Exploring Different Execution Paradigms in Exposed Datapath Architectures with Buffered Processing Units

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

Abstract—Exposed datapath processor architectures allow the software to bypass register usage by directly moving intermediate results between processing units with suitable instructions. Synchronous Control Asynchronous Dataflow (SCAD) is a new exposed datapath architecture consisting of a grid of processing units with buffers to store their input and output values. Code generation inspired from queue machines can utilize the bypassing capability of such exposed datapath architectures with buffered processing units and can completely eliminate the use of registers this way. In this paper, we consider different execution paradigms of SCAD that make a compromise between hardware complexity, hardware scalability, and the use of instruction-level parallelism. To that end, we study both resource- and time-optimal code for all variants that we determine by satisfiability modulo theories (SMT) solvers. Experimental results show that the execution paradigm followed by our original SCAD architecture makes a reasonable compromise between hardware complexity and the use of instruction-level parallelism while maintaining hardware scalability.

I. INTRODUCTION

A. Motivation

Use of instruction-level parallelism (ILP) by dynamic scheduling as introduced by the Tomasulo algorithm [1] for processors with out-of-order execution and static scheduling as introduced for very long instruction word (VLIW) processors [2], face limits on their further scalability [3]. One inherent reason for these limitations is the limited number of registers: Since the execution time of memory accesses did not improve as fast as that of other instructions, the use of memory accesses had to be avoided as much as possible. A simple way to avoid them is to load values into local memories like registers and to work on the local copies as long as possible. However, the limited number of registers and the consequent need of load and store instructions limit the use of ILP. For example, consider the expression tree shown in Figure 1a. By the Sethi-Ullmann algorithm [4], at least 3 registers are required to evaluate the expression tree if no load/store instructions shall be used. Using 4 registers, one can evaluate the expression in only 3 parallel steps. If only 2 registers would be available, one has to insert spill code in that the obtained result of \(x_1 + x_2\) is stored in memory and loaded in a register after having evaluated \(x_3 + x_4\) as shown in Figure 1b. It now takes 5 steps to evaluate the program (since there are 5 levels in the obtained syntax tree), irrespective of the number of processing units (PUs). Note that more memory accesses not only reduce the performance, but also adversely affect the timing-predictability of applications, which is an important metric in real-time embedded systems.

Increasing the number of registers is however difficult: First, this number is directly encoded in the instruction sets. Changing it requires corresponding changes in processors and compilers. Second, increasing the number of registers and PUs quickly leads to a bottleneck in wiring these on the chips. For the latter reason, clustered architectures [5] have been introduced where PUs have only access to predefined register clusters.

Exposed datapath architectures propose an interesting alternative by avoiding the use of programmer-accessible registers. Instead, these architectures provide a large number of PUs and allow the compiler to move values directly from one PU to another. Another reason for the recent popularity of exposed datapath architectures is that they also allow the compiler to mitigate the increasing communication delays in hardware [3], [6]: Instructions may be statically assigned to PUs such that the physical distance that a data value must travel from its producer PU to its consumer PU is minimized. These architectures have already been studied to a great detail. However, their current compiler technology [7]–[11], although utilizes register bypassing, still relies on classic code generators where the optimal use of registers is in the focus.

B. Background

In [12], we suggested a code generation technique for exposed datapath architectures based on a breadth-first traversal rather than the classic depth-first traversal over the syntax trees. To that end, we considered the Synchronous Control

Fig. 1. (a) An example expression tree, (b) the same expression tree with spill code, and (c) the corresponding assembler code.
Asynchronous Dataflow (SCAD) architecture, that is a special exposed datapath architecture whose PUs have buffers for input and output values. The classic depth-first traversal was motivated by the reuse of registers, while the breadth-first version was motivated by classic queue machines to exploit maximal ILP and to eliminate the use of registers completely. The breadth-first traversal ensures that the operands are found in the correct order in the buffers and therefore ensures that there is no need for an additional memory. It is also emphasized in [12] that a simple code generation as done for queue machines does however not lead to optimal code for exposed datapath architectures (due to multiple PUs and buffers in these architectures). An optimal code generation is encoded in [13] as a satisfiability (SAT) problem to determine the minimal number of PUs required to execute programs without any computational overhead. Furthermore, we refined in [14] the SAT encoding of [13] to an encoding as a satisfiability modulo theories (SMT) problem, introducing execution time as an additional parameter for optimization and demonstrating the superiority of queue-based code generation compared to register-based code generation to exploit concurrency in programs.

C. Contribution

In this paper, we explore three different execution paradigms for exposed datapath architectures with buffered PUs (in the context of the SCAD architecture):

1) The ‘classic’ execution paradigm where ordering of values and addresses of output buffers are used at PU inputs to identify correct operands for the next operation. In the following, we call these architectures simply SCAD or classic SCAD architectures.

2) \textit{Statically ordered} SCAD architectures where PUs only rely on ordering of values at its input to identify correct operands for executing the next operation. In the following, we call these SO-SCAD architectures.

3) \textit{Dynamically ordered} SCAD architectures where explicit tags are used to match operands for executing operations (as used in classic dataflow computers). In the following, we call these DO-SCAD architectures.

