On Memory Optimal Code Generation for Exposed Datapath Architectures with Buffered Processing Units

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

Abstract—One reason for the limited use of instruction level parallelism (ILP) by conventional processors is their use of registers. Therefore, some recent processor architectures expose their datapaths to the compiler so that the compiler can move values directly between processing units. In particular, the Synchronous Control Asynchronous Dataflow (SCAD) machine is an exposed datapath architecture that uses FIFO buffers at the input and output ports of its processing units. Code generation techniques inspired by classic queue machines can completely eliminate the use of conventional registers in SCAD. However, bounded buffer sizes may still make spill code necessary to store values temporarily in main memory. Since memory access is expensive, it has to be avoided to improve the execution time of programs. Memory optimal code generation problems have been extensively studied in the case of register machines and were proven to be NP-complete.

In this paper, we prove that memory optimal code generation for SCAD is also NP-complete by presenting a polynomial-time transformation from memory optimal register code to memory optimal SCAD code. In particular, we present a one to one correspondence between the registers in register machines and the entries of buffers in SCAD machines which indicates that these architectures are closer to each other than expected. Still, SCAD machines offer important advantages: The size of circuit implementations of buffers scales much better compared to register files so that more space is available on SCAD machines with the same chip size. Second, the instruction set of SCAD does not depend on a fixed number of registers or buffers. We therefore present experimental results to compare the execution time of memory optimal SCAD code with FIFO buffers and memory optimal code based on conventional register allocation.

I. INTRODUCTION

A. Motivation

Most current processor architectures are so-called load/store architectures which means that the main memory is only accessed by load/store instructions while operands and results of all other instructions are stored in registers. The success of these architectures stems from the fact that the execution time of memory accesses did not improve as fast as that of other instructions. Therefore, a simple way to improve the runtime of programs is to reduce the number of main memory accesses by loading values into local memories like registers and using these as often as possible. While introducing registers was a good idea for sequential processors, it now limits the use of instruction level parallelism (ILP).

For example, consider the expression tree shown in Figure 1(a). At least 3 registers are required to evaluate the expression tree without using load/store instructions. This is achieved by ordering and assigning registers to tree nodes by a depth-first traversal [20], [25]. Table 1(b) shows the operation order and the corresponding register assignment. The horizontal lines divide the sequence of operations into segments, where all operations in a segment can be executed concurrently. Thus, it takes 4 steps to evaluate the expression tree in parallel this way, while it can be evaluated in only 3 steps if all levels of the tree would be executed in parallel. Note that even without the overhead of load and store instructions (i.e., given a sufficient number of registers to avoid memory accesses), it is apparent that the register-based code limits the use of ILP. Moreover, if only 2 registers would be available, one has to insert spill code in that the obtained result of \( x_1 + x_2 \) is temporarily stored in memory and loaded in a register after having evaluated \( x_3 + x_4 \) as shown in Figure 1(c). At least 5 steps are then required to evaluate the expression (since there are 5 levels in the obtained syntax tree), irrespective of the number of processing units (PUs). Hence, the number of programmer-accessible registers influences the use of ILP.

The use of registers not only restricts the amount of available ILP, but also requires additional time to read/write values. This additional time can be saved if the value is transported directly from the producer to the consumer PUs. Exposed datapath architectures [5], [10], [12], [16], [22]–[24] offer this feature by exposing their datapaths to the compiler so that the compiler can take care of moving values directly between PUs. While these architectures have already been studied to a great detail, their compilers still rely on classic code generators where the optimal use of registers is vital to improve performance. We observed in [6] 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, Synchronous Control Asynchronous Dataflow (SCAD) architecture, 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. 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, with a limited number of PUs, computational overhead is required in the form of duplication and swap operations [6]. Also, with limited buffer sizes, spill code is required to store values temporarily in main memory and to reload them when necessary. We used SAT and SMT solvers to generate SCAD code with both minimal memory accesses and minimal computational overhead [7], [8]. The SAT solver generated resource constrained SCAD code (that works with a minimal number of PUs) and the SMT solver generated time constrained SCAD code (that minimizes runtime). Understanding the complexities of the ‘memory optimal SCAD code generation’ problem and the ‘computational overhead optimal SCAD code generation’ problem would alleviate the risk of potentially overlooking simple and efficient optimal algorithms. In this paper, we address the former.

