A Formal Semantics of Exposed Datapath Architectures with Buffered Processing Units

Markus Anders and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—Synchronous control asynchronous dataflow (SCAD) architectures are a new kind of exposed datapath processor architectures which reveal all of their processing units (PU) and datapaths to the compiler to increase the use of instruction-level parallelism (ILP). They employ FIFO buffers at the input and output ports of their PUs, consuming operands and producing results similar to dataflow computers. Because the arrival of operands triggers the execution of instructions, SCAD architectures only need a single type of instruction that transports data from output buffers to input buffers. Among other advantages, this allows a simple integration of application-specific PUs, since a PU may implement in principle any function that can be realized with reasonable hardware without changing the instruction set. However, simply allowing any computable function for PUs introduces problems: Because of the asynchronous nature of data moves and operation execution, not all functions yield scheduling-independent, deterministic results. In this paper, we therefore first describe an operational semantics for SCAD architectures which enables formal reasoning about the underlying model of computation. Then, we create an equivalent model using well-known dataflow process networks. From this model, the desired restrictions for functions can then be derived, giving a configurable and sound class of SCAD architectures. Finally, we discuss further restrictions that should be applied and propose a notion for well-structured SCAD machine code.

Keywords—exposed data path architectures; instruction-level parallelism, dataflow process networks

I. INTRODUCTION

A. Motivation

Essentially all conventional processor architectures today are register architectures, i.e., memory is only accessed by load and store instructions, and all other operations read their operands from registers and store their results also in registers. The number of available registers therefore limits the number of operations that can be executed in parallel. Increasing the number of registers is however quite difficult: In particular, the chip size of the central register file grows quadratically with the number of processing units (PU) and registers while the access time of registers increases accordingly, which compromises the main advantage of registers. Recent years have therefore seen the appearance of exposed datapath architectures [1]–[4]. They expose all of their PUs and datapaths to the compiler so that the compilers can avoid the use of the central register file by forwarding intermediate results directly from a producer PU to a consumer PU. In many cases, this leads to an increase of the available instruction-level parallelism (ILP).

Transport triggered architectures (TTA) [4] are a well-known class of exposed datapath architectures utilizing only a single instruction. This instruction moves values from the outputs of one PU of the processor to the inputs of another or the same PU. The arrival of data then triggers the operations of the PUs and the produced results can be moved later on by further move instructions. Since values can be transported directly from PUs to other PUs, the compiler can often avoid the use of a central register file. This approach also leads to a great flexibility without having the need to change the instruction set. TTAs utilize static scheduling to enjoy a better power efficiency and a predictable execution time analysis compared to dynamically scheduled architectures. Synchronous control asynchronous dataflow (SCAD) architectures modify TTAs by replacing registers at input/output ports of PUs by FIFO buffers. This has the following advantages: First, PUs may have arbitrary delay times and can implement a variety of functions without fundamentally changing the instruction set or compilation tools. The FIFO buffers abstract from precise execution times and allow unforeseen exceptions in the PUs. This also enables late design changes that may map a function to a special PU instead to software running on standard PUs. Therefore, we generally assume that the compiler does not need to know the times required for computation and communication for scheduling instructions to the available PUs. Second, worst-case execution time (WCET) analysis will be comparatively simple even when a lot of ILP is exploited [5]. This is due to ILP being utilized explicitly in SCAD programs without the need of dynamically scheduled execution. Furthermore, in [5] it was shown that this approach enables the utilization of a substantial number of ILP. Third, memory calls can be greatly reduced when using the same number of wiring as register architectures, albeit for a severe runtime penalty [6]. This is still desirable, since caches are known to be hardly predictable for real-time systems. Reducing the total number of memory calls can therefore have a tremendous effect on the WCET. Furthermore, good compilation has been shown to reduce or even eliminate the runtime penalty [5], [7].

![Fig. 1. A SCAD PU with its firing rules (a) may produce different results (b) depending on the schedule.](image-url)
However, the development of reliable systems on application-specific SCAD architectures requires more than the previously mentioned techniques. In particular, formal verification is a common requirement in safety critical environments. However, if PUs may implement any computable function, we cannot even guarantee a deterministic execution: For example, Figure 1 shows that the given firing rules to copy one of the inputs to the output may obviously lead to different results depending on the specific run.

Furthermore, the correctness of machine programs – even if they are the result of an automated model-based design flow where intermediate models are verified – may be required to be proven in order to avoid bugs introduced by compilers. The translation of programs to SCAD machine code may not be trusted, in particular, since SCAD machine code turned out to be very difficult to read and to reason about. Even worse, SCAD programs are not by definition bounded in space within the processor – which is trivially the case for conventional register-based machines, since the number of registers is hard-coded into the instruction set. A formal verification methodology for SCAD architectures will therefore also depend on better understanding the underlying theoretical model of computation.

B. Main Contributions

The contributions of this paper include:

- The definition of operational semantics for SCAD architectures, enabling formal reasoning about the underlying model of computation.
- A bisimilar dataflow process network (DPN) model for every SCAD architecture with minor restrictions. Using the semantics of DPNs, it can thus be treated as an alternative formal semantics of SCAD architectures.
- We prove that SCAD architectures restricted to Kahn PUs – as known from DPNs – produce scheduling-independent results, thus allowing compilers to ignore the execution times of PUs.
- Furthermore, a proposal for well-structured SCAD machine code is made, that has properties comparable to conventional register-based machine code. This structure will guarantee that programs have bounded space usage within the processor.

C. Outline

First, we will discuss related work in Section II. Then, necessary preliminaries for DPNs and SCAD architectures are given in Section III. Section IV will deal with the definition of SCAD operational semantics. In Section V, a DPN model will be given that we will then prove to be bisimilar to the previously defined definition. Finally, we will make the observation that more restrictions are necessary to generate machine code that is more akin to conventional register-based machine code in Section VI.

II. RELATED WORK

There is an abundance of hybrid dataflow/von-Neumann architectures [1] and more specifically exposed datapath architectures [2]–[4] that share characteristics with SCAD architectures. There is however not much work regarding formal semantics of these architectures. In [8], the semantics of the Synchronous Transfer Architecture (STA) is discussed, aimed towards formal verification applications. SCAD architectures share some similarities with STA [9]. STA employs buffers at the outputs of its processing units and utilizes static scheduling similar to VLIW architectures. However, as STA maintains strictly deterministic system behaviour [8], these results are not applicable here.