Different execution paradigms lead to subtle differences in the SMT encoding for optimal code generation. We use Microsoft’s Z3 SMT solver [15] to obtain both resource-optimal (that uses minimal number of PUs) and time-optimal code (that uses maximal ILP to execute in minimum time) for the above three execution paradigms. Experimental results show that our SCAD architecture makes a reasonable compromise between hardware complexity and the use of ILP while maintaining hardware scalability. We also study the feasibility of the SMT-based optimal code generation in terms of program sizes it can handle. Although practical basic blocks (up to 12 instructions) are successfully processed by the SMT solver, heuristics are required to handle larger basic blocks. To this end, we have developed a cycle-accurate SCAD machine simulator [16] to run and observe the execution of different basic blocks to learn key features for heuristics.

D. Paper Outline

The paper is organized as follows: Section II describes the three execution paradigms of SCAD. The idea behind queue-based code generation for exposed datapath architectures is discussed briefly in Section III. The encoding of optimal code generation for the three variants of exposed datapath architectures as SMT problem and the corresponding experimental results are provided in Sections IV and V, respectively. Section VI mentions some recent exposed datapath architectures. The final section summarizes the paper and mentions future work.

II. \textbf{Execution Paradigms in Exposed Datapath Architectures}

The organization and functionality of SCAD architectures is first described followed by statically ordered and dynamically ordered variants.

A. \textbf{SCAD Architectures}

The organization of PUs in a SCAD architecture is shown in Figure 2. Each PU has queues at its input and output ports. Input and output buffers are connected to two interconnection networks: There is the \textit{move-instruction bus (MIB)} (given in red color) which is used to synchronously send values from the control unit to the PUs, and the \textit{data transport network (DTN)} (given in green color) which is used by the PUs to asynchronously send values to each other whenever these are available. Figure 3a shows a PU with input and output buffers connected to MIB and DTN. Buffers hold pairs \((adr, val)\) of entries. For an input buffer, \(adr\) is the address of the output buffer of the PU that has produced or that will produce the value \(val\). An entry \((adr,⊥)\) with the special value \(⊥\) is used to indicate that the required value is not yet available and will later be sent from the output buffer \(adr\). Similarly, for an output buffer, \(adr\) is the address of the input buffer of the PU that will consume the value \(val\). An entry \((adr,⊥)\) with the special value \(⊥\) is used to indicate that the required value is not yet available and will later be produced by the PU and can then be sent to the input buffer \(adr\).

SCAD is programmed by a sequence of \textit{move instructions} \((src, tgt)\) whose semantics is to move a value from the head of output buffer \(src\) to the tail of input buffer \(tgt\). Although only two-input one-output PUs are shown in Figure 2, a PU in a SCAD architecture may implement any function with an arbitrary number of inputs and outputs. Similarly, any interconnection network ranging from a simple set of buses and sockets to more complex parallel networks such as Omega, Banyan, or Beneš networks can be used as DTN. These properties recommend SCAD as an interesting candidate for application-specific processors.

The execution of a move program works as follows: Using the program counter, the \textit{control unit (CU)} will fetch the next move instruction \((src, tgt)\) from the instruction memory and will broadcast it via the MIB to all PUs. The input buffer with address \(tgt\) will add the entry \((src,⊥)\) to its tail, and the output buffer with address \(src\) will add the entry \((tgt,⊥)\) to its...
new tail. If one of the two buffers should be full, it will signal this via a feedback signal fullBuffer to the control unit. The other buffer will then also not store the entry, and the control unit will resend the move instruction (src,tgt) in the next cycle (it is stalled at this point of time). The data transport related with a move instruction (src,tgt) is deferred to a later point of time when the data is available. It is important to note that all move instructions’ addresses are stored in the buffers in the order in which they were issued by the control unit, i.e., as specified by the program. To see in more detail how a move program is executed, let us consider the behaviors of the PUs, and their input and output buffers.

If a PU will find entries \((adr_1,x_1),\ldots,(adr_m,x_m)\) with \(x_i \neq \perp\) at the heads of its \(m\) input buffers and there is free space in its \(n\) output buffers, it can react and will consume entries \((adr_1,x_1),\ldots,(adr_m,x_m)\) to produce new result values \(y_1 := f_1(x_1,\ldots,x_m),\ldots,y_n := f_n(x_1,\ldots,x_m)\) where \(f_1,\ldots,f_n\) are the functions associated with that PU. Each output value \(y_i\) is then stored in that entry \((tgt,\perp)\) of output buffer number \(i\) that is closest to the head of the output buffer, i.e., that entry is replaced with \((tgt,y_i)\). If there should be no such entry, then a new entry \((\perp,y_i)\) is placed at the tail of the output buffer \(i\), and the next target address for this output buffer will be stored in this entry. Note that it is possible that the result value has been computed before a move instruction has been issued by the control unit to move it to another place.

The output buffers are responsible for the final transport of data by sending messages between PUs over the DTN. Such a message \((src,tgt,val)\) consists of the address of the sending output buffer \(src\), the address of the input target buffer \(tgt\), and the value \(val\) that is transported by the message. A message \((src,tgt,val)\) is created when the output buffer with address \(src\) has a completed entry \((tgt,val)\) as its head. This message is then sent to input buffer \(tgt\) via the DTN. When it will finally reach input buffer \(tgt\), the input buffer will replace the entry \((src,\perp)\) closest to its head with \((src,val)\), and this may trigger a new operation of its PU. Additionally, the output buffers snoop the MIB for receiving new target addresses for their values. If output buffer \(src\) will see the move instruction \((src,tgt)\) on the MIB, it will check whether it contains an entry \((\perp,y_i)\). If so, it will replace the one closest to its head with the address \((tgt,y_i)\). Otherwise, it will create a new tail \((tgt,\perp)\), if there is still space available. Otherwise, it will signal fullBuffer to the control unit, which then has to stall and resend the move instruction later. The input buffers also always snoop the two interconnection networks, i.e., the MIB and the DTN. As explained above, address entries \((src,\perp)\) are put in order in the input buffer \(tgt\) whenever a move instruction \((src,tgt)\) is seen on the MIB, and an available entry \((src,\perp)\) is completed with the value \(val\) when a message \((src,tgt,val)\) arrives.

