Optimal Compilation for Exposed Datapath Architectures with Buffered Processing Units by SAT Solvers

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

Abstract—Conventional processor architectures are restricted in exploiting instruction level parallelism (ILP) due to the limited number of available registers in their instruction sets. Therefore, recent processor architectures expose their datapaths so that the compiler not only schedules instructions to functional units, but also takes care of directly moving values between functional units avoiding the need of registers at all. However, the current compiler technology is still based on classic register architectures where a nearly optimal register mapping is the key for the quality of the generated assembly code.

The Synchronous Control Asynchronous Dataflow (SCAD) architecture is a new exposed datapath architecture where processing units (PUs) are equipped with first-in first-out (FIFO) buffers at their inputs and outputs. Code generation for SCAD machines can be done as known for classic queue machines to completely eliminate the use of registers, and to improve the degree of exploited ILP. However, the SCAD code generated this way is not optimal since compared to queue machines, SCAD machines can contain many PUs and buffers which offers the compiler more freedom to reduce unnecessary computational overhead. In this paper, we map the SCAD code generation problem to a satisfiability problem, and then use SAT solvers to generate code without overhead that works with the minimal number of PUs. The generated optimal code will serve as a reference to judge the quality of heuristics that will be finally used in SCAD compilers.

I. INTRODUCTION

Compared to high-performance computing systems, the shift from single core to multicore architectures proceeds slower for embedded systems. One reason for this is that for most embedded systems, one has to ensure in addition to the correctness of multithreaded programs also some non-functional properties like real-time constraints. Due to the required thread synchronization [1] and weak memory consistency issues [2]–[4], both the correctness problem and the timing analysis [5] becomes more complicated for multicore architectures. In hard real-time embedded systems, it is however paramount to guarantee both safe and tight upper bounds on the execution time to meet deadlines and to ensure proper hardware utilization. Usually, these real-time bounds ask for higher performance of the processor architectures which is given by multicore architectures, but with the additional problems mentioned above.

An alternative for improving the performance of processor architectures is still to increase the use of instruction-level parallelism (ILP) [6]–[8] contained in the programs. In particular, code that has been automatically generated in model-based embedded system design offers usually a large amount of ILP. The two main approaches to exploit ILP in modern processor architectures are dynamic scheduling as introduced by the Tomasulo algorithm [9] and static scheduling as used by very long instruction word (VLIW) processors [10], [11] and other embedded processors. In dynamic scheduling, the processor keeps track of data dependencies of the instructions and schedules them for execution on its functional units (out-of-order execution). Dynamic scheduling simplifies compilers, but requires increased chip size and power consumption since in addition to the computation also the instruction scheduling is done by the processor at runtime. Static scheduling avoids this since the compiler handles all scheduling decisions at compile-time. To this end, many sophisticated compiler techniques like trace scheduling [11], [12] and software pipelining [13], [14] schedule instructions across basic blocks, often finding this way enough independent instructions.

However, both traditional variants of ILP face unavoidable limits on their further scalability. One inherent reason for these limitations is simply given by the use of a limited number of registers that are encoded even in the instruction sets of the processor: Most current processor architectures are so-called load/store architectures where only load and store instructions have access to the main memory, while all other instructions use registers as operands and target addresses. The main reason for the success of load/store architectures is that the execution time of memory accesses did not improve as fast as that of other instructions, so that the number of memory accesses had to be limited as much as possible. A simple way to reduce them is to load values into local memories like registers and to work on the local copies as long as possible. While the introduction of registers was a good idea for sequential processors, it imposes new limits for ILP: The use of load and store instructions in conventional architectures due to limitation in number of available registers breaks the instruction level concurrency in programs, thus limiting the use of ILP. Furthermore the number of available registers is the maximal number of instructions that can be executed at a time in VLIW processors, since all instructions in an instruction bundle have to write their result to different registers.
Consider, for example, a 3-level perfect expression tree, as illustrated in Figure 1. By the Sethi-Ullmann algorithm [15], at least 3 registers are required to execute the expression tree on a RISC machine without loading/storing data from/to memory. Consider the execution of the expression tree on a superscalar and VLIW machine each containing 2 registers. Inevitably, the code generator has to store the output from \( +_1 \) to memory and load the data back to a register once \( +_2 \) is completed. The final data flow graph that gets executed on the superscalar machine is shown in Figure 2. Also shown is the VLIW code obtained by packing the RISC instructions in bundles. Clearly, the superscalar and VLIW machines take 5 steps to completely execute the program (since there are 5 levels in the data flow graph where all instructions belonging to one level can be fired in parallel), irrespective of the number of functional units. However, if the parallelism offered by the expression tree is utilized at the fullest, it can be executed in only 3 steps where all nodes per level are executed at the same time.