The goal of this paper is to identify criteria to guarantee that results of a SCAD program are independent of scheduling choices in SCAD architectures. The motivation of using formal semantics to provide a framework for formal verification in [8] does however carry over to our discussion. The described issue is not really rooted in processor architecture research, but more akin to the Kahn principle of DPNs [10], [11]. As we will show, the model of computation of SCAD architectures is similar enough to DPNs such that those results carry over to SCAD. DPNs restricted to Kahn criteria as stated by [11] are known to produce scheduling-independent results, which we will exploit in this paper. Meanwhile, even more results are known: isochronous, endochronous [12] and weakly endochronous DPNs [13] have relaxed notions that still produce deterministic results. We will however not use these more advanced notions in this paper, even though they do actually apply as well.

III. PRELIMINARIES

A. Dataflow Process Networks

DPNs as first described in [11] consist of processes that exclusively communicate via FIFO-buffers connecting them. The descriptions used are inspired by [11], [14]. The definition of operational semantics for SCAD architectures will also make direct use of DPN operational semantics: Nodes read values from their incoming FIFO buffers and write to their outgoing FIFO buffers. A priori, there are no restrictions on functions implemented by the nodes. As stated in [11], nodes can be thought of as Turing machines.

The abbreviation Stream$_D^m := (D^*)^m$ is used to denote a stream, where $m$ is the number of buffers (e.g. Stream$_D^n \rightarrow$ Stream$_D^n$ denotes a function of a DPN node consuming from $m$ buffers and producing to $n$ buffers). The content of buffers will be described using a Stream$_D^n$. The state $S$ of a DPN is simply a mapping from buffers to a respective Stream$_D^n$, i.e., if a DPN has a set of buffers $Z$ over data type $D$, then $S$ has type $Z \rightarrow$ Stream$_D^n$. We will not consider any internal computational states of nodes and only refer to state changes according to the transformation of buffers. In particular, we will assume that the firing of a node happens instantaneously – consuming and producing all necessary values in one step – meaning that nodes do not need any internal state.

DPNs generally do not produce deterministic (scheduling-independent) results. We will later imply the following restrictions to firing rules due to [11]:

**Definition 1.** Kahn criteria.

1) Firing rules do not check for emptiness of a buffer.
2) Only blocking reads of buffers are allowed.
3) Process nodes must implement sequential functions.
As shown in [11], we can then state the following:

**Lemma 1.** If all process nodes of a DPN fulfill the Kahn criteria, its results are scheduling-independent and latency-insensitive.

### B. SCAD Architectures

SCAD architectures, as defined in [5], [6], [15], are mainly a set of PUs connected by the move instruction bus (MIB) and the data transfer network (DTN). The PUs have a number of input ports and output ports. At each port, there is a FIFO buffer split into an address part (adr) and a value part (val).

The adr part contains addresses of other ports, while the val part contains actual values. Data is always sent from output buffers to input buffers. A data move is initiated by a corresponding move instruction src → tgt in the machine program. The meaning is that one value should be sent from output buffer src to input buffer tgt. Decoding instructions and keeping a program counter (pc) is the job of the control unit (CU). After the CU decodes src → tgt, it will send the according information to the PUs via the MIB. The PUs snoop the MIB for new instructions. If output buffer src snoops instruction src → tgt, the target address of the instruction (tgt) will be pushed into the first free (empty or ⊥) adr entry of the output buffer. If no value in the corresponding val entry is already present, it will be set to ⊥, denoting absence. In the input buffer tgt, the source address of the instruction (src) is pushed into the first empty slot, setting its corresponding val entry to ⊥. Note that adding src and tgt to the respective adr cells always happens synchronously. Only then, the pc is increased and the next instruction will be fetched. Once (tgt, x) (where x ≠ ⊥ is some value and tgt is the target address we just added) is at the head of output buffer src, the data is actually sent via the DTN. (tgt, x) is popped from src, while (src, x) replaces (src, ⊥) closest to the head of tgt (which is not a strict queue operation, as we will demonstrate in the following). Firing of the operations inside the PUs happens once enough values are present. The corresponding values (and their addresses) are consumed from the inputs, and the outputs are produced to the output buffers. When producing values, free entries in the val part closest to the head of output buffers are overwritten. Formal operational semantics will be provided in Section IV. Therefore, instead of describing it in more detail informally, we now show an example of how SCAD architectures work in action.

We will consider a SCAD architecture with 2 PUs and input/output ports {a,...,f}, as shown in Figure 2. The red cells indicate the adr part of buffers while the green cells indicate the val part. The initial configuration contains 4 and 5 in the val part of output buffer a. The PUs will simply perform addition. The move program to be executed is the following:

- a → e;
- a → f;
- a → b;
- d → b;

A possible run of this program goes as follows: the CU first broadcasts a → e, a → f and a → b on the MIB. The corresponding buffers add their respective counterpart of the move operation to their adr buffer, which results in the first configuration (1) shown in Figure 2. Then, some actual data transports via the DTN may take place: (e, 4) is at the head of output buffer a, which means that value 4 can be sent to input buffer e. Next, (f, 5) is at the head, which is also sent. Then, (b, ⊥) has no value yet, which means that this move operation can not yet be performed. The CU broadcasts the last operation of our program, d → b. The resulting configuration is the second one (2) in Figure 2. Now a PU can fire, since the second PU finds both operands necessary to perform an addition: (a, 4) and (a, 5) are consumed, and 4 + 5 = 9 is produced to the first free cell of the output buffer. The result is shown in the third configuration (3). Finally, (b, 9) is now ready in output buffer d. Value 9 is therefore sent to input buffer b. It replaces (d, ⊥) closest to the head of the buffer, which is indicated in the last configuration (4). No further rule can be applied now and the program terminates.

### IV. Operational Semantics of SCAD Architectures

In this section, we will first provide formal descriptions for the components of SCAD architectures. Then, based on these descriptions, operational semantics will be given.

#### A. Formal Description

A SCAD architecture is a triple $\mathcal{A} = (D, P, F)$, consisting of a data type $D$, ports $P$ and PUs $F$. The data type $D$ always contains a special symbol $\bot \in D$ denoting the absence of a value. $P \subseteq ((i, o) \times N) \cup \{c\}$ are the available input and output ports. Note that while port names could be chosen freely (as done in Figure 2), we will stick to this convention in our discussions concerning the theoretical model of computation. All SCAD architectures have a finite number of ports $P$. Input ports are defined as $\text{in}(P) = P \cap ((i) \times N)$ and output ports as $\text{out}(P) = P \cap ((o) \times N)$, $c$ denotes a special port reserved...
for the CU. It is always assumed that $c \in P$, $F$ is the finite set of PUs. Every PU $(P', f) \in F$ also has a subset of input ports $in(P') \subseteq in(P)$, output ports $out(P') \subseteq out(P)$ and an associated set of firing rules $f$ just as used by DPNs, consuming values from a Stream$_{in}(P')$ and producing values to Stream$_{out}(P')$. The streams ought to be those associated with the input and output ports. As the special port $c$ is part of the CU and not of any PU, $c \notin P'$ must hold. The ports of two PUs $p, q \in F$ are mutually exclusive, meaning $P(p) \cap P(q) = \emptyset$. For simplicity, we demand that all input and output ports are connected to a PU.