We must assume at least one store unit (SU) that has two input buffers, one for the memory addresses and another one for the values to be stored at the corresponding addresses. There is no output buffer. Instead, the SU stores the values in the order as specified by the input buffers (in the program order) to the main memory. Clearly, there is also at least a load unit (LU) that has just one input buffer for the addresses and an output buffer for the values loaded from memory. They will be sent through the DTN similar to output values of other PUs.

Branch instructions are handled as follows by the CU: if the target of a move instruction is the CU itself, it is meant to be the program counter. In this case, the CU stops fetching move instructions and has to wait until this value arrives at the head of its input buffer associated with the program counter. Otherwise, it will simply increment the program counter and place it on its input buffer’s head associated with the program counter so that the next move instruction in the program order is fetched in the subsequent clock cycle.

B. Statically Ordered SCAD (SO-SCAD) Architectures

In SCAD, both ordering of values and synchronously registered output buffer addresses are used to identify correct operands for executing the next operation. In SO-SCAD, the PUs only rely on ordering of values to identify operands for executing the next operation. Figure 3b shows a PU in a SO-SCAD architecture (note that there are no address slots in the input buffers). Input and output values of PUs now reside in ‘pure’ first-in-first-out (FIFO) buffers unlike the more difficult buffers of the classic SCAD architecture, thus, reducing the hardware complexity and simplifying the execution of a move program. When a move instruction \((src,tgt)\) is broadcast via the MIB to all PUs, only the output queue with address \(src\) will add the entry \((tgt,\perp)\) to its tail. Similar to SCAD, the result of execution of a PU is added to the value slot in the entry \((adr,\perp)\) at the tail of its output queue. The DTN snoops the head of output queues for values to be transported to the addressed input queues. However, the transported values are then simply enqueued to the tail of the respective input queues.

Since output buffer addresses are not used to order values at PU inputs during runtime, the control unit must synchronously control firings of PUs to ensure that values from different output buffers are moved to the same input buffer in the correct order (details are given in Section IV). This adversely
affects scalability of SO-SCAD. Latencies of PUs in SO-SCAD must be known beforehand to derive a correct static schedule while SCAD machines can tolerate variable latencies of PUs. Furthermore, the DTN in these machines must guarantee that values are transported in the order that they are produced. This is however not a major drawback since all multistage interconnection networks satisfy this requirement by construction.

C. Dynamically Ordered SCAD (DO-SCAD) Architectures

Figure 3c shows a PU in a DO-SCAD architecture. Instead of queues, we have a set of entries at PU inputs and outputs, since unlike the notion of ordering used in SCAD and SO-SCAD, tag matching is used by PUs in DO-SCAD architectures to find corresponding operands to execute an operation. For the sake of simplicity, we assume that all tag values are determined at compile time. A PU’s input pool holds pairs of entries \((val,otag)\), where \(val\) is the operand value and \(otag\) is the tag that is associated with that operand. In the PU’s outputs, we have two kinds of tuples: (1) \((val,otag)\) where \(val\) is the result of the execution of an operation and \(otag\) is the tag associated with its operands, and (2) \((adr,rtag,otag)\) where \(adr\) is the address of that input of destination PU to which the value \(val\) associated with operand tag \(otag\) have to be transported and \(rtag\) is the tag associated with and transported along with \(val\) for operand matching in destination PU. Clearly, the move instructions must now be updated with operand and result tags as follows: \((src\{otag\},tgt\{rtag\})\). For example, consider two instructions \(i\) and \(j\) whose operands are associated with tags \(t_i\) and \(t_j\), respectively.

\[
x_{tgt(i)} = x_{srcL(i)}(t_i) \odot_i x_{srcR(i)}(t_i)
\]

\[
x_{tgt(j)} = x_{tgt(i)}(t_j) \odot_j x_{srcR(j)}(t_j)
\]

Note that the left operand of instruction \(j\) is the target of instruction \(i\) \((x_{tgt(i)})\). Assume that instructions \(i\) and \(j\) are assigned to PUs \(k\) and \(l\), respectively. Then, the move instruction \((PU_{k.out}(t_i),PU_{l.left}(t_j))\) transports the target of instruction \(i\) from the output of PU \(k\) to the left input of PU \(l\) to execute instruction \(j\). The execution of a move program now proceeds as follows: a move instruction is broadcast via the MIB to all PUs. The PU output with address \(src\) will add the entry \((tgt,rtag,otag)\) to its pool of entries. A PU can fire if it finds operand values with matching tags for execution. The result of an execution along with operand tag is added to the output pool of tuples \((val,otag)\). At the PU’s outputs, the operand tag in the pool of tuples \((adr,rtag,otag)\) is matched with that in the pool of tuples \((val,otag)\). If a match is found, the \(val\) and \(rtag\) is made available to the DTN for transporting to the PU input with address \(adr\).

