Exploring the Potential of Instruction-Level Parallelism of 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—Recent processor architectures expose their datapaths to the compiler so that the compiler not only takes care of scheduling instructions to the available processing units but also of scheduling the data transports between the processing units. Bypassing register usage this way generally allows the compiler to improve the degree of instruction-level parallelism. However, the current compiler technology is still based on code generation done by a depth-first traversal on the syntax trees that makes use of as few registers as possible. Code generators inspired from queue machines can better utilize the register bypassing capability of exposed datapath architectures with buffered processing units.

In this paper, we encode the decision version of optimal code generation as a satisfiability modulo theories (SMT) problem and use SMT solvers to generate optimal code that maximizes the instruction-level parallelism (ILP) for a given number of processing units. Our experimental results clearly demonstrate the potential of exposed datapath architectures to utilize ILP contained in basic blocks to the fullest by relying on our recently suggested queue-based code generation. Second, it is also shown that the queue-based code generation technique produces more efficient code than classic compiler techniques.

1. INTRODUCTION

A. Motivation

Traditional approaches to exploit ILP in processor architectures are 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]. However, both variants of ILP face limits on their further scalability [3]. One inherent reason for these limitations is the limited number of registers as defined by the instruction sets: 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 targets. 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.

However, limited numbers 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 1(a). 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 1(b). However, 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). More memory accesses not only reduce ILP, but also adversely affects the timing-predictability of applications [5] which is an important metric in real-time embedded systems.

Hence, the number of programmer-accessible registers influences the use of ILP. Increasing the number of registers is however difficult: First, this number is directly encoded in the instruction sets. Changing it requires corresponding changes in machines, compilers, and even operating systems. 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 [6] have been introduced where PUs have only access to predefined register clusters.

Recently introduced exposed datapath architectures [7]–[13] propose an interesting alternative by avoiding the use of programmer-accessible registers at all. Instead, these architectures provide a large number of PUs and allow the compiler to move values directly from one PU to another one. Another reason for the recent popularity of exposed datapath architectures is that they also allow the compiler to mitigate communication delays by appropriate instruction placement, which minimizes the physical distance that the data from a producer PU must travel to reach the consumer PUs. Execution
in recent many-core architectures are becoming increasingly dominated by the communication between threads/cores [3], [14]. While exposed datapath architectures have already been studied to a great detail, their current compiler technology still relies on classic code generators where the optimal use of registers is the main focus.

B. Background

We observed in [15] that more adequate code generators are required for exposed datapath architectures and suggested a code generation based on a breadth-first traversal rather than the classic depth-first traversal over the syntax trees. To that end, a special exposed datapath architecture is considered 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 the 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. However, it is also emphasized in [15] that a simple code generation as done for queue machines does not lead to optimal code for exposed datapath architectures, due to the possibility to make use of multiple PUs and buffers in these architectures. In [15], we refer to optimality in terms of minimal computational overhead (i.e., \( dup \) and \( swap \) operations, see Section III). The code generation is therefore encoded in [16] as a satisfiability (SAT) problem to determine the minimal number of PUs required to execute programs without any computational overhead.

C. Contribution

In this paper, we refine the SAT encoding of [16] to an encoding as a satisfiability modulo theories (SMT) problem. The reason for this refinement is our wish to consider execution time as an additional parameter for optimization, so that we can generate important experimental results on the possible use of ILP in exposed datapath architectures with buffered PUs. The experimental results not only demonstrate the superiority of queue-based code generation compared to register-based code generation, but also shows that the queue-based compilation enables exposed datapath architectures to exploit concurrency in programs to the fullest. We use Microsoft’s Z3 SMT solver [17] to obtain optimal code (in this work, we refer to optimality in terms of minimal execution time or maximal use of ILP with given resources without incurring any computational overhead) and study the feasibility of this approach in terms of program sizes it can handle. Moreover, the generated optimal code will serve in future as a reference to judge the quality of heuristics that will be used in compilers.

