}}) &= \varnothing & OR(O'_{\mathsf{exec}}) \cap IR(O'_{\mathsf{prep}}) &= \varnothing\\
OR(O'_{\mathsf{exec}}) \cap OR(O'_{\mathsf{fetch}}) &= \varnothing & OR(O'_{\mathsf{exec}}) \cap IR(O'_{\mathsf{fetch}}) &= \varnothing\\
OR(O'_{\mathsf{prep}}) \cap OR(O'_{\mathsf{fetch}}) &= \varnothing & OR(O'_{\mathsf{prep}}) \cap IR(O'_{\mathsf{fetch}}) &= \varnothing.
\end{aligned}$$

Hence we only need to take a closer look at the conditions O'_{postp} con O'_{fetch} and O'_{postp} tra O'_{fetch}; and we only need to consider the memory element pc. Now, take an arbitrary state S'. It is easy to see that if O'_{postp} changes pc in state S', then O'_{exec} must not have set isf one step back and O'_{prep} must have set jdf two steps back. It is also easy to see that, as a consequence, O'_{fetch} does not change any memory element in states S' and $O'_{\mathsf{prep}}(S')$. Hence, both the consistency condition and the transparency condition are trivially met. □

The proof of Theorem 3 shows that stalling pipelined instruction processing when O'_{prep} has decoded a jump instruction is crucial for parallel composability. It is easy to see that O_{step} is not parallel composable of O'_{postp}, O'_{exec}, O'_{prep}, O'_{fetch} and O_{plctr}. This is to be expected. For example, the flags jdf, isf and jpf are set by O'_{prep}, O'_{exec} and O'_{postp} to influence how plsr is updated by O_{plctr}.

10. Conditional jump instructions

In this section, we extend PGA with conditional jump instructions and look at the effect of this on non-pipelined and pipelined instruction processing.

We add to PGA the following primitive instructions:

— for each $a \in \mathfrak{A}$ and $k \in \mathbb{N}$, a *positive conditional jump instruction* $+a\#k$
— for each $a \in \mathfrak{A}$ and $k \in \mathbb{N}$, a *negative conditional jump instruction* $-a\#k$.

A positive conditional jump instruction $+a\#k$ has the same effect as $+a\,;\#k$, but counts as one instruction; and a negative conditional jump instruction $-a\#k$ has the same effect as $-a\,;\#k$, but counts as one instruction. In Bergstra and Loots (2002), PGA is extended with a *unit instruction* operator $\mathbf{u}$ that turns PGA programs into single instructions. In that extension of PGA, called $\mathrm{PGA}_{\mathrm{u}}$, $+a\#k$ and $-a\#k$ can be taken as abbreviations for $\mathbf{u}(+a\,;\#k)$ and $\mathbf{u}(-a\,;\#k)$, respectively. In Ponse (2002), thread extraction for $\mathrm{PGA}_{\mathrm{u}}$ programs is described by means of a mapping from $\mathrm{PGA}_{\mathrm{u}}$ programs to PGA programs.

The SP-NPL-enhancement of a Maurer machine changes only slightly when conditional jump instructions are added. Only the set IT and the auxiliary functions *dec*, *opc* and *pcu* used in the definition of the SP-NPL-enhancement of a Maurer machine from Section 7 have to be redefined. The set IT is redefined because the two kinds of conditional jump instructions give rise to two additional instruction types: pcfjmp and ncfjmp. The function

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

dec is redefined in order to deal with the decoding of conditional jump instructions. The function *opc* is redefined because conditional jump instructions cause an operation to be performed. The function *pcu* is redefined in order to deal with the adjustment of the program counter in the case of conditional jump instructions.

IT is redefined to be the set $\{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{fjmp}, \mathsf{pcfjmp}, \mathsf{ncfjmp}, \mathsf{term}\}$.

The function $dec : \mathscr{S}' \to IT \times A \times \mathsf{MA}_{\mathsf{prog}}$ is redefined as follows:

$$\begin{aligned}
dec(S') &= (\mathsf{bsc}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = a\\
dec(S') &= (\mathsf{ptst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = +a\\
dec(S') &= (\mathsf{ntst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = -a\\
dec(S') &= (\mathsf{fjmp}, S'(\mathsf{bar}), k) && \text{if } S'(\mathsf{ir}) = \#k\\
dec(S') &= (\mathsf{pcfjmp}, a, k) && \text{if } S'(\mathsf{ir}) = +a\#k\\
dec(S') &= (\mathsf{ncfjmp}, a, k) && \text{if } S'(\mathsf{ir}) = -a\#k\\
dec(S') &= (\mathsf{term}, S'(\mathsf{bar}), S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = !.
\end{aligned}$$