Since every instruction uses unique tags for its operands, tag matching at PU inputs ensures that correct operand values are consumed by the PU for execution. In other words, the compiler simply has to assign instructions to PUs and any assignment is a valid one (see details in Section IV). Similar to SCAD, DO-SCAD is also latency-insensitive. However, this comes at the cost of a considerably more complex hardware that is needed to accommodate and match tags. Importantly, DO-SCAD architectures face the same token matching problem that early dynamic dataflow computers suffered from [17].

III. QUEUE-BASED CODE GENERATION

In this section, we briefly explain a code generation technique for exposed datapath architectures with buffered PUs (again in the context of SCAD) that utilizes the capability of these architectures to bypass registers by moving values directly from one PU’s output buffer to the same or another PU’s input buffer.

A. Code Generation for Queue Machine

A queue machine (see Figure 4a) [18] reads operands for executing an operation from the head of a queue and adds the results to the tail of that queue. Generating a queue program to evaluate an expression tree is done by a breadth-first traversal of the tree [18]. A consistent left to right or right to left traversal ensures that operands required to execute operations at one level are available in the queue in the correct order. An expression tree and the content of the queue after executing each instruction of the expression tree.
machine after executing each instruction of the expression tree is shown in Figure 4b and 4c.

Basic blocks of programs are often represented by DAGs. Generating a queue program for an expression tree is easy since an expression tree is by definition a level-planar graph [19]. However, generating queue programs for general expression DAGs involves first converting the DAG into a level-planar graph and then performing a breadth-first traversal of the graph [19] as shown in Figure 5. The given expression DAG is first levelized which means that operations must only refer to operands at the same level. This can be easily achieved by introducing dup operations which take a value from the head of the queue and add some copies of it to the tail of the queue. Then, the graph is planarized which means that crossing edges are removed by inserting swap operations which take two values from the head of the queue, and add them in exchanged order to the tail of the queue. One can sometimes avoid the introduction of swap operations by a suitable ordering of input or output nodes, but not in general. Finally, another levelization is usually required since swap operations may be placed at new levels.

Fig. 5. (a) An expression DAG, (b) levelized version, (c) planarized version, (d) the final level-planar expression DAG, and (e) the queue contents after executing each instruction in the final level-planar expression DAG.

B. Generating SCAD Code from Queue Code

In [12], we simulated a queue machine by a SCAD machine by mapping each queue instruction to a sequence of moves for a universal SCAD machine. A universal SCAD machine is a SCAD machine with a single universal PU. It has one output queue out and four input queues: left and right to store the operands, opc to store the operation to be executed, and cps to store the number of copies of a result to be added to the output queue. The content of the queue in the queue machine and the output queue in the universal SCAD machine will be the same after each execution of a queue instruction and the corresponding SCAD move instruction, respectively [12]. Given an assignment of each queue instruction to a PU in a SCAD machine with multiple PUs, it is not difficult to adapt the mapping to derive code for the SCAD machine with multiple PUs. However, the derived SCAD code is not optimal.

Since a SCAD machine contains multiple PUs and many queues compared to one central queue of a queue machine, the SCAD machine might not require as much overhead (dup and swap operations) as the queue machine. Hence, the SCAD code generated from the queue code is not necessarily optimal. For example, consider a SCAD machine containing one load-store unit (lsu), one adder (add) and one multiplier (mul). The expression DAG in Figure 5a can be executed on this SCAD machine without any dup and swap operations, while the queue machine required 3 dup and 1 swap operations to execute the same DAG. The dup operator \( D_1 \) (in Figure 5b) is not required since the output from the node \( +_1 \) is accessible from the adder unit’s output queue without duplicating the values \( x_2 \) that resides in the load-store unit’s output queue. Similarly, the swap operator \( S \) is not required since both the moves (represented by the crossing edges) dequeue the values from different output queues, namely from the output queue of the adder unit and the output queue of the load-store unit. For more details, see [12].

IV. OPTIMAL CODE GENERATION

Overhead due to dup and swap operations not only degrades the performance, but also increases the code size and power consumption of exposed datapath machines with buffered PUs.

A minimal number of PUs ensures that code can be generated without any additional overhead. Furthermore, time-optimal code for a given machine (with a given number of PUs) must execute in minimal possible time (or exploit maximal possible ILP). To determine both resource-optimal and time-optimal code, we encode in this section the decision version of the optimal code generation problem as an equivalent SMT problem. The formulation of buffer constraints in SMT encodings differs for the three execution paradigms considered in Section II.

A. Code Generation Problem

Given the following:

- a basic block (DAG) as three-address code in static single assignment (SSA) form consisting of \( l \) instructions, i.e.,

\[
x_{\text{tgt}}(0) = x_{\text{srcL}}(0) \odot 0 \quad x_{\text{srcR}}(0)
\]

\[
\vdots
\]

\[
x_{\text{tgt}}(\ell-1) = x_{\text{srcL}}(\ell-1) \odot \ell-1 \quad x_{\text{srcR}}(\ell-1)
\]

for some variables \( V := \{ x_0, \ldots, x_{n-1} \} \), where \( \odot \) denotes some binary operation,
- a SCAD machine (respectively SO-SCAD or DO-SCAD machine) with one load-store unit (LSU) and \( p \) universal PUs that may execute any binary operation. Let \( \delta_t \) denote the time taken (in clock cycles) to produce the variable \( x_1 \) (by either the LSU or a universal PU),
- a desired execution time \( t \).

Determine if the basic block can be executed on the given machine in time \( t \) without any dup and swap operations. If so, determine the schedule of the basic block on the given machine.

