Towards Code Generation for Synchronous Control Asynchronous Dataflow (SCAD) Architectures

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

Abstract. Many recent processor architectures expose their datapaths so that the compiler can not only schedule instructions to increase instruction-level concurrency, but can even take care of moving values between the processing units of the processor to optimize their allocation at compile-time. In this paper, we introduce with the Synchronous Control Asynchronous Dataflow (SCAD) paradigm another exposed datapath architecture and discuss how code can be generated best for SCAD architectures. While traditional compilers focus on register usage and therefore evaluate expressions usually by a depth-first traversal, we show that compiler techniques more adequate for SCAD should better focus on a breadth-first evaluation of expressions as known from queue machines. This way, they can forward values from one processing unit to another one without using registers at all. However, these machines sometimes have to make use of additional swap and duplication operations that add some overhead to the actual computation. Since a queue machine can be simulated by a universal SCAD machine, we can derive SCAD programs from queue programs. Moreover, if the queue program does not contain swap or duplication operations, the SCAD program is optimal for a single processing unit, where 'optimal' refers to the minimal number of swap and duplication operations. However, we also show that sometimes SCAD programs can avoid this overhead even if queue machines need it, which makes SCAD code generation more difficult.

1. Introduction

Since almost a decade, all newly announced microprocessors are multicore processors for various technical reasons. This tremendous shift in processor architecture demands for multithreaded programming that is however not possible for sequential problems and very difficult for others [13]. An alternative is still to increase the use of instruction-level parallelism (ILP). In particular, code that is automatically generated in model-based design offers usually a big amount of ILP. There are essentially two main approaches to exploit ILP in processor architectures: First, using dynamic scheduling as introduced by the Tomasulo algorithm [21], and second by static scheduling as preferred by very long instruction word (VLIW) architectures [5, 10] and other embedded processors to reduce the power consumption.

Both variants of ILP face limits on their further scalability: Dynamic scheduling implies 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. However, also VLIW processors face limits on ILP:

R. Wimmer (Hrsg.): MBMV 2016
ISBN: 978-3-00-052380-9

Most compiler techniques like trace scheduling [9, 10] and software pipelining [12, 16] focus on scheduling instructions across basic blocks, and find this way often enough independent instructions. However, the *number of registers* is an upper bound for the number of instructions that can be bundled in a VLIW word, since the RISC instruction format demands that all instructions finally write their results into registers. Hence, while the introduction of registers was a good idea for sequential processors, it imposes now limits for ILP. Adding more registers changes the instruction set, and also the wiring of the register file becomes another bottleneck that already lead to clustered architectures [2].

It can be observed that new ideas in processor architecture somehow all try to eliminate the explicit use of registers in that the architectures expose not only the processing units, but also all datapaths between them, and are therefore often called *exposed datapath architectures*. Examples of exposed datapath architectures are Raw [14] with its commercial variant Tilera [3], WaveScalar [19], TRIPS [4], Flexcore [20], explicit datapath wide SIMD [23], and the transport-triggered architectures (TTAs) [6]. These architectures provide a large number of processing units, and the compiler is not only responsible to schedule instructions to these processing units, but also to move data directly from one processing unit to another one. This way, the use of a central register file can be bypassed. While these architectures have been studied already in great detail, current compiler technology is still based on the classic register architectures where expressions are evaluated in a depth-first manner to reduce the need of registers [18]. However, we demonstrate that compilers more adequate for exposed datapath architectures should rather evaluate expressions in a breadth-first manner as known from queue machines [22, 8] which exposes more ILP and allows bypassing of registers (and consequently memory accesses).