The function $opc : \mathscr{S}' \to \mathbb{B}$ is redefined as follows:

$$opc(S') = \mathsf{T} \text{ iff } S'(\mathsf{ditr}) \in \{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{pcfjmp}, \mathsf{ncfjmp}\}.$$

The function $pcu : \mathscr{S}' \to \mathsf{MA}'_{\mathsf{prog}}$ is redefined as follows:

$$\begin{aligned}
pcu(S') &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{eitr}) = \mathsf{bsc} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ptst} \ \&\ S'(\mathsf{irr}) = \mathsf{T} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \ \&\ S'(\mathsf{irr}) = \mathsf{F} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{F} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{T} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{term}\\
pcu(S') &= S'(\mathsf{pc}) + 1 && \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \ \&\ S'(\mathsf{irr}) = \mathsf{F} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \ \&\ S'(\mathsf{irr}) = \mathsf{T}) \ \&\\
&&& \quad S'(\mathsf{pc}) + 1 \leqslant S'(\mathsf{pcbr})\\
pcu(S') &= S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) && \text{if } (S'(\mathsf{eitr}) = \mathsf{fjmp} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{T} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{F}) \ \&\\
&&& \quad S'(\mathsf{dr}) \neq 0 \ \&\\
&&& \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \leqslant S'(\mathsf{pcbr})\\
pcu(S') &= S'(\mathsf{pcbr}) + 1 && \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \ \&\ S'(\mathsf{irr}) = \mathsf{F} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \ \&\ S'(\mathsf{irr}) = \mathsf{T}) \ \&\\
&&& \quad S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \vee\\
&&& \quad (S'(\mathsf{eitr}) = \mathsf{fjmp} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{T} \vee\\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \ \&\ S'(\mathsf{irr}) = \mathsf{F}) \ \&\\
&&& \quad (S'(\mathsf{dr}) = 0 \vee\\
&&& \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})).
\end{aligned}$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

Like the SP-NPL-enhancement of a Maurer machine, the SP-PL-enhancement of a Maurer machine changes only slightly when conditional jump instructions are added. The memory has to be extended with a *conditional jump flag* (cjf), which, like the other flags, contains a Boolean value. Also, the set M'_{plc}, the auxiliary functions *jpc* and *pcu'*, the suboperation O'_{exec} and the operation O_{plctr} used in the definition of the SP-PL-enhancement of a Maurer machine from Section 8 have to be redefined. The flag cjf is needed in order to control the pipelined processing of instructions in the presence of conditional jump instructions. The set M'_{plc} is redefined because of the addition of the flag cjf. The function *jpc* is redefined because, after adjustment of the program counter on conditional jump instructions, pipelined instruction processing must be restarted as in the case of unconditional jump instructions. Just like *pcu* before, the function *pcu'* is redefined in order to deal with the adjustment of the program counter in the case of conditional jump instructions. The suboperation O'_{exec} is redefined in order to set the additional flag cjf when, in the case of conditional jump instructions, the reply value is produced on which the jump concerned must actually take place. The operation O_{plctr} is redefined in order to control the pipelined processing of instructions in the presence of conditional jump instructions.

M'_{plc} is redefined to be the set $\{\text{isf}, \text{jdf}, \text{jpf}, \text{cjf}, \text{plsr}, \text{rr}\}$.

The function $jpc : \mathscr{S}' \to \mathbb{B}$ is redefined as follows:

$$
\begin{aligned}
&jpc(S') = \text{T iff}\\
&\quad S'(\text{eitr}) = \text{fjmp} \vee\\
&\quad S'(\text{eitr}) = \text{pcfjmp} \ \&\ S'(\text{irr}) = \text{T} \vee S'(\text{eitr}) = \text{ncfjmp} \ \&\ S'(\text{irr}) = \text{F}.
\end{aligned}
$$

The function $pcu' : \mathscr{S}' \to \text{MA}'_{\text{prog}}$ is redefined as follows:

$$
\begin{aligned}
&pcu'(S') = S'(\text{pc}) && \text{if } S'(\text{eitr}) \in \{\text{bsc}, \text{ptst}, \text{ntst}, \text{term}\} \vee\\
&&& \quad S'(\text{eitr}) = \text{pcfjmp} \ \&\ S'(\text{irr}) = \text{F} \vee\\
&&& \quad S'(\text{eitr}) = \text{ncfjmp} \ \&\ S'(\text{irr}) = \text{T}\\
&pcu'(S') = S'(\text{pc}) - 2 + S'(\text{dr}) && \text{if } S'(\text{eitr}) = \text{fjmp} \ \&\ S'(\text{dr}) \neq 0 \ \&\\
&&& \quad S'(\text{pc}) - 2 + S'(\text{dr}) \leqslant S'(\text{pcbr})\\
&pcu'(S') = S'(\text{pc}) - 3 + S'(\text{dr}) && \text{if } (S'(\text{eitr}) = \text{pcfjmp} \ \&\ S'(\text{irr}) = \text{T} \vee\\
&&& \quad S'(\text{eitr}) = \text{ncfjmp} \ \&\ S'(\text{irr}) = \text{F}) \ \&\\
&&& \quad S'(\text{dr}) \neq 0 \ \&\\
&&& \quad S'(\text{pc}) - 3 + S'(\text{dr}) \leqslant S'(\text{pcbr})\\
&pcu'(S') = S'(\text{pcbr}) + 1 && \text{if } S'(\text{eitr}) = \text{fjmp} \ \&\\
&&& \quad (S'(\text{dr}) = 0 \vee\\
&&& \quad S'(\text{pc}) - 2 + S'(\text{dr}) > S'(\text{pcbr})) \vee\\
&&& \quad (S'(\text{eitr}) = \text{pcfjmp} \ \&\ S'(\text{irr}) = \text{T} \vee\\
&&& \quad S'(\text{eitr}) = \text{ncfjmp} \ \&\ S'(\text{irr}) = \text{F}) \ \&\\
&&& \quad (S'(\text{dr}) = 0 \vee\\
&&& \quad S'(\text{pc}) - 3 + S'(\text{dr}) > S'(\text{pcbr})).
\end{aligned}
$$

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

The suboperation O'_{exec} is redefined as follows:

$$\begin{aligned}
O'_{\mathsf{exec}}(S') &= S' && \text{if } \mathsf{execst} \notin S'(\mathsf{plsr})\\
O'_{\mathsf{exec}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\\
O'_{\mathsf{exec}}(S')(\mathsf{isf}) &= \mathit{isc}(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\\
O'_{\mathsf{exec}}(S')(\mathsf{cjf}) &= \mathit{cjc}(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\\
O'_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}, \mathsf{cjf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}, \mathsf{cjf}\}) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}),
\end{aligned}$$

where $\mathit{isc} : \mathscr{S}' \to \mathbb{B}$ is defined as in the case without conditional jump instructions and $\mathit{cjc} : \mathscr{S}' \to \mathbb{B}$ is the unique function from $\mathscr{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathscr{S}'$,

$$\begin{aligned}
\mathit{cjc}(S') = \mathsf{T} \text{ iff}\\
S'(\mathsf{ditr}) = \mathsf{pcfjmp} \ \&\ O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{T} \vee\\
S'(\mathsf{ditr}) = \mathsf{ncfjmp} \ \&\ O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{F}.
\end{aligned}$$

O_{plctr} is redefined as follows:

$$\begin{aligned}
O_{\mathsf{plctr}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= S' \restriction (M' \setminus M'_{\mathrm{plc}})\\
O_{\mathsf{plctr}}(S')(\mathsf{jdf}) &= \mathsf{F}\\
O_{\mathsf{plctr}}(S')(\mathsf{isf}) &= \mathsf{F}\\
O_{\mathsf{plctr}}(S')(\mathsf{jpf}) &= \mathsf{F}\\
O_{\mathsf{plctr}}(S')(\mathsf{cjf}) &= \mathsf{F}\\
O_{\mathsf{plctr}}(S')(\mathsf{plsr}) &= \mathit{plsu}(S')\\
O_{\mathsf{plctr}}(S')(\mathsf{rr}) &= \mathit{ru}(S'),
\end{aligned}$$

where $\mathit{plsu} : \mathscr{S}' \to \mathscr{P}(PS)$ is the unique function from $\mathscr{S}'$ to $\mathscr{P}(PS)$ such that for all $S' \in \mathscr{S}'$,

$$\begin{aligned}
\mathsf{fetchst} \in \mathit{plsu}(S') \ \text{iff}\ & S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\\
& (\mathsf{fetchst} \in S'(\mathsf{plsr}) \ \&\ S'(\mathsf{jdf}) = \mathsf{F} \ \&\ S'(\mathsf{cjf}) = \mathsf{F} \vee\\
& S'(\mathsf{isf}) = \mathsf{T} \vee S'(\mathsf{jpf}) = \mathsf{T})\\
\mathsf{prepst} \in \mathit{plsu}(S') \ \text{iff}\ & S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \ \&\\
& (\mathsf{fetchst} \in S'(\mathsf{plsr}) \ \&\ S'(\mathsf{jdf}) = \mathsf{F} \ \&\ S'(\mathsf{cjf}) = \mathsf{F} \vee\\
& S'(\mathsf{isf}) = \mathsf{T})\\
\mathsf{execst} \in \mathit{plsu}(S') \ \text{iff}\ & \mathsf{prepst} \in S'(\mathsf{plsr}) \ \&\ S'(\mathsf{isf}) = \mathsf{F} \ \&\ S'(\mathsf{cjf}) = \mathsf{F}\\
\mathsf{postpst} \in \mathit{plsu}(S') \ \text{iff}\ & \mathsf{execst} \in S'(\mathsf{plsr}).
\end{aligned}$$