B. Encoding as SMT Problem

In SSA form, every variable \( x_i \) occurs at most once as left-hand side in the three-address code, but it may occur several
times on the right-hand sides. This defines three kinds of variables:

- **target variables** \( V_{\text{tgt}} \) occur as left-hand side
- **source variables** \( V_{\text{src}} \) occur on the right-hand side
- **load variables** \( V_{\text{id}} \) only occur on the right-hand side, i.e., \( V_{\text{id}} := V_{\text{src}} \setminus V_{\text{tgt}} \)

For the rest of this section, we shall refer to the variables \( x_i \) as nodes to avoid ambiguity with the following variables introduced in the SMT encoding.

1) **Binary Integer Variables:** Assuming that PU 0 is the single LSU in the given machine with \( p \) universal PUs \( \{0, \ldots, p-1\} \), we define the following binary variables \( \alpha_{i,j} \) and \( \theta_{i,j} \):

- \( \alpha_{i,j} \) (PU assignment) means that \( x_i \in V \) is produced by PU \( j \in \{0, \ldots, p-1\} \). This determines the instructions of the basic block which are executed by PU \( j \). We demand that all \( x_i \in V_{\text{id}} \) are produced by PU 0 (load/store unit).
- \( \theta_{i,j} \) (scheduling) means that \( x_i \in V \) is scheduled in the time slot \( j \in \{0, \ldots, t-1\} \).

2) **Constraints:** In the following, we set up linear integer and logical constraints, starting with constraining the values of the variables to be either 0/1. Then, every assignment of binary variables that satisfies these constraints is a valid schedule for the given machine and vice versa.

**Binary values:**

\[
\bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{p-1} 0 \leq \alpha_{i,j} \leq 1 \quad \text{and} \quad \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{t-1} 0 \leq \theta_{i,j} \leq 1 \quad (1)
\]

**Schedule exactly once and Unique PU assignment:** The constraint 2 ensures that each node \( x_i \) is scheduled exactly once in one of the \( t \) time slots and that every node \( x_i \) is assigned to one and only one PU.

\[
\bigwedge_{i=0}^{n-1} \sum_{j=0}^{t-1} \theta_{i,j} = 1 \quad \text{and} \quad \bigwedge_{i=0}^{n-1} \sum_{j=0}^{p-1} \alpha_{i,j} = 1 \quad (2)
\]

**Resource constraint:** To comply with the available resources, at most one load instruction and \( p \) non-load instructions may be scheduled in each time slot:

\[
\bigwedge_{j=0}^{t-1} \bigwedge_{\forall x_i \in V_{\text{id}}} \theta_{i,j} \leq 1 \quad \text{and} \quad \bigwedge_{j=0}^{p-1} \bigwedge_{\forall x_i \in V \setminus V_{\text{id}}} \theta_{i,j} \leq p \quad (3)
\]

**Data dependency:** For each node \( x_i \), \( \tau_i \) gives the time slot in which the node is scheduled:

\[
\tau_i = \sum_{j=0}^{t-1} j \times \theta_{i,j} \quad (4)
\]

Consider instruction \( x_{\text{tgt}(i)} = x_{\text{srcL}(i)} \odot_i x_{\text{srcR}(i)} \) of the basic block: It requires that operands \( x_{\text{srcL}(i)} \) and \( x_{\text{srcR}(i)} \) must have already been produced before producing \( x_{\text{tgt}(i)} \), thus we demand:

\[
\bigwedge_{i=0}^{t-1} (\tau_{\text{tgt}(i)} - \tau_{\text{srcL}(i)}) \geq \delta_{\text{srcL}(i)}) \land (\tau_{\text{tgt}(i)} - \tau_{\text{srcR}(i)}) \geq \delta_{\text{srcR}(i)}) \quad (5)
\]

**Buffer constraint:** Consider two instructions \( x_{\text{tgt}(i)} = x_{\text{srcL}(i)} \odot_i x_{\text{srcR}(i)} \) and \( x_{\text{tgt}(j)} = x_{\text{srcL}(j)} \odot_j x_{\text{srcR}(j)} \) of the basic block. If the instructions are executed on different PUs, it is possible to move their operands to the corresponding inputs irrespective of the machine type (SCAD or SO-SCAD or DO-SCAD). Assume that the instructions are executed on the same PU \( k \). In a DO-SCAD machine, the operands can be moved to the respective inputs of PU \( k \) in any order since operands for each instruction are associated with unique tags that are matched at runtime to identify the correct operands for executing instructions. Now consider a SCAD machine: If \( x_{\text{tgt}(i)} \) (respectively \( x_{\text{tgt}(j)} \)) is produced before \( x_{\text{tgt}(j)} \) (respectively \( x_{\text{tgt}(i)} \)), then we should be able to move the operand \( x_{\text{srcL}(j)} \) (respectively \( x_{\text{srcL}(i)} \)) before the operand \( x_{\text{srcR}(j)} \) (respectively \( x_{\text{srcR}(i)} \)), to the left input buffer of PU \( k \). This is possible if the left operands \( x_{\text{srcL}(i)} \) and \( x_{\text{srcL}(j)} \) are produced by different PUs because the synchronously registered output buffer addresses of these PUs at the left input of PU \( k \) ensures correct ordering of these left operands at runtime providing correct operands to PU \( k \) for executing instructions. However, if both operands are produced by the same PU, we must enforce the schedule ordering \( \tau_{\text{srcL}(i)} \leq \tau_{\text{srcL}(j)} \) (respectively \( \tau_{\text{srcR}(i)} \leq \tau_{\text{srcR}(j)} \)). Meanwhile, in a SO-SCAD machine, irrespective of whether the left operands are produced by the same PU or not, we must enforce the schedule ordering \( \tau_{\text{srcL}(i)} \leq \tau_{\text{srcL}(j)} \) (respectively \( \tau_{\text{srcR}(i)} \leq \tau_{\text{srcR}(j)} \)). This is because the SO-SCAD architecture does not register any buffer addresses at PU inputs. Instead, they simply rely on a static ordering of operands to identify the correct operands for execution. Similar arguments apply for the right-hand side operands.