D. Paper Organization

The rest of the paper is organized as follows: Section II mentions some recent exposed datapath architectures; in particular, the SCAD architecture that we consider for our experiments. Code generation for exposed datapath architectures (in the context of SCAD architectures) is discussed in Section III. The core of this paper, i.e., the encoding of optimal SCAD code generation as an SMT problem and the corresponding experimental results are provided in Sections IV and V, respectively. The final section summarizes the paper and mentions future work.

II. EXPOSED DATAPATH ARCHITECTURES

The RAW machine [7] consists of PUs that are arranged in a 2D tiled architecture with routers between them. RAW uses compiler-determined issue of operations. Wavescalar [9] and TRIPS [10] 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 PUs with buffers at their inputs and outputs. In TRIPS, the compiler maps operations to be executed to the PUs and execution is carried out in dataflow fashion. 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 still uses to fetch the sequence of frames of instructions.

In Flexcore [11], PUs are connected by a flexible network via a set of control signals. A 91-bit native instruction set architecture (N-ISA) encodes both connections enabled by the flexible interconnect status of control signals and the operations to be executed on individual PUs. Similarly, in the explicit datapath wide single instruction multiple data (SIMD) architecture [12], a set of PUs 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 PUs. 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 transports of intermediate results without using registers, it is still based on traditional compiler technology that optimizes the use of registers.

Transport-triggered architectures (TTAs) [13] are exposed datapath architectures that use registers at input and output ports of PUs. The output ports are connected to input ports using an interconnect network. TTAs are programmed by move instructions where computation is performed as a side effect whenever new inputs arrive at a PU. The compiler is responsible not only for scheduling these moves, but also for bundling independent moves where each bundle can be executed by the hardware in one step.

Finally, Synchronous Control Asynchronous Dataflow (SCAD) [15], [16], [18] is a new paradigm for exposed datapath architectures where each PU is equipped with buffers for storing its input and output values. The organization and functionality of SCAD architectures is explained in the following section.
move instruction \( (\text{src}, \text{tgt}) \) from the instruction memory and will broadcast it via the MIB to all PUs. The input buffer with address \( \text{tgt} \) will add the entry \( (\text{src}, \bot) \) to its tail, and the output buffer with address \( \text{src} \) will add the entry \( (\text{tgt}, \bot) \) to its 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 \( (\text{src}, \text{tgt}) \) in the next cycle (it is stalled at this point of time). The data transport related with a move instruction \( (\text{src}, \text{tgt}) \) 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 PU will find entries \( (\text{adr}_i, x_1), \ldots, (\text{adr}_m, x_m) \) 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 \( (\text{adr}_1, x_1), \ldots, (\text{adr}_m, x_m) \) 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 \( (\bot, \bot) \) of output buffer number \( i \) that is closest to the head of the output buffer, i.e., that entry is replaced with \( (\text{tgt}, y_i) \). 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 \( (\text{src}, \text{tgt}, \text{val}) \) consists of the address of the sending output buffer \( \text{src} \), the address of the input target buffer \( \text{tgt} \), and the value \( \text{val} \) that is transported by the message. A message \( (\text{src}, \text{tgt}, \text{val}) \) is created when the output buffer with address \( \text{src} \) has a completed entry \( (\text{tgt}, \text{val}) \) as its head. This message is then sent to input buffer \( \text{tgt} \) via the DTN. When it will finally reach input buffer \( \text{tgt} \), the input buffer will replace the entry \( (\bot, \bot) \) closest to its head with \( (\text{src}, \text{val}) \), 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 \( \text{src} \) will see the move instruction \( (\text{src}, \text{tgt}) \) 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 \( (\text{tgt}, y_i) \). Otherwise, it will create a new tail \( (\text{tgt}, \bot) \), 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 \( (\bot, \bot) \) are put in order in the input buffer \( \text{tgt} \) whenever a move instruction \( (\text{src}, \text{tgt}) \) is seen on the MIB, and an available entry \( (\bot, \bot) \) is completed with the value \( \text{val} \) when a message \( (\text{src}, \text{tgt}, \text{val}) \) arrives.