11. Backward jump instructions

In this short section we discuss backward jump instructions and sketch the effect of their inclusion on non-pipelined and pipelined instruction processing.

In the preceding sections we have only considered finite PGA programs, that is, closed terms of PGA in which the repetition operator does not occur. This means that programs that are infinite sequences of primitive instructions are excluded. In other words, programs for which the execution goes on indefinitely are not covered. However, in a setting with backward jump instructions, there exists for each such program a behaviourally equivalent program that is a finite sequence of primitive instructions.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

In a setting with backward jump instructions, there are, in addition to the primitive instructions of PGA introduced earlier, the following primitive instructions:

— for each $k \in \mathbb{N}$, a *backward jump instruction* $\backslash\#k$.

We write $\mathfrak{I}'$ for the set consisting of all primitive instructions of PGA and all backward jump instructions. A PGLB *program* is a closed term that can be built from:

— for each $u \in \mathfrak{I}'$, an instruction constant u
— the concatenation operator $\_ \,;\, \_$.

The meaning of PGLB programs is described in Bergstra and Loots (2002) by means of a mapping from PGLB programs to PGA programs. For each PGA program, there exists a PGLB program that is mapped to a PGA program with the same behaviour. In other words, the expressiveness is not decreased by replacing the repetition operator by backward jump instructions.

The addition of backward jump instructions gives rise to trivial changes of the SP-NPL- and SP-PL-enhancements of Maurer machines: forward jump instructions and backward jump instructions can be treated in the same way.

Only the set IT and the auxiliary functions *dec* and *pcu* used in the definition of the SP-NPL-enhancement of a Maurer machine from Section 7 and the auxiliary function *pcu′* used in the definition of the SP-PL-enhancement of a Maurer machine from Section 8 have to be redefined. The set IT must be redefined because the backward jump instructions give rise to an additional instruction type: bjmp. The function *dec* must be redefined in order to deal with the decoding of backward jump instructions. The function *pcu* and *pcu′* must be redefined in order to deal with the adjustment of the program counter in the case of backward jump instructions.

It is easy to see that with the correct redefinitions, Theorems 1 and 2 go through after the addition of backward jump instructions. Conditional backward jump instructions can be added in the same way as conditional forward jump instructions were added in Section 10.

12. Instruction set architectures

In this section, we introduce the concept of a strict load/store Maurer instruction set architecture. This concept gets its name because:

— it is described in the setting of Maurer's model for computers;
— it is concerned only with load/store architectures;
— the load/store architectures it deals with are strict in some respects, which will be explained after it has been formalised.

The concept of a strict load/store Maurer instruction set architecture, or, for short, a strict load/store Maurer ISA, is an approximation of the concept of a load/store instruction set architecture. It is focussed on instructions for data manipulation and data transfer. Instructions for transfer of program control are treated in a uniform way over different strict load/store Maurer ISAs. Instances of the concept of a strict load/store Maurer ISA are those Maurer machines for which SP-NPL- and SP-PL-enhancements are

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

primarily intended. The SP-NPL- and SP-PL-enhancements of a strict load/store Maurer ISA can be viewed as implementations of that ISA.

Each Maurer machine has a number of basic actions with which an operation is associated. In this section, when we refer to Maurer machines that are strict load/store Maurer ISAs, such basic actions are loosely called basic instructions. The term basic action is not commonly used when we are talking about ISAs, and, moreover, basic instructions and basic actions are identified in the semantics of PGA.

The basic idea underlying the concept of a strict load/store Maurer ISA is that there is a main memory whose elements contain data, an operating unit with a small internal memory through which data can be manipulated, and an interface between the main memory and the operating unit for data transfer between them. For the sake of simplicity, data is restricted to the natural numbers between 0 and some upper bound. Other types of data that could be supported can always be represented by the natural numbers provided. Moreover, the data manipulation instructions offered by a strict load/store Maurer ISA are not restricted and may include ones that are tailored to manipulation of representations of other types of data. Therefore, we believe that nothing essential is lost by the restriction to natural numbers.