Increasing the number of programmer-accessible registers is however difficult: First, this number is directly encoded in the instruction set. Changing it requires corresponding changes in the compilers and operating systems (as happened during the shift from 32 bit to 64 bit architectures). Second, increasing the number of registers and functional units quickly leads to a bottleneck in connecting registers and functional units on the chips: If every register can be accessed by every functional unit, the chip size increases quadratically with their number. For the latter reason, clustered architectures [16] have been introduced where the registers are partitioned into clusters and the functional units have then only access to some of the register clusters.

Recent processor architectures somehow all try to eliminate the use of registers in that the architectures expose not only the processing units, but also all data paths between them. They are often called exposed datapath architectures. Examples are Raw [17] with the commercial variant Tilera [18], WaveScalar [19], TRIPS [20], Flexcore [21], explicit datapath wide SIMD [22] and the Transport-Triggered architectures (TTAs) [23]. These architectures provide a large number of processing units and the compiler is not only responsible to schedule the instructions to these processing units but also to move data from one processing unit to another. This way, the use of registers for storing intermediate results can be avoided. While these architectures have been studied already to a great detail, current compiler technology is still based on the classic register architectures where a nearly optimal register mapping is the essential key for the quality of generated assembler code. We notice that more adequate code generators are required for exposed datapath architectures that must focus on the capability of these architectures to directly move values from one processing unit to another.

Synchronous Control Asynchronous Dataflow (SCAD) [24], [25] is a new paradigm for exposed datapath architectures where each processing unit is equipped with queues for storing its input and output values (see Figure 3). Output queues of all processing units are connected to input queues via a Data Transport Network (DTN). SCAD machines execute a sequential program consisting of move instructions whose effect is to transport a value from the head of an output queue of a processing unit to the tail of an input queue of the same or another processing unit. A processing unit can ‘fire’ if it finds enough operands in the heads of its input queues to execute the operation. Since this point of time is independently determined of other processing units, the dataflow is asynchronous tolerating variable latencies of processing units. The registration of move instructions at the queues is however synchronous to guarantee a correct execution. Similar to other exposed datapath architectures, the main advantage of SCAD is its ability to avoid the need of programmable registers and therefore to break the limits imposed by register files for ILP. A second advantage is the simple extension to application-specific processing units without having the need to change the instruction set.

Initially, we used a code generation technique for SCAD architectures based on classic queue machines that evaluate expression trees along a breadth-first traversal. The breadth-first traversal along expression trees ensures that the operands are found in the correct order in the queue and therefore ensures that there is no need for an additional memory like a register file. However, to follow this code generation for general directed acyclic graphs (DAGs), we have to levelize and planarize the DAG which introduces overhead due to the use of dup (duplicate) and swap operators. While this is required for queue machines, it is not always required for SCAD machines: For a SCAD machine with multiple processing units, we observe that the required number of dup and swap operations can be reduced depending on the
chosen mapping of processing units to nodes of the DAG, and even non-level non-planar DAGs can lead to code without any overhead. It is desirable to find the optimal code to execute a given DAG on a given SCAD machine, where we refer to optimality in terms of minimum computational overhead or minimum program length.

In this paper, we map the decision version of optimal SCAD code generation to a satisfiability problem, which in general can be used for any exposed datapath architecture with buffered processing units. We use a custom SAT solver to obtain optimal SCAD code and study the feasibility of this approach in terms of DAG sizes it can handle. The generated optimal code will serve in future as a reference to judge the quality of heuristics that will be used in SCAD compilers. Moreover, we prove this way the membership of this code generation problem in NP, and in future work, we will also study the completeness of this problem.

The outline of the paper is as follows: Section II briefly compares the SCAD paradigm with other exposed datapath architectures. Section III describes the organization and functionality of SCAD architectures. Code generation for SCAD architectures based on queue machines is explained in Section IV. Mapping of optimal SCAD code generation to satisfiability problem is described in Section V and the corresponding experimental results are provided in Section VI. The final section summarizes the paper and concludes by mentioning the future work.