![Fig. 2. Architecture of a SCAD Processor](image-url)
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 DTU similar to output values of other PUs, and whether and how 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’s 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 describe 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, to the fullest.

III. CODE GENERATION FOR SCAD ARCHITECTURES

In this section, we consider the code generation problem for exposed datapath architectures with buffered PUs (explained in the context of SCAD architectures).

A. Code Generation for Queue Machine

A queue machine (see Figure 3(a)) [19] 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 [19]. 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 an expression tree and contents of the queue after executing each instruction of that program is shown in Figure 3. A list of queue instructions is listed in Table I.

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 [20]. 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 [20] as shown in Figure 4. 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 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load addr</td>
<td>Load data from memory address addr and add n copies of the loaded value to the tail of the queue.</td>
</tr>
<tr>
<td>store addr</td>
<td>Store the value from the head of the queue to the memory address addr.</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>jGoto PC,L</td>
<td>Conditional Branch: Transfer the control from PC to PC+L if the head of the queue holds else PC+L.</td>
</tr>
</tbody>
</table>

TABLE I

LIST OF QUEUE MACHINE INSTRUCTIONS

Fig. 4. An expression DAG with its levelized and planarized versions, the final level-planar expression DAG, and the obtained queue program.

Classic register-based compilers perform a depth-first traversal on expression trees to minimize the use of registers in the generated assembler code. This operation order is inspired from stack machines. In Table II(A), a stack machine organized operation order and the corresponding register assignment for the expression tree in Figure 3(b) are shown. At least 3 registers are needed (since there are 3 levels in the expression tree) to
avoid memory load/store. In Table II(B), a queue machine organized operation order for the same expression tree is shown. In both tables, we have divided the sequence of operations into sets, where all operations in a set can be executed concurrently. Notice that even without the overhead of any load and store instructions (i.e., given sufficient number of registers), it is apparent that the register-based code offers less ILP compared to code organized by a queue machine. However, it should be noted that dynamically scheduled superscalar machines with large enough instruction windows (reservation stations) to hold the entire register-based code could also exploit maximal ILP since execution proceeds in dataflow order for instructions in the reservation station. However, these machines are not scalable due to quadratic growth in hardware complexity since in addition to the computation also the instruction scheduling is done by the processor at runtime.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Register</th>
<th>Operation</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1</td>
<td>x_1</td>
<td>x_2</td>
<td>x_2</td>
</tr>
<tr>
<td>x_2</td>
<td>x_1</td>
<td>x_3</td>
<td>x_3</td>
</tr>
<tr>
<td>x_3</td>
<td>x_1</td>
<td>x_4</td>
<td>x_4</td>
</tr>
<tr>
<td>x_4</td>
<td>x_2</td>
<td>x_1</td>
<td>x_1</td>
</tr>
<tr>
<td>x_1</td>
<td>x_0</td>
<td>x_0</td>
<td>x_0</td>
</tr>
</tbody>
</table>

(A) STACK-BASED OPERATION ORDER AND REGISTER ASSIGNMENT

(B) QUEUE-BASED OPERATION ORDER

B. Generating SCAD Code from Queue Code

To simulate a queue machine by a SCAD machine, we mapped each queue instruction to a sequence of move instructions for a universal SCAD machine as listed in Table III. A universal SCAD machine is a SCAD machine with a single universal PU. It has one output queue out and four input queues: inp1 and inp2 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. Note that the contents of the only queue in the queue machine and the output queue in the universal SCAD machine will be the same after execution of a queue instruction and the corresponding SCAD move instructions [15]. It is not difficult to adapt the mapping for a SCAD machine with multiple PUs, given a mapping from each queue instruction to the PUs of the SCAD machine.