The concept of a strict load/store Maurer ISA is parametrised by:

— an address width k
— a word length l
— a bit size m of the operating unit
— a number u of pairs of address and data registers for load instructions
— a number v of pairs of address and data registers for store instructions
— a set A' of basic instructions for data manipulation.

It is assumed that a fixed but arbitrary set $\mathsf{M}_{\mathsf{data}}$ of cardinality 2^k and a fixed but arbitrary bijection $\mathsf{m}_{\mathsf{data}} : [0, 2^k - 1] \to \mathsf{M}_{\mathsf{data}}$ have been given. $\mathsf{M}_{\mathsf{data}}$ is called the *data memory*. The data memory is a memory whose elements can be addressed by means of natural numbers in the interval $[0, 2^k - 1]$. The address width k can be regarded as the number of bits used for the binary representation of addresses of data memory elements. We write $\mathsf{B}_{\mathsf{addr}}$ for $[0, 2^k - 1]$. The data memory elements are meant for containing data. They can contain natural numbers in the interval $[0, 2^l - 1]$. The word length l can be regarded as the number of bits used to represent data in data memory elements. We write $\mathsf{B}_{\mathsf{data}}$ for $[0, 2^l - 1]$.

It is assumed that a fixed but arbitrary set M_{ou} of cardinality m, called the *operating unit memory*, has been given. The operating unit memory is a memory whose elements can contain natural numbers in the set $\{0, 1\}$, that is, bits. We write Bit for $\{0, 1\}$. The bit size m can be regarded as the number of bits that the internal memory of the operating unit contains. Usually, a part of the operating unit memory is partitioned into groups to which data manipulation instructions can refer.

It is assumed that fixed but arbitrary sets M_{ld} and M_{la} of cardinality u and fixed but arbitrary bijections $\mathsf{m}_{\mathsf{ld}} : [0, u - 1] \to \mathsf{M}_{\mathsf{ld}}$ and $\mathsf{m}_{\mathsf{la}} : [0, u - 1] \to \mathsf{M}_{\mathsf{la}}$ have been given. It is also assumed that fixed but arbitrary sets M_{sd} and M_{sa} of cardinality v and fixed but arbitrary bijections $\mathsf{m}_{\mathsf{sd}} : [0, v - 1] \to \mathsf{M}_{\mathsf{sd}}$ and $\mathsf{m}_{\mathsf{sa}} : [0, v - 1] \to \mathsf{M}_{\mathsf{sa}}$ have been given.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

The members of $\mathsf{M_{la}}$ and $\mathsf{M_{ld}}$ are called *load address registers* and *load data registers*, respectively. The members of $\mathsf{M_{sa}}$ and $\mathsf{M_{sd}}$ are called *store address registers* and *store data registers*, respectively. The load and store registers are special memory elements meant for transferring data between the data memory and the operating unit memory. The members of $\mathsf{M_{la}}$ and $\mathsf{M_{sa}}$ can contain addresses, that is, members of $\mathsf{B_{addr}}$. The members of $\mathsf{M_{ld}}$ and $\mathsf{M_{sd}}$ can contain data, that is, members of $\mathsf{B_{data}}$. It is assumed that $\mathsf{M_{data}}$, $\mathsf{M_{ou}}$, $\mathsf{M_{ld}}$, $\mathsf{M_{sd}}$, $\mathsf{M_{la}}$, $\mathsf{M_{sa}}$ and $\{\mathsf{rr}_a \mid a \in A\}$ are pairwise disjoint sets.

Let $n \in [0, 2^k - 1]$, $n' \in [0, u-1]$ and $n'' \in [0, v-1]$. Then, we write $\mathsf{M_{data}}[n]$ for $\mathsf{m_{data}}(n)$, $\mathsf{M_{ld}}[n']$ for $\mathsf{m_{ld}}(n')$, $\mathsf{M_{la}}[n']$ for $\mathsf{m_{la}}(n')$, $\mathsf{M_{sd}}[n'']$ for $\mathsf{m_{sd}}(n'')$ and $\mathsf{M_{sa}}[n'']$ for $\mathsf{m_{sa}}(n'')$.