B. Operational Semantics

To define operational semantics, a notion of state is introduced. A configuration $C = (\mathcal{P}, pc, st, A, V)$ of a SCAD architecture must contain the contents of the $adr$ and $val$ buffers of all ports $P$ as described in Section III-B. Thus, for all ports the function $A : P \rightarrow \text{Stream}_{p}$ with $P_\perp = P \cup \{ \perp \}$ maps to a sequence of port addresses $P$ and the special symbol $\perp$ denoting absence of a value. The same holds for values $V : P \rightarrow \text{Stream}_{p}$ which maps to a sequence of the data type $D$ (note that $\perp \in D$ holds by definition). $\mathcal{P}$ is the program currently in execution and $pc \in \mathbb{N}$ is the current program counter. The st $\in \mathbb{B}$ flag indicates whether the CU is currently stalling or not. A program $\mathcal{P}$ consists of move instructions $l : src \rightarrow tgt$ where $l \in \mathbb{N}$, $src \in out(P)$ and $tgt \in in(P) \cup \{ \perp \}$. The labels $l$ of instructions must be mutually exclusive. If $(l_1 : src_1 \rightarrow tgt_1), (l_2 : src_2 \rightarrow tgt_2) \in \mathcal{P}$ and $l_1 = l_2$, then $src_1 = src_2$ and $tgt_1 = tgt_2$ must hold. Therefore, the labels of two distinct instructions always differ. $src \rightarrow tgt$ will transport a value from output port $src$ to input port $tgt$, $c$ is the special input port of the control unit. If the control unit is not stalling which is true if $st = 0$ holds – it always fetches instructions according to the label $l$ and the state of the program counter $pc$. If a value is sent to $c$, it is used as the new value of the pc. As long as the control unit waits for this value, it is stalling, meaning $st = 1$.

The following definition of operational semantics follows closely the descriptions in [5], [15]. A step $C \rightarrow C'$ in $S$ can be made if a new move instruction is read from $\mathcal{P}$ and added to the address buffers (Rules 1, 2), a new value for the program counter $pc$ is set and the CU is not stalled (Rule 3), a move between PUs is performed (Rule 4) or an enabled firing rule of a PU is used (Rule 5). In order to simplify the formal construction of these rules, we will first define some notations and functions. First, $head : \text{Stream}_{X} \rightarrow X$ and $\text{tail} : \text{Stream}_{X} \rightarrow \text{Stream}_{X}$ for any $X$ with the expected behaviour are defined:

**Definition 2. Head and tail functions.**

$$\text{head}(S) := \begin{cases} x & \text{for } S = (x : S') \\ \perp & \text{for } S = \epsilon \end{cases}$$

$$\text{tail}(S) := \begin{cases} S' & \text{for } S = (x : S') \\ \epsilon & \text{for } S = \epsilon \end{cases}$$

Furthermore, as a lot of substitutions will be necessary on the buffers of configurations when data is read or written, some abbreviations that can be used to transform the mappings $A$ and $V$ of the configurations should be defined. In the following, $M \in \{ A, V \}$ can be assumed for all definitions. $M[p' \rightarrow S]$ will replace the content of the stream connected to port $p'$ in $M$ with stream $S$.

**Definition 3. Substitution on configuration mappings.**

$$M[p' \rightarrow S](p) := \begin{cases} M(p) & \text{for } p \neq p' \\ S & \text{for } p = p' \end{cases}$$

We may write $M[p_1 \rightarrow S_1, p_2 \rightarrow S_2]$ for $(M[p_1 \rightarrow S_1])[p_2 \rightarrow S_2]$. Next, the substitution of Definition 3 can be used to define queue operations directly on the configuration mappings. $M[p \leftarrow x]$ enqueues $x$ in the stream of port $p$. $M[p \rightarrow]$ dequeues one value from the stream of $p$.

**Definition 4. Triangle notation.**

$$M[p \leftarrow x] := M[p \rightarrow M(p), x]$$

$$M[p \rightarrow] := M[p \rightarrow \text{tail}(M(p))]$$

Again, we may directly chain the operations, allowing $M[p_1 \leftarrow x_1, p_2 \leftarrow x_2]$ for $(M[p_1 \leftarrow x_1])[p_2 \leftarrow x_2]$ (or using $\circ$ instead). Finally, $\text{first}(X, x, Y, y)$ will find the first position where both stream $X$ is $x$ and stream $Y$ is $y$. $S[i \rightarrow x]$ replaces position $i$ with value $x$ in stream $S$. These will be used to define the execution of data moves. Unfortunately, this definition is a bit cumbersome as a specific position inside the input buffer must be replaced. The $\text{first}$ function is necessary to find the first position inside an input buffer where the value is $x$ and the address is from the source output buffer. The substitution will then be used to replace that position of the stream.

**Definition 5. First function and substitution on streams.**

$$\text{first}(X, x, y, y) := i \text{ with } X_i = x, Y_i = y \text{ and } \forall j : X_j = x \land Y_j = y \rightarrow j \geq i.$$ 

$$S[i \rightarrow x](j) := \begin{cases} S_j & \text{for } i \neq j \\ x & \text{for } i = j \end{cases}$$

Using these definitions, the operational firing rules can now be given.

**Rule 1. ADVANCE.** If the CU is not stalling (meaning $st = 0$), try to read instruction $pc : src \rightarrow tgt$. If it exists, enqueue $tgt$ in $A'(src)$ and $src$ in $A'(tgt)$. Then, increment the program counter $pc' = pc + 1$ if $tgt \neq \epsilon$. Additionally, enqueue $\perp$ in $\mathcal{V}'(tgt)$:

$$pc : src \rightarrow tgt \in \mathcal{P}$$

$$\mathcal{V}'(tgt) := (\mathcal{V}, pc', 0, A', V')$$

with

$$A' = A[\mathcal{src} \rightarrow tgt, tgt \mathcal{src}], \mathcal{V}' = V'[\mathcal{src} \rightarrow \perp]$$

Note that $\perp$ is not enqueued in the $src$ output buffer, only the value part of an output buffer acts as a FIFO-buffer for the results of the PU. Therefore, $\perp$ values are always replaced in production order. In a way, the $\text{head}$ function – which returns $\perp$ on the empty stream – simulates the correct behaviour.