<table>
<thead>
<tr>
<th>Queue Instr</th>
<th>Corresponding SCAD Move Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>load addr</td>
<td>load-&gt;inp1; load-&gt;opc; n-&gt;cps;</td>
</tr>
<tr>
<td>store addr</td>
<td>load-&gt;inp1; out-&gt;inp2; store-&gt;opc;</td>
</tr>
<tr>
<td>op n</td>
<td>out-&gt;inp1; out-&gt;inp2; op-&gt;opc; n-&gt;cps;</td>
</tr>
<tr>
<td>swap</td>
<td>out-&gt;inp1; out-&gt;inp2; swap-&gt;opc;</td>
</tr>
<tr>
<td>dup n</td>
<td>out-&gt;inp1; dup-&gt;opc; n-&gt;cps;</td>
</tr>
<tr>
<td>goto PC</td>
<td>pc-&gt;inp1; goto-&gt;opc; L-&gt;cps;</td>
</tr>
<tr>
<td>if/goto PC</td>
<td>pc-&gt;inp1; out-&gt;inp2; if/goto-&gt;opc; L-&gt;cps;</td>
</tr>
</tbody>
</table>

TABLE III

MAPPING QUEUE MACHINE INSTRUCTIONS TO MOVE INSTRUCTIONS OF A UNIVERSAL SCAD MACHINE.

Since a queue machine has a 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, since a SCAD machine contains multiple PUs and many queues, it 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 (dup and swap operations) as the queue machine [15]. Hence, the SCAD code generated from queue code is not optimal. For example, consider a SCAD machine containing one load-store unit (lsu), one adder (add) and one multiplier (mul). Table IV lists a sequence of SCAD move instructions to execute the DAG in Figure 4 without using 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 4) 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. 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.

<table>
<thead>
<tr>
<th>SCAD program</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr(1)-&gt;lsu.inp1;</td>
</tr>
<tr>
<td>addr(2)-&gt;lsu.inp1;</td>
</tr>
<tr>
<td>lsu.out-&gt;add.inp1;</td>
</tr>
<tr>
<td>add.out-&gt;mul.inp1;</td>
</tr>
<tr>
<td>add.out-&gt;add.inp1;</td>
</tr>
<tr>
<td>addr(y1)-&gt;lsu.inp1;</td>
</tr>
<tr>
<td>addr(y2)-&gt;lsu.inp1;</td>
</tr>
</tbody>
</table>

TABLE IV

SCAD PROGRAM TO EXECUTE THE DAG IN FIGURE 4 ON A SCAD MACHINE WITH MULTIPLE PROCESSING UNITS.

IV. OPTIMAL SCAD CODE GENERATION

Additional dup and swap operations not only degrade the performance, but also increase the code size and power consumption of SCAD machines. A sufficient number of PUs ensures that SCAD code can always be generated without any additional overhead as determined by the SAT encoding in [16]. However, optimal code for a given SCAD machine (with a given number of PUs) must execute in minimal possible time (to exploit maximal possible ILP). To determine the same, we encode in this section the decision version of the optimal SCAD code generation problem as an equivalent SMT problem.

A. SCAD Code Generation Problem

Given the following:

- a basic block (DAG) as three-address code in static single assignment (SSA) form [21], i.e.,

\[
\begin{align*}
   x_{tgt(0)} &= x_{src1(0)} \oplus 0 \cdot x_{src2(0)} \\
   &\vdots \\
   x_{tgt(\ell-1)} &= x_{src1(\ell-1)} \oplus \ell-1 \cdot x_{src2(\ell-1)}
\end{align*}
\]

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

The problem is to check whether the basic block can be executed on the SCAD machine in time $t$ without any dup and swap operations. If yes, determine the schedule of the basic block on the SCAD 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 $V_{\text{tgt}}$ are those that occur as left-hand side
- source variables $V_{\text{src}}$ are those that occur as right-hand side
- load variables $V_{\text{ld}}$ are those that only occur as right-hand side $V_{\text{ld}} := V_{\text{src}} \setminus V_{\text{tgt}}$

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

Furthermore, basic blocks can also be partitioned into levels by defining sets of variables $V_{\text{def}}$ defined in level $j$:
- Level 0 contains all instructions that only read variables $V_{\text{ld}}$. Their target variables are the variables $V_{\text{def}}$ defined at this level.
- Level $j+1$ are all instructions that only read variables $V_{\text{ld}} \cup \bigcup_{i=0}^{j} V_{\text{def}}$ and where at least one variable of $V_{\text{def}}$ is read.