A *strict load/store Maurer instruction set architecture* with parameters k, l, m, u, v and A' is a Maurer machine $H = (M, B, \mathscr{S}, \mathscr{O}, A, [\![\_]\!])$ with

$$
\begin{aligned}
M &= \mathsf{M_{data}} \cup \mathsf{M_{ou}} \cup \mathsf{M_{ld}} \cup \mathsf{M_{sd}} \cup \mathsf{M_{la}} \cup \mathsf{M_{sa}} \cup \{\mathsf{rr}_a \mid a \in A\} \\
B &= \mathsf{B_{data}} \cup \mathsf{B_{addr}} \cup \mathbb{B} \\
\mathscr{S} &= \{S : M \to B \mid \\
&\qquad \forall m \in \mathsf{M_{data}} \cup \mathsf{M_{ld}} \cup \mathsf{M_{sd}} \cdot S(m) \in \mathsf{B_{data}} \ \ \& \\
&\qquad \forall m \in \mathsf{M_{la}} \cup \mathsf{M_{sa}} \cdot S(m) \in \mathsf{B_{addr}} \ \ \& \\
&\qquad \forall m \in \mathsf{M_{ou}} \cdot S(m) \in \mathsf{Bit} \ \ \& \ \ \forall a \in A \cdot S(\mathsf{rr}_a) \in \mathbb{B}\} \\
\mathscr{O} &= \{O_a \mid a \in A\} \\
A &= \{\mathsf{load}{:}n \mid n \in [0, u-1]\} \cup \{\mathsf{store}{:}n \mid n \in [0, v-1]\} \cup A' \\
[\![a]\!] &= (O_a, \mathsf{rr}_a) \quad \text{for all } a \in A,
\end{aligned}
$$

where, for all $n \in [0, u-1]$, $O_{\mathsf{load}:n}$ is the unique function from $\mathscr{S}$ to $\mathscr{S}$ such that for all $S \in \mathscr{S}$,

$$
\begin{aligned}
O_{\mathsf{load}:n}(S) \upharpoonright (M \setminus \{\mathsf{M_{ld}}[n], \mathsf{rr}_{\mathsf{load}:n}\}) &= S \upharpoonright (M \setminus \{\mathsf{M_{ld}}[n], \mathsf{rr}_{\mathsf{load}:n}\}) \\
O_{\mathsf{load}:n}(S)(\mathsf{M_{ld}}[n]) &= S(\mathsf{M_{data}}[S(\mathsf{M_{la}}[n])]) \\
O_{\mathsf{load}:n}(S)(\mathsf{rr}_{\mathsf{load}:n}) &= \mathsf{T},
\end{aligned}
$$

and, for all $n \in [0, v-1]$, $O_{\mathsf{store}:n}$ is the unique function from $\mathscr{S}$ to $\mathscr{S}$ such that for all $S \in \mathscr{S}$,

$$
\begin{aligned}
O_{\mathsf{store}:n}(S) \upharpoonright (M \setminus \{\mathsf{M_{data}}[S(\mathsf{M_{sa}}[n])], \mathsf{rr}_{\mathsf{store}:n}\}) &= S \upharpoonright (M \setminus \{\mathsf{M_{data}}[S(\mathsf{M_{sa}}[n])], \mathsf{rr}_{\mathsf{store}:n}\}) \\
O_{\mathsf{store}:n}(S)(\mathsf{M_{data}}[S(\mathsf{M_{sa}}[n])]) &= S(\mathsf{M_{sd}}[n]) \\
O_{\mathsf{store}:n}(S)(\mathsf{rr}_{\mathsf{store}:n}) &= \mathsf{T},
\end{aligned}
$$

and, for all $a \in A'$, O_a is a function from $\mathscr{S}$ to $\mathscr{S}$ such that,

$$
\begin{aligned}
IR(O_a) &\subseteq \mathsf{M_{ou}} \cup \mathsf{M_{ld}} \\
OR(O_a) &\subseteq \mathsf{M_{ou}} \cup \mathsf{M_{sd}} \cup \mathsf{M_{la}} \cup \mathsf{M_{sa}} \cup \{\mathsf{rr}_a\}.
\end{aligned}
$$

We have made the deliberate decision to restrict consideration to load/store architectures. We believe that load/store architectures give rise to a relatively simple interface between the data memory and the operating unit. For example, with an architecture other than a load/store architecture, it is more difficult to identify statically, when we are concerned with instructions for data manipulation and/or data transfer, those cases in which the operations associated with instructions that follow each other can be safely performed in a different order or in parallel.

https://doi.org/10.1017/S0960129507006548 Published online by Cambridge University Press

A strict load/store Maurer ISA is strict in the following respects:

— with data transfer between the data memory and the operating unit, there is a strict separation between memory elements used for loading data, loading addresses, storing data, and storing addresses;
— from these memory elements, only the memory elements used for loading data are allowed in the input regions of data manipulation operations;
— a data memory whose size is less than the number of addresses determined by the address width is not allowed.