**Rule 2.STALL.** This is almost the same case as Rule 1, but $tgt = \epsilon$. In this case, do not increment $pc$ and stall further.
instructions with \( st' = 1 \) to wait for the availability of the value:

\[
\text{pc} : \text{src} \rightarrow \epsilon \in \mathcal{P}
\]

with

\[
A' = A[\text{src} \triangleleft \epsilon, \epsilon \triangleleft \text{src}], V' = V[\epsilon \triangleleft \bot]
\]

Rule 4, CONTINUE. If the CU is stalling \( (st = 1) \), but a value \( v = \text{head}(V(\epsilon)) \neq \bot \) is available, consume it and set \( \text{pc}' = v \).

Set \( st' = 0 \) to un stall the CU and also to deque que the value and address from \( \epsilon \):

\[
\text{pc}' = \text{head}(V(\epsilon)) \neq \bot
\]

\[
(\mathcal{P}, \text{pc}, 0, A, V) \rightarrow (\mathcal{P}, \text{pc}', 0, A(\epsilon'), V(\epsilon'))
\]

Rule 4, DATAFLOW. In order to fire this rule, for some port \( p \in \text{out}(\mathcal{P}) \), there must be a value \( v \) and an address \( q \) available. So, \( v = \text{head}(V(p)) \neq \bot \) and \( q = \text{head}(A(p)) \neq \bot \). The corresponding move instruction therefore was \( p \rightarrow q \). As \( v \) is transported to a different port, it has to be dequeued with \( V'(p) = \text{tail}(V(p)) \) and \( \forall a' \neq \bot \)(tail(A(p))) = \bot.\) For input port \( q \), we want to replace the value closest to the head of the queue where \( V(q) = \bot \) and \( A(q) = p \) with \( v \). This can be described with \( i = \text{first}(A(q), p, V(q), \bot) \) and \( V'(q)[i \rightarrow v] \).

To summarize, this yields the following rule:

\[
p \in \text{out}(\mathcal{P}) \land v = \text{head}(V(p)) \neq \bot \land q = \text{head}(A(p)) \neq \bot
\]

\[
(\mathcal{P}, \text{pc}, \text{st}, A, V) \rightarrow (\mathcal{P}, \text{pc}', \text{st}, A', V')
\]

with

\[
A' = A[\text{pc}], V' = V[\text{pc}, q \rightarrow V(q)[i \rightarrow v]]
\]

and

\[
i = \text{first}(A(q), p, V(q), \bot).
\]

Rule 5, FIRE. The PU firing rule will mostly be reduced to DPN operational semantics. PU functions are DPN functions, as stated in Section IV. For the PU firing, we therefore treat the PU as a DPN node on the value part (\( V \)) of the connected input and output buffers. We simply use the informational firing rules of DPNs to determine whether it can fire, and how streams are transformed. The only difference is that whenever values from \( V \) are consumed, their respective addresses are also consumed from \( A \). So, if \( n \) values are dequeued from \( V(p) \) by the firing rule, \( n \) values are dequeued from \( A(p) \) as well.

Execution terminates if all rules have fired exhaustively. A (possibly infinite) run \( R \) of a SCAD architecture is a sequence of configurations where for all adjacent configurations \( C_i, C_{i+1} \) in the sequence \( C_0, \ldots, C_j \) holds by one of the rules above.

A run \( R \) is exhaustive, iff it terminates or is infinite, i.e., we keep applying rules for as long as possible. Note that Rule 4 is only well-defined if for a move instruction \( \text{src} \rightarrow \text{tgt} \) the corresponding entries in both address buffers of \( \text{src} \) and \( \text{tgt} \) exist (with a \( \bot \) value in the value buffer of \( \text{tgt} \) at the corresponding position, of course). Since we always add addresses to both buffers synchronously in Rules 1 and 2, applying rules will not cause any issues with the definition: addresses are always added and removed simultaneously from \( \text{src} \) and \( \text{tgt} \). A problem can therefore only occur when starting from a configuration that does not have corresponding entries in the address buffers. In this paper, it is therefore required that any initial configuration \( C_0 \) has empty address buffers, meaning

\[
A(C_0)(p) = \epsilon \ \forall p \in \mathcal{P}.
\]

Using this assumption, the issue can be safely ignored. It is also not a severe restriction: addresses stored in the initial configuration can be added back by prepending corresponding move instructions to the program. Why exactly do we introduce nondeterminism, though? While real hardware implementations are of course deterministic, we intentionally choose this abstract characterization: the interconnection networks can experience varying communication delays, while the order or duration of PU firings may alternate. By allowing all possible runs, we can prove that scheduling and timing are not relevant for the correctness of produced results (in the sense of Lemma 1). Therefore, the compiler can then ignore low-level implementation details that may even vary dynamically during execution.

V. A SOUND CLASS OF SCAD ARCHITECTURES

In this section, a sound class of SCAD architectures producing scheduling-independent, latency-insensitive results is derived. In order to achieve this, a given SCAD architecture \( A \) is first modelled using a DPN \( c(A) \). Then, bisimulation of \( A \) and \( c(A) \) is proven. By using Lemma 1, we can then show that requiring Kahn criteria will yield similar results for SCAD architectures.

A. DPN Model for SCAD Architectures

The DPN has a node for every port \( j : o_j \) and \( c \in \mathcal{P} \). Furthermore, it has a node for every PU in \( F \). To read and distribute move instructions, there is also a CU node. Figure 3 illustrates the basic structure of \( c(A) \). Now, nodes have to be connected. The control unit (CU) is connected to all input ports with \( \text{src} \) and to all outputs with \( \text{tgt} \). These buffers are similar to the \( a_{dr} \) buffers of the SCAD architecture. Furthermore, the CU is connected to itself with three buffers \( \mathcal{P} \), \( \text{pc} \) and \( \text{st} \). Also, the input port \( c \) feeds into the CU. \( \mathcal{P} \) keeps a copy of the program in execution, \( \text{pc} \) is the program counter and \( \text{st} \) is the stall flag. Data transports are simulated by the port nodes. For every PU \( p = (P', f) \in F \) we connect the input ports \( \text{in}(P') \) to the PU node \( p \) with \( i_j \) and then \( p \) to the output ports \( \text{out}(P') \) with \( o_j \). The actual values that the PUs consume and produce are in the \( i_j \) and \( o_j \) buffers. These are similar to the \( \text{val} \) buffers of the SCAD machine. One difference is that \( i_j \) buffers only contain values that can already be consumed. \( \text{val} \) buffers may contain \( \bot \) values and even valid values after that. The \( i_j \) buffer can only contain values up to the first \( \bot \) value. This is due to the fact that SCAD input buffers actually behave like FIFO buffers for every respective source address. This is then similar to the
Buffers – as these are literally FIFO buffers for every possible source address: every output port \( o_i \) is connected to all input ports \( i_j \), labeled with \( dtm_{i,j} \). Additionally, all output ports are connected in the same manner to \( c \). If input or output buffers are left without connections to or from a PU node, we add input edges to output buffers and output edges to input buffers from the environment. Next, firing rules of nodes are given.

