University of Amsterdam
Programming Research Group

On the Operating Unit Size of Load/Store Architectures

J.A. Bergstra
C.A. Middelburg

Report PRG0703 August 2007

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

C.a. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

On the Operating Unit Size of Load/Store Architectures*

J.A. Bergstra[1,2] and C.A. Middelburg[1]

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

Abstract. We introduce a strict version of the concept of a load/store instruction set architecture in the setting of Maurer machines. We take the view that transformations on the states of a Maurer machine are achieved by applying threads as considered in thread algebra to the Maurer machine. We study how the transformations on the states of the main memory of a strict load/store instruction set architecture that can be achieved by applying threads depend on the operating unit size, the cardinality of the instruction set, and the maximal number of states of the threads.

Keywords: load/store instruction set architecture, operating unit size, Maurer machine, thread algebra, thread powered function class.

1998 ACM Computing Classification: C.0, F.1.1, F.1.3.

1 Introduction

In [4], we introduced Maurer machines, which are based on the model for computers proposed by Maurer in [11], and extended basic thread algebra, which is introduced in [3] under the name basic polarized process algebra, with operators for applying threads to Maurer machines. Threads can be looked upon as the behaviours of deterministic sequential programs as run on a machine. By applying threads to a Maurer machine, transformations on the states of the Maurer machine are achieved. In [5], we proposed a strict version of the concept of a load/store instruction set architecture for theoretical work relevant to micro-architecture design. We described the concept in the setting of Maurer machines. The idea underlying it is that there is a main memory of which the elements contain data, an operating unit with a small internal memory by which data can be manipulated, and an interface between the main memory and the operating unit for data transfer between them.

* This research was partly carried out in the framework of the GLANCE-project MICROGRIDS, which is funded by the Netherlands Organisation for Scientific Research (NWO).

In this paper, we study how the transformations on the states of the main memory of a strict load/store instruction set architecture that can be achieved by applying threads to it depend on the operating unit size, the cardinality of the instruction set, and the maximal number of states of the threads. In order to present certain results in a conveniently arranged way, we introduce the concept of a thread powered function class. The idea underlying this concept is that the transformations on the main memory of strict load/store instruction set architectures that can be achieved by applying threads to them are primarily determined by the address width, the word length, the operating unit size, and the cardinality of the instruction set of the instruction set architectures and the number of states of the threads that can be applied to them.

Why did we choose to use Maurer machines and basic thread algebra to study issues relevant to the design of instruction set architectures? Maurer machines are based on the view that a computer has a memory, the contents of all memory elements make up the state of the computer, the computer processes instructions, and the processing of an instruction amounts to performing an operation on the state of the computer which results in changes of the contents of certain memory elements. The design of instruction set architectures must deal with these aspects of real computers. Turing machines and the other kinds of machines known from theoretical computer science (see e.g. [10]) abstract from these aspects of real computers. Basic thread algebra is a form of process algebra. Well-known process algebras, such as ACP [1], CCS [12], and CSP [9], are too general for our purpose. Basic thread algebra has been designed as an algebra of deterministic sequential processes that interact with a machine. In [7], 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 [6]. However, it is quite awkward to describe and analyse processes of this kind using such a general process algebra.

The structure of this paper is as follows. First, we review basic thread algebra (Section 2), Maurer machines (Section 3) and the operators for applying threads to Maurer machines (Section 4). Next, we introduce the concept of a strict load/store Maurer instruction set architecture (Section 5). Then, we study the consequences of reducing the operating unit size of a strict load/store Maurer instruction set architecture (Section 6). After that, we give conditions under which all possible transformations on the states of the main memory of a strict load/store Maurer ISA with a certain address width and word length can be achieved by applying a thread to such a strict load/store Maurer ISA (Section 7). Following this, we give a condition under which not all possible transformations can be achieved (Section 8). Finally, we make some concluding remarks (Section 9).

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), a form of process algebra which was first presented in [3] under the name BPPA (Basic Polarized Process

Table 1. 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 $X \in V(E)$ RSP

Algebra). It is a form of process algebra which is tailored to the description of 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* $\mathcal{A}$. BTA has the following constants and operators:

- the *deadlock* constant D;
- the *termination* constant S;
- for each $a \in \mathcal{A}$, 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 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).

Each closed term of BTA denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Guarded recursive specifications give rise to infinite threads.

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is a term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' terms of BTA that contain only variables from V. We write $V(E)$ for the set of all variables that occur on the left-hand side of an equation in 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 [2].

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 $X \in V(E)$, we add a constant standing for the unique solution of E for X to the constants of BTA. The constant standing for the unique solution of E for X is denoted by $\langle X|E\rangle$. Moreover, we add the axioms for guarded recursion given in Table 1 to BTA, where we write $\langle t_X|E\rangle$ for t_X with, for all $Y \in V(E)$, all occurrences of Y in t_X replaced by $\langle Y|E\rangle$. In this table, X, t_X and E stand for an arbitrary variable, an arbitrary term of BTA and an arbitrary guarded recursive specification, re-

Table 2. Approximation induction principle

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

Table 3. 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

spectively. Side conditions are added to restrict the variables, terms and guarded recursive specifications for which X, t_X and E stand.

We will write BTA+REC for BTA extended with the constants for solutions of guarded recursive specifications and axioms RDP and RSP. We will 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.

Closed terms of BTA+REC that denote the same infinite thread cannot always be proved equal by means of the axioms of BTA+REC. We introduce the approximation induction principle to remedy this. The approximation induction principle, AIP in short, is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a thread is obtained by cutting it off after performing a sequence of actions of length n.

AIP is the infinitary conditional equation given in Table 2. Here, following [2], approximation of depth n is phrased in terms of a unary *projection* operator $\pi_n(\_)$. The axioms for the projection operators are given in Table 3. In this table, a stands for an arbitrary member of $\mathcal{A}$.

Henceforth, we write $\mathcal{E}_{\mathsf{fin}}(A)$, where $A \subseteq \mathcal{A}$, for the set of all finite guarded recursive specifications over BTA that contain only postconditional operators $\_ \trianglelefteq a \trianglerighteq \_$ for which $a \in A$. Moreover, we write $\mathcal{T}_{\mathsf{finrec}}(A)$, where $A \subseteq \mathcal{A}$, for the set of all closed terms of BTA+REC that contain only postconditional operators $\_ \trianglelefteq a \trianglerighteq \_$ for which $a \in A$ and only constants $\langle X|E\rangle$ for which $E \in \mathcal{E}_{\mathsf{fin}}(A)$.

A *linear recursive specification* over BTA is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$, where each t_X is a term of the form D, S or $Y \trianglelefteq a \trianglerighteq Z$ with $Y, Z \in V$. For each closed term $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, there exist a linear recursive specification $E \in \mathcal{E}_{\mathsf{fin}}(A)$ and a variable $X \in V(E)$ such that $p = \langle X|E\rangle$ is derivable from the axioms of BTA+REC.

Henceforth, we write $\mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A)$, where $A \subseteq \mathcal{A}$, for the set of all linear recursive specifications from $\mathcal{E}_{\mathsf{fin}}(A)$.

Below, the interpretations of the constants and operators of BTA+REC in models of BTA+REC are denoted by the constants and operators themselves. Let $\mathcal{A}$ be some model of BTA+REC, and let p be an element from the domain of

$\mathcal{A}$. Then the set of *states* or *residual threads* of p, written $Res(p)$, is inductively defined as follows:

- $p \in Res(p)$;
- if $q \trianglelefteq a \trianglerighteq r \in Res(p)$, then $q \in Res(p)$ and $r \in Res(p)$.

We are only interested in models of BTA+REC in which $card(Res(\langle X|E\rangle)) \leq card(E)$ for all finite linear recursive specifications E, such as the projective limit model of BTA presented in [2].

3 Maurer Machines

In this section, we introduce the concept of a Maurer machine. This concept was first introduced in [4].