B. Main Contributions

Traditional code generation problems that focus on the efficient use of registers to minimize memory accesses are known to be NP-complete [2], [4], [13]. However, these results are not directly applicable to SCAD code generation since SCAD code bypasses registers. In this paper, we therefore make the following contributions:

- We present polynomial-time transformations between memory optimal register code and memory optimal SCAD code. This implies NP-completeness of the memory optimal SCAD code generation problem.
- We find a strong correspondence between the number of registers in register machines and the size of buffers in SCAD machines. Still, the better scalability of circuit size of buffers compared to registers and the independence of a fixed number of registers or buffers still recommends SCAD as the better choice.
- We present experimental results to demonstrate the improvement of the performance of memory optimal SCAD code compared to memory optimal code that is based on register allocation.

C. Outline

Section II describes the model and code generation problem that we consider for both register machines and SCAD machines. The basic transformations from memory optimal register code to memory optimal SCAD code and vice versa, followed by the NP-completeness proof is given in Section III. We also discuss architectural implications of these transformation by considering the impact of scaling buffers and PUs on circuit complexity and computational overhead. Experimental results are provided in Section IV. Section V briefly discusses related exposed datapath architectures and code generators.

II. PRELIMINARIES

A. Register Architectures

For register machines, we consider the simple architecture of [2], i.e., the following non-commutative register-memory instruction set is used:

1) \( r_i \leftarrow m \) (load)
2) \( m \leftarrow r_i \) (store)
3) \( r_i \leftarrow r_i \odot m \) (memop)
4) \( r_i \leftarrow r_j \) (assign)
5) \( r_i \leftarrow r_i \odot r_j \) (regop)

where \( r \) and \( m \) refer to values in register and memory locations, respectively, and \( \odot \) refers to any binary operation.

A commutative architecture will additionally include the instruction \( r \leftarrow m \odot r_i \). Straight-line programs also called basic blocks are considered for code generation in the following. Basic blocks are often represented by directed acyclic graphs (DAGs) where the nodes of the graph denote computations and directed edges denote data dependencies. For simplicity, we consider only a one-register machine to derive our complexity result. Clearly, a one-register machine cannot use instructions 4 and 5 above. It is then not difficult to see that a program with a minimal number of memory accesses will also have a minimal number of instructions, since all the remaining instructions 1, 2 and 3 access memory. The memory optimal code generation problem is now defined as follows:

Problem 1 (Optimal Code Generation (OCG)). Given an expression DAG \( D \), determine the shortest register machine program to evaluate \( D \) and to store its root nodes in memory.

It is shown in [2] that OCG is NP-complete for a one-register machine using the non-commutative instruction set. The following lemma is therefore stated without proof:

Lemma 1. OCG for a one-register machine using the non-commutative instruction set is NP-complete [2].

Exponential time algorithms for commutative and non-commutative instructions with any number of registers are given in [2].

B. SCAD Architectures

In this section, we first describe the basic structure and mode of operation of SCAD machines. Then, the code generation problem is defined. For a more detailed description of SCAD architectures, see [7].