**CU.** The CU node manages the control state and adds move instructions to the respective buffers. The inputs are \( \mathcal{P}, \mathcal{S}, st, pc \) and \( c \). The output consists of the same (except \( c \)) and additionally \( src_{c_j} \) (for all \( i_n \)) and \( tgt_{i} \) (for all \( out_i \)), which are similar to the \( a_{dr} \) buffers of the SCAD architecture. This node works very similarly to the CU rules of Section IV, which is why we skip the informal discussion. The only noteworthy difference is that if the correct instruction is not at the head of the \( \mathcal{P} \) buffer, the buffer is simply looped until the right instruction is found. The cases will be annotated with their respective name. The firing rule of this node is the following:

\[
\begin{align*}
\text{st'} &= \text{pop(st)} // \text{read stall state} \\
\text{pc'} &= \text{pop(pc)} // \text{and program counter}
\end{align*}
\]

- if(\( st' = 0 \)) { // not stalling
  - if(l != pc') { // \( l \) not correct
    - push(\( \mathcal{P}_{,} \), l) and (\( l : x \to y \))
    - push(pc, pc')
  } else { // \( l \) correct
    - push(src_{c_j}, x) // ADVANCE
    - push(tgt_{i}, y)
    - push(\( \mathcal{P}_{,} \), l)
    - if(y == c) { // if target is CU
      - push(pc, pc',) // STALL
      - push(st, 1)
    } else {
      - push(pc, pc' + 1)
      - push(st, 0)
    }
  }
}
else { // stalling
  - v = pop(i) // CONTINUE
  - push(pc, v)
  - push(st, 0)
}

Note that the description as given clearly only needs sequential blocking reads and does not check for emptiness of a buffer. It therefore fulfills the Kahn criteria given in Definition 1.

**Nodes** \( i_j, \forall j \in \{1,...,m\} \). This node handles the values that will go to \( i_j \) (which is then directly connected to a PU). The incoming buffers are \( src_{c_j} \) (from CU) and \( dtm_{i,j} \) (from \( o_i \)). First, address \( x \) must be read from \( src_{c_j} \). The next value for \( i_j \) will come from output buffer \( x \) as was scheduled by the CU. Value \( v \) from buffer \( dtm_{i,j} \) is then read accordingly. \( v \) is then written to \( i_j \). These values can then be consumed by the PU. The rules are summarized in the following firing table:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
src_{c_j} & dtm_{i,j} & \ldots & dtm_{i,m} & i_j \\
\hline
1 : S & x : X_1 & \ldots & X_n & x & \\
\vdots & \vdots & \ddots & \vdots & \vdots & \\
n : S & X_1 & \ldots & x : X_n & x
\end{array}
\]

**Nodes** \( o_i, \forall i \in \{1,...,n\} \). This node manages where values of \( o_i \) will be sent. The inputs are \( tgt_{i} \) (from CU) and \( o_i \) (from PU), while outputs are \( dtm_{i,j} \) to all \( i_j \). First, address \( y \) is read from \( tgt_{i} \). The next value \( v \) of \( o_i \) is going to be sent to input buffer \( y \) as scheduled by the CU. Therefore, \( v \) is written to \( dtm_{i,j} \). Again, the following firing table summarizes the rules:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
tgt_{i} & o_i & dtm_{i,1} & \ldots & dtm_{i,m} & o_i \\
\hline
1 : T & x : X & x & \ldots & \epsilon & \\
\vdots & \vdots & \vdots & \ddots & \vdots & \\
m : T & x : X & \epsilon & \ldots & x
\end{array}
\]

**PU** \( p \). This node implements PU firings of PU \( p = (P', f) \in F \). It will thus simply use the firing rules \( f \) of \( p \).

The DPN model does have significant differences that will influence the bisimulation proof in Section V-B:

**Fixed-size Patterns.** Fixed-size patterns are DPN node functions that essentially are able to check if a buffer is empty in addition to the conventional queue operations (push and pop). It is important to note that \( X \) and \( c(X) \) may exhibit different behaviours if fixed-size patterns are used in PU firing rules. Consider the example in Figure 4, where the rule of \( in_{2} \) denotes checking for emptiness. Now using SCAD operational semantics, if \( out \rightarrow in_{2} \) and then \( out \rightarrow in_{1} \) are to be executed, the sequence in Figure 4(b) will occur. Thus, the firing rule can never match the input stream and execution terminates without the PU ever firing. In the DPN this is not necessarily the case, though. In the following, a run is given that enables the PU to fire in the corresponding DPN: first, move instructions are added by the CU to \( src_{c_j}, src_{c_j}, tgt_{i} \) and \( tgt_{i} \). Then, \( o_1 \) may fire twice and put \( a \) into \( dtm_{i_{1,2}} \), as well as \( b \) into \( dtm_{i_{1,1}} \). Now \( i_1 \) and \( i_2 \) can both fire. Since they are independent, they can fire in any order. Therefore, it is possible for \( i_1 \) to fire first and put \( b \) into \( i_1 \). The PU then only sees \( b \) in \( i_1 \) while \( i_2 \) is empty, and can fire. This clearly implies that the constructed DPN and SCAD architecture are not bisimilar if fixed-sized pattern matching is present in a firing rule. As will be shown however, this problem is indeed limited to fixed-size pattern matching. If only prefixes of streams can be matched, this is no problem: the DPN always sees less or equal values in its PU input buffers. If only prefixes are matched, firing rules active in the DPN are also active in the SCAD architecture.

**Consumption Limitation.** Furthermore, the models are only equivalent if the PUs of the SCAD architecture can not consume and read values stored after the first \( \bot \) value in an input buffer. The DPN stores these values in different buffers (the \( dtm_{i,j} \) buffers) not accessible to the PU. Therefore, for the sake of making the models equivalent, we will assume that SCAD architectures can not do this either.
B. Weak Bisimulation

Using model c(\(\mathfrak{A}\)) and DPN operational semantics, we get a different characterization of operational semantics for a SCAD architecture \(\mathfrak{A}\). In this section, we will show using weak bisimulation that both notions agree. Then, conclusions from Kahn criteria (Lemma 1) can be applied to SCAD architectures.

First, we have to define how configurations of SCAD architectures and the constructed DPNs are related with each other. To that end, we give a relation \(\mathcal{R}\) between SCAD architecture configurations of \(\mathfrak{A}\) and the state of the derived DPN \(c(\mathfrak{A})\). If \((p,q)\) is in \(\mathcal{R}\), where \(p = (Q, D, pc, st, A, V)\) is the SCAD configuration and \(q\) the DPN configuration (mapping buffer names to streams), iff all of the following requirements are fulfilled.