A *Maurer machine* H consists of the following components:

- a non-empty set M;
- a set B with $card(B) \geq 2$;
- a set $\mathcal{S}$ of functions $S : M \to B$;
- a set $\mathcal{O}$ of functions $O : \mathcal{S} \to \mathcal{S}$;
- a set $A \subseteq \mathcal{A}$;
- a function $[\![\_]\!] : A \to (\mathcal{O} \times M)$;

and satisfies the following conditions:

- if $S_1, S_2 \in \mathcal{S}$, $M' \subseteq M$, 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 $S_3 \in \mathcal{S}$;
- if $S_1, S_2 \in \mathcal{S}$, then the set $\{x \in M \mid S_1(x) \neq S_2(x)\}$ is finite;
- if $S \in \mathcal{S}$, $a \in A$, and $[\![a]\!] = (O, m)$, then $S(m) \in \{\mathsf{T}, \mathsf{F}\}$.

M is called the *memory* of H, B is called the *base set* of H, the members of $\mathcal{S}$ are called the *states* of H, the members of $\mathcal{O}$ are called the *operations* of H, the members of A are called the *basic actions* of H, and $[\![\_]\!]$ is called the *basic action interpretation function* of H.

We write M_H, B_H, $\mathcal{S}_H$, $\mathcal{O}_H$, A_H and $[\![\_]\!]_H$, where $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ is a Maurer machine, for M, B, $\mathcal{S}$, $\mathcal{O}$, A and $[\![\_]\!]$, respectively.

A Maurer machine has much in common with a real computer. The memory of a Maurer machine consists of memory elements which have as content an element from its base set. The contents of all memory elements together make up a state of the Maurer machine. State changes are accomplished by performing its operations. Every state change amounts to changes of the contents of certain memory elements. The Maurer machine processes each of its basic actions by performing the operation associated with the basic action by its basic action interpretation function. At completion of the processing, the content of the memory element associated with the basic action by the basic action interpretation function is the reply produced by the Maurer machine. The term basic action originates from BTA. Where real computers are concerned, basic actions are usually called instructions.

In [11], Maurer proposed a model for computers. In [4], we introduced the term Maurer computer for what is a computer according to Maurer's definition. Leaving out the set of basic actions and the basic action interpretation function from a Maurer machine yields a Maurer computer. The set of basic actions and the basic action interpretation function constitute the interface of a Maurer machine with its environment.

The notions of input region of an operation and output region of an operation, which originate from [11], are used in subsequent sections.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, and let $O : \mathcal{S} \to \mathcal{S}$. 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:

$$IR(O) = \big\{x \in M \;\big|\; \exists S_1, S_2 \in \mathcal{S} \bullet (\forall z \in M \setminus \{x\} \bullet S_1(z) = S_2(z) \wedge \exists y \in OR(O) \bullet O(S_1)(y) \neq O(S_2)(y))\big\}\,,$$

$$OR(O) = \big\{x \in M \;\big|\; \exists S \in \mathcal{S} \bullet S(x) \neq O(S)(x)\big\}\,.^{3}$$

$OR(O)$ 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.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, let $S_1, S_2 \in \mathcal{S}$, and let $O \in \mathcal{O}$. Then $S_1 \restriction IR(O) = S_2 \restriction IR(O)$ implies $O(S_1) \restriction OR(O) = O(S_2) \restriction OR(O)$.[4] 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.

4 Applying Threads to Maurer Machines

In this section, we add for each Maurer machine H a binary *apply* operator $\_ \bullet_H \_$ to BTA+REC and introduce a notion of computation in the resulting setting.

The apply operators associated with Maurer machines are related to the apply operators introduced in [8]. 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 $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ must be a term from $\mathcal{T}_{\mathsf{finrec}}(A)$ and its second argument must be a state from $\mathcal{S} \cup \{\uparrow\}$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, let $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, and let $S \in \mathcal{S}$. Then $p \bullet_H S$ is the state that results if all basic actions performed

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

[4] 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 $d \in \mathrm{dom}(g)$, $g(d) = f(d)$.

Table 4. Defining equations for apply operator

$x \bullet_H \uparrow = \uparrow$
$\mathsf{S} \bullet_H S = S$
$\mathsf{D} \bullet_H S = \uparrow$
$(x \trianglelefteq a \trianglerighteq y) \bullet_H S = x \bullet_H O_a(S)$ 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 5. Rule for divergence

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

by thread p are processed by the Maurer machine H from initial state S. The processing of a basic action a by H amounts to a state change according to the operation associated with a by $[\![\_]\!]$. In the resulting state, the reply produced by H is contained in memory element associated with a by $[\![\_]\!]$. If p is S, then there will be no state change. If p is D, then the result is $\uparrow$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the apply operator $\_ \bullet_H \_$ is defined by the equations given in Table 4 and the rule given in Table 5. In these tables, a stands for an arbitrary member of A and S stands for an arbitrary member of $\mathcal{S}$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, let $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, and let $S \in \mathcal{S}$. Then p *converges* from S on H if there exists an $n \in \mathbb{N}$ such that $\pi_n(p) \bullet_H S \neq \uparrow$. The rule from Table 5 can be read as follows: if x does not converge from S on H, then $x \bullet_H S$ equals $\uparrow$.

Below, we introduce a notion of computation in the current setting. First, we introduce some auxiliary notions.

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

- 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 $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ 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 $(p_{n+1}, S_{n+1}) \in \mathcal{T}_{\mathsf{finrec}}(A) \times \mathcal{S}$ 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 $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, and let $S \in \mathcal{S}$. 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, then 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, $\|(p_0, S_0)\|_H$ is the least $n \in \mathbb{N}$ such that $\pi_n(p_0) \bullet_H S_0 \neq \uparrow$.

5 Instruction Set Architectures

In this section, we introduce the concept of a strict load/store Maurer instruction set architecture. This concept, which was first introduced in [5], takes its name from the following: it is described in the setting of Maurer machines, it concerns only load/store architectures, and the load/store architectures concerned are strict in some respects that will be explained after its formalization.

The concept of a strict load/store Maurer instruction set architecture, or shortly 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 considered to be treated in a uniform way over different strict load/store Maurer ISAs.

Each Maurer machine has a number of basic actions with which an operation is associated. Henceforth, when speaking about Maurer machines that are strict load/store Maurer ISAs, such basic actions are loosely called basic instructions. The term basic action is uncommon where we are concerned with ISAs.

The idea underlying the concept of a strict load/store Maurer ISA is that there is a main memory of which the elements contain data, an operating unit with a small internal memory by 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 parametrized by:

- an address width aw;
- a word length wl;
- an operating unit size ous;
- a number $nrpl$ of pairs of address and data registers for load instructions;
- a number $nrps$ of pairs of address and data registers for store instructions;
- a set A_{dm} of basic instructions for data manipulation;

where $aw, ous \geq 0$, $wl, nrpl, nrps > 0$ and $A_{dm} \subseteq \mathcal{A}$.

The address width aw can be regarded as the number of bits used for the binary representation of addresses of data memory elements. The word length wl can be regarded as the number of bits used to represent data in data memory

elements. The operating unit size *ous* can be regarded as the number of bits that the internal memory of the operating unit contains.

It is assumed that, for each $n \in \mathbb{N}$, a fixed but arbitrary countably infinite set $\mathsf{M}^n_{\mathsf{data}}$ and a fixed but arbitrary bijection $\mathsf{m}^n_{\mathsf{data}} : \mathbb{N} \to \mathsf{M}^n_{\mathsf{data}}$ have been given. The members of $\mathsf{M}^n_{\mathsf{data}}$ are called *data memory elements*. The contents of data memory elements are taken as data. The data memory elements from $\mathsf{M}^n_{\mathsf{data}}$ can contain natural numbers in the interval $[0, 2^n - 1]$.

It is assumed that a fixed but arbitrary countably infinite set M_{ou} and a fixed but arbitrary bijection $\mathsf{m}_{\mathsf{ou}} : \mathbb{N} \to \mathsf{M}_{\mathsf{ou}}$ have been given. The members of M_{ou} are called *operating unit memory elements*. They can contain natural numbers in the set $\{0, 1\}$, i.e. bits. Usually, a part of the operating unit memory is partitioned into groups to which data manipulation instructions can refer.