In this paper, we present with the *Synchronous Control Asynchronous Dataflow (SCAD)* architecture a new exposed datapath architecture. In SCAD, each processing unit is equipped with queues for storing its inputs and outputs (see Figures 1 and 5). Output queues of all processing units are connected to input queues via a data transport network (DTN). The SCAD machine is programmed by a sequence 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 its input queues to execute the operation, so that the dataflow is asynchronous. The registration of move instructions at the queues is however done synchronously to guarantee a correct execution. Similar to other exposed datapath architectures, the main advantage of SCAD is to directly move values between processing units and therefore to break the limit for ILP given by the number of available registers. A second advantage is the simple extension to application-specific processing units without having the need to change the instruction set.

We motivate a code generation technique for our SCAD architecture based on queue machines that also do not make use of registers. We show that while a queue machine can be simulated by a SCAD machine, the reverse simulation is not always possible. As a consequence, optimal code can be generated this way for SCAD machines with a single processing unit if the queue program does not require additional swap and duplication operations. However, SCAD code can avoid overhead for data transport in cases where the queue machine needs it.

The outline of the paper is as follows: our SCAD architectures are described in Section 3. In Section 4, we explain the code generation for queue machines followed by a comparison of queue and SCAD machines in Section 4.3. The final section summarizes the contributions of this paper and concludes by mentioning important future work.

2. Related Work

There are already many different kinds of exposed datapath architectures: Transport-triggered architectures (TTAs) [6] 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 side effect of the data transport. The compiler is responsible not only to order these move instructions in a sequence, but also to pack 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 arbitrarily complex processing units both in TTA and SCAD.

The RAW machine [14] consists of processing units arranged in a 2D tiled architecture with routers between them. It uses compiler-determined 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.

Both Wavescalar [19] and TRIPS [4] are 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 static dataflow fashion (static placement dynamic issue [15]). 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 [20], 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 [23], 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.

3. SCAD Architectures

The organization of processing units in a SCAD architecture is shown in Figure 1. 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,⊥)* with the special value ⊥ is used to indicate that the required value is not yet available and will later be sent from the output buffer *adr*. Similarly for an output buffer, *adr* is the address of the input buffer of the PU that will consume the value *val*. An entry *(adr,⊥)* with the special value ⊥ is used to indicate that the required value is not yet available and will later be produced by the PU and can then be sent to the input buffer *adr*.