II. RELATED WORK
There are already many different kinds of exposed datapath architectures: Transport-triggered architectures (TTAs) [23] use registers at output and input ports of processing units unlike queues in our SCAD machines. The output ports are connected to input ports using an interconnection network. Similar to SCAD machines, TTAs only need move instructions that transport values from output ports to input ports, and computation is done as a side effect of the data transport. The compiler is responsible not only for ordering these move instructions in a sequence, but also for packing independent moves into parallel bundles, where each set of moves can be executed by the hardware in one step. This is an extreme case of static scheduling, since in addition to instruction scheduling also the allocation of processing units is done at compile time. Having move instructions only allows application-specific processing units both in TTA and SCAD.

The RAW machine [17] consists of processing units that are arranged in a 2D tiled architecture with routers between them. It uses compiler-detemined issue of operations and data transports via inter-ALU routers, while in SCAD, operations are executed dynamically in dataflow order which allows arbitrary latencies of processing units.

Wavescalar [19] and TRIPS [20] are both based on the explicit dataflow graph execution (EDGE) paradigm. Both fetch (basic) blocks of instructions (called frames in TRIPS and waves in Wavescalar) and execute them on an array of processing units. In TRIPS, the compiler maps operations to be executed to the processing units and execution is carried out in dataflow fashion (static placement dynamic issue [26]). In Wavescalar, both placement and issue of operations are performed during runtime. Unlike TRIPS, Wavescalar uses dynamic dataflow execution using wave numbers as tags for matching operands for a function application across waves. Due to this, Wavescalar could completely abandon the program counter inherited from von Neumann execution, which TRIPS relies on to fetch the sequence of frames of instructions. Also in SCAD, we use the program counter to fetch the sequence of move instructions.

In Flexcore [21], processing units are connected by a flexible network via a set of control signals. A 91-bit native instruction set architecture (N-ISA) encodes connections enabled by the flexible interconnect status of control signals and operations to be executed on individual processing units. Similarly, in the explicit datapath wide single instruction multiple data (SIMD) architecture [22], a set of processing units are arranged in a circular layout where each unit is connected to its left and right neighbors. There is a control processor that can talk to all processing units. It is programmed using very long instructions that encode for each execution unit the source and destination of its operands in addition to the role of the control processor. Even though code generation for the above architectures obviously utilize data transport of intermediate results without using registers, it is still based on traditional compiler technology that optimizes the use of registers.

III. SCAD ARCHITECTURES
A. Organization
The organization of processing units in a SCAD architecture is shown in Figure 3. Each processing unit (PU) has queues (or first-in-first-out (FIFO) buffers) at its input and output ports. Input and output buffers are connected to two interconnection networks: There is the move-instruction bus (MIB) (given in red color) which is used to synchronously send values from the control unit to the PUs, and the 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.

Buffers hold pairs $(adr, val)$ of entries. For an input buffer, $adr$ is the address of the output buffer of the PU that produced or that will produce the value $val$. An entry $(adr, \bot)$ with the special value $\bot$ 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 $(adr, \bot)$ with the special value $\bot$ 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 $adr$.

B. Execution of a Move Program
SCAD is programmed by a sequence of 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
Banyan, or Beneš networks can be used as DTN. These and sockets to more complex parallel networks such as Omega, an interconnection network ranging from a simple set of buses in a SCAD architecture may implement any function with only two-input one-output PUs are shown in Figure 3, 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 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 \((src, \bot)\) to its tail, and the output buffer with address \(src\) will add the entry \((tgt, \bot)\) to its new tail. If one of the two buffers should be full, it will signal this via a feedback signal \(\text{fullBuffer}\) 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. Therefore, the control flow is synchronous and the dataflow is carried out asynchronously and in dataflow order. It is important to note that all move instructions 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 its input and output buffers.

If a processing unit will find entries \((adr_1, x_1), \ldots, (adr_m, x_m)\) with \(x_i \neq \bot\) 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 \((tgt, \bot)\) 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 \((\bot, y_i)\) 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 \((src, \bot)\) 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 \((\bot, y_i)\). If so, it will replace the one closest to its head with the address \((tgt, y_i)\). Otherwise, it will create a new tail \((tgt, \bot)\), if there is still space available. Otherwise, it will signal \(\text{fullBuffer}\) 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 \((src, \bot)\) are put in order in the input buffer \(tgt\) whenever a move instruction \((src, tgt)\) is seen on the MIB, and an available entry \((src, \bot)\) 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, and whether the SU and the LU have to be synchronized depends on a chosen weak memory model.