It is assumed that, for each $n \in \mathbb{N}$, fixed but arbitrary countably infinite sets $\mathsf{M}^n_{\mathsf{ld}}$, $\mathsf{M}^n_{\mathsf{sd}}$, $\mathsf{M}^n_{\mathsf{la}}$ and $\mathsf{M}^n_{\mathsf{sa}}$ and fixed but arbitrary bijections $\mathsf{m}^n_{\mathsf{ld}} : \mathbb{N} \to \mathsf{M}^n_{\mathsf{ld}}$, $\mathsf{m}^n_{\mathsf{sd}} : \mathbb{N} \to \mathsf{M}^n_{\mathsf{sd}}$, $\mathsf{m}^n_{\mathsf{la}} : \mathbb{N} \to \mathsf{M}^n_{\mathsf{la}}$ and $\mathsf{m}^n_{\mathsf{sa}} : \mathbb{N} \to \mathsf{M}^n_{\mathsf{sa}}$ have been given. The members of $\mathsf{M}^n_{\mathsf{ld}}$, $\mathsf{M}^n_{\mathsf{sd}}$, $\mathsf{M}^n_{\mathsf{la}}$ and $\mathsf{M}^n_{\mathsf{sa}}$ are called *load data registers*, *store data registers*, *load address registers* and *store address registers*, respectively. The contents of load data registers and store data registers are taken as data, whereas the contents of load address registers and store address registers are taken as addresses. The load data registers from $\mathsf{M}^n_{\mathsf{ld}}$, the store data registers from $\mathsf{M}^n_{\mathsf{sd}}$, the load address registers from $\mathsf{M}^n_{\mathsf{la}}$ and the store address registers from $\mathsf{M}^n_{\mathsf{sa}}$ can contain natural numbers in the interval $[0, 2^n - 1]$. The load and store registers are special memory elements designated for transferring data between the data memory and the operating unit memory.

It is assumed that, for each $n, n' \in \mathbb{N}$, $\mathsf{M}^n_{\mathsf{data}}$, M_{ou}, $\mathsf{M}^n_{\mathsf{ld}}$, $\mathsf{M}^n_{\mathsf{sd}}$, $\mathsf{M}^{n'}_{\mathsf{la}}$, $\mathsf{M}^{n'}_{\mathsf{sa}}$ and $\{\mathsf{rr}\}$ are pairwise disjoint sets.

If $M \subseteq \mathsf{M}^n_{\mathsf{data}}$ and $\mathsf{m}^n_{\mathsf{data}}(i) \in M$, then we write $M[i]$ for $\mathsf{m}^n_{\mathsf{data}}(i)$. If $M \subseteq \mathsf{M}^n_{\mathsf{ld}}$ and $\mathsf{m}^n_{\mathsf{ld}}(i) \in M$, then we write $M[i]$ for $\mathsf{m}^n_{\mathsf{ld}}(i)$. If $M \subseteq \mathsf{M}^n_{\mathsf{sd}}$ and $\mathsf{m}^n_{\mathsf{sd}}(i) \in M$, then we write $M[i]$ for $\mathsf{m}^n_{\mathsf{sd}}(i)$. If $M \subseteq \mathsf{M}^n_{\mathsf{la}}$ and $\mathsf{m}^n_{\mathsf{la}}(i) \in M$, then we write $M[i]$ for $\mathsf{m}^n_{\mathsf{la}}(i)$. If $M \subseteq \mathsf{M}^n_{\mathsf{sa}}$ and $\mathsf{m}^n_{\mathsf{sa}}(i) \in M$, then we write $M[i]$ for $\mathsf{m}^n_{\mathsf{sa}}(i)$.

Let $aw, ous \geq 0$, $wl, nrpl, nrps > 0$ and $A_{dm} \subseteq \mathcal{A}$. Then a *strict load/store Maurer instruction set architecture* with parameters aw, wl, ous, $nrpl$, $nrps$ and A_{dm} is a Maurer machine $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ with

$$
\begin{aligned}
M &= M_{data} \cup M_{ou} \cup M_{ld} \cup M_{sd} \cup M_{la} \cup M_{sa} \cup \{\mathsf{rr}\} \,, \\
B &= [0, 2^{wl} - 1] \cup [0, 2^{aw} - 1] \cup \mathbb{B} \,, \\
\mathcal{S} &= \{S : M \to B \mid \\
&\qquad \forall m \in M_{data} \cup M_{ld} \cup M_{sd} \bullet S(m) \in [0, 2^{wl} - 1] \wedge \\
&\qquad \forall m \in M_{la} \cup M_{sa} \bullet S(m) \in [0, 2^{aw} - 1] \wedge \\
&\qquad \forall m \in M_{ou} \bullet S(m) \in \{0, 1\} \wedge S(\mathsf{rr}) \in \mathbb{B}\} \,, \\
\mathcal{O} &= \{O_a \mid a \in A\} \,, \\
A &= \{\mathsf{load}{:}n \mid n \in [0, nrpl - 1]\} \cup \{\mathsf{store}{:}n \mid n \in [0, nrps - 1]\} \cup A_{dm} \,, \\
[\![a]\!] &= (O_a, \mathsf{rr}) \quad \text{for all } a \in A \,,
\end{aligned}
$$

where

$$
\begin{aligned}
M_{data} &= \{\mathsf{m}^{wl}_{\mathsf{data}}(i) \mid i \in [0, 2^{aw} - 1]\} \ , \\
M_{ou} &= \{\mathsf{m}_{\mathsf{ou}}(i) \mid i \in [0, ous - 1]\} \ , \\
M_{ld} &= \{\mathsf{m}^{wl}_{\mathsf{ld}}(i) \mid i \in [0, nrpl - 1]\} \ , \\
M_{sd} &= \{\mathsf{m}^{wl}_{\mathsf{sd}}(i) \mid i \in [0, nrps - 1]\} \ , \\
M_{la} &= \{\mathsf{m}^{aw}_{\mathsf{la}}(i) \mid i \in [0, nrpl - 1]\} \ , \\
M_{sa} &= \{\mathsf{m}^{aw}_{\mathsf{sa}}(i) \mid i \in [0, nrps - 1]\} \ ,
\end{aligned}
$$

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

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

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

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

and, for all $a \in A_{dm}$, O_a is a function from $\mathcal{S}$ to $\mathcal{S}$ such that:

$$IR(O_a) \subseteq M_{ou} \cup M_{ld} \ ,$$

$$OR(O_a) \subseteq M_{ou} \cup M_{sd} \cup M_{la} \cup M_{sa} \cup \{\mathsf{rr}\} \ .$$

We will write $\mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, nrpl, nrps, A_{dm})$ for the set of all strict load/store Maurer ISAs with parameters aw, wl, ous, $nrpl$, $nrps$ and A_{dm}.

In our opinion, load/store architectures give rise to a relatively simple interface between the data memory and the operating unit.

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

- with data transfer between the data memory and the operating unit, a strict separation is made 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 of which the 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 concern 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 the design of instruction set architectures. The third way in which a strict load/store Maurer ISA is strict saves the need to deal with addresses that do not 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 the design of instruction set architectures.

A strict separation between memory elements used for loading data, loading addresses, storing data, and storing addresses is also made in Cray and Thornton's design of the CDC 6600 computer [13], which is arguably the first implemented load/store architecture. However, in their design, the memory elements used for storing data are also allowed in the input regions of data manipulation operations.

6 Reducing the Operating Unit Size

In a strict load/store Maurer ISA, data manipulation takes place in the operating unit. This raises questions concerning the consequences of changing the operating unit size. One of the questions is whether, if the operating unit size is reduced by one, it is possible with new instructions for data manipulation to transform each thread that can be applied to the original ISA into one or more threads that can each be applied to the ISA with the reduced operating unit size and together yield the same state changes on the data memory. This question can be answered in the affirmative.