A SCAD machine consists of a control unit (CU), a load/store unit (LSU), and processing units (PUs). These are connected by a synchronous move-instruction bus (MIB) and an asynchronous data transport network (DTN). SCAD is programmed by \textit{move} instructions that transport values between PUs. The MIB is used to send move instructions from the control unit to the processing units while the DTN is used to transport values between the processing units. Each PU – shown in orange color in Figure 2 – can have several input and output ports with unique addresses (for simplicity, only one output two input PUs are shown in Figure 2). Every port has a FIFO (first-in first-out) buffer to store the values arriving at that port. A PU reads values from its input buffers.
and writes the computed results to its output buffers. A buffer may have several entries, each of which is a pair \((adr, val)\). For input buffers, \(adr\) is the address of that output buffer that produces value \(val\). If \(val\) has not yet been transported, the special symbol \(\perp\) is used to denote this. For output buffers, \(adr\) denotes that input buffer to which \(val\) will be sent. \(\perp\) indicates a value that has not yet been produced. In Figure 2, the red cells indicate \(adr\) fields and green cells are \(val\) fields. Note that values in output buffers are produced by corresponding PUs while the values in the input buffers are transported from output buffers.

The meaning of a move instruction \(src \rightarrow tgt\) is to transport a value from the head of the output buffer \(src\) to the tail of the input buffer \(tgt\). The move instruction bus – shown in red color in Figure 2 – is used to register move instructions fetched by the CU, on the PUs. The PUs continuously snoop the MIB for instructions \((src, tgt)\). The output buffer \(src\) will add the entry \((tgt, \perp)\) to its tail. Similarly, the input buffer \(tgt\) will add the entry \((src, \perp)\) to its tail. If any of these buffers is full, a feedback signal \(fullBuffer\) is given. In that case, no buffer is written and the CU is stalled. Then, the instruction will be repeated until there is space in both buffers to successfully register the move instruction. Therefore, the move instructions are synchronously registered. In contrast, the execution in the PUs and the actual data transports are carried out asynchronously.

Operations are fired inside a PU if operand values are available at all its input buffer heads and if space is available in its output buffers to store the computed results. Assume a PU with \(n\) inputs and \(m\) outputs has values \(x_1, \ldots, x_n\) at the heads of its input buffers. If each of the \(m\) output buffers has enough free space, the PU fires and the output values \(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)\) are computed. These output values overwrite the pairs \((adr, \perp)\) closest to the heads of the corresponding output buffer, i.e., the entry \((adr, \perp)\) closest to the head of output buffer \(j\) will be replaced by \((adr, f_j(x_1, \ldots, x_n))\). Firing of PUs create messages to be transported by the DTN to appropriate input buffers.

A message in the DTN – shown in green color in Figure 2 – is a triple \((src, tgt, val)\) where \(src\) is the source output buffer where the value \(val\) was produced and \(tgt\) is the target input buffer to where the value \(val\) must be sent. Whenever an output buffer \(src\) finds the pair \((tgt, val)\) with \(val \neq \perp\) at its head, it will produce the message \((src, tgt, val)\) for transmission by DTN. Upon receiving the message, the input buffer \(tgt\) will replace the entry \((src, \perp)\) closest to its head with \((src, val)\). Note that both firing of PUs and transmission of data by the DTN proceeds asynchronously, as and when data is available.

The load store unit (LSU) – shown in Figure 3 – is a special processing unit that handles memory operations. It has the following four inputs: opcode \(opc\) (either \(store\) or \(load\)), a value \(val\) (for store operations), a memory address \(adr\), and the number of copies \(cps\) to be generated (by load operations). Clearly, \(load\) will load the value at memory address \(adr\) and enqueue \(cps\) copies of the same to the output buffer \(out\) of LSU. \(store\) will store the given value \(val\) at memory address \(adr\). We use two different kinds of stores in this paper: first, destructive store \(store_d\) that consumes the value \(val\) and produces nothing in the output buffer \(out\). Second, preserving store \(store_p\) that consumes value \(val\) and produces this \(val\) in \(out\).

In this paper, a simplified version of the SCAD architecture is considered. Every PU is a universal PU with four inputs and one output. As shown in Figure 3, the inputs are the left operand \(opl\), the right operand \(opr\), the opcode \(opc\), and the number of copies \(cps\). The opcode \(opc\) determines the type of binary operation, and \(opl, opr\) are the left/right operands of this binary operation. \(cps\) is the number of copies of the result that will be produced in the output buffer \(out\) of the PU. Note that a universal PU and the LSU can be combined into a single unit with four inputs and one output.