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 head associated with the program counter so that the next move instruction in the program order
is fetched in the subsequent clock cycle.

Note that we have not mentioned register files or other local storage although it is possible to use them just like any other PU in the SCAD architecture. In the following section, we motivate a code generation technique that does not require local memory other than the buffers. In other words, it utilizes the capability of the SCAD architecture to move values from one PU’s output buffer to the same or another PU’s input buffer.

IV. SCAD CODE ORGANIZED BY QUEUE MACHINE

A queue machine [27], [28] reads operands for executing an operation from the head of a queue and adds the results to the tail of that queue. The architecture of a queue machine is shown in Figure 4.

![Fig. 4. Architecture of a queue machine](image)

<table>
<thead>
<tr>
<th>Queue Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load adr,n</td>
<td>Load data from memory address adr and add n copies of the loaded value to the tail of the queue.</td>
</tr>
<tr>
<td>store adr</td>
<td>Store the value from the head of the queue to the memory address adr.</td>
</tr>
<tr>
<td>opcode n</td>
<td>Dequeue necessary operands from the head of the queue to execute the operation opcode and add n copies of the result to the tail of the queue.</td>
</tr>
<tr>
<td>swap</td>
<td>Dequeue two operands from the head of the queue, swap them, and add them to the tail of the queue.</td>
</tr>
<tr>
<td>dup n</td>
<td>Dequeue one operand from the head of the queue, and add n copies of it to the tail of the queue.</td>
</tr>
<tr>
<td>goto PC,L</td>
<td>Unconditional Branch: Transfer the control from PC to PC+L.</td>
</tr>
<tr>
<td>ifGoto PC,L</td>
<td>Conditional Branch: Transfer the control from PC to PC+L if the head of the queue holds else PC+1</td>
</tr>
</tbody>
</table>

TABLE I

A LIST OF QUEUE INSTRUCTIONS

A. Code Generation for Queue Machine

Generating a queue program to evaluate an expression tree is done by a breadth-first traversal of the tree [28] as shown in Figure 5. 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. The queue program for the expression tree and the contents of the queue after executing each instruction of that program is also shown in Figure 5. A list of queue instructions is listed in Table I.

<table>
<thead>
<tr>
<th>given expression</th>
<th>levelized expression</th>
<th>planar expression</th>
<th>level-planar expression</th>
<th>queue program</th>
</tr>
</thead>
<tbody>
<tr>
<td>load x1,1</td>
<td>[x1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load x2,1</td>
<td>[x1,x2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load x3,1</td>
<td>[x1,x2,x3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add 1</td>
<td>[x3,x1+x2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load x4,1</td>
<td>[x3,x1+x2,x4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>load x5,1</td>
<td>[x3,x1+x2,x4,x5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul 1</td>
<td>[x3*(x1+x2),x4,x5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>div 1</td>
<td>[x3*(x1+x2),x4,x5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub 1</td>
<td>[x3*(x1+x2),x4,x5]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE II

A SCAD MACHINE IMPLEMENTATION

B. SCAD Code from Queue Code

To generate SCAD code from queue code, we map each queue instruction to a sequence of move instructions for the universal SCAD machine as listed in Table II. A universal SCAD machine is a SCAD machine with a single universal processing unit. It has one output queue out to store the result.
of each operation and four input queues: \textit{inp1} to store the first operand, \textit{inp2} to store the second operand, \textit{opc} to store the operation to be executed, and \textit{cps} to store the number of copies of a result to be added to the output queue. Note that the contents of the queue in the queue machine and the output queue in the universal SCAD machine will be one and the same after execution of each queue instruction on the queue machine and the corresponding move instructions on the universal SCAD machine [25]. It is not difficult to adapt the mapping for a SCAD machine with multiple processing units, given a mapping from each queue instruction to that processing unit which executes the queue instruction.