Theorem 1. *Let* $aw \geq 0$, $wl, ous, nrpl, nrps > 0$ *and* $A_{dm} \subseteq \mathcal{A}$, *let* $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!]) \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, nrpl, nrps, A_{dm})$, *and let* $M_{data} = \{\mathsf{m}^{wl}_{\mathsf{data}}(i) \mid i \in [0, 2^{aw}-1]\}$ *and* $\mathsf{bc} = \mathsf{m}_{\mathsf{ou}}(ous-1)$. *Then there exist an* $A'_{dm} \subseteq \mathcal{A}$ *and an* $H' = (M', B, \mathcal{S}', \mathcal{O}', A', [\![\_]\!]') \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous-1, nrpl, nrps, A'_{dm})$ *such that for all* $p \in \mathcal{T}_{\mathsf{finrec}}(A)$ *there exist* $p'_0, p'_1 \in \mathcal{T}_{\mathsf{finrec}}(A')$ *such that*

$$\{(S \restriction M_{data}, (p \bullet_H S) \restriction M_{data}) \mid S \in \mathcal{S} \wedge S(\mathsf{bc}) = 0\}$$
$$= \{(S' \restriction M_{data}, (p'_0 \bullet_{H'} S') \restriction M_{data}) \mid S' \in \mathcal{S}'\}$$

and

$$\{(S \restriction M_{data}, (p \bullet_H S) \restriction M_{data}) \mid S \in \mathcal{S} \wedge S(\mathsf{bc}) = 1\}$$
$$= \{(S' \restriction M_{data}, (p'_1 \bullet_{H'} S') \restriction M_{data}) \mid S' \in \mathcal{S}'\}\ .$$

In the proof of Theorem 1 given below, we take A'_{dm} such that, for each instruction a in A_{dm}, there are four instructions $a(0)$, $a(1)$, $\overline{a}(0)$ and $\overline{a}(1)$ in A'_{dm}. $O_{a(0)}$ and $O_{a(1)}$ affect the memory elements of H' like O_a would affect them if the content of the missing operating unit memory element would be 0 and 1, respectively. The effect that O_a would have on the missing operating unit memory

element is made available by $O_{\overline{a}(0)}$ and $O_{\overline{a}(1)}$, respectively. They do nothing but replying F if the content of the missing operating unit memory element would become 0 and T if the content of the missing operating unit memory element would become 1.

Proof (of Theorem 1). Instead of the result to be proved, we prove that there exist an $A'_{dm} \subseteq \mathcal{A}$ and an $H' = (M', B, \mathcal{S}', \mathcal{O}', A', [\![\_]\!]') \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous - 1, nrpl, nrps, A'_{dm})$ such that for all $p \in \mathcal{T}_{\mathsf{finrec}}(A)$ there exist $p'_0, p'_1 \in \mathcal{T}_{\mathsf{finrec}}(A')$ such that

$$\begin{aligned}&\{(S \restriction (M' \setminus \{\mathsf{rr}\}), (p \bullet_H S) \restriction (M' \setminus \{\mathsf{rr}\})) \mid S \in \mathcal{S} \wedge S(\mathsf{bc}) = 0\}\\ &\quad = \{(S' \restriction (M' \setminus \{\mathsf{rr}\}), (p'_0 \bullet_{H'} S') \restriction (M' \setminus \{\mathsf{rr}\})) \mid S' \in \mathcal{S}'\}\end{aligned}$$

and

$$\begin{aligned}&\{(S \restriction (M' \setminus \{\mathsf{rr}\}), (p \bullet_H S) \restriction (M' \setminus \{\mathsf{rr}\})) \mid S \in \mathcal{S} \wedge S(\mathsf{bc}) = 1\}\\ &\quad = \{(S' \restriction (M' \setminus \{\mathsf{rr}\}), (p'_1 \bullet_{H'} S') \restriction (M' \setminus \{\mathsf{rr}\})) \mid S' \in \mathcal{S}'\}\ .\end{aligned}$$

This is sufficient because $M_{data} \subseteq M' \setminus \{\mathsf{rr}\}$.

We take $A'_{dm} = \{a(k), \overline{a}(k) \mid a \in A_{dm} \wedge k \in \{0,1\}\}$, and we take $H' = (M', B, \mathcal{S}', \mathcal{O}', A', [\![\_]\!]')$ such that, for each $a \in A_{dm}$ and $k \in \{0,1\}$, $O_{a(k)}$ and $O_{\overline{a}(k)}$ are the unique functions from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}O_{a(k)}(S') &= O_a(\rho_k(S')) \restriction M'\ ,\\ O_{\overline{a}(k)}(S') \restriction (M' \setminus \{\mathsf{rr}\}) &= S' \restriction (M' \setminus \{\mathsf{rr}\})\ ,\\ O_{\overline{a}(k)}(S')(\mathsf{rr}) &= \gamma(O_a(\rho_k(S'))(\mathsf{bc}))\ ,\end{aligned}$$

where, for each $k \in \{0,1\}$, ρ_k is the unique function from $\mathcal{S}'$ to $\mathcal{S}$ such that

$$\begin{aligned}\rho_k(S') \restriction M' &= S'\ ,\\ \rho_k(S')(\mathsf{bc}) &= k\end{aligned}$$

and $\gamma : \{0,1\} \to \mathbb{B}$ is defined by

$$\begin{aligned}\gamma(0) &= \mathsf{F}\ ,\\ \gamma(1) &= \mathsf{T}\ .\end{aligned}$$

We restrict ourselves to $p \in \{\langle X|E\rangle \mid E \in \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A) \wedge X \in V(E)\}$, because each term from $\mathcal{T}_{\mathsf{finrec}}(A)$ can be proved equal to some constant from this set by means of the axioms of BTA+REC.

We define transformation functions $\phi_k : \{\langle X|E\rangle \mid E \in \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A) \wedge X \in V(E)\} \to \{\langle X|E\rangle \mid E \in \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A') \wedge X \in V(E)\}$, for $k \in \{0,1\}$, as follows:

$$\phi_k(\langle X|E\rangle) = \langle X_k|\phi'_k(E)\rangle\ ,$$

where $\phi'_k : \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A) \to \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A')$, for $k \in \{0,1\}$ is defined as follows:

$$\begin{aligned}
\phi'_k(\{X = \mathsf{S}\}) &= \{X_k = \mathsf{S}\}\ ,\\
\phi'_k(\{X = \mathsf{D}\}) &= \{X_k = \mathsf{D}\}\ ,\\
\phi'_k(\{X = Y \trianglelefteq a \trianglerighteq Z\}) &= \{X_k = Y_k \trianglelefteq a \trianglerighteq Z_k\} \qquad \text{if } a \notin A_{dm}\ ,\\
\phi'_k(\{X = Y \trianglelefteq a \trianglerighteq Z\}) &= \{X_k = X'_k \trianglelefteq \overline{a}(k) \trianglerighteq X''_k,\\
&\qquad\quad X'_k = Y_1 \trianglelefteq a(k) \trianglerighteq Z_1,\\
&\qquad\quad X''_k = Y_0 \trianglelefteq a(k) \trianglerighteq Z_0\} \quad \text{if } a \in A_{dm}\ ,\\
\phi'_k(E' \cup E'') &= \phi'_k(E') \cup \phi'_k(E'')\ .
\end{aligned}$$

Here, for each variable X, the new variables X_0, X'_0, X''_0, X_1, X'_1 and X''_1 are taken such that: (i) they are pairwise different variables; (ii) for each variable Y different from X, $\{X_0, X'_0, X''_0, X_1, X'_1, X''_1\}$ and $\{Y_0, Y'_0, Y''_0, Y_1, Y'_1, Y''_1\}$ are disjoint sets.