All instructions in one level are independent of each other, i.e., can be fired in parallel. The levels are also defined by an ASAP (as soon as possible) scheduling of the basic block. 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 SCAD machine with $p$ universal PUs \( \{1, \ldots, p\} \), 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 $j \in \{0, \ldots, p\}$. This determines the instructions of the basic block which are executed by PU $j$. We demand that all $x_i \in V_{\text{ld}}$ 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 SCAD machine and vice versa.

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

Schedule exactly once: The next constraint ensures that each node $x_i$ is scheduled exactly once in one of the $t$ time slots:

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

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:

\[
\bigwedge_{j=0}^{t-1} \sum_{i=0}^{n-1} \theta_{i,j} \leq 1 \quad \text{and} \quad \bigwedge_{j=0}^{t-1} \sum_{i \in V_{\text{ld}}} \theta_{i,j} \leq p \quad \text{(3)}
\]

Unique PU assignment: The constraint 4 ensures that every node $x_i$ is assigned to one and only one PU,

\[
\bigwedge_{i=0}^{n-1} \sum_{j=0}^{t-1} \alpha_{i,j} = 1 \quad \text{(4)}
\]

Data dependency: For each node $x_i$, $\tau_i$ is the time slot in which the node is scheduled:

\[
\tau_i = \sum_{j=0}^{t-1} j \times \theta_{i,j} \quad \text{(5)}
\]

Consider instruction $x_{\text{tgt}(i)} = x_{\text{srcL}(i)} \otimes_i x_{\text{srcR}(i)}$ 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:

\[
\begin{align*}
\bigwedge_{i=0}^{t-1} \tau_{\text{tgt}(i)} - \tau_{\text{srcL}(i)} & \geq \delta_{\text{srcL}(i)} \\
\bigwedge_{i=0}^{t-1} \tau_{\text{tgt}(i)} - \tau_{\text{srcR}(i)} & \geq \delta_{\text{srcR}(i)}
\end{align*}
\]

Buffer constraint: A total schedule order must exist for those nodes $x_i$ that are at some time in the same buffer. Consider two instructions $x_{\text{tgt}(i)} = x_{\text{srcL}(i)} \otimes_i x_{\text{srcR}(i)}$ and $x_{\text{tgt(j)}} = x_{\text{srcL}(j)} \otimes_j x_{\text{srcR}(j)}$ 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 $x_{\text{srcL}(j)}$ (respectively $x_{\text{srcL}(i)}$), 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. However, if both operands are produced by the same PU, $x_{\text{srcL}(i)}$ (respectively $x_{\text{srcL}(j)}$) must be produced before $x_{\text{srcL}(j)}$ (respectively $x_{\text{srcL}(i)}$). In other words, we must
enforce the schedule ordering \( \tau_{\text{srcL}(i)} \leq \tau_{\text{srcL}(j)} \) (respectively \( \tau_{\text{srcR}(i)} \leq \tau_{\text{srcR}(j)} \)). Similar arguments apply for the right operands.