In the following, we explain the computation of a SCAD machine with the help of a simple example. Consider a SCAD machine with only one PU and one LSU to execute the following program:

\[
\begin{align*}
y_1 &= x_1 \odot x_2 \\
y_2 &= x_2 \odot x_1
\end{align*}
\]

where \(x_1\) and \(x_2\) are variables stored in memory at addresses \(adr(x_1)\) and \(adr(x_2)\), respectively. First, we load these values from memory into the output buffer of LSU by moving corresponding addresses and the \(load\) opcode to the respective input buffers of the LSU. Since both \(x_1\) and \(x_2\) are used twice in the above computation, two copies of each one must be produced in the output buffer. Therefore, \(cps\) is set to 2. Note
that the load operation does not consume any value from input buffer \( \text{LSU.val} \).

\[
\begin{align*}
\text{adr}(x_1) & \rightarrow \text{LSU.adr} \\
\text{load} & \rightarrow \text{LSU.opc} \\
2 & \rightarrow \text{LSU.cps} \\
\text{adr}(x_2) & \rightarrow \text{LSU.adr} \\
\text{load} & \rightarrow \text{LSU.opc} \\
2 & \rightarrow \text{LSU.cps}
\end{align*}
\]

After registering these move instructions, the load operations can fire. All inputs are consumed and the output buffer \( \text{LSU.out} \) then contains the values \([x_2, x_2, x_1, x_1]\) (from tail to head). To evaluate \( y_1 \) and \( y_2 \), these copies of \( x_1 \) and \( x_2 \) must now be moved to the input buffers of the PU. The first copy of \( x_1 \) (at the head of \( \text{LSU.out} \)) should be moved to the left operand input buffer \( \text{PU.opl} \) of the PU.

\[
\text{LSU.out} \rightarrow \text{PU.opl} \text{ (first } x_1 \text{)}
\]

After this move instruction is registered over the move instruction bus (MIB), the head of \( \text{LSU.out} \) will have the entry \((\text{PU.opl}, x_1)\) and the head of \( \text{PU.opl} \) will have the entry \((\text{LSU.out}, \bot)\). Next, the message \((\text{LSU.out}, \text{PU.opl}, x_1)\) is sent from \( \text{LSU.out} \) to \( \text{PU.opl} \) over the DTN. Thus, \( x_1 \) arrives at the head of the input buffer \( \text{PU.opl} \), but the other values must be moved as well.

\[
\begin{align*}
\text{LSU.out} & \rightarrow \text{PU.opr} \text{ (second } x_1) \\
\text{LSU.out} & \rightarrow \text{PU.opl} \text{ (first } x_2) \\
\text{LSU.out} & \rightarrow \text{PU.opr} \text{ (second } x_2)
\end{align*}
\]

The above sequence of move instructions transport one copy of \( x_1 \) to \( \text{PU.opl} \) and the other copy to \( \text{PU.opr} \) (analogously for \( x_2 \)). Once all moves are executed, both input buffers \( \text{PU.opl} \) and \( \text{PU.opr} \) will contain \([x_2, x_1]\). If the binary operation \( \circ \) were to fire now, \( x_1 \circ x_1 \) will be calculated since \( x_1 \) is at the head of both input buffers. Therefore, the values need to be reordered in the buffers in order to fire the operation with the correct operands. In order to be able to reorder values in buffers, the PUs implement duplication operations: \( \text{dup}_{\text{op}} \) and \( \text{dup}_{\text{op}} \). They consume a value from the head of the respective input buffer and produce \( \text{cps} \) copies of the value in the output buffer. For the example, note that only values in \( \text{PU.opr} \) need to be reordered as follows:

\[
\begin{align*}
\text{dup}_{\text{op}} & \rightarrow \text{PU.opc} \\
1 & \rightarrow \text{PU.cps} \\
\text{PU.out} & \rightarrow \text{PU.opr}
\end{align*}
\]