**Requirement 1. Program Equivalence.** Program \(\mathfrak{P}\) is equivalent to \(q(\mathfrak{P})\) when treated as a set (note that the labels make instructions unique):

\[ \mathfrak{P} = \{ i \mid i \in q(\mathfrak{P}) \} \]

Both models must be executing the same program.

**Requirement 2. Control Flow.** It is true that \(pc = q(pc)\) and \(st = q(st)\). This also implies \(|q(pc)| = 1\) and \(|q(st)| = 1\). This just means that the control state of both models must be equivalent.

**Requirement 3. Output Equivalence.** For all \(o_1, ..., o_n\):

\[ V(o_i) = q(o_i) \text{ and } A(o_i) = q(tgt_i) \]

Output buffer values and addresses must be equivalent.

**Requirement 4. Input Equivalence 1.** For all \(i_1, ..., i_m\): exhaustively simulate firing of the \(i_j\) nodes in \(c(\mathfrak{A})\) by replacing missing \(dtn_{i,j}\) values with \(\perp\) until \(src_{i,j}\) would be empty – then the resulting streams \(s \subseteq Stream_i\) must fulfill \(s = V(ij)\). This characterizes the notion that the DPN stores input values after the first \(\perp\) symbol in its \(dtn\) buffers. To define this formally, we first create function \(\text{con}_{ij}\) to simulate the exhaustive firing, where \(d_i\) will denote the content of the incoming \(dtn_{i,j}\) buffer:

**Definition 6. Exhaustive firing simulation.**

\[
\text{con}_{ij}(src, d_1, ..., d_n) = \begin{cases} 
\{ \epsilon \text{ if head}(src) = \perp \} & \text{head}(d_{head(src)}) \text{. con}'_{ij} \text{ else}.
\end{cases}
\]

where

\[
\text{con}'_{ij} = \text{con}_{ij}(\text{tail}(src), d'_1, ..., d'_n) \]

and

\[
d'_i = \begin{cases} 
\text{tail}(d_i) & \text{if head}(src) = i \\
 d_i & \text{else}.
\end{cases}
\]

Then, stream \(s_j\) can be expressed in terms of the initial configuration of \(q(ij)\), with the results of the exhaustive firing appended to it:

\[
s_j = q(ij) . \text{con}_{ij}(q(src_{ij}), q(dtn_{i,j}), ..., q(dtn_{n,j}))
\]

However, we need to further require that this is actually exhaustive. Let \(dtn^*_{i,j} \forall i \in \{ 1, ..., n \} \) be the \(d_i\) parameters used in the last recursive call of \(\text{con}_{ij}\). Then, \(dtn^*_{i,j}\) must be empty and Requirement 4 simply becomes:

\[
s_j = V(ij) \text{ and } dtn^*_{i,j} = \epsilon
\]

**Requirement 5. Input Equivalence 2.** For all \(i_1, ..., i_m\): after removing the first \(\{ q(i_j) \}\) elements from \(A(ij)\) (by applying the tail function \(|q(ij)|\) times), it must be equivalent to \(q(src_{ij})\): the DPN does not contain addresses of values that are ready to consume which is why the corresponding addresses in the SCAD architecture have to be ignored.

\[
tail^{|q(ij)|}(A(ij)) = q(src_{ij})
\]

To reiterate, the tuple \((p, q)\) consisting of SCAD configuration \(p\) and DPN configuration \(q\) is in \(\mathcal{R}\), iff Requirements 1-5 hold for them. Using \(\mathcal{R}\), we can now show weak bisimulation of \(\mathfrak{A}\) and \(c(\mathfrak{A})\).

**Lemma 2.** \(\mathfrak{A}\) and \(c(\mathfrak{A})\) are weakly bisimilar.

**Proof:** For the proof, we can always assume Requirements 1-5 hold prior to firing any operation. Then, it must be shown that all of them are satisfied again after firing the respective operations. First, we show simulation of \(c(\mathfrak{A})\) by \(\mathfrak{A}\). Assume \((q, p) \in \mathcal{R}^{-1}\). For \(q \rightarrow q'\) any of the DPN nodes can fire. Therefore, we make a case distinction on the type of node (CU, \(ij\), oi or PU) that is firing:

**Case CU.** Either no action is necessary, or we do the equivalent one in SCAD. We further split this case into which rule has to be fired in the SCAD architecture.

**(No Action)** If \(q(st) = 0\), instruction \(l : i \rightarrow j\) is read from \(q(\mathfrak{P})\). If \(l \neq pc\), all values are written back. The resulting state \(q'\) is still equivalent to \(p\). The only thing changed is the order of the \(\mathfrak{P}\) buffer, which does not matter for relation \(\mathcal{R}\), since only set equivalence of \(\mathfrak{P}\) and \(q(\mathfrak{P})\) is needed by Requirement 1.

**(Continue)** If \(q(st) = 1\), \(st = 1\) must hold as well (Requirement 2). We can only fire if \(q(c)\) contains a value \(v\), and if that is the case, does \(V(c)\) (Requirement 3). In SCAD, we then fire the \text{CONTINUE} rule. In both models, the values are consumed and \(pc' = v\), as well as \(st' = 0\) are written. Therefore, Requirement 2 and Requirement 3 are fulfilled for the resulting states. The other requirements are not touched and therefore still hold.

**(Advance)** If \(q(st) = 0\), \(q(\mathfrak{P}) = l : i \rightarrow j\), \(q(pc) = l\) and \(j \neq c\), the \text{ADVANCE} rule must be fired in the SCAD architecture. Source address \(i\) (of instruction \(l : i \rightarrow j\)) is enqueued in \(A(ij)\) and \(q(src_{ij})\), as well as \(j\) in \(A(oi)\) and \(q(tgt_{ij})\). SCAD additionally adds a \(\perp\) to the tail of \(V(j)\). Because \(A(oi) = q(tgt_{ij})\) (Requirement 3), clearly \(A(oi).j = q(tgt_{ij}).j\). Therefore, Requirement 3 still holds. Requirements 1 and 2 are not touched, and therefore still hold. It remains to show that Requirements 4 and 5 are fulfilled. For the input buffer in the DPN, the missing \(\perp\) will be inserted by the exhaustive firing \(\text{con}_{ij}\), which now has an additional recursion because it will be called on \(q(src_{ij})\) with \(i \neq \perp\), instead of just \(q(src_{ij})\). Note that since before, \(dtn^*_{i,j} = \epsilon\) (the state of the last recursion, Requirement 4), the added address \(i\) will definitely yield a \(\perp\) at the tail of \(s_j\), with the rest being unchanged. This is because \(head(\epsilon) = \perp\). This is convenient, since the SCAD architecture
enqueues a $\perp$ to $\mathcal{V}(ij)$. Therefore, $s_j, \perp = \mathcal{V}(ij), \perp$, which then fulfills Requirement 4. Now, because of Requirement 4 and the fact that SCAD keeps an address for every value, it is clear that $|q(ij)| \leq |A(ij)|$. Values stored in $q(ij)$ must be a prefix of the ones in $\mathcal{V}(ij)$, and $|\mathcal{V}(ij)| = |A(ij)|$. Therefore, the tail function will at most remove $|A(ij)|$ many values. Thus, for $|A(ij)| = 1$ values, at least the last value will remain. This implies $tail^{(|q(ij)|)}(A(ij), i) = q(src_i), i$, which satisfies Requirement 5.