We introduce a boolean relation \( \beta_{i,j} \) that is true iff \( x_i \) and \( x_j \) are produced by the same PU:

\[
\beta_{i,j} = \bigvee_{k=0}^{p} \alpha_{i,k} \land \alpha_{j,k} \quad (7)
\]

Note that \( \alpha_{i,k} \) (appearing in constraint 7 as a boolean version of the corresponding binary variable) means that \( x_i \) is produced in the target buffer of PU \( k \). Finally, constraint 8 ensures a correct ordering of the variables in the input and output buffers:

\[
\bigwedge_{i,j=0}^{\ell-1} \bigwedge_{i,j=0}^{\ell-1} \beta_{\text{tgt}(i),\text{tgt}(j)} \Rightarrow \left\{ \begin{array}{l}
(\tau_{\text{tgt}(i)} < \tau_{\text{tgt}(j)}) \land \\
(\beta_{\text{srcL}(i),\text{srcL}(j)} \rightarrow \tau_{\text{srcL}(i)} \leq \tau_{\text{srcL}(j)}) \land \\
(\beta_{\text{srcR}(i),\text{srcR}(j)} \rightarrow \tau_{\text{srcR}(i)} \leq \tau_{\text{srcR}(j)}) \\
(\tau_{\text{tgt}(j)} < \tau_{\text{tgt}(i)}) \land \\
(\beta_{\text{srcL}(j),\text{srcL}(i)} \rightarrow \tau_{\text{srcL}(j)} \leq \tau_{\text{srcL}(i)}) \land \\
(\beta_{\text{srcR}(j),\text{srcR}(i)} \rightarrow \tau_{\text{srcR}(j)} \leq \tau_{\text{srcR}(i)})
\end{array} \right. \quad (8)
\]

3) Solution: With the above constraints, we have defined a schedule for executing the considered basic block on the SCAD machine. 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 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 level 0:

- Arrange the target variables in this level by \( \prec \). 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 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

Similar to ILP encodings of register-based code generation, concurrency in queue-based code is limited by the buffer constraint 8 that enforces the existence of a variable ordering. Here, we show a simple example to demonstrate the same. Consider the following input program:

\[
x_3 = x_0 \odot_0 x_1 \\
x_4 = x_2 \odot_1 x_3 \\
x_5 = x_1 \odot_2 x_4
\]

The program can be executed without overhead on a SCAD machine with only one universal PU and one LSU. Clearly, variables \( \{x_0, x_1, x_2\} \) are load variables assigned to LSU, while variables \( \{x_3, x_4, x_5\} \) are assigned to the universal PU.

First, we determine the minimal number of clock cycles required to execute the program by using an ideal scheduler, assuming one universal PU and one LSU both having unit latencies. An ideal scheduler is neither constrained by any variable ordering (buffer) constraint nor by any storage (number of registers) constraint. To determine the minimal execution time, we simply remove the buffer constraint from the SMT encoding and derive a schedule for the program satisfying other constraints. The derived minimal execution time is 5 clock cycles and a corresponding schedule is shown in Table V(A).

\[
\begin{array}{c|c|c}
\text{clk} & \text{instructions} & \text{clk} \\
1 & x_0 & 1 & x_0 \\
2 & x_1 & 2 & x_2 \\
3 & x_2 & 3 & x_3 \\
4 & x_4 & 4 & x_3 \\
5 & x_5 & 5 & x_4 \\
\end{array}
\]

**TABLE V**

**SCHEDULE OF PROGRAM 9 GENERATED BY (A) AN IDEAL SCHEDULER (B) QUEUE-BASED SCHEDULER**

Next, we determine the minimal number of clock cycles required to execute the same program by using our queue-based scheduler, assuming a SCAD machine with the same configuration (i.e., one universal PU and one LSU), by adding back the buffer constraint to SMT encoding. The derived minimal execution time is now 6 clock cycles and the corresponding schedule is shown in Table V(B).