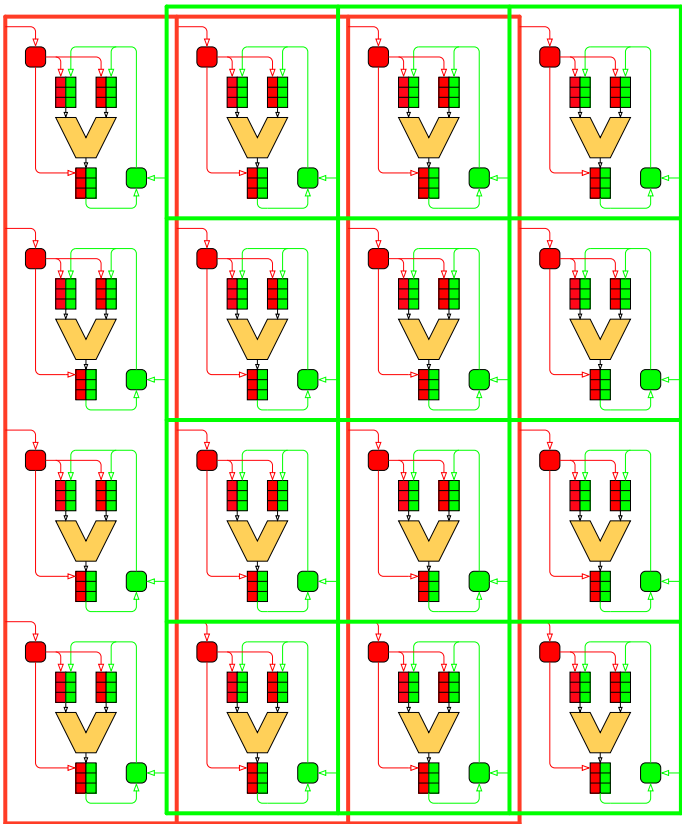


Figure 1: Architecture of a SCAD Processor

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 only two-input one-output PUs are shown in Figure 1, a PU in SCAD architecture may implement any arbitrary functionality 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,⊥)* to its tail, and the output buffer with address *src* will add the entry *(tgt,⊥)* to its new tail. If one of the two buffers should be full, it will signal this via a feedback signal *fullBuffer* to the control unit. The other buffer will then also not store the entry, and the control unit will resend the move instruction *(src,tgt)* in the next cycle (it is stalled at this point of time). The data transport related with a move instruction *(src,tgt)* is deferred to a later point of time when the data is available. Therefore, control flow is synchronous and 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),...,(adr_m, x_m) with $x_i \neq \perp$ at the heads of its m input buffers and there is free space in its n output buffers, it can react and will consume entries (adr_1, x_1),...,(adr_m, x_m) to produce new result values $y_1 := f_1(x_1, \ldots, x_m)$, ..., $y_n := f_n(x_1, \ldots, x_m)$ 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,⊥)* 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 *(⊥,y_i)* is placed at the tail of 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,⊥)* 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 *(⊥,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,⊥)*, if there is still space available. Otherwise, it will signal *fullBuffer* to the control unit, which then has to stall and resend the move instruction later. The input buffers also always snoop the two interconnection networks, i.e., the MIB and the DTN. As explained above, address entries *(src,⊥)* 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,⊥)* 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 either the SU and the LU have to be synchronized or implement a weak memory model.

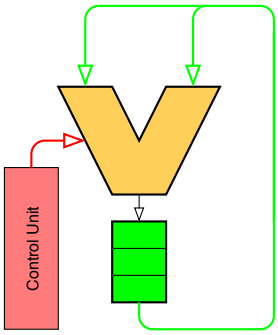
Branch instructions are handled as follows by the control unit (CU): if the target of a move instruction is the CU itself, it is meant to be the program counter, and thus, the CU has to wait until this value arrives at its input buffer associated with the program counter. Otherwise, it will generate the next program counter on its own and will put it in that input buffer.

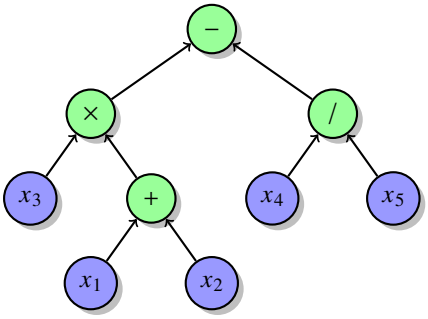
Note that we have not mentioned register files or other local storage although it is possible to use them just like any other PU in the SCAD architecture. In the following section, we motivate a code generation technique that does not require local memory other than the buffers. In other words, it utilizes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

4. Simulation of Queue Machines

4.1. Code generation for Queue Machines

A queue machine [22, 8] 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 on the left of Figure 2, with a queue program and the contents of the queue after executing each instruction of that program. A list of queue instructions is listed in Table 1.





program	queue content
`load x1,1`	`[x1]`
`load x2,1`	`[x1,x2]`
`load x3,1`	`[x1,x2,x3]`
`add 1`	`[x3,x1+x2]`
`load x4,1`	`[x3,x1+x2,x4]`
`load x5,1`	`[x3,x1+x2,x4,x5]`
`mul 1`	`[x4,x5,x3*(x1+x2)]`
`div 1`	`[x3*(x1+x2),x4/x5]`
`sub 1`	`[x3*(x1+x2)-(x4/x5)]`

Figure 2: Architecture of a queue machine (left), an expression with its queue program and the content of the queue after executing each instruction.