To formalize the buffer constraint, we first introduce a boolean relation \( \beta_{i,j} \) that is true iff \( x_i \) and \( x_j \) are produced by the same PU:

\[
\beta_{i,j} = \bigvee_{k=0}^{p} \alpha_{i,k} \land \alpha_{j,k} \quad (6)
\]

Recall that \( \alpha_{i,k} \) means that \( x_i \) is produced in the target buffer of PU \( k \). Now, constraints 7 and 8 ensure a correct ordering of the variables in the input and output buffers for a SO-SCAD machine and a classic SCAD machine, respectively, while no constraints on variable ordering are required for DO-SCAD machines.

\[
\bigwedge_{i,j=0}^{t-1} \beta_{\text{tgt}(i), \text{tgt}(j)} \rightarrow \left( \begin{array}{ll}
\left( \tau_{\text{tgt}(i)} < \tau_{\text{tgt}(j)} \right) & \land \\
\left( \tau_{\text{srcL}(i)} \leq \tau_{\text{srcL}(j)} \right) & \land \\
\left( \tau_{\text{srcR}(i)} \leq \tau_{\text{srcR}(j)} \right)
\end{array} \right) \land \left( \begin{array}{ll}
\left( \tau_{\text{tgt}(j)} < \tau_{\text{tgt}(i)} \right) & \land \\
\left( \tau_{\text{srcL}(j)} \leq \tau_{\text{srcL}(i)} \right) & \land \\
\left( \tau_{\text{srcR}(j)} \leq \tau_{\text{srcR}(i)} \right)
\end{array} \right) \quad (7)
\]
Clearly, a total ordering on variables (even if they are produced by different PUs) is required for SO-SCAD machines while classic SCAD machines only require a total ordering of those variables that are produced by the same PU. Therefore, unlike SO-SCAD, control units in a classic SCAD machine do not need to synchronously control firings of PUs making it easy to scale. The buffer constraint 7 for a SO-SCAD is stricter compared to buffer constraint 8 for a classic SCAD machine. That is, any solution for the SMT encoding (equivalently any valid program) for SO-SCAD is also a solution for SCAD. Similarly, any valid program for a classic SCAD machine is also a valid program for a DO-SCAD machine. Therefore, as we consider $SO-SCAD \leftrightarrow SCAD \leftrightarrow DO-SCAD$, the following statements hold: (1) Minimal number of PUs required to execute any given basic block decreases and (2) use of ILP for a given number of PUs increases.

3) Solution: To derive the program executed on a particular PU, simply extract the instructions whose target variable is assigned to that PU and sort them according to the target variables using $\tau_i$. Finally, partition the basic block into levels defined by as soon as possible (ASAP) scheduling. For variables belonging to each level, the derived schedule $\tau_i$ imposes a partial order $\prec$ defined by $x_i \prec x_j$ if $\tau_i < \tau_j$. The execution of the basic block can now proceed level-wise starting with bottommost level (containing all instructions that only read variables $\mathcal{V}_{\text{id}}$):

- Arrange the target variables in this level by $\prec$. The variables read in this level are available in output buffers in $\prec$.
- Move them to the input buffers of the PUs that will fire in this level. This is possible due to the buffer constraint that assures that the source variables (variables read) are available in the order $\prec$ in output buffers.
- Fire the instructions of this level on each PU which makes the target variables of this level available in the output buffers, again ordered by $\prec$.

After one round, we can repeat the above for the next level, since after each round, all source variables read in the next level are available in the output buffers in $\prec$ order.

C. Example

Clearly, any basic block can be executed without overhead on a DO-SCAD machine using only 1 PU. Consider the following program:

\begin{align*}
x_2 &= x_1 \odot 0 \cdot x_0 \\
x_3 &= x_2 \odot 1 \cdot x_2 \\
x_4 &= x_0 \odot 2 \cdot x_3 \\
x_5 &= x_3 \odot 3 \cdot x_1
\end{align*}

(9)

Load variables $x_0$ and $x_1$ are assigned to the LSU, while variables $\{x_2, x_3, x_4, x_5\}$ are assigned to any available universal PUs. Assignment of variables (equivalently instructions) to a minimal number of PUs in both classic SCAD and SO-SCAD are shown in Table I. Note that 2 PUs (excluding LSU / PU 0) are required in classic SCAD to execute the basic block without any overhead. Due to data dependencies, $\tau_2 < \tau_4$ and $\tau_2 < \tau_5$. If $x_2$ and $x_4$ are assigned to the same PU 1, the buffer constraint 8 demands that $\tau_1 < \tau_0$ must hold since $x_1$ and $x_0$ are left operands of $x_2$ and $x_4$, respectively. Similarly, if $x_2$ and $x_5$ are assigned to the same PU 1, $\tau_0 < \tau_1$ must hold. Due to these contradicting conditions, an extra PU 2 is used in classic SCAD to execute instruction $x_4$.