Since a queue machine has one central queue, it requires a total ordering of all input values read in a level in the level-planar DAG so that all output values from this level are available in the right order for execution of operations at the next level. However, a SCAD machine contains multiple processing units and many queues. Therefore, it has more freedom to order values compared to a queue machine. A SCAD machine requires only a total ordering of those values that pass through the same queue. As a result, the SCAD machine might not require as much overhead (\textit{dup} and \textit{swap} operations) as the queue machine [25]. Hence, the SCAD code generated from queue code is not optimal.

1) Example: Consider a SCAD machine containing one load-store unit (\textit{lsu}), one adder (\textit{add}) and one multiplier (\textit{mul}). Table III lists a sequence of SCAD move instructions to execute the DAG in Figure 6 without using any \textit{dup} and \textit{swap} operations, while the queue machine required 3 \textit{dup} and 1 \textit{swap} operations to execute the same DAG.

<table>
<thead>
<tr>
<th>Queue Instr</th>
<th>Corresponding SCAD Move Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>load \textit{adr}</td>
<td>\textit{adr}→\textit{inp1}; load→\textit{opc}; \textit{n}→\textit{cps};</td>
</tr>
<tr>
<td>store \textit{adr}</td>
<td>\textit{adr}→\textit{inp1}; \textit{out}→\textit{inp2}; store→\textit{opc};</td>
</tr>
<tr>
<td>\textit{op n}</td>
<td>\textit{out}→\textit{inp1}; \textit{out}→\textit{inp2}; \textit{op}→\textit{opc}; \textit{n}→\textit{cps};</td>
</tr>
<tr>
<td>\textit{swap}</td>
<td>\textit{out}→\textit{inp1}; \textit{out}→\textit{inp2}; \textit{swap}→\textit{opc};</td>
</tr>
<tr>
<td>\textit{dup n}</td>
<td>\textit{out}→\textit{inp1}; \textit{dup}→\textit{opc}; \textit{n}→\textit{cps};</td>
</tr>
<tr>
<td>\textit{goto PC,L}</td>
<td>\textit{pc}→\textit{inp1}; \textit{goto}→\textit{opc}; \textit{L}→\textit{cps};</td>
</tr>
<tr>
<td>if\textit{Goto PC,L}</td>
<td>\textit{pc}→\textit{inp1}; \textit{out}→\textit{inp2}; \textit{if\textit{Goto}→\textit{opc}}; \textit{L}→\textit{cps};</td>
</tr>
</tbody>
</table>

\textbf{TABLE III}

\textbf{SCAD PROGRAM TO EXECUTE THE DAG IN FIGURE 6 ON A SCAD MACHINE WITH MULTIPLE PROCESSING UNITS}

\textbf{Unnecessary \textit{dup} overhead}: The \textit{dup} operator \(D_1\) (in Figure 6) is not required since the output from the node \(+1\) is accessible from the adder unit output queue without duplicating the values \(x_2\) that resides in the load-store unit output queue. As a general rule, a value at the head of an output queue need not be duplicated as long as all the values that have to be transported before this value are accessible from other output queues.

\textit{Unnecessary swap overhead}: Similarly, the \textit{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. Also, both the moves enqueue values to different input queues, namely the first input queue of the adder unit and the second input queue of the multiplier unit. As a general rule, consider the four possibilities shown in Figure 7 for the moves associated with a crossing edge where the values \(val_1\) and \(val_2\) need to be transported from one or more output queues to one or more input queues. Assume that if the values \(val_1\) and \(val_2\) are enqueued to the same input queue, the move instructions must store the values in the order (from head to tail of the queue) \([val_2, val_1]\) in the input queue.

We observe that except for case (a), we can generate move code for the edge crossing without introducing any \textit{swap} operation. In case (b), this is possible as we can access \(val_2\) before \(val_1\) since both values reside in different output queues. We simply move the value \(val_2\) first, and then move the value \(val_1\). If both the values are enqueued to different input queues (cases (c) and (d)), we can always generate valid \textit{swap-free} move code irrespective of the order of access of values from the output queue(s). In case (c), since both values reside in same output queue in the order \([val_1, val_2]\), we first move \(val_1\) to the relevant input queue followed by \(val_2\) to the other input queue. In case (d), since both values reside in different output queues, we may schedule both the moves in any order.