Queue Instruction	Description
`load adr,n`	Load data from memory address `adr` and add `n` copies of the loaded value to the tail of the queue.
`store adr`	Store the value from the head of the queue to the memory address `adr`.
`opcode n`	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.
`swap`	Dequeue two operands from the head of the queue, *swap* them, and add them to the tail of the queue.
`dup n`	Dequeue one operand from the head of the queue, and add *n* copies of it to the tail of the queue.
`goto L`	Unconditional Branch: replace current program counter `pc` by `pc+L`.
`ifGoto L`	Conditional Branch: replace current program counter `pc` by `pc+L` if the head of the queue is different to zero.

Table 1: List of queue instructions

Generating a queue program to evaluate an expression without `swap` and `dup` instructions is done by a breadth-first traversal of the expression tree [8] as shown in Figure 2. A consistent left to right or right to left traversal ensures that operands required to execute operations at one level are available at the head of the queue in the correct order.

Basic blocks of programs are often represented by directed acyclic graphs (DAGs). Generating a queue program for an expression *tree* is easy since an expression tree is by definition a level-planar graph [17]. 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 [17] as shown in Figure 3: 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 crossings of 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` operation may be placed at new levels.

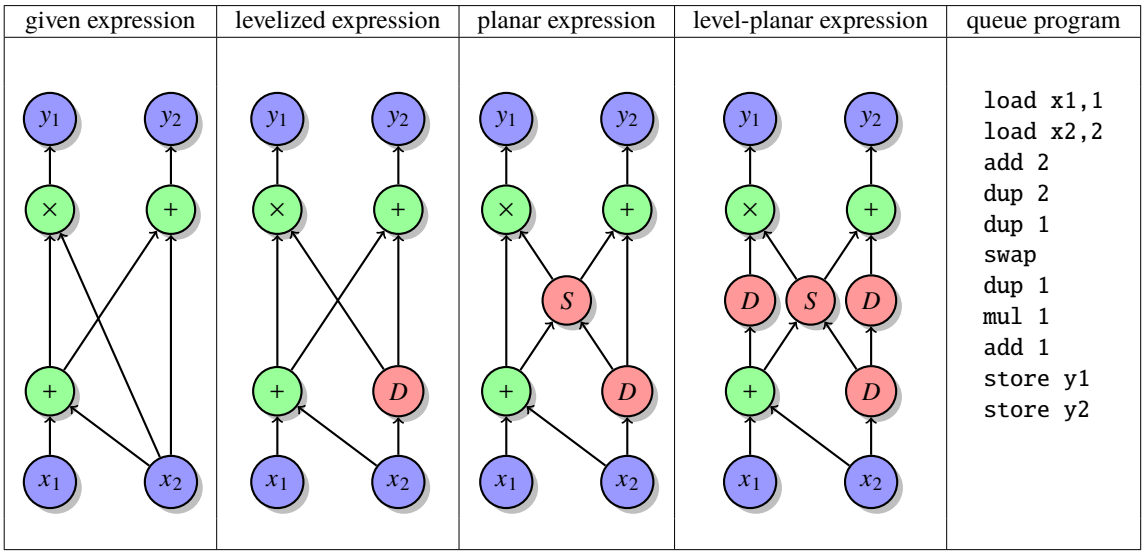


Figure 3: A given expression DAG with its levelized version, its planarized version, the final level-planar expression DAG, and the corresponding queue program.

4.2. Bypassing Usage of Registers

Exposed datapath architectures try to avoid the use of registers, and the examples of queue programs shown in Figures 2 and 3 show how the considered expressions are evaluated without the use of registers at all. The advantage is that the size of the queue can be much larger than the number of registers available in RISC instruction sets. In contrast, traditional register machines which are the basis of RISC instruction sets [11] expect operands in registers and also write back the results in registers. For expression trees, it is simple to implement optimal code generators in the sense of minimal register usage (or minimal use of load/store operations) [18]. This is done by a depth-first traversal that computes the Strahler number [7] of the subtrees which is the minimal number of required registers. Intuitively, this is the height of the highest full binary subtree in the expression tree which is 3 in the example of Figure 2. An optimal program using three registers for this expression is shown in Figure 4.

```
load x1,r1
load x2,r2
add r1,r2,r1
load x3,r2
mul r2,r1,r1
load x4,r2
load x5,r3
div r2,r3,r2
sub r1,r2,r1
store y,r1
```

Figure 4: Optimal RISC code for the expression of Figure 2.