Let $p \in \{\langle X|E\rangle \mid E \in \mathcal{E}^{\mathsf{lin}}_{\mathsf{fin}}(A) \wedge X \in V(E)\}$, let $S \in \mathcal{S}$ and $S' \in \mathcal{S}'$ be such that $S \restriction M' = S'$, let (p_i, S_i) be the $(i+1)$st element in the full path of (p, S) on H, and let (p'_i, S'_i) be the $(i+1)$st element in the full path of $(\phi_{S(\mathsf{bc})}(p), S')$ on H' of which the first component does not equal $p \trianglelefteq a(k) \trianglerighteq q$ for any $p, q \in \mathcal{T}_{\mathsf{finrec}}$, $a \in A_{dm}$ and $k \in \{0,1\}$. Moreover, let a_i be the unique $a \in A$ such that $p_i = p \trianglelefteq a \trianglerighteq q$ for some $p, q \in \mathcal{T}_{\mathsf{finrec}}$. Then, it is easy to prove by induction on i that if $a_i \in A_{dm}$:

$$O_{a_i}(S_i)(\mathsf{bc}) = \gamma^{-1}(O_{\overline{a_i}(S_i(\mathsf{bc}))}(S'_i)(\mathsf{rr}))\ , \tag{1}$$

$$O_{a_i}(S_i)(\mathsf{rr}) = O_{a_i(S_i(\mathsf{bc}))}(O_{\overline{a}_i(S_i(\mathsf{bc}))}(S'_i))(\mathsf{rr}) \tag{2}$$

(if $i + 1 < \|(p, S)\|_H$ in case p converges from S on H). Now, using (1) and (2), it is easy to prove by induction on i that:

$$\begin{aligned}
&\phi_{S_i(\mathsf{bc})}(p_i) = p'_i\ ,\\
&S_i \restriction (M \setminus \{\mathsf{rr}\}) = \rho_{S_i(\mathsf{bc})}(S'_i) \restriction (M \setminus \{\mathsf{rr}\})
\end{aligned}$$

(if $i < \|(p, S)\|_H$ in case p converges from S on H). From this, the result follows immediately. □

The proof of Theorem 1 gives us some upper bounds:

- for each thread that can be applied to the original ISA, the number of threads that can together produce the same state changes on the data memory of the ISA with the reduced operating unit does not have to be more than 2;
- the number of states of the new threads does not have to be more than 6 times the number of states of the original thread;
- the number of steps that the new threads take to produce some state change does not have to be more than 2 times the number of steps that the original thread takes to produce that state change;
- the number of instructions of the ISA with the reduced operating unit does not have to be more than 4 times the number of instructions of the original ISA.

Moreover, the proof indicates that more efficient new threads are possible: equations $X = Y \trianglelefteq a \trianglerighteq Z$ with $a \in A_{dm}$ can be treated as if $a \notin A_{dm}$ in the case where the missing operating unit memory element is not in $IR(O_a)$.

As a corollary of the proof of Theorem 1, we have that only one transformed thread is needed if the input region of the operation associated with the first instruction performed by the original thread does not include the operating unit memory element that is missing in the ISA with the reduced operating unit size.

Corollary 1. *Let $aw \geq 0$, $wl, ous, nrpl, nrps > 0$ and $A_{dm} \subseteq \mathcal{A}$, let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!]) \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, nrpl, nrps, A_{dm})$, and let $M_{data} = \{\mathsf{m}^{wl}_{\mathsf{data}}(i) \mid i \in [0, 2^{aw}-1]\}$ and $\mathsf{bc} = \mathsf{m}_{\mathsf{ou}}(ous-1)$. Moreover, let $\mathcal{T}' = \{q \trianglelefteq a \trianglerighteq r \mid q, r \in \mathcal{T}_{\mathsf{finrec}}(A) \wedge a \in A \wedge \mathsf{bc} \in IR(O_a)\}$. Then there exist an $A'_{dm} \subseteq \mathcal{A}$ and an $H' = (M', B, \mathcal{S}', \mathcal{O}', A', [\![\_]\!]') \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous-1, nrpl, nrps, A'_{dm})$ such that for all $p \in \mathcal{T}_{\mathsf{finrec}}(A) \setminus \mathcal{T}'$ there exists a $p' \in \mathcal{T}_{\mathsf{finrec}}(A')$ such that*

$$\{(S \restriction M_{data}, (p \bullet_H S) \restriction M_{data}) \mid S \in \mathcal{S}\} = \{(S' \restriction M_{data}, (p' \bullet_{H'} S') \restriction M_{data}) \mid S' \in \mathcal{S}'\} .$$

As another corollary of the proof of Theorem 1, we have that if the operating unit size is reduced to zero, it is still possible to transform each thread that can be applied to the original ISA into a number of threads that can each be applied to the ISA of which the operating unit size is reduced to zero and together yield the same state changes on the data memory.

Corollary 2. *Let $aw \geq 0$, $wl, ous, nrpl, nrps > 0$ and $A_{dm} \subseteq \mathcal{A}$, let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!]) \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, nrpl, nrps, A_{dm})$, and let $M_{data} = \{\mathsf{m}^{wl}_{\mathsf{data}}(i) \mid i \in [0, 2^{aw}-1]\}$ and $M_{ou} = \{\mathsf{m}_{\mathsf{ou}}(i) \mid i \in [0, ous-1]\}$. Then there exist an $A'_{dm} \subseteq \mathcal{A}$ and an $H' = (M', B, \mathcal{S}', \mathcal{O}', A', [\![\_]\!]') \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, 0, nrpl, nrps, A'_{dm})$ such that for all $p \in \mathcal{T}_{\mathsf{finrec}}(A)$ and $S_{ou} \in \{S \restriction M_{ou} \mid S \in \mathcal{S}\}$ there exists a $p' \in \mathcal{T}_{\mathsf{finrec}}(A')$ such that*

$$\{(S \restriction M_{data}, (p \bullet_H S) \restriction M_{data}) \mid S \in \mathcal{S} \wedge S \restriction M_{ou} = S_{ou}\} = \{(S' \restriction M_{data}, (p' \bullet_{H'} S') \restriction M_{data}) \mid S' \in \mathcal{S}'\} .$$

The cardinality of $\{S \restriction M_{ou} \mid S \in \mathcal{S}\}$ is 2^{ous}. Therefore, for each thread that can be applied to the original ISA, the number of threads that can together produce the same state changes on the data memory of the ISA of which the operating unit size is reduced to zero does not have to be more than 2^{ous}. Corollary 2 does not go through if the number of states of the new threads is bounded.

7 Thread Powered Function Classes

A simple calculation shows that, for a strict load/store Maurer ISA with address width aw and word length wl, the number of possible transformations on the states of the data memory is $2^{(2^{aw} \cdot wl + aw) \cdot wl)}$. This raises questions concerning the possibility to achieve all these state transformation by applying a thread to

a strict load/store Maurer ISA with this address width and word length. One of the questions is how this possibility depends on the operating unit size of the ISAs, the size of the instruction set of the ISAs, and the maximal number of states of the threads. This brings us to introduce the concept of a thread powered function class.

The concept of a thread powered function class is parametrized by:

- an address width aw;
- a word length wl;
- an operating unit size ous;
- an instruction set size iss;
- a state space bound ssb;
- a working area flag waf;

where $aw, ous \geq 0$, $wl, iss, ssb > 0$ and $waf \in \mathbb{B}$.

The instruction set size iss is the number of basic instructions, excluding load and store instructions. To simplify the setting, we consider only the case where there is one load instruction and one store instruction. The state space bound ssb is a bound on the number of states of the threads that can be applied. The working area flag waf indicates whether a part of the data memory is taken as a working area. A part of the data memory is taken as a working area if we are not interested in the state transformations with respect to that part. To simplify the setting, we always set aside half of the data memory for working area if a working area is in order.

Intuitively, the thread powered function class with parameters aw, wl, ous, iss, ssb and waf are the transformations on the states of the data memory or the first half of the data memory, depending on waf, that can be achieved by applying threads with not more than ssb states to a strict load/store Maurer ISA of which the address width is aw, the word length is wl, the operating unit size is ous, the number of register pairs for load instructions is 1, the number of register pairs for store instructions is 1, and the cardinality of the set of instructions for data manipulation is iss. Henceforth, we will use the term *external memory* for the data memory if $waf = \mathsf{F}$ and for the first half of the data memory if $waf = \mathsf{T}$. Moreover, if $waf = \mathsf{T}$, we will use the term *internal memory* for the second half of the data memory.