\textbf{V. OPTIMAL SCAD CODE GENERATION}

Increasing numbers of \textit{dup} and \textit{swap} operations not only degrade the performance, but also increase the code size and power consumption of SCAD machines. Therefore, it is
desirable to obtain optimal code for a given DAG and a given SCAD machine, where optimal refers to a minimum number of additional operations. In this section, we map the decision version of the optimal SCAD code generation problem to an equivalent satisfiability problem.

A. Problem Statement
To formulate the precise problem, we assume that the following is given:

- a basic block (DAG) in the form of three-address code in static single assignment (SSA) form, i.e.,
  \[ x_{\text{tgt}(0)} = x_{\text{srcL}(0)} \odot 0 \ x_{\text{srcR}(0)} \]
  \[ \vdots \]
  \[ x_{\text{tgt}(\ell - 1)} = x_{\text{srcL}(\ell - 1)} \odot \ell - 1 \ x_{\text{srcR}(\ell - 1)} \]
  for some variables \( V := \{x_0, \ldots, x_{n-1}\} \), where \( \odot \) denotes some binary operation
- a SCAD machine with one load-store unit and \( p \) universal processing units that may execute any binary operation

Determine if the basic block can be executed on the SCAD machine without any \textit{dup} and/or \textit{swap} computation overhead. If so, determine the schedule of the basic block on the PUs of the SCAD machine.

B. Mapping to SAT
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 side. This defines three different kinds of variables:

- target variables \( V_{\text{tgt}} \) are those that occur as left hand sides
- source variables \( V_{\text{src}} \) are those that occur on the right hand sides
- load variables \( V_{\text{ld}} \) are those that only occur as right hand sides \( V_{\text{ld}} := V_{\text{src}} \setminus V_{\text{tgt}} \)

If a variable is in \( V_{\text{src}} \setminus V_{\text{tgt}} \), then we assume that all its read operations occur after its unique write operation (no shadowing of variables).

Furthermore, the basic block can also be partitioned into levels.

- level 0 are all instructions that only read variables \( V_{\text{ld}} \)
- their target variables \( V_{\text{def}}^0 \) are written by this level
- level \( j + 1 \) are all instructions that only read variables \( V_{\text{ld}} \cup \bigcup_{i=0}^{j} V_{\text{def}}^i \) and where at least one variable of \( V_{\text{def}}^i \) is read

If viewed as an expression tree, the level 0 are the leaf nodes, level 1 are the nodes that are only connected to leaves and so on. All instructions in one level are independent and can be fired in parallel. The levels are also defined by ASAP (as soon as possible) scheduling of the basic block.

1) Relations: Assuming that PU 0 is the one load-store unit in the given SCAD machine with \( p \) universal processing units \( \{1, \ldots, p\} \), we define the following relations to determine the allowed schedules for the basic block on the SCAD machine:

- PU assignment \( \alpha_{i,j} \) for \( x_i \in V \) and \( j \in \{0, \ldots, p\} \)
  - \( \alpha_{i,j} \) means that \( x_i \in V \) is produced by PU \( j \)
  - we may fix that all \( x_i \in V_{\text{ld}} \) are produced by PU 0 (load/store unit)
  - this determines the instructions of the basic block executed by PU \( j \)

- variable order \( x_i \prec x_j \) for \( x_i, x_j \in V \)
  - \( x_i \prec x_j \) means that \( x_i \) and \( x_j \) occurs in some buffer in that order, which includes
  - production order: \( x_i \) and \( x_j \) are target variables produced by the same PU. Therefore they both occur in the output buffer of that PU.
  - consumption order: \( x_i \) and \( x_j \) are source variables (same argument) of instructions assigned to the same PU. Therefore they both occur in the corresponding input buffer of that PU.

2) Constraints: In the following, we set up boolean constraints, whose conjunction provides a constraint formula for the given basic block. Every satisfying assignment of that formula is a valid schedule for the given SCAD machine and vice versa.