Traditional code generators therefore prefer a depth-first traversal which is however bad for queue programs that would then have to introduce a lot of `swap` and `dup` instructions. Following a breadth-first evaluation of expression trees allows one even to evaluate expression trees of arbitrary size without any `swap` and `dup` instructions and any other memory than just the queue. The maximal size of the queue is then the width of the widest level of the expression tree. In contrast, the Strahler number can become the height of the tree, and since that can exceed the number of available registers, it will require the use of load and store instructions on any RISC architecture.

The same is the case for expression DAGs where code generation for RISC architectures is known to be NP-complete [1].

We observe that *registers are not required at all* for queue machines if we follow a breadth-first evaluation of the expression trees or DAGs where DAGs first have to be made level-planar as shown in Figure 3. This operation order ensures the availability of results from one level of the expression tree/DAG before the consuming operation at the next level is executed. It is moreover well-known that queue machines offer an *increased instruction-level parallelism* in that all operations at each level of the expression DAG can be executed in parallel.

4.3. Simulation of Queue Machines by SCAD Machines

In the following, we show that a queue machine can be simulated by a SCAD machine, but not vice versa. Consider the architecture of a universal SCAD machine shown in Figure 5. 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 (Figure 5 only shows two of them): `inp1` to store the first operand, `inp2` to store the second operand, `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. As usual in SCAD, a universal PU fires if relevant data is available at the heads of its input buffers to execute the operation. Note that the universal SCAD machine's execution is close to the queue machine's execution in that instead of using a single queue for storing operands and results of computations, a universal SCAD machine uses separate queues. In fact, each queue instruction can be mapped to a sequence of move instructions for the universal SCAD machine as listed in Table 2.

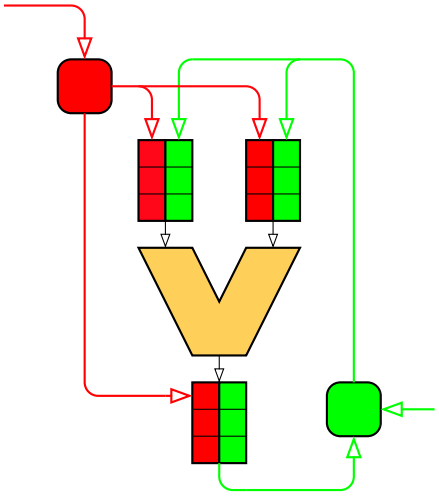


Figure 5: A SCAD Machine with a single universal PU.

Queue Instruction	Corresponding SCAD Move Instructions			
`load adr,n`	`adr->inp1;`		`load->opc;`	`n->cps;`
`store adr`	`adr->inp1;`	`out->inp2;`	`store->opc;`	
`op n`	`out->inp1;`	`out->inp2;`	`op->opc;`	`n->cps;`
`swap`	`out->inp1;`	`out->inp2;`	`swap->opc;`	
`dup n`	`out->inp1;`		`dup->opc;`	`n->cps;`
`goto PC,L`	`pc->inp1;`		`goto->opc;`	`L->cps;`
`ifGoto PC,L`	`pc->inp1;`	`out->inp2;`	`ifGoto->opc;`	`L->cps;`

Table 2: Mapping Queue Machine Instructions to SCAD Move Instructions of a Universal SCAD Machine.