Now \( \text{PU.opr} \) contains values \([x_1, x_2]\) while \( \text{PU.opl} \) still contains values \([x_2, x_1]\). The binary operation \( \circ \) can now be fired twice to get the desired results.

\[
\begin{align*}
1 & \rightarrow \text{PU.cps} \\
\circ & \rightarrow \text{PU.opc} \text{ (compute } y_1) \\
1 & \rightarrow \text{PU.cps} \\
\circ & \rightarrow \text{PU.opc} \text{ (compute } y_2)
\end{align*}
\]

After all operations and data moves are executed, \( \text{PU.out} \) contains \([y_2, y_1]\). It is not difficult to see that duplication operations are necessary and cannot be avoided for this example. Clearly, duplications and subsequent data transports degrade performance and increase the power consumption. This kind of runtime overhead appears due to the restriction that one can access values only from the head of a FIFO buffer. This is referred to as queue overhead later in this paper. For more details see [6].

The limited number of registers of a register machine may make it necessary to temporarily store the contents of a register in main memory. The corresponding store/load instructions are often called spill code. Similarly, when the buffers in a SCAD machine have a bounded size, values have to be spilled to main memory as well. Since memory accesses are expensive, it is useful to consider memory optimal compilation that minimizes the amount of spills.

**Problem 2** (\( k \)-PU \( n \)-Bounded Memory Optimal Compilation). Given an expression DAG \( D \) and a SCAD machine with \( k \) PUs with size of all buffers \( n \) (\( k \) and \( n \) positive integers), determine a move program that evaluates and stores all values of \( D \), using a minimum number of memory accesses.

Clearly, the number of memory accesses in a SCAD machine is given by the number of load/store operations fired in the LSU.

![Fig. 4. Combined LSU/PU (a) with their respective buffers of the restricted SCAD architecture (opc and cps buffers are omitted here) and (b) a non-commutative one-register machine](image)

### III. Transformations

In this section, the optimal code generation problem for non-commutative one-register machines will be reduced to memory optimal code generation problem for 1-PU 1-bounded SCAD machines. The SCAD machine will use a combined LSU/PU as shown in Figure 4. Therefore, the labels \( \text{op}/\text{adr} \) and \( \text{opl}/\text{val} \) are used interchangeably. The reduction will use polynomial-time transformations between optimal SCAD code and optimal register code. In the following, we first present the transformations and then derive complexity results. Finally, extensions to arbitrary numbers of registers, buffer sizes, and PUs are discussed.

#### A. From SCAD Code to Register Code

In order to minimize memory accesses, it is necessary to keep previously computed values as long as possible in the processor. There are three buffers of unit size inside the 1-PU 1-bounded SCAD machine where values can be stored (see Figure 4), while in a one-register machine there is only one register to store a value. Recall that one-register machine can perform operations of the form \( r \leftarrow r \circ m \) and load (and store) value from memory to register (register to memory). While a SCAD machine has more space, we will show that it can still not perform any actions that would allow it to save memory accesses compared to the register machine. For that reason, the
SCAD machine would have to keep an additional value that is not directly loaded from memory in the processor and be still able to perform a computation independent of this value. This is similar to a second register in the register machine.

Two different types of values will be considered in the following: \( l \) (load) will denote a value that was directly loaded from memory. In the register machine, this would be a register value right after a load operation or the \( m \) operand of an operation. Therefore, \( l \)-values may in some cases not be stored in the register, as operands may come directly from memory. \( r \) (register) means that the value is the result of some binary operation. Because \( r \) is the result of a binary operation, it will always be stored in the register of the register machine. The difference between \( l \) and \( r \)-values will be of utmost importance in the following discussion. If the SCAD machine is able to produce two distinct \( r \)-values through operations, it can save memory accesses compared to the register machine. \( _-\) means that no value is currently present in the buffer. As there are not many cases for what value type is in which buffer inside the SCAD machine, they are all enumerated in Figure 5. Computations with immediate operands will not be considered for the sake of simplicity. Doing calculations with immediate operands cannot contribute to the desired results of the given problem as they only consist of operations on load variables and intermediate results. The only case where immediate operands are loaded into \( adr \) is when they are used directly as an address for memory operations. It will not be considered that load variables or intermediate results are moved into the \( cps \) or \( opc \) buffer.