For $aw \geq 0$ and $wl > 0$, we define $\mathsf{M}_{\mathsf{data}}^{aw,wl}$, $\mathsf{B}_{\mathsf{data}}^{wl}$, and $\mathsf{S}_{\mathsf{data}}^{aw,wl}$ as follows:

$$\begin{aligned}
\mathsf{M}_{\mathsf{data}}^{aw,wl} &= \{\mathsf{m}_{\mathsf{data}}^{wl}(i) \mid i \in [0, 2^{aw} - 1]\} \,, \\
\mathsf{B}_{\mathsf{data}}^{wl} &= [0, 2^{wl} - 1] \,, \\
\mathsf{S}_{\mathsf{data}}^{aw,wl} &= \{S \mid S : \mathsf{M}_{\mathsf{data}}^{aw,wl} \to \mathsf{B}_{\mathsf{data}}^{wl}\} \,, \\
\mathsf{T}_{\mathsf{data}}^{aw,wl} &= \{T \mid T : \mathsf{S}_{\mathsf{data}}^{aw,wl} \to \mathsf{S}_{\mathsf{data}}^{aw,wl}\} \,.
\end{aligned}$$

Let $aw, ous \geq 0$ and $wl, nrpl, nrps > 0$. Then, for all $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!]) \in \bigcup_{A_{dm} \subseteq \mathcal{A}} \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, nrpl, nrps, A_{dm})$, we have $\mathsf{M}_{\mathsf{data}}^{aw,wl} \subseteq M$, $\mathsf{B}_{\mathsf{data}}^{aw,wl} \subseteq B$, and $\mathsf{S}_{\mathsf{data}}^{aw,wl} = \{S \restriction \mathsf{M}_{\mathsf{data}}^{aw,wl} \mid S \in \mathcal{S}\}$.

Let $aw, ous \geq 0$ and $wl, iss, ssb > 0$, and let $waf \in \mathbb{B}$ be such that $waf = \mathsf{F}$ if $aw = 0$. Then the *thread powered function class* with parameters aw, wl, ous, iss, ssb and waf, written $\mathcal{TPFC}(aw, wl, ous, iss, ssb, waf)$, is the subset of $\mathsf{T}_{\mathsf{data}}^{aw,wl}$ that is defined as follows:

$$
\begin{aligned}
&T \in \mathcal{TPFC}(aw, wl, ous, iss, ssb, waf) \\
&\quad \Leftrightarrow \exists A_{dm} \subseteq \mathcal{A} \bullet \\
&\qquad \exists H \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, 1, 1, A_{dm}) \bullet \\
&\qquad\quad \exists p \in \mathcal{T}_{\mathsf{finrec}}(A_H) \bullet \\
&\qquad\qquad \big(card(A_{dm}) = iss \wedge card(Res(p)) \leq ssb \wedge \\
&\qquad\qquad\ \forall S \in \mathcal{S}_H \bullet \\
&\qquad\qquad\quad \big(\big(waf = \mathsf{F} \Rightarrow T(S \restriction \mathsf{M}_{\mathsf{data}}^{aw,wl}) = (p \bullet_H S) \restriction \mathsf{M}_{\mathsf{data}}^{aw,wl}\big) \wedge \\
&\qquad\qquad\quad\ \big(waf = \mathsf{T} \Rightarrow \\
&\qquad\qquad\quad\quad T(S \restriction \mathsf{M}_{\mathsf{data}}^{aw,wl}) \restriction \mathsf{M}_{\mathsf{data}}^{aw-1,wl} = (p \bullet_H S) \restriction \mathsf{M}_{\mathsf{data}}^{aw-1,wl}\big)\big)\big) \,.
\end{aligned}
$$

We say that $\mathcal{TPFC}(aw, wl, ous, iss, ssb, waf)$ is *complete* if $\mathcal{TPFC}(aw, wl, ous, iss, ssb, waf) = \mathsf{T}_{\mathsf{data}}^{aw,wl}$.

The following theorem states that $\mathcal{TPFC}(aw, wl, ous, iss, ssb, waf)$ is complete if $ous = 2^{aw} \cdot wl + aw + 1$, $iss = 5$ and $ssb = 8$. Because $2^{aw} \cdot wl$ is the data memory size, i.e. the number of bits that the data memory contains, this means that completeness can be obtained with 5 data manipulation instructions and threads of which the number of states is less than or equal to 8 by taking the operating unit size slightly greater than the data memory size.

Theorem 2. *Let $aw \geq 0$, $wl > 0$ and $waf \in \mathbb{B}$, and let $dms = 2^{aw} \cdot wl$. Then $\mathcal{TPFC}(aw, wl, dms + aw + 1, 5, 8, waf)$ is complete.*

The idea behind the proof of Theorem 2 given below is that first the content of the whole data memory is copied data memory element by data memory element via the load data register to the operating unit, after that the intended state transformation is applied to the copy in the operating unit, and finally the result is copied back data memory element by data memory element via the store data register to the data memory. The data manipulation instructions used to accomplish this are an initialization instruction, a pre-load instruction, a post-load instruction, a pre-store instruction, and a transformation instruction. The pre-load instruction is used to update the load address register before a data memory element is loaded, the post-load instruction is used to store the content of the load data register to the operating unit after a data memory element has been loaded, and the pre-store instruction is used to update the store address register and to load the content of the store data register from the operating unit before a data memory element is stored. The transformation instruction is used to apply the intended state transformation to the copy in the operating unit.

Proof (of Theorem 2). For convenience, we define

$$
\begin{aligned}
M_{data} &= \{\mathsf{m}^{wl}_{\mathsf{data}}(i) \mid i \in [0, 2^{aw}-1]\}\ ,\\
M_{ou} &= \{\mathsf{m}_{\mathsf{ou}}(j) \mid j \in [0, dms+aw]\}\ ,\\
M^d_{ou} &= \{\mathsf{m}_{\mathsf{ou}}(j) \mid j \in [0, dms-1]\}\ ,\\
M^a_{ou} &= \{\mathsf{m}_{\mathsf{ou}}(j) \mid j \in [dms, dms+aw]\}\ ,\\
M^d_{ou}\langle i\rangle &= \{\mathsf{m}_{\mathsf{ou}}(j) \mid j \in [i\cdot wl, (i+1)\cdot wl-1]\}\ ,\ \text{for } i \in [0, 2^{aw}-1]\ ,\\
\mathsf{ldr} &= \mathsf{m}^{wl}_{\mathsf{ld}}(0)\ ,\\
\mathsf{sdr} &= \mathsf{m}^{wl}_{\mathsf{sd}}(0)\ ,\\
\mathsf{lar} &= \mathsf{m}^{aw}_{\mathsf{la}}(0)\ ,\\
\mathsf{sar} &= \mathsf{m}^{aw}_{\mathsf{sa}}(0)\ .
\end{aligned}
$$

We have that $M^d_{ou} = \bigcup_{i\in[0,2^{aw}-1]} M^d_{ou}\langle i\rangle$ and $M_{ou} = M^d_{ou} \cup M^a_{ou}$.

We have to deal with the binary representations of natural numbers in the operating unit. The set of possible binary representations of natural numbers in the operating unit is

$$\mathcal{R} = \bigcup_{n\in[0,dms+aw],m\in[n,dms+aw]} \{R \mid R : \{\mathsf{m}_{\mathsf{ou}}(i) \mid i \in [n,m]\} \to \{0,1\}\}\ .$$

For each $R \in \mathcal{R}$, the natural number of which R is a binary representation is given by the function $\nu : \mathcal{R} \to \mathbb{N}$ that is defined as follows:

$$\nu(R) = \textstyle\sum_{i \text{ s.t. } \mathsf{m}_{\mathsf{ou}}(i)\in\mathrm{dom}(R)} R(\mathsf{m}_{\mathsf{ou}}(i)) \cdot 2^{i-\min\{j\mid \mathsf{m}_{\mathsf{ou}}(j)\in\mathrm{dom}(R)\}}\ .$$

To prove the theorem, we take a fixed but arbitrary $T \in \mathsf{T}^{aw,wl}_{\mathsf{data}}$ and show that $T \in \mathcal{TPFC}(aw, wl, dms+aw+1, 5, 8, waf)$.