Note that \( x_1 \) is the right-hand side operand of the target variable \( x_3 \) (\( x_3 = x_0 \odot_0 x_1 \)). Therefore, in the ideal schedule, variable \( x_1 \) should be produced before variable \( x_2 \) in order to schedule both \( x_2 \) and \( x_3 \) in the same time slot. However, this is not possible in the queue-based schedule for the SCAD machine due to the following buffer constraint:

\[
\beta_{4,5} \rightarrow \left\{ \begin{array}{l}
(\tau_4 < \tau_5) \land \\
(\beta_{4,2} \rightarrow \tau_2 \leq \tau_1) \land \\
(\beta_{4,4} \rightarrow \tau_3 \leq \tau_4) \\
\end{array} \right. \quad (10)
\]

\( \beta_{4,5} \) holds since both \( x_4 \) and \( x_5 \) are mapped to the single available universal PU. Since \( x_3 = x_1 \odot_2 x_4 \) (\( x_4 \) is the right-hand side operand of target variable \( x_3 \)), variable \( x_4 \) must be produced before producing variable \( x_5 \), i.e., \( \tau_4 < \tau_5 \).
Also, $\beta_{1,2}$ holds since both load variables $x_1$ and $x_2$ are mapped to the single available LSU. Therefore, from buffer constraint 10, it must hold that $\tau_2 < \tau_1$, i.e., variable $x_2$ must be produced before variable $x_1$. In the following section, we prove experimentally that compromises in exploited ILP due to the buffer constraint (or the ordering constraint) are an exception and not the expectation.

V. EXPERIMENTAL RESULTS

A. Experimental Set up

We have implemented a random basic block generator 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 $(n, l)$, 1k basic blocks were generated. Wrapper functions around the Z3 SMT solver [17] were implemented to carry out the following steps for every basic block: (1) Derive the minimal number of PUs required in a SCAD machine to execute the input DAG without any dup and swap overhead. To derive the minimal number of PUs, we simply set the desired execution time to the theoretical maximum value (by adding up latencies of all nodes in the DAG) so that the execution time does not affect the satisfiability of the SMT problem. (2) With a minimal number of PUs, determine the minimal time taken to execute the basic block (DAG) by using an ideal scheduler, a \{2,4\}-register-based scheduler and our queue-based scheduler. For the ideal scheduler (assuming infinite local storage and no ordering constraints), we remove the buffer constraints and run the SMT solver on the input basic block. For the \{2,4\}-register-based scheduler, we first allocate basic block variables to \{2,4\} registers using the well-known Chaitin-Briggs heuristic [22] (that yields nearly optimal results). Load and store instructions are then inserted for the variables mapped to main memory. The resulting data flow graph is used as input by the SMT solver, that is run again without including the buffer constraints. For the queue-based scheduler, we run the SMT solver directly on the input basic block by including the buffer constraints. In all cases, a binary search between the theoretical maximum and minimum values of execution time is performed to derive the minimal time taken to execute the basic block. Therefore, optimal schedules are generated for basic blocks by all scheduler types using the SMT solver.

B. Results

Two sets of experimental results are shown: (1) Abstract-time: Assuming unit latencies for all DAG nodes gives the degree of ILP exploited in abstract time (or steps). The average value of the minimal times (in steps) taken to execute DAGs of different sizes (numbers of nodes) by all scheduler types is shown in Figure 5. (2) Real-time: We randomly choose a latency of 1 or 2 clock cycles for non-load and non-store nodes in DAG. 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. The average value of the minimal times (in clock cycles) taken to execute DAGs of different sizes by all scheduler types is shown in Figure 6. All the runs are performed on an 2x Intel Xeon CPU X5450 (4x3.0GHz) 64bit computer with 32 GB RAM running the Ubuntu 16.04 operating system.