\textit{Strict order relation} \( \prec \): The variable ordering \( \prec \) is both transitive and irreflexive, which defines the constraints 1 and 2 respectively. Both together implies that \( \prec \) is acyclic, that defines constraint 3.

\[
\bigwedge_{x_i, x_j, x_k \in V} x_i \prec x_j \land x_j \prec x_k \rightarrow x_i \prec x_k \tag{1}
\]
\[
\bigwedge_{x_i \in V} \neg x_i \prec x_i \tag{2}
\]
\[
\bigwedge_{x_i, x_j \in V} x_i \prec x_j \rightarrow \neg x_j \prec x_i \tag{3}
\]

\textit{Buffer constraints}: A total variable ordering must exist for those variables \( x_i \) that are at some time in the same buffer. 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 input buffers irrespective of the ordering of operand values in some buffers. Assume that the instructions are executed on the same PU \( k \). 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 \( x_{\text{srcL}(i)} \) (respectively \( x_{\text{srcL}(j)} \)) before the operand \( x_{\text{srcR}(i)} \) (respectively \( x_{\text{srcR}(j)} \)), 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. However if both operands are produced by the same PU, we must enforce the ordering \( x_{\text{srcL}(i)} \preceq x_{\text{srcL}(j)} \) (respectively \( x_{\text{srcL}(j)} \preceq x_{\text{srcL}(i)} \)). Similar argument applies for the right operands.

We introduce a new relation \( \beta_{i,j} \) that is true if variables \( x_i \) and \( x_j \) are produced by the same PU and false otherwise. i.e.

\[
\beta_{i,j} = \bigvee_{k=0}^{p} \alpha_{i,k} \land \alpha_{j,k} \tag{4}
\]
Note that $\alpha_{i,k}$ means that $x_i$ is produced in the target buffer of PU $k$. Now constraint 5 ensures that if the two instructions are executed on the same PU $k$, then the variables will occur in the right order in the input and output buffers.

\[
\bigwedge_{i,j=0}^{\ell-1} \beta_{\text{tgt}(i),\text{tgt}(j)} \rightarrow \begin{cases} x_{\text{tgt}(i)} \prec x_{\text{tgt}(j)} \\
\beta_{\text{srcL}(i),\text{srcL}(j)} \rightarrow x_{\text{srcL}(i)} \preceq x_{\text{srcL}(j)} \\
\beta_{\text{srcR}(i),\text{srcR}(j)} \rightarrow x_{\text{srcR}(i)} \preceq x_{\text{srcR}(j)} \end{cases}
\]

\[
\bigwedge_{i,j=0}^{\ell-1} \beta_{\text{srcL}(i),\text{srcL}(j)} \rightarrow x_{\text{srcL}(i)} \preceq x_{\text{srcL}(j)} 
\bigvee
\beta_{\text{srcR}(i),\text{srcR}(j)} \rightarrow x_{\text{srcR}(i)} \preceq x_{\text{srcR}(j)}
\]

(5)

**Data dependency:** Consider instruction $x_{\text{tgt}(i)} = x_{\text{srcL}(i)} \odot i$ of the basic block. It requires that operands $x_{\text{srcL}(i)}$ and $x_{\text{srcR}(i)}$ must have already been produced before producing $x_{\text{tgt}(i)}$. Therefore:

\[
\bigwedge_{i=0}^{\ell-1} x_{\text{srcL}(i)} \prec x_{\text{tgt}(i)} \land x_{\text{srcR}(i)} \prec x_{\text{tgt}(i)}
\]

(6)

**Unique PU assignment:** The final constraint 7 demands that every variable has to be produced by one and only one PU.

\[
\left(\bigwedge_{x_i \in \mathcal{V}_d} \bigvee_{k=0}^{p} \alpha_{i,k}\right) \land \left(\bigwedge_{x_i \in \mathcal{V}_d} \bigwedge_{k=0}^{p} \alpha_{i,k} \rightarrow \bigwedge_{j=0}^{p} \neg \alpha_{i,j}\right)
\]

(7)

We may determine that all $x_i \in \mathcal{V}_d$ are assigned to PU 0, then

- replace for all $x_i \in \mathcal{V}_d$ all $\alpha_{i,0}$ with true
- replace for all $x_i \not\in \mathcal{V}_d$ all $\alpha_{i,0}$ with false
- replace for all $x_i \in \mathcal{V}_d$ and $j > 0$ all $\alpha_{i,j}$ with false

$\Rightarrow$ only $\alpha_{i,j}$ remain where $x_i \not\in \mathcal{V}_d$ and $j > 0$

3) **Solution:** In the following, we construct a schedule for the given basic block for execution on the given SCAD machine with a variable ordering $\prec$ and PU assignment $\alpha_{i,j}$ that satisfies the constraints. This also proves that the given constraints are sufficient.