As can be seen in Figure 5, all operations incorporate at most one previously computed \( r \)-value in the left operand. In a one-register machine, this value can be stored in the register. Storing \( l \)-values is of no interest to the reduction as this is obviously an unnecessary memory operation that cannot occur in an optimal program. The \( r \)-value can be stored either destructively or preserving. In register machines, values are always stored in the preserving fashion, but overwriting it afterwards is equivalent to the destructive one. Loading a \( l \)-value while an \( r \)-value is present in the machine and then using it in a binary operation is equivalent to the \( r \leftarrow r \odot m \) operation of the one-register machine. Note that \( r \) cannot be moved to \( opr \) while an \( l \)-value is present in the machine – and no variable can be loaded when \( r \) is in \( opr \) as this is also the \( adr \) field. Therefore, the data moving capabilities and additional buffer space in the SCAD machine do not matter. All 1-PU 1-bounded SCAD machine programs can be transformed to one-register programs using the same number of memory operations.

B. From Register Code to SCAD Code

Given a memory-optimal register program, the following method will produce SCAD move code with the same amount of memory operations. Operations on immediate values are again ignored for simplicity. It is assumed that on each store, it is known whether the content of register \( r \) will be used again or not. This can be achieved through dataflow analysis or by simply looking one instruction ahead and checking whether it overwrites \( r \) or not, as consecutive stores are not optimal. The transformation reads the program sequentially and uses a case distinction on the register instruction set given in Section II-A.

<table>
<thead>
<tr>
<th>state</th>
<th>PU values</th>
<th>description and successor states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>No valid values in the buffers. The only possible action is that a load using ( adr ) with an immediate operand produces a ( l )-value to ( out ). Possible successor states: 2.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A load variable ( l ) is present in the ( out ) buffer. As this blocks any further operations until moved away, there are only two non-stalling possibilities: ( l ) is moved to ( opr ) or ( opr ). Possible successor states: 3,4.</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>As ( adr ) is not free, another load variable ( l ) cannot be loaded. ( l ) can be moved away to ( out ) using a ( disp_{opr} ) operation. Possible successor states: 2.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>As ( adr ) and ( out ) are free, another variable can be loaded to ( out ). Alternatively, ( l ) can be moved to ( out ) or stored. Possible successor states: 1,2,6.</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>The only non-stalling option is to move ( l_2 ) to ( opr ). Possible successor states: 7.</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>The only non-stalling option is to move ( l_2 ) to ( opr ) or to destructively store ( l_1 ). Possible successor states: 2,7.</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Both ( l_1 ) and ( l_2 ) can be moved to ( out ). A store cannot occur as ( adr ) is not free. Alternatively, a binary operation can be performed producing ( r ) to ( out ). Possible successor states: 5,6,8.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>The only non-stalling option is to move ( r ) to ( opr ) or ( opr ). Possible successor states: 9,10.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>As ( adr ) is not free, another variable ( l ) cannot be loaded. ( r ) can only be moved to ( out ). Possible successor states: 8.</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>As ( adr ) and ( out ) are free, another variable ( l ) can be loaded. Alternatively, ( r ) can be moved to ( out ) or stored. Possible successor states: 1,8,11.</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>( l ) can either be moved to ( opr ) or ( r ) can be stored destructively. Possible successor states: 2,12.</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>A binary operation can be performed producing a ( r )-value to ( out ). Alternatively, ( r ) can be stored or moved to ( out ). ( l ) can be moved to ( out ) as well. Possible successor states: 3,8,11,13.</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>( r ) can be moved to ( opr ). ( l ) cannot be moved or stored as ( adr ) and ( out ) are both not free. Possible successor states: 12.</td>
</tr>
</tbody>
</table>

Fig. 5. Possible states of a 1-PU 1-bounded SCAD machine.
As we are transforming memory optimal register code to memory optimal SCAD code, we may obviously assume the optimality of the register code whenever necessary.