We take $A_{dm} = \{\mathsf{init}, \mathsf{preload}, \mathsf{postload}, \mathsf{prestore}, \mathsf{transform}\}$, and we take $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!]) \in \mathcal{MISA}_{\mathrm{sls}}(aw, wl, dms+aw+1, 1, 1, A_{dm})$ such that O_{init}, O_{preload}, O_{postload}, O_{prestore}, and $O_{\mathsf{transform}}$ are the unique functions from $\mathcal{S}$ to $\mathcal{S}$ such that for all $S \in \mathcal{S}$:

$$
\begin{aligned}
&O_{\mathsf{init}}(S) \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{rr}\})) = S \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{rr}\}))\ ,\\
&\nu(O_{\mathsf{init}}(S) \restriction M^a_{ou}) = 0\ ,\\
&O_{\mathsf{init}}(S)(\mathsf{rr}) = \mathsf{T}\ ,\\
&\\
&O_{\mathsf{preload}}(S) \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{lar}\} \cup \{\mathsf{rr}\}))\\
&\qquad = S \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{lar}\} \cup \{\mathsf{rr}\}))\ ,\\
&\nu(O_{\mathsf{preload}}(S) \restriction M^a_{ou}) = \nu(S \restriction M^a_{ou}) + 1 \text{ if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ ,\\
&O_{\mathsf{preload}}(S) \restriction M^a_{ou} = S \restriction M^a_{ou} \quad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ ,\\
&O_{\mathsf{preload}}(S)(\mathsf{lar}) = \nu(S \restriction M^a_{ou}) \quad \text{if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ ,\\
&O_{\mathsf{preload}}(S)(\mathsf{lar}) = S(\mathsf{lar}) \quad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ ,\\
&O_{\mathsf{preload}}(S)(\mathsf{rr}) = \mathsf{T} \quad \text{if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ ,\\
&O_{\mathsf{preload}}(S)(\mathsf{rr}) = \mathsf{F} \quad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ ,
\end{aligned}
$$

$$
\begin{aligned}
&O_{\mathsf{postload}}(S) \restriction (M \setminus (M^d_{ou}\langle \nu(S \restriction M^a_{ou})\rangle \cup \{\mathsf{rr}\})) \\
&\qquad\qquad = S \restriction (M \setminus (M^d_{ou}\langle \nu(S \restriction M^a_{ou})\rangle \cup \{\mathsf{rr}\}))\ , \\
&\nu(O_{\mathsf{postload}}(S) \restriction M^d_{ou}\langle \nu(S \restriction M^a_{ou})\rangle) = S(\mathsf{ldr})\ , \\
&O_{\mathsf{postload}}(S)(\mathsf{rr}) \qquad\qquad = \mathsf{T}\ ,
\end{aligned}
$$

$$
\begin{aligned}
&O_{\mathsf{prestore}}(S) \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{sar},\mathsf{sdr}\} \cup \{\mathsf{rr}\})) \\
&\qquad\qquad = S \restriction (M \setminus (M^a_{ou} \cup \{\mathsf{sar},\mathsf{sdr}\} \cup \{\mathsf{rr}\}))\ , \\
&\nu(O_{\mathsf{prestore}}(S) \restriction M^a_{ou}) = \nu(S \restriction M^a_{ou}) + 1 \qquad \text{if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S) \restriction M^a_{ou} \quad = S \restriction M^a_{ou} \qquad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{sar}) \quad = \nu(S \restriction M^a_{ou}) \qquad \text{if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{sar}) \quad = S(\mathsf{sar}) \qquad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{sdr}) \quad = \nu(S \restriction M^d_{ou}\langle \nu(S \restriction M^a_{ou})\rangle) \text{ if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{sdr}) \quad = S(\mathsf{sdr}) \qquad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{rr}) \quad = \mathsf{T} \qquad \text{if } \nu(S \restriction M^a_{ou}) < 2^{aw}\ , \\
&O_{\mathsf{prestore}}(S)(\mathsf{rr}) \quad = \mathsf{F} \qquad \text{if } \nu(S \restriction M^a_{ou}) \geq 2^{aw}\ ,
\end{aligned}
$$

$$
\begin{aligned}
&O_{\mathsf{transform}}(S) \restriction (M \setminus (M_{ou} \cup \{\mathsf{rr}\})) = S \restriction (M \setminus (M_{ou} \cup \{\mathsf{rr}\}))\ , \\
&O_{\mathsf{transform}}(S) \restriction M^d_{ou} \quad = T'(S \restriction M^d_{ou})\ , \\
&\nu(O_{\mathsf{transform}}(S) \restriction M^a_{ou}) = 0\ , \\
&O_{\mathsf{transform}}(S)(\mathsf{rr}) \quad = \mathsf{T}\ ,
\end{aligned}
$$

where T' is the unique function from $\{S \restriction M^d_{ou} \mid S \in \mathcal{S}\}$ to $\{S \restriction M^d_{ou} \mid S \in \mathcal{S}\}$ such that, for all $S^d_{ou} \in \{S \restriction M^d_{ou} \mid S \in \mathcal{S}\}$, there exists an $S_{data} \in \{S \restriction M_{data} \mid S \in \mathcal{S}\}$ such that:

$$
\begin{aligned}
&\forall i \in [0, 2^{aw} - 1] \bullet \\
&\quad (\nu(S^d_{ou} \restriction M^d_{ou}\langle i\rangle) = S_{data}(M_{data}[i]) \wedge \\
&\quad\ \nu(T'(S^d_{ou}) \restriction M^d_{ou}\langle i\rangle) = T(S_{data})(M_{data}[i]))\ .
\end{aligned}
$$

Moreover, we take $p \in \mathcal{T}_{\mathsf{finrec}}(A)$ such that $p = \langle X|E\rangle$ with E consisting of the following equations:

$$
\begin{aligned}
&X = \mathsf{init} \circ Y\ , \\
&Y = (\mathsf{load{:}0} \circ \mathsf{postload} \circ Y) \trianglelefteq \mathsf{preload} \trianglerighteq (\mathsf{transform} \circ Z)\ , \\
&Z = (\mathsf{store{:}0} \circ Z) \trianglelefteq \mathsf{prestore} \trianglerighteq \mathsf{S}\ .
\end{aligned}
$$

Let $S \in \mathcal{S}$ and let (p_i, S_i) be the $(i{+}1)$st element in the full path of $(\langle X|E\rangle, S)$ on H of which the first component equals $\langle X|E\rangle$, $\langle Y|E\rangle$, $\langle Z|E\rangle$ or S. Then it is easy to prove by induction on i that

$$
\begin{aligned}
&i = 1 \Rightarrow p_i = \langle Y|E\rangle \wedge \nu(S_i \restriction M^a_{ou}) = 0\ ,\\
&i \in [2, 2^{aw}+2] \Rightarrow\\
&\quad p_i = \langle Y|E\rangle \wedge\\
&\quad \forall j \in [0, i-2] \bullet \nu(S_i \restriction M^d_{ou}\langle j\rangle) = S(M_{data}[j]) \wedge\\
&\quad \nu(S_i \restriction M^a_{ou}) = i-1\ ,\\
&i = 2^{aw}+3 \Rightarrow\\
&\quad p_i = \langle Z|E\rangle \wedge\\
&\quad \forall j \in [0, 2^{aw}-1] \bullet \nu(S_i \restriction M^d_{ou}\langle j\rangle) = T(S \restriction M_{data})(M_{data}[j]) \wedge\\
&\quad \nu(S_i \restriction M^a_{ou}) = 0\ ,\\
&i \in [2^{aw}+4, 2^{aw+1}+4] \Rightarrow\\
&\quad p_i = \langle Z|E\rangle \wedge\\
&\quad \forall j \in [0, i-(2^{aw}+4)] \bullet S_i(M_{data}[j]) = T(S \restriction M_{data})(M_{data}[j]) \wedge\\
&\quad \nu(S_i \restriction M^a_{ou}) = i-(2^{aw}+3)\ ,\\
&i = 2^{aw+1}+5 \Rightarrow p_i = \mathsf{S} \wedge S_i \restriction M_{data} = T(S \restriction M_{data})\ .
\end{aligned}
$$

Hence, $(\langle X|E\rangle \bullet_H S) \restriction M_{data} = T(S \restriction M_{data})$. That is, T can be achieved by applying $\langle X|E\rangle$ to H. □

As a corollary of the proof of Theorem 2, we have that in the case where $waf = \mathsf{T}$ completeness can also be obtained if we take about half the external memory size as the operating unit size.

Corollary 3. *Let $aw > 0$ and $wl > 0$, and let $ems = 2^{aw-1} \cdot wl$. Then $\mathcal{TPFC}(aw, wl, ems + aw, 5, 8, \mathsf{T})$ is complete.*

As a corollary of the proofs of Theorems 1 and 2, we have that completeness can even be obtained if we take zero as the operating unit size. However, this may require quite a large number of data manipulation instructions and threads with quite a large number of states.