For the queue program of the level-planar DAG of Figure 3, the corresponding move program for a universal SCAD machine obtained using this mapping is shown in Table 3. We can prove now the following theorem:

Queue Instruction	Corresponding SCAD Move Instructions			
`load x1,1`	`x1->inp1;`		`load->opc;`	`1->cps;`
`load x2,2`	`x2->inp1;`		`load->opc;`	`2->cps;`
`add 2`	`out->inp1;`	`out->inp2;`	`add->opc;`	`2->cps;`
`dup 2`	`out->inp1;`		`dup->opc;`	`2->cps;`
`dup 1`	`out->inp1;`		`dup->opc;`	`1->cps;`
`swap`	`out->inp1;`	`out->inp2;`	`swap->opc;`	
`dup 1`	`out->inp1;`		`dup->opc;`	`1->cps;`
`mul 1`	`out->inp1;`	`out->inp2;`	`mul->opc;`	`1->cps;`
`add 1`	`out->inp1;`	`out->inp2;`	`add->opc;`	`1->cps;`
`store y1`	`y1->inp1;`	`out->inp2;`	`store->opc;`	
`store y2`	`y2->inp1;`	`out->inp2;`	`store->opc;`	

Table 3: Queue Program and SCAD Move Program for the Expression of Figure 3

Theorem 1 *Queue machines can be simulated by SCAD machines with a single universal processing element. Moreover, for any queue program without* `swap` *and* `dup` *instructions, there is a corresponding SCAD program without* `swap` *and* `dup` *move instructions. However, there are SCAD programs without* `swap` *and* `dup` *move instructions where the queue machine requires* `swap` *and* `dup` *instructions.*

Proof: Given any queue program and the corresponding SCAD program generated by the mapping given in Table 2. It is not difficult to see that the contents of the queue of the queue machine and the output queue of 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. Hence, a queue machine can be simulated by a universal SCAD machine.

expression DAG	level-planar DAG	queue program	SCAD program

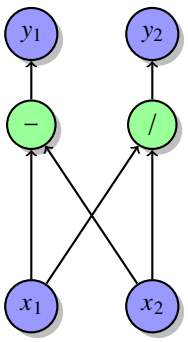
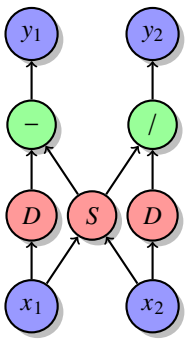
		`load x1,2`
		`load x2,2`
		`dup 1`
		`swap`
		`dup 1`
		`sub 1`
		`div 1`
		`store y1`
		`store y2`
			`x1->inp1; load->opc; 2->cps;`
			`x2->inp1; load->opc; 2->cps;`
			`out->inp1; out->inp1;`
			`out->inp2; out->inp2;`
			`sub->opc; 1->cps;`
			`div->opc; 1->cps;`
			`y1->inp1; out->inp2; store->opc;`
			`y2->inp1; out->inp2; store->opc;`

Figure 6: A given expression DAG with its planarized version, the corresponding queue program with `swap` and `dup` instructions, and an equivalent SCAD program without `swap` and `dup` instructions.

The converse is however not true. For the simple basic block `y1 = x1-x2; y2 = x1/x2`, we obtain the expression DAG and its level-planar version as shown in Figure 6. As can be seen, the expression DAG is not planar and therefore, we have to introduce additional `swap` and `dup`

instructions for the queue machine. This is required since all operands have to be brought into a *total* order in the queue machine since it has only one queue. After loading, duplicating, and swapping, the content of the queue is then `x1,x2,x1,x2` so that the two binary operations can read their operands in the right order to compute `x1-x2` and `x1/x2`. For the SCAD machine, we can do without `swap` and `dup` instructions: To this end, we first load two copies of `x1` and then two copies of `x2`, so that the content of the output buffer is `x1,x1,x2,x2` after the first two lines. We then move the two copies of `x1` to input buffer `inp1` and then the two copies of `x2` to input buffer `inp2` so that `inp1` holds values `x1,x1` and input buffer `inp2` holds values `x2,x2`. We then move the opcodes of `sub` and `div` to the `opc` buffer, and can then get the results from the output buffer for storing. ■

Hence, the SCAD machine has more freedom to order values because of two separate input buffers that only require total ordering of the operands that pass these buffers. It is still the case that SCAD machines are closer to queue machines than to register machines, and therefore their code generation should also be better done in a similar way as for queue machines.