Case $r \leftarrow m$. If there was some content in the register before, it cannot be used anymore. However, in an optimal program, this had to be stored before since otherwise, there would have been unnecessary results, and thus unnecessary memory operations. It may therefore be assumed that no value is present in the SCAD machine. The following is thus added to the SCAD program:

\begin{align*}
\text{load} & \rightarrow \text{opc} \\
m & \rightarrow \text{opr} \\
\text{out} & \rightarrow \text{opl}
\end{align*}

The register token is moved to \text{opl} as this is assumed by the following operations.

Case $m \leftarrow r$. There must be a register token in the PU, otherwise, the register code would attempt to store an empty register which would not be optimal. The register token is in \text{opl}. If $r$ will be overwritten in the next instruction, a destructive read is used in the SCAD code:

\begin{align*}
\text{store}_d & \rightarrow \text{opc} \\
m & \rightarrow \text{opr}
\end{align*}

If $r$ is reused in the next instruction, a preserving read is used in SCAD. The register token is then moved back to \text{opl}:

\begin{align*}
\text{store}_p & \rightarrow \text{opc} \\
m & \rightarrow \text{opr} \\
\text{out} & \rightarrow \text{opl}
\end{align*}

Case $r \leftarrow r \odot_i m$. Again, the register token is in \text{opl}. Otherwise, the operation would be calculated on an empty register. Calculations on empty registers cannot contribute to the results of the program and are therefore at least not optimal.

\begin{align*}
\text{load} & \rightarrow \text{opc} \ (\text{load} \ m \ \text{to} \ \text{opr}) \\
m & \rightarrow \text{opr} \\
\text{out} & \rightarrow \text{opr} \\
\odot_i & \rightarrow \text{opc} \ (\text{perform} \ \odot_i) \\
\text{out} & \rightarrow \text{opl}
\end{align*}

C. Optimality of Transformed Code

As already mentioned, optimality of SCAD code and one-register machine code is determined by the minimal number of memory accesses.

Lemma 2. The register code $P$ resulting from the presented transformation of memory optimal SCAD code $S$ is optimal.

Proof: Proof by contradiction: Assume that $P$ with $n$ operations produced from $S$ with $n$ memory operations would not be optimal. Then, there exists a equivalent register machine program $P'$ with $n' < n$ operations. By the construction given before, this can be transformed to SCAD code $S'$ with $n'$ memory operations. However, this contradicts the assumption that $S$ is memory optimal.

Lemma 3. Problem 2 for $k = 1$ and $n = 1$ is NP-complete.

D. Extensions

Some parameters of the architecture and the problem can be generalized as follows.

Commutative One-Register Instructions. The instruction set may be changed to the commutative one-register instruction set discussed in Section II-A. The transformation will still work if the SCAD architecture is changed accordingly. As now $r \leftarrow m \odot_i r$ operations become possible in the register machine, these have to be enabled in the SCAD machine as well. This is done by making the LSU buffers commutative. Enabling both \text{opl} and \text{opr} buffers to become \text{adr} will make it possible to move $r$ to \text{opr} and still load a variable that is then put into \text{opl}. States shown in Figure 6 are thus added to the ones in the non-commutative case.

Fig. 6. Possible SCAD states added by enabling both \text{opl} and \text{opr} to become \text{adr}.

Fig. 7. Some of the states possible in an \text{out} buffer of size 2.

Proof: Since Problem 1 is in NP (Lemma 1), chaining the given polynomial-time transformation to an algorithm $A$ solving Problem 1 in NP will yield an algorithm solving Problem 2 for $k = 1$ and $n = 1$ (Lemma 2) that is in NP. Thus, Problem 2 for $k = 1$ and $n = 1$ is in NP. Now assume there exists an algorithm $A'$ solving Problem 2 in polynomial time. If this would be the case, chaining the polynomial time transformation to $A'$ will yield a polynomial time algorithm for Problem 1. Therefore, as Problem 1 is NP-hard, so is Problem 2.