(Stall) If $q(st) = 0$, $q(\mathcal{V}) = l : i \rightarrow j$, $q(pe) = l$ and $j = c$, the STALL rule must be fired in the SCAD architecture. Then, $s'_i = 1$ and $q' (st) = 1$ are set, which satisfies Requirement 2. The rest of this case is equivalent to Advance.

Case $ij$. No action in the SCAD architecture is necessary. Since $\mathcal{R}$ only requires the result of exhaustive firing of $ij$ nodes to be equivalent to the SCAD configuration, Requirement 4 is still satisfied. For Requirement 5, there is one value less in $src_j$, but we also remove one more value from $A(ij)$, which is fine: $tail^{(|q(ij)|)}(A(ij)) = q(src_i)$ (Requirement 5) directly implies $tail^{(|q(ij)|)+1}(A(ij)) = tail(q(src_i))$. Therefore, the new DPN state $q' st$ still satisfies $(q', p) \in \mathcal{R}^{-1}$.

Case $oi$. Is simulated using the DATAFLOW rule in SCAD. Values and addresses are dequeued from the output buffer and are then transported to the respective position in the input buffer. The DPN uses the $\text{dtn}$ buffers to determine this respective position, while the SCAD architecture uses the first function. Clearly, $j = head(q(tgl)) = head(A(oi))$ and $v = head(q(oj)) = head(V(oi))$ must hold (Requirement 3). Also, if the firing rule of $oi$ is active, $v \neq \perp$ and $j \neq \perp$ must hold, which implies that DATAFLOW can be fired in SCAD. The data move being executed is $i \rightarrow j$, transporting value $v$ from output buffer $oi$ to input buffer $ij$. Because of Requirement 3, $\mathcal{V}(oi) = q(oi)$ and $A(oi) = q(tgl)$ hold. This directly implies $tail(V(oi)) = tail(q(oi))$ and $tail(A(oi)) = tail(q(tgl))$. Therefore, Requirement 3 is still satisfied. The premises of Requirements 1 and 2 are not touched and therefore they remain valid, but the content of input buffer $j$ will change. Therefore, Requirements 4 and 5 remain to be shown. In the SCAD architecture the first$(A(ij), i, V(ij), \perp)$ will find the closest address $i$ to the head of the buffer where $V(ij)$ is $\perp$, and then replaces that $\perp$ with $v$. In the DPN, exhaustive firing must put a $\perp$ symbol at some position in the sequence $s_j$, because the SCAD architecture has a $\perp$ symbol and Requirement 4 was valid. Note here that this line of argument is only valid because it was required in Section IV that configurations start with no values in address buffers, and we just assume that corresponding entries always exist. When pushing $v$ into $\text{dtn}_{i, j}$, this clearly replaces the one closest to the head where the corresponding address is $i$. This is because all values with the corresponding address are read from the $\text{dtn}_{i, j}$ buffer. Therefore, Requirement 4 still holds true. The number of values removed at the head of $A(ij)$ did not change, and $src_j$ has not been touched. Therefore, Requirement 5 still holds as well.

Case $PU$. The content of the input streams $ij$ in the DPN is a prefix of the one in SCAD. If none of the $ij$ nodes can fire, the first value that is not in $ij$, but in the SCAD architecture can only be a $\perp$. The SCAD architecture can not read values beyond the first $\perp$. Therefore, the SCAD architecture sees more or equal numbers of values in the buffers. Since we defined semantics of PU firing as a DPN using the same firing rule, a rule firing in the DPN must be active in the SCAD architecture as well if matching with fixed size patterns is not allowed. The SCAD architecture additionally consumes its addresses when the rule fires. This is however fine, since we do not care for the addresses that can be consumed, namely the first $|q(ij)|$ in the SCAD architecture (Requirement 5). The rule firing can at most consume $|q(ij)|$ tokens. Therefore, this case can simply be simulated using the FIRE rule using the same firing rule as the DPN. Requirement 5 still holds by the argument before. As equal values are consumed and produced in both models, Requirements 3 and 4 still hold as well. Requirements 1 and 2 are not touched.

Next, we show simulation of $\mathcal{Q}$ by $c(\mathcal{Q})$. Assume $(p, q) \in \mathcal{R}$. For $p' \rightarrow p'$ we can perform any of the given rules 1-5 from Section IV.

Case $\text{ADVANCE}$. In this case $st = 0$ holds, directly implying $q(st) = 0$. The DPN might need to find the appropriate instruction $l$. We therefore fire the CU node as often as necessary, until the desired instruction $l : i \rightarrow j$ is found. This will surely succeed, since set equivalence of $\mathcal{Q}$ and $c(\mathcal{Q})$ was required. At this point, this case becomes analogous to the one previously discussed for the DPN CU node: corresponding addresses are enqueued in the appropriate buffers in both models. Source address $i$ is enqueued in $src_j$ in the DPN. In SCAD, it is enqueued in $A(ij)$. Since the amount removed at the head of this buffer does not change, Requirement 5 still holds. The same is true for Requirement 3. Target address $j$ is enqueued in the respective output buffers $tgl_i$ and $A(oi)$. From the analogous case, we can see that this clearly yields equivalent sates $(p', q') \in \mathcal{R}$.

Case $\text{STALL}$. This case is analogous to $\text{ADVANCE}$. We additionally just set $st = 1$ in both models.

Case $\text{CONTINUE}$. In this case $st = 1$, so clearly $q(st) = 1$ holds. The DPN CU node can only fire once a value is available at $q(\epsilon)$. This is the same case for SCAD ($\text{ADVANCE}$ and $\text{STALL}$ cannot fire if $st = 1$), and both models will set the $pe$ to the value of $c$ and $st = 0$.

Case $\text{DATAFLOW}$. A move action from output port $i$ is simulated in the DPN by firing the appropriate $oi$ node. Consumption is equivalent in both models. Thus, this case becomes analogous to the case of firing a $oi$ DPN node, in which we already detailed why this yields equivalent states.