Corollary 4. *Let $aw \geq 0$ and $wl > 0$, let $waf \in \mathbb{B}$ be such that $waf = \mathsf{F}$ if $aw = 0$, and let $dms = 2^{aw} \cdot wl$. Then $\mathcal{TPFC}(aw, wl, 0, 5{\cdot}4^{dms+aw+1}, 8{\cdot}6^{dms+aw+1}, waf)$ is complete.*

8 On Incomplete Thread Powered Function Classes

From Corollary 4, we know that it is possible to achieve all transformations on the states of the external memory of a strict load/store Maurer ISA with given address width and word length even if the operating unit size is zero. However, this may require quite a large number of data manipulation instructions and threads with quite a large number of states. This raises the question whether the operating unit size of the ISAs, the size of the instructions set of the ISAs

and the maximal number of states of the threads can be taken such that it is impossible to achieve all transformations on the states of the external memory.

Below, we will give a theorem concerning this question, but first we give a lemma that will be used in the proof of that theorem.

Lemma 1. *Let* $aw > 1$ *and* $wl, ous, iss, ssb > 0$*, and let* $ems = 2^{aw-1} \cdot wl$*. Then* $\mathcal{TPFC}(aw, wl, ous, iss, ssb, \mathsf{T})$ *is not complete if* $ous \leq ems/2$ *and* $iss \leq 2^{ems/2}$ *and there are no more than* 2^{ems} *threads that can be applied to the members of* $\bigcup_{A_{dm} \subseteq \mathcal{A}} \mathcal{MISA}_{\mathrm{sls}}(aw, wl, ous, 1, 1, A_{dm})$*.*

Proof. We have that, if the operating unit size is no more than $ems/2$, no more than

$$(2^{ems/2})^{(2^{ems/2})}$$

transformations on the states of the operating unit can be associated with one data manipulation instruction. It follows that, if there are no more than $2^{ems/2}$ data manipulation instructions, no more than

$$\left((2^{ems/2})^{(2^{ems/2})}\right)^{(2^{ems/2})}$$

transformations on the states of the external memory can be achieved with one thread. Hence, if no more than 2^{ems} threads can be applied, no more than

$$\left((2^{ems/2})^{(2^{ems/2})}\right)^{(2^{ems/2})} \cdot 2^{ems}$$

transformations on the states of the external memory can be achieved. Using elementary arithmetic, we easily establish that

$$\left((2^{ems/2})^{(2^{ems/2})}\right)^{(2^{ems/2})} \cdot 2^{ems} < (2^{ems})^{(2^{ems})} .$$

It follows that $\mathcal{TPFC}(aw, wl, ous, iss, ssb, \mathsf{T})$ is not complete because $(2^{ems})^{(2^{ems})}$ is the number of transformations that are possible on the states of the external memory. □

In Lemma 1, the bound on the number of threads that can be applied does not appear out of the blue. It is the number of threads that can at most be represented in the internal memory: with the most efficient representations we cannot have more than one thread per state of the internal memory.

The following theorem states that $\mathcal{TPFC}(aw, wl, ous, iss, ssb, \mathsf{T})$ is not complete if the operating unit size is not greater than half the external memory size, the instruction set size is not greater than $2^{wl} - 4$, and the maximal number of states of the threads is not greater than 2^{aw-2}. Notice that 2^{wl} is the number of instructions that can be represented in memory elements with word length wl.

Theorem 3. *Let* $aw, wl > 1$ *and* $ous, iss, ssb > 0$*, and let* $ems = 2^{aw-1} \cdot wl$*. Then* $\mathcal{TPFC}(aw, wl, ous, iss, ssb, \mathsf{T})$ *is not complete if* $ous \leq ems/2$ *and* $iss \leq 2^{wl} - 4$ *and* $ssb \leq 2^{aw-2}$*.*

Proof. A transformation on the states of the external memory cannot be achieved by a thread with at most ssb states if it cannot be achieved by a thread with exactly ssb states using one additional instruction. This is easy to see considering that the identity transformation on the states of the entire memory can be associated with the additional instruction. It follows that it is sufficient to show that $\mathcal{TPFC}(aw, wl, ous, iss + 1, ssb, \mathsf{T})$ is not complete if only the threads with ssb states can be applied.

If there is one additional instruction, the number of threads with ssb states is

$$\big((iss + 3) \cdot ssb^2 + 2\big)^{ssb}$$

(recall that there is also one load instruction and one store instruction). Because $ssb > 0$,

$$\big((iss + 3) \cdot ssb^2 + 2\big)^{ssb} \;< \big((iss + 4) \cdot ssb^2\big)^{ssb} \;.$$

Using elementary arithmetic, we easily establish that

$$\big((iss + 4) \cdot ssb^2\big)^{ssb} < 2^{ems} \;.$$

Consequently, the number of threads with ssb states is less than 2^{ems}. From this, and the facts that $ous \leq ems/2$ and $iss < 2^{ems/2}$, it follows by Lemma 1 that $\mathcal{TPFC}(aw, wl, ous, iss + 1, ssb, \mathsf{T})$ is not complete if only the threads with ssb states can be applied. □

9 Conclusions

In [4, 5], we have worked at a formal approach to micro-architecture design based on Maurer machines and basic thread algebra. In those papers, we made hardly any assumption about the instruction set architectures for which new micro-architectures are designed, but we put forward strict load/store Maurer instruction set architectures as preferable instruction set architectures. In the current paper, we have established general properties of strict load/store Maurer instruction set architectures. Some of these properties are presumably non-trivial insights among practitioners involved in the design of instruction set architectures. At the least, they clarify in some degree existing trends in the design of load/store instruction set architectures, such as the ever increasing operating unit size. We believe that the work presented in this paper may grow into a theoretical basis for the design of instruction set architectures.

One of the options for future work is to improve upon the results given in this paper. For example, we know from Theorem 2 that, in order to obtain completeness with 5 data manipulation instructions and threads of which the number of states is less than or equal to 8, it is sufficient to take the operating unit size slightly greater than the data memory size. However, we do not yet know what the smallest operating unit size is that will do. Another option for future work is to establish results that bear upon the use of half the data memory

as internal memory. No such results are given in this paper. In Lemma 1, it is assumed that half the data memory is used as internal memory because it provides a good case for the number taken as the maximal number of threads that can be applied. However, Lemma 1, as well as Theorem 3, goes through without internal memory.

The speed with which transformations on the states of the external memory are achieved depends largely upon the way in which the strict load/store Maurer instruction set architecture in question is implemented. This hampers establishing general results about it. However, the speed with which transformations on the states of the external memory are achieved depends also on the volume of data transfer needed between the external memory and the operating unit. Establishing a connection between this volume and the parameters of thread powered function classes is still another option for future work.

In this paper, we have taken the view that transformations on the states of the external memory are achieved by applying threads. We could have taken the less abstract view that transformations on the states of the external memory are achieved by running stored programs. This would have led to needless complications: only the threads that are represented by those programs are relevant to the transformations on the states of the external memory that are achieved.

References

1. J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.
2. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
3. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
4. J. A. Bergstra and C. A. Middelburg. Maurer computers with single-thread control. To appear in *Fundamenta Informaticae*. Preliminary version: Computer Science Report 05-17, Department of Mathematics and Computer Science, Eindhoven University of Technology.
5. J. A. Bergstra and C. A. Middelburg. Maurer computers for pipelined instruction processing. To appear in *Mathematical Structures in Computer Science*. Preliminary version: Computer Science Report 06-12, Department of Mathematics and Computer Science, Eindhoven University of Technology.
6. J. A. Bergstra and C. A. Middelburg. Splitting bisimulations and retrospective conditions. *Information and Computation*, 204(7):1083–1138, 2006.
7. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
8. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

10. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, second edition, 2001.
11. W. D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226–235, 1966.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989.
13. J. Thornton. *Design of a Computer – The Control Data 6600*. Scott, Foresman and Co., Glenview, IL, 1970.

Within this series the following reports appeared.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures*, Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory*, Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital*, Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation*, Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads*, Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading*, Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises*, Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields*, Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops*, Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration)*, Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction*, Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures*, Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF*, Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics*, Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA*, Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads*, Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result*, Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services*, Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving*, Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot*, Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax*, Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives*, Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA*, Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/