At least 3 PUs (excluding LSU / PU 0) are needed in a SO-SCAD architecture to execute the same basic block without overhead. This is because in a SO-SCAD machine, we can no longer schedule $x_3$ and $x_5$ in the same PU 1 since the buffer constraint 7 demands that $\tau_2 < \tau_1$ must hold which is not possible due to data dependencies. This was not a concern in classic SCAD since $x_2$ and $x_1$ are produced by different PUs, and thus have no need to be ordered.

<table>
<thead>
<tr>
<th>PU</th>
<th>instr order</th>
<th>PU</th>
<th>instr order</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$x_0 \prec x_1$</td>
<td>0</td>
<td>$x_0 \prec x_1$</td>
</tr>
<tr>
<td>1</td>
<td>$x_2 \prec x_3 \prec x_5$</td>
<td>1</td>
<td>$x_2 \prec x_3$</td>
</tr>
<tr>
<td>2</td>
<td>$x_4$</td>
<td>2</td>
<td>$x_4$</td>
</tr>
<tr>
<td>3</td>
<td>$x_5$</td>
<td>3</td>
<td>$x_5$</td>
</tr>
</tbody>
</table>

TABLE I: MINIMAL PU ASSIGNMENTS FOR PROGRAM 9 ON A (A) CLASSIC SCAD, AND (B) SO-SCAD.

V. EXPERIMENTAL RESULTS

A. Random Input Generation

A random basic block generator was implemented that accepts the number of nodes $n$ and the number of levels $l$ as input. The basic block is generated by randomly choosing two predecessors of every node ensuring that the DAG has $l$ levels. Clearly, for a $n$ node basic block, $l := \{2, \ldots, n-1\}$ levels are possible. For every pair $(n, l)$, a thousand basic blocks were generated. All the experimental runs are performed on an 2x Intel Xeon CPU X5450 (4x 3.0GHz) 64bit computer with 32 GB RAM running the Ubuntu 16.04 operating system.

B. Results

1) Resource Usage: As the first set of experiments, we evaluate for every basic block the minimal number of PUs required in SO-SCAD and classic SCAD architectures to execute the basic block without any dup and swap overhead (recall that only 1 PU is required in DO-SCAD to execute any basic block without overhead). To derive the minimal number
of PUs, we assume unit latencies for all nodes in the basic block. Then, we simply set the desired execution time to a large enough value (by adding up latencies of all nodes in the DAG) so that execution time does not affect satisfiability of the SMT problem. A wrapper function implemented around the Z3 SMT solver [15] finds minimal numbers of PUs by incrementing the number of PUs \( p \) until a solution is found for the SMT problem.

Figure 6a shows the minimal number of PUs required in both SO-SCAD and classic SCAD architectures in terms of the number of nodes in basic blocks. Both average and worst case values are shown. The minimal number of PUs increases with increasing number of nodes. Note that in the worst case, only 3 PUs are required in a SCAD machine to execute any basic block up to size 12. However, 6 PUs are needed in a SO-SCAD machine for 8 to 12-node basic blocks. On an average, classic SCAD requires 11% less PUs compared to its statically ordered counterpart. However, note that this difference grows with increasing number of nodes in the basic block. For 12-node basic blocks of different levels, the minimal number of PUs is shown in Figure 6b. The minimal number of PUs increases with an increasing number of levels. Again, note that in the worst case, the minimal number of PUs required in a SO-SCAD machine is twice as that required in a classic SCAD machine for 6 to 11-levels, 12-node basic blocks.

2) Performance: As the second set of experiments, we determine the minimal time taken to execute the basic blocks on all SCAD machine variants. Clearly, the minimal number of PUs required in SCAD is less than or equal to that required in a SO-SCAD. For fairness, we assume the same number of PUs in all SCAD machines. This number is given by the minimal number of PUs required in a SO-SCAD machine to execute the basic block without any computational overhead. Again, a wrapper function is implemented around the SMT solver that derives minimal execution time using a binary search. To study the effect of different node latencies on execution times, we conduct these experiments on three sets of input data:

1) Abstract-time: Assuming unit latencies for all DAG nodes gives the degree of ILP exploited in abstract time (or steps)
2) Real-time: With probabilities of 10% and 5%, we assign latencies of 5 clock cycles (for example multiplication) and 10 clock cycles (for example division) for non-load and non-store DAG nodes while retaining unit latency for other nodes.
3) Real-time*: We maintain the latencies derived for non-load and non-store nodes in the real-time case. A cache miss latency of 10 cycles and cache (store buffer) hit latency of 1 cycle is used for load nodes. All store node latencies are fixed to 1 cycle to take into account the presence of a store buffer that is available in most recent architectures. Furthermore, a cache hit probability of 90% was assumed. Average of minimal time taken to execute DAGs of different sizes (number of nodes) with different latency configurations on all SCAD machine variants is shown in Figure 7.

Note that the ILP exploited by SO-SCAD machines deviates from maximal ILP (exploited by DO-SCAD machine) with increase in basic block size. Although the deviation in the use of ILP by SO-SCAD becomes slightly more apparent when node latencies are increased (note that same scale is used in abstract-time, real-time, and real-time* cases), surprisingly neither the memory access node latencies nor other node latencies have any considerable effect on the relative use of ILP by different SCAD machine variants. Importantly, ILP exploited by our classic SCAD machine stays near to maximal ILP in all scenarios. This is due to more relaxed variable order or buffer constraints for SCAD compared to the SO-SCAD architecture. Therefore, the execution paradigm adopted by our classic SCAD architecture can maximize the use of ILP by making an intelligent compromise on hardware complexity while maintaining scalability.