The minimal number of PUs required to generate overhead-free SCAD code increases with an increasing number of nodes. However, even for basic block size 12, we find that only 3 PUs are required in the worst case (consistent with our previous results using the SAT encoding [16]). Therefore, all
practical basic blocks could be handled without any overhead using SCAD machines with only 3 PUs. Importantly, note from Figures 5 and 6 that in both abstract-time and real-time cases, the ILP exploited using the \(\{2, 4\}\)-register-based scheduler deviates farther away from the maximal ILP (by the ideal scheduler) as the number of nodes increases. This reduction in exploited ILP already starts with basic blocks with size around three times the number of registers (from 6 node basic blocks for 2 registers and from 10 node basic blocks for 4 registers). This is because more load and store instructions are inserted to deal with the increasing number of variables in the basic block, and this in turn destroys its ILP. The effect is clearly more apparent in the real-time case when we take into account realistic latency numbers, due to slower memory accesses (note that same scale is used in both figures). However, the ILP exploited by using our queue-based scheduler stays near to the maximal ILP in all scenarios. The slight deviation is due to the buffer constraints that enforce an ordering for basic block variables so that only buffers and no registers are needed to store these variables. Clearly, with more PUs (and hence more buffers), this ordering requirement becomes less difficult to satisfy allowing the queue-based scheduler to exploit maximal ILP, while adding PUs does not help to mitigate the performance degradation in register-based schedules. Moreover, FIFO buffers or queues are easier to scale compared to register files.

C. Feasibility

We also study the feasibility of our optimal code generation approach using SMT solvers. Average and maximal times taken by the SMT solver to derive queue-based schedules for DAGs of different sizes (with both unit node latency in both the abstract-time experimental set up and realistic node latencies in real-time experimental set up) are shown in Figure 7. As expected, in both abstract-time and real-time cases, the time taken by the SMT solver to derive a feasible schedule increases with increasing numbers of nodes because larger numbers of nodes means larger PU assignments \((\alpha_{i,j})\) and scheduling \((\theta_{i,j})\) variables to consider in the SMT problem. Figure 8 shows the SMT solver’s runtime to derive queue-based schedules for 11 and 12 node DAGs of different levels with unit node latency. DAGs with realistic node latencies also follow similar patterns. While the average time taken gradually increases with the number of levels, the SMT solver ran for longest time in case of two-level and three-level DAGs. On the one hand, with increasing numbers of levels in a basic block, the SMT solver has to deal with more data dependencies leading to an increase in its runtime. On the other hand, as data dependencies increase, more variable orders are predefined by these data dependencies leaving less choices for the SMT solver to satisfy the buffer constraint, leading to a decrease in its runtime. Therefore, in general, for two-level or three-level DAGs, the SMT solver is able to find a feasible schedule in lesser time due to many choices of variable orderings. However, for certain exceptions (difficult DAG structures), the SMT solver has to explore all possible variable orderings before finding a feasible schedule.

With a timeout of 60 seconds, DAG sizes up to 12 nodes in the abstract-time set up were successfully processed, while nearly 800 seconds were taken in the worst case (65 seconds in the average case) to process 12 node DAGs in the real-time set up. Therefore, the proposed code generation technique using SMT solvers can process basic blocks of practical size (nearly 85% of basic blocks in SPEC 2000 benchmark suite consists of up to 10 instructions [23]. Average dynamic basic block sizes for SPEC 2000 integer benchmarks are below 12 instructions [24]). However, generating code for superblocks [2] and hyperblocks [2] will require heuristics. Nevertheless, the performance comparison results of register-based and queue-based code generation motivates well to develop heuristics to employ queue-based code generation for larger programs. To that end, we have developed a cycle-accurate SCAD machine simulator [25] to run and observe the execution of different basic blocks to learn key features for heuristics.

VI. Conclusions

By a suitable encoding as a SMT problem, we experimentally showed that exposed datapath architectures with buffered processing units have the potential to exploit the maximal ILP contained in basic blocks by following a novel queue-based code generation. Furthermore, we demonstrated the superiority of queue-based code generation compared with the classic register-based code generation. The optimal code generation technique using SMT solvers as described in this paper is even feasible for basic blocks of practical size (with up to 12 variables in SSA form). Nevertheless, it is important to develop heuristics to deal with larger basic blocks. To that end, the generated optimal code will serve as a reference to measure the quality of future heuristics. Furthermore, it is necessary to
analyze the buffer sizes required in exposed datapath architectures. Also, 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.

REFERENCES