To derive the program executed on each PU, simply extract the instructions whose target variable is assigned to that PU and sort them according to the target variables using $\prec$. From the buffer constraints, it is guaranteed that both the left and right hand side arguments will occur ordered with respect to $\prec$ in the input buffers with this schedule. Now, the execution of the basic block can proceed level-wise starting with level 0:

- the variables read in this level are available in output buffers in order $\prec$
- 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 for each input buffer, the order $\prec$ is also used
- 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 the right order.

VI. EXPERIMENTAL RESULTS

We implemented a random basic block generator function that accepts the number of nodes $n$ and number of levels $l$ as input. The basic block is generated by randomly choosing the predecessors of every node ensuring that the DAG has $l$ levels. Clearly for a node basic block, $l := \{1, \ldots, n-1\}$ levels are possible. For every $(n, l)$ pair, 1k basic blocks were generated. A wrapper function around a custom SAT solver based on the famous DPLL algorithm [30] was used to derive the minimum number of processing units required in a SCAD machine to execute the input DAG without any overhead of **dup** and **swap** operations. All the runs are performed on an Intel Core-i5 (4 x 2.67 GHz) desktop computer with 8 GB RAM running the Ubuntu 14.04 operating system.

As expected, the time taken (both average case and worst case) by the SAT solver to produce the result grows with the number of nodes in the basic block as shown in Figure 8. Increasing the number of nodes leads to an increase in the number of boolean constraints that forms the propositional logic formula. With a timeout of 5 seconds, basic block sizes up to 15 nodes were successfully processed. Therefore, the SAT solver can process basic blocks of practical size. However, generating code for superblocks [11] and hyperblocks [11] will require heuristics (or optimistically a polynomial time optimal algorithm). Figure 9 shows the variation in the average time taken by the SAT solver with varying numbers of levels for basic block sizes $n = \{13, 14, 15\}$. The time taken is neither monotonically increasing nor decreasing. This is because, although the number of boolean constraints increase with more number of levels, the choice of variable ordering decreases since more variable orders are now given by the data dependency leaving lesser choices for the SAT solver to reorder variables to satisfy the constraints.

Figures 10 and 11 show the variation in the minimum number of processing units required by a SCAD machine (both average and worst case) to generate overhead-free code, with the number of nodes and number of levels in the basic block, respectively. The number of processing units increases with both increasing number of nodes and levels. This is more apparent with the increasing number of nodes since it is more probable that a DAG with higher number of nodes lead to
buffer constraints that are not satisfiable with less number of processing units. However, note that even for basic block size 15, only 4 processing units are required in the worst case. Therefore, all practical basic blocks are handled without any overhead using SCAD machines with only 4 processing units.

VII. CONCLUSIONS
Exposed datapath architectures allows the compiler to bypass the use of registers improving the degree of exploited ILP. However the current compiler technology still relies on an optimal register mapping for the quality of the generated assembly code. SCAD is an exposed datapath architecture where inputs and outputs of processing units are buffered in FIFO queues. Code generation for SCAD (or any exposed datapath architecture with buffered processing units) based on classic queue machines completely eliminates register usage, improving the exploited ILP compared to conventional superscalar and VLIW architectures. However, some computational
overhead in the form of \textit{dup} and \textit{swap} operations might be required to generate SCAD code for basic blocks represented by non-level non-planar DAGs. In this paper, by mapping to a satisfiability problem, we determined the minimum number of processing units required in a SCAD machine and the corresponding SCAD code, to execute a given basic block with no computational overhead. Experimental results indicate that this technique is applicable for all practical basic blocks (with up to 15 variables in SSA form). Furthermore, a SCAD machine with only as few as four processing units could execute all generated basic blocks without any overhead. The generated optimal code will serve as a reference to measure the quality of heuristics that will be finally used in SCAD compilers.

Research is underway to determine various optimal SCAD code generation algorithms targeting different objective functions. For example, to determine the minimum program length to execute a given DAG on a given SCAD machine, to determine the minimum number of processing units in a SCAD machine to execute a given DAG within a given time, to determine the program to execute a given DAG on a given SCAD machine that minimizes the size of buffers in the SCAD machine, etc. While it is necessary to compare the hardware resources required for prototyping a SCAD machine with those required for comparable superscalar and VLIW machines, it is also important to compare these machines for performance, power consumption and timing predictability.

REFERENCES