In particular, based on the above theorem, we can say that if we generate for a given basic block a level-planar DAG, we can then generate a SCAD program without having the need of additional local storage or registers. The number of `swap` and `dup` operations is then exactly the number of `swap` and `dup` nodes of the level-planar DAG.

However, as observed by the example shown in Figure 6, we can see that it is not always necessary to generate a level-planar graph for a given expression DAG. The reason for this is that *any edge crossing* in the DAG is a violation for the queue machine, but not necessarily for the SCAD machine. In the SCAD machine, we have several queues, and we therefore only have to consider edge crossings of the same buffers that have to be made planar. In the example of Figure 6, `x1` and `x2` were put in different input buffers `inp1` and `inp2`, respectively, and were also used as different operands (either left or right operand) of all operations that used them. As a consequence, the edge crossing is no problem, and we can generate SCAD code without overhead. Bytheway, the same is the case for the example in Figure 3.

Hence, SCAD machines have less overhead due to operand ordering, but this even makes their code generation more difficult. In particular, for SCAD machines with many processing units, the allocation of processing units and buffers for given instructions is crucial for code generation.

5. Conclusion and Future Work

In this paper, we presented the SCAD architecture – a new exposed datapath architecture – that avoids the use of registers, so that instruction-level parallelism is not limited by the number of registers as it is the case for VLIW architectures. Instead, the only instructions are move instructions, and the compiler has to generate move instructions in a way that values available in output buffers of processing elements can be moved to input buffers of other processing elements. Once processing elements have available inputs, they consume them and produce corresponding output values that are then sent by the output buffers to other processing elements.

To this end, we suggested a code generation technique based on a breadth-first evaluation of level-planar expression DAGs instead of the depth-first evaluation of traditional compilers. This code generation is known for queue machines, and since we proved that queue machines can be simulated by SCAD machines, it is clear by our translation that queue code programs without `swap` and `dup` operations yield optimal programs for SCAD machines with a single universal

processing unit. However, if the queue program contains `swap` and `dup` operations, we can still derive an equivalent SCAD program, but as shown by the example given in Figure 6, there could be SCAD programs with fewer (`swap` and `dup`) instructions. SCAD machines offer more freedom for maintaining the values inside the many buffers that are not available in queue machines. We therefore have to refine the code generation technique in that only *critical crossings* are made planar in the expression DAG for SCAD code generation.

It is moreover important to analyze buffer sizes required to execute various benchmarks in the SCAD machine. Along with performance comparisons, it is also necessary to compare the hardware resource usage and power consumption of SCAD architectures with that of conventional superscalar and VLIW architectures with a similar configuration.

References

[1] Aho, A.V., S.C. Johnson, and J.D. Ullman: *Code generation with common subexpressions*. Journal of the ACM (JACM), 24(1):146–160, 1977.

[2] Aletà, A., J.M. Codina, A. González, and D.R. Kaeli: *Heterogeneous clustered VLIW microarchitectures*. In *Code Generation and Optimization (CGO)*, pages 354–366, San Jose, California, USA, 2007. IEEE Computer Society.

[3] Bell, S., B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, C.C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook: *TILE64 - processor: A 64-core SoC with mesh interconnect*. In *International Solid-State Circuits Conference (ISSCC)*, pages 88–598, San Francisco, CA, USA, 2008. IEEE Computer Society.

[4] Burger, D., S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder: *Scaling to the end of silicon with EDGE architectures*. IEEE Computer, 37(7):44–55, July 2004.

[5] Colwell, R.P., R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman: *A VLIW architecture for a trace scheduling compiler*. In Katz, R. (editor): *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 180–192, Palo Alto, California, USA, 1987. ACM.