Case $\text{FIRE}$. The SCAD architecture may have more values available in an input buffer $ij$. However, since it can only read up to the first $\perp$ value, we know that we can make the DPN $ij$ buffer equivalent up to the first $\perp$ by firing the $ij$ nodes (as explicitly ensured by Requirement 4). Clearly, both models then have the same activated firing rules, which will yield equivalent results since output buffers are directly related in $\mathcal{R}$.

This case concludes the bisimulation proof. Using the same relation $\mathcal{R}$, $\mathcal{Q}$ can simulate $c(\mathcal{Q})$ and vice versa.

Next, our desired result can be given – yielding a sound, configurable class of SCAD architectures:
Lemma 3. If the firing rules of the PU nodes fulfill the Kahn criteria and cannot consume values stored after ⊥, the results of c(Ẋ) and therefore of Ẋ are scheduling-independent and latency-insensitive.

Proof: All given functions of the DPN fulfill the Kahn criteria. If additionally the PU nodes do, the statement follows for c(Ẋ) from Lemma 1. From Lemma 2 and the fact that Ẋ directly related output buffers (Requirement 3) and eventual content of input buffers (Requirement 4), it therefore also follows for Ẋ.

VI. A FRAMEWORK FOR GUARANTEED SPACE BOUNDEDNESS

By choosing a SCAD architecture that simulates a queue machine, it should come with no surprise that SCAD machine programs can be Turing-complete using the given semantics, even if the Kahn criteria are adhered and no memory beyond the FIFO buffers is used. The queue is simply stored within the buffers.

Lemma 4. There exists a Turing-complete SCAD architecture without external memory fulfilling the Kahn criteria.

In the following, a program Ẋ on a SCAD architecture is space bounded, if there exists a n ∈ N and a exhaustive run Ẋ such that the length of all buffers is bounded by n for all Č ∈ Ẋ. In particular, using a simple diagonalization argument, the following lemma can then be shown:

Lemma 5. Space boundedness for SCAD machine programs without external memory is undecidable.

Note that these lemmas would not hold for conventional register-based systems: they are trivially space bounded without external memory, since the number of registers is hardcoded into the instruction set. However, this situation is not desirable: space within processors is usually tightly restricted. Space needed within a processor should probably not even depend on the problem size. In this section, we will therefore make a proposal to structure move programs and architectures such that space boundedness can be guaranteed easily. Thus, we create a setting more comparable to register machines. First, the necessary restrictions are given. Then, a method to guarantee boundedness is described.

Let us first consider an example, using the PUs of Figure 2 as a reference:

1: a → b
2: l → c

Why does this program not have bounded space requirements? After the first iteration, there is one entry in the adr buffer of a and b. PUs cannot fire, no data move can be executed. Clearly, this will not change in the following iterations: After i iterations, i many entries will be present in each buffer. Data transports are never actually executed, while buffers are being filled endlessly. The idea of the following restrictions is to make sure that every time the basic block induced by label i is entered, it is entered with the same kind of configuration – unlike the example. In essence, we require well-defined interfaces between basic blocks, such that it is not possible to build up junk in buffers.

Architectures will be restricted to static PU functions (as known from static DPNs [16]). The number of produced values and number of consumed values from buffers cannot vary with firings – they are constants. In particular, they are independent of the actual values present in value buffers – only the number of values present matters to determine if they can fire. This fact is crucial for the method that will be presented in the following. Clearly, these static PU functions automatically fulfill the Kahn criteria, meaning that Lemma 3 applies.

A. Well-Structured Move Code

We now introduce labels and a pseudo-op for more structured branching, that are to be translated and checked by an assembler.

Branches. We add pseudo-op (b, l₁, l₂) → c, meaning that depending on whether the value sent from b is 0, control will either jump to label l₁ or l₂. Furthermore, a normal branch b → c as defined previously is disallowed.

Well-defined Interfaces. All labels i that are reachable via a branch must be marked with a configuration layout L = (L₁, L₂) with L₁ : P → Stream and L₂ : P → Stream : (l₁, ⊤) : for every buffer of the architecture, it includes the expected location of values stored in V and the expected addresses in Ā (compare to the definition of configurations in Section IV). ⊤ denotes that there is a value and ⊥ denotes absence, as before. Note that we need to provide a map that can include ⊥, since there can be gaps (compare Figure 2, bottom right). Every time control reaches this label, the current configuration must fit the layout. A configuration with Ā and V fits a configuration layout (L₁, L₂), if Ā = L₁ and V(b₁) = x operand in and output to ⊤. Every time a PU would consume an actual value in Ā, it will now consume ⊤ in Ā. Every time it would produce a value, it will produce ⊤. For example, if we are given a PU that reads x, y from two input buffers and produces their sum x + y, the equivalent PU in Ā would simply read ⊤, ⊤ while producing ⊤. Data moves will move ⊤ symbols just like actual values. Since we required static PU functions, execution will only really depend on values of D if we perform our pseudo-op (b, l₁, l₂) → c. However, as
long as no branching occurs and we are only interested in how configuration layouts look, it clearly suffices to perform simulated execution in \( \mathcal{A} \). Configuration layouts can be easily extended to configurations of \( \mathcal{A} \) (by using the defined label and program).

Next, check structured move code using \( \mathcal{A} \): for every label \( l_0 \) that has a specified configuration layout \( L_0 \), we will execute \( L_0 \). From exhaustive execution in \( \mathcal{A} \) until a pseudo-op \( (b, l_1, l_2) \to c \) or a label \( l_1 \) with layout \( L_1 \) is encountered, we get a resulting configuration layout \( L_0 \) and possible outgoing labels \( l_1, l_2 \). We remove a \( \sqcap \) from buffer \( b \) (including its address, thus simulating the data move required for the pseudo-op) and get \( L_{0}^\prime \). If there is no \( \sqcup \) entry, we reject the program. Then, we check whether the resulting configuration layout \( L_{0}^\prime \) matches the defined configuration layout \( L_{1}, L_{2} \) of the outgoing labels. If all do, we know that the program is space bounded, since all actual configurations would fit \( L_{0} \). Note that because of Lemma 3 it suffices to choose any exhaustive run to make this claim. If layouts do not match, we know that the specification as given is not correct and therefore reject the program. Clearly, this method is not complete in terms of space bounded programs, since actual executions might not be able to reach both labels of a branch.

VII. Conclusions

We have shown a close relationship between DPNs and the model of computation used by SCAD architectures. This relationship gave rise to a configurable and sound class of SCAD architectures. Furthermore, we have seen that there are other problems related to the nature of SCAD machine code, i.e., being not implicitly space bounded. Nevertheless, it was possible to achieve a trustworthy foundation for further development by limiting its possibilities.

References