C. Feasibility
Average times taken by the SMT solver to derive schedules for DAGs of different sizes on different SCAD machine variants are shown in Figure 8. Only abstract-time and real-time
Fig. 7. Average minimal time to execute basic blocks of different sizes with (a) unit node latencies (abstract-time), (b) realistic node latencies (real-time), and (c) realistic node latencies (real-time*) on SO-SCAD, classic SCAD, and DO-SCAD machines.

Fig. 8. SMT solver runtime to compile basic blocks of different sizes with unit node latencies (abstract-time) and realistic node latencies (real-time) for SO-SCAD, classic SCAD, and DO-SCAD architectures.

configurations are shown to avoid cluttering the graph while the real-time* configuration follow similar patterns. Note that in all latency configurations, the SMT solver takes longest to derive schedules for the SO-SCAD machine, and the schedules for DO-SCAD machines are obtained in shortest time due to absence of any buffer constraints. The buffer constraints 8 for classic SCAD are more relaxed (less strict) compared to the buffer constraints 7 for SO-SCAD, and is therefore solved in less time by SMT solvers. For classic SCAD, DAG sizes up to 12 nodes were successfully processed with maximum average times of 12, 37 and 44 seconds for abstract-time, real-time, and real-time* configurations, respectively.

VI. RELATED WORK

The RAW machine [7] consists of processor cores (every core containing a register file) that are arranged in a 2D tiled architecture with routers between them. The compiler statically schedules programs to run across all processor cores. Wavescalar [20] and TRIPS [8] are based on the explicit dataflow graph execution (EDGE) paradigm. Both fetch blocks of instructions and execute them on an array of PUs with buffers at their inputs and outputs. While TRIPS compilation [8] relies on register accesses to execute instruction blocks, Wavescalar does not use registers at all. However, Wavescalar requires complex tag-matching hardware similar to the DO-SCAD architecture. Similar to SCAD, transport-triggered architectures (TTAs) [21] are programmed by move instructions where computation is performed as a side effect whenever new inputs arrive at a PU. Outputs of PUs are connected to the inputs using an interconnect network. The compiler is responsible not only for scheduling these moves, but also for bundling independent moves where each bundle is executed by the hardware in one step. Even though many code generation techniques are proposed for TTAs [9] that bypass the use of registers, TTAs still need register files to store some intermediate results since PUs in TTA have registers instead of buffers at their input and output ports. The MOVE-Pro architecture [22] improves the freedom of bypassing by employing multiple registers at the outputs of PUs along with other optimizations to the TTA architecture. The statically scheduled Mill architecture [23] uses a fixed-length FIFO (called Belt) to store operands and results of execution and this way removes the general purpose register file completely. Old results are pushed out when the new ones are enqueued. However unlike the buffers in SCAD machine, the PUs in Mill CPU may directly read operand value from any location in the Belt. The execution paradigm of the Synchronous Transfer Architecture (STA) [10] is closer to SCAD with buffers at the outputs of PUs and a configurable interconnect that allows PUs to receive operands for execution from these output buffers. Both operations to be executed on PUs and the interconnect configurations are encoded in instructions and execute a static schedule of instruction bundles (like in standard VLIW). Code generation proposed for STAs [10] bypasses register usage only within adjacent instruction bundles. In Flexcore [11], a 91-bit native instruction set architecture encodes both the operations to be executed on individual PUs and connections enabled by flexible network that connects PUs. Similarly, in
the explicit datapath wide single instruction multiple data (SIMD) architecture [24], a set of PUs are arranged in a circular layout where each unit is connected to its left and right neighbors. It is programmed using very long instructions that encode for each PU, the source and destination of its operands. Although compilation for both Flexcore [11] and explicit datapath wide SIMD [24] tries to maximize bypassing, registers are still required while our queue-based code generation for SCAD completely eliminates the use of registers.

VII. CONCLUSIONS

In this paper, we considered three different execution paradigms for exposed datapath architectures with buffered PUs, described in the context of SCAD architectures, namely the statically ordered SCAD (SO-SCAD), the classic SCAD, and the dynamically ordered SCAD (DO-SCAD) architectures. These variants differ in the way corresponding operands are found for executing an instruction: While the corresponding operands are determined in SO-SCAD by the corresponding places in input FIFO buffers, the places in these buffers are identified in the classic SCAD architecture by the address tags registered from the program. Meanwhile in the DO-SCAD, operands are associated with unique tags and tag matching is used to find corresponding operands.

We compared and discussed the hardware complexity of these three architectures and then adapted our SMT-based code generation [14] for these variants. It is instructive to see which parts of the SMT-encoding was influenced by the SCAD paradigms. Moreover, the different variants were compared empirically in terms of minimal resource usage and maximal performance. Experimental results clearly show that the execution paradigm in our original SCAD architecture maximizes the use of ILP while maintaining hardware scalability.

On the compiler front, it is necessary to develop heuristics to compile larger basic blocks. More importantly, the control flow boundaries in programs must be dealt with in an effective way (minimizing memory accesses) to efficiently apply our queue-based code generation to the entire program, considering representative benchmarks in embedded systems. It is also necessary to analyze buffer sizes required in SCAD architectures to execute queue-based SCAD code. On the hardware front, SCAD machines must be compared with traditional superscalar and VLIW machines to reveal both area and power consumption benefits.

REFERENCES