[6] Corporaal, H.: *TTAs: Missing the ILP complexity wall*. Journal of Systems Architecture, 45(12-13):949–973, June 1999.

[7] Esparza, J., M. Luttenberger, and M. Schlund: *A brief history of Strahler numbers*. In Dediu, A. H., C. Martin-Vide, J.L. Sierra-Rodriguez, and B. Truthe (editors): *Language and Automata Theory and Applications (LATA)*, volume 8370 of *LNCS*, pages 1–13, Madrid, Spain, 2014. Springer.

[8] Feller, M. and M.D. Ercegovac: *Queue machines: An organization for parallel computation*. In Brauer, W., P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and Wolfgang Händler (editors): *Conpar 81*, volume 111 of *LNCS*, pages 37–47, Nürnberg, Germany, 1981. Springer.

[9] Fisher, J.A.: *Trace scheduling: A technique for global microcode compaction*. IEEE Transactions on Computers (T-C), C-30(7):478–490, July 1981.

[10] Fisher, J.A., P. Faraboschi, and C. Young: *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[11] Hennessy, J., N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill: *MIPS: A microprocessor architecture*. In *Microarchitecture (MICRO)*, pages 17–22, Palo Alto, California, USA, 1982. IEEE Computer Society.

[12] Lam, M.: *Software pipelining: an effective scheduling technique for VLIW machines*. In Wexelblat, R.L. (editor): *Programming Language Design and Implementation (PLDI)*, pages 318–328, Atlanta, Georgia, USA, 1988. ACM.

[13] Lee, E.A.: *The problem with threads*. IEEE Computer, 39(5):33–42, 2006.

[14] Lee, W., R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe: *Space-time scheduling of instruction-level parallelism on a raw machine*. In Bhandarkar, D. and A. Agarwal (editors): *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, San Jose, California, USA, 1998. ACM.

[15] Nagarajan, R., S.K. Kushwaha, D. Burger, K.S. McKinley, C. Lin, and S.W. Keckler: *Static placement, dynamic issue (SPDI) scheduling for EDGE architectures*. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 74–84, Antibes Juan-les-Pins, France, 2004. IEEE Computer Society.

[16] Ramakrishna Rau, B.: *Iterative modulo scheduling: an algorithm for software pipelining loops*. In *Microarchitecture (MICRO)*, pages 63–74, San Jose, California, USA, 1994. IEEE Computer Society.

[17] Schmit, H., B. Levine, and B. Ylvisaker: *Queue machines: hardware compilation in hardware*. In Arnold, J. and K.L. Pocek (editors): *Field-Programmable Custom Computing Machines (FCCM)*, pages 152–160, Napa, California, USA, 2002. IEEE Computer Society.

[18] Sethi, R. and J.D. Ullman: *The generation of optimal code for arithmetic expressions*. Journal of the ACM (JACM), 17(4):715–728, October 1970.

[19] Swanson, S., A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S.J. Eggers: *The WaveScalar architecture*. ACM Transactions on Computer Systems (TOCS), 25(2):1–54, May 2007.

[20] Thuresson, M., M. Själander, M. Björk, L.J. Svensson, P. Larsson-Edefors, and P. Stenström: *FlexCore: Utilizing exposed datapath control for efficient computing*. In Blume, H., G. Gaydadjiev, C.J. Glossner, and P.M.W. Knijnenburg (editors): *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, pages 18–25, Samos, Greece, 2007. IEEE Computer Society.

[21] Tomasulo, R.M.: *An efficient algorithm for exploiting multiple arithmetic units*. IBM Journal of Research and Development, 11(1):25–33, 1967.

[22] Vollmar, R.: *Über einen Automaten mit Pufferspeicherung*. Computing, 5(1):57–70, 1970.

[23] Waeijen, L., D. She, H. Corporaal, and Y. He: *A low-energy wide SIMD architecture with explicit datapath*. Journal of Signal Processing Systems, 80(1):65–86, 2015.