The first two ways in which a strict load/store Maurer ISA is strict relate to the interface between the data memory and the operating unit. We believe that they yield the most conveniently arranged interface for theoretical work relevant to micro-architecture design. More complicated interfaces are found in many load/store architectures for which there are implementations. The third way in which a strict load/store Maurer ISA is strict saves us from having to deal with addresses that do not actually address a memory element. Such addresses can be dealt with in many different ways, each of which complicates the architecture considerably. We consider their exclusion desirable in much theoretical work relevant to micro-architecture design.

An anonymous referee drew our attention to the fact that a strict separation between memory elements used for loading data, loading addresses, storing data, and storing addresses was also made in Cray and Thornton's design of the CDC 6600 computer (Thornton 1970), which was arguably the first load/store architecture to be implemented. However, in their design, the memory elements used for storing data are also allowed in the input regions of data manipulation operations.

13. Conclusions

We have modelled micro-architectures with non-pipelined instruction processing and pipelined instruction processing, using Maurer machines, basic thread algebra and program algebra. Because our descriptions of micro-architectures are more precise than those usually given, we have been able to verify that stored programs are executed as intended with these micro-architectures. Also, a thorough understanding of the issues relevant to pipelined instruction processing can be acquired by modelling micro-architectures based on different pipeline organisations.

In this paper, pipelined instruction processing deals with control conflicts, but does not deal with data conflicts. Because memory access is not made explicit, data conflicts simply do not occur in the model presented in this paper. Models in which memory access is made explicit may have it placed in a separate pipeline stage, as a result of which data conflicts may occur. In those models, additional assumptions are needed about the instruction set architecture. Such additional assumptions are incorporated in the concept of a strict load/store Maurer instruction set architecture introduced in this paper.

Several techniques for speeding up instruction processing involve multi-threading, a form of concurrency where some interleaving strategy determines how threads that exist concurrently are interleaved (see also Bergstra and Middelburg (2007c) and Bergstra

and Middelburg (2005)). When modelling micro-architectures for those techniques, the enabledness of basic actions, as discussed in Section 4, is likely to be relevant. It is certainly relevant in the case of micro-threading (Bolychevsky *et al.* 1996; Jesshope and Luo 2000).

There are many options for future work. We will just mention the modelling of micro-architectures for different combinations of instruction set architectures and techniques for speeding up instruction processing. Through this, the work presented in this paper may grow into a theoretical basis for micro-architecture design.

The work presented in this paper, as well as the preceding work presented in Bergstra and Middelburg (2007a), has convinced us that a special notation for the description of micro-architectures is desirable. For example, it is annoying that there has to be an explicit description for each memory element that is not affected by an operation. However, we found that fixing an appropriate notation still requires some significant design decisions. We aim at a notation for which the semantics can be given simply by a translation to logical formulas, much in the spirit of predicative methodology (Hehner *et al.* 1986). The following alternative description of the operation O_{fetch} from Section 7 shows what an appropriate notation might look like:

$$O_{\mathsf{fetch}} : \mathbf{if}\ \mathsf{pc} + 1 \leqslant \mathsf{pcbr}\ \mathbf{then}\ \mathsf{pc} := \mathsf{pc} + 1\,,\\ \mathbf{if}\ \mathsf{pc} \leqslant \mathsf{pcbr}\ \mathbf{then}\ (\mathsf{ir} := \mathsf{M}_{\mathsf{prog}}[\mathsf{pc}]\ ;\ \mathsf{rr} := \mathsf{T})\ \mathbf{else}\ (\mathsf{ir} := \#0\ ;\ \mathsf{rr} := \mathsf{F})\,.$$

The work presented in Bergstra and Middelburg (2007a) and this paper has also convinced us that modularity is material to this work: it is about combining and extending models and about renaming and hiding names used in those models. All this has until now been done informally, but in the future there may arise a need to formalise it. We believe that module algebra Bergstra *et al.* (1990) is a suitable formalism on which to base that formalisation.

Parallel composability in connection with pipelined instruction processing is studied in a different setting in Hoe and Arvind (2004). Using algebraic techniques from Harman and Tucker (1996), three simple pipelined systems and a pipelined implementation of a micro-processor are both modelled and verified in Fox and Harman (2003) and Fox (1998), respectively. The simple pipelined systems as well as the pipelined implementation of a micro-processor are modelled as iterated maps. By modelling a pipelined micro-processor as an iterated map, we can use a level of abstraction that is higher than that at which micro-architecture design takes place. We focus our attention on modelling at the latter level of abstraction. A very extensive and up-to-date overview of interesting work on modelling and verifying pipelined micro-processors can also be found in Fox and Harman (2003).

Acknowledgements

We thank Bob Diertens from the University of Amsterdam, Programming Research Group, for carefully reading a draft of this paper, for contributing the pictures included in this paper, and for implementing a simulator for the micro-architectures modelled in this paper. We are grateful to an anonymous referee for his/her valuable comments both on technical points and matters of presentation.

References

Baeten, J. C. M. and Weijland, W. P. (1990) *Process Algebra*, Cambridge Tracts in Theoretical Computer Science **18**, Cambridge University Press.

Bergstra, J. A. and Bethke, I. (2003) Polarized process algebra and program equivalence. In: Baeten, J. C. M., Lenstra, J. K., Parrow, J. and Woeginger, G. J. (eds.) Proceedings 30th ICALP. *Springer-Verlag Lecture Notes in Computer Science* **2719** 1–21.

Bergstra, J. A., Heering, J. and Klint, P. (1990) Module algebra. *Journal of the ACM* **37** (2) 335–372.

Bergstra, J. A. and Klop, J. W. (1984) Process algebra for synchronous communication. *Information and Control* **60** (1/3) 109–137.

Bergstra, J. A. and Loots, M. E. (2002) Program algebra for sequential code. *Journal of Logic and Algebraic Programming* **51** (2) 125–156.

Bergstra, J. A. and Middelburg, C. A. (2005) A thread algebra with multi-level strategic interleaving. In: Cooper, S. B., Löwe, B. and Torenvliet, L. (eds.) CiE 2005. *Springer-Verlag Lecture Notes in Computer Science* **3526** 35–48.

Bergstra, J. A. and Middelburg, C. A. (2006a) Splitting bisimulations and retrospective conditions. *Information and Computation* **204** (7) 1083–1138.

Bergstra, J. A. and Middelburg, C. A. (2006b) Thread algebra with multi-level strategies. *Fundamenta Informaticae* **71** (2/3) 153–182.

Bergstra, J. A. and Middelburg, C. A. (2007a) Maurer computers with single-thread control. *Fundamenta Informaticae* **80** (4) 333–362.

Bergstra, J. A. and Middelburg, C. A. (2007b) Simulating Turing machines on Maurer machines. *Journal of Applied Logic*, doi:10.1016/j.jal.2007.04.001.

Bergstra, J. A. and Middelburg, C. A. (2007c) Thread algebra for strategic interleaving. *Formal Aspects of Computing* **19** (4) 445–474.

Bergstra, J. A. and Ponse, A. (2002) Combining programs and state machines. *Journal of Logic and Algebraic Programming* **51** (2) 175–192.

Bolychevsky, A., Jesshope, C. R. and Muchnick, V. (1996) Dynamic scheduling in RISC architectures. *IEE Proceedings Computers and Digital Techniques* **143** (5) 309–317.

Brookes, S. D., Hoare, C. A. R. and Roscoe, A. W. (1984) A theory of communicating sequential processes. *Journal of the ACM* **31** (3) 560–599.

Fox, A. J. C. (1998) *Algebraic Representation of Advanced Microprocessors*, Ph.D. thesis, Department of Computer Science, Swansea University.

Fox, A. J. C. and Harman, N. A. (2003) Algebraic models of correctness for abstract pipelines. *Journal of Logic and Algebraic Programming* **57** (1/2) 71–107.

Harman, N. A. and Tucker, J. V. (1996) Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica* **33** 421–456.

Hehner, E. C. R., Gupta, L. E. and Malton, A. J. (1986) Predicative methodology. *Acta Informatica* **23** 487–505.

Hennessy, J. L. and Patterson, D. A. (2003) *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*, Prentice-Hall.

Hoe, J. C. and Arvind (2004) Operation-centric hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **23** (9) 1277–1288.

Hopcroft, J. E., Motwani, R. and Ullman, J. D. (2001) *Introduction to Automata Theory, Languages and Computation*, second edition, Addison-Wesley.

Jesshope, C. R. and Luo, B. (2000) Micro-threading: A new approach to future RISC. In: *ACAC 2000*, IEEE Computer Society Press 34–41.

Maurer, W. D. (1966) A theory of computer instructions. *Journal of the ACM* **13** (2) 226–235.
Maurer, W. D. (2006) A theory of computer instructions. *Science of Computer Programming* **60** 244–273.
Milner, R. (1980) A Calculus of Communicating Systems. *Springer-Verlag Lecture Notes in Computer Science* **92**.
Milner, R. (1989) *Communication and Concurrency*, Prentice-Hall.
Ponse, A. (2002) Program algebra with unit instruction operators. *Journal of Logic and Algebraic Programming* **51** (2) 157–174.
Sima, D. (2004) Decisive aspects in the evolution of microprocessors. *Proceedings of the IEEE* **92** (12) 1896–1926.
Thornton, J. (1970) *Design of a Computer – The Control Data 6600*, Scott, Foresman and Co.