Clearly, NP-hardness for the general case follows immediately.

Lemma 4. Problem 2 is NP-hard.
Increasing Buffer Sizes. Consider the commutative case and a combined PU/LSU unit with larger buffers. For simplicity, assume that only the size of buffer out is increased. It is easy to see that it does not matter which buffer’s size is increased as long as buffer out has at least size 2 to enable duplications. Let $n$ be the size of the buffer out. It can then be shown that this is equal to a commutative two-address $n$-register machine with an instruction set as given in Section II-A concerning memory operations. For each additional entry in the out buffer, a $r$-value can be stored independently from computations involving load variables and other $r$-values. In $n$ registers with a commutative instruction set, there can be $n$ independent computed $r$-values. They can interact with load variables in any way using the memory operations. As the register machine is a two-address machine, one of the registers involved in a binary operation is overwritten. Now consider the case of 2 registers and an out buffer of size 2. At most one of the $r$-values can be duplicated in the SCAD machine before performing a binary operation, as can be seen in Figure 7. Having 4 values in the 4 possible buffer positions would stall the machine. This is equivalent to one register being overwritten and the other one not by the operation in the register machine. Figure 7 only shows a selection of possible states, but note that a third $r$-value differing from the others cannot be produced. If no $r$-value is duplicated, a $l$-value can be loaded and perform an operation with any of the $r$-values, just like the memory operations in the register machine. The argument still holds when further extending the buffer sizes. If there is enough space in the SCAD machine to duplicate an additional value, there is also an additional register to which a value can be assigned with $r_i \leftarrow r_j$. As seen in the one-register machine, it also does not matter how often values are shuffled inside the SCAD machine. The $r$-values used for operations in the register machine can always be moved into the relevant position in the SCAD machine. If queue overhead is ignored, register assignments are ignored and only memory operations are considered, the space in each of the machines is therefore equally useful. The transformation method given before can thus be extended to give a transformation for the general case of longer buffers. However, queue overhead may considerably slow down the execution. If the worst case is assumed for transforming register code to SCAD code, the following holds.

Lemma 5. Minimal queue overhead when transforming memory optimal register code to memory optimal SCAD code is in $O(n \cdot l)$, where $l$ is the combined length of input and output buffers of the PU and $n$ is the number of operations.

Proof: When performing an operation $op_l \odot op_r$, the operands $op_l$ and $op_r$ have to be moved to the tail of their respective input buffers. Assuming the worst case for each of them, they are at the head of the wrong input buffer. So $op_l$ is at the head of the right input buffer. It then has to be duplicated through the right input buffer, the output buffer and the left input buffer. This yields $O(l)$ steps for a single operation. Assuming the worst case for all $n$ operations gives overhead $O(n \cdot l)$. □

Increasing the Number of PUs. Only one of the PUs is the combined PU/LSU to keep memory accesses consistent. However, apart from this, the previous discussion still holds. The space in addition to the one in the one-register machine can be used just like registers as long as there is at least one out buffer with size 2. Even if operations are executed in parallel, it is easy to see that this can be sequentialized for the $n$-register machine and executed with the same amount of memory operations. Using this argument, memory optimal code for any number of PUs and buffer sizes can be derived from the exponential-time algorithms for register machines. Therefore, following the same line of arguments as in Lemma 3, Problem 2 is also seen to be in NP.

Lemma 6. Problem 2 is NP-complete.

For a given total buffer size, increasing the number of PUs reduces the worst case minimal overhead. As shown in Lemma 5, the worst case minimal overhead is only dictated by the buffer depth $l$ and the number of operations. So, assuming buffer space is distributed evenly among $k$ PUs, buffer depth will be $\frac{l}{k}$, and therefore the overhead is reduced.

IV. EXPERIMENTAL RESULTS

As shown in the previous section, the same amount of internal processor memory in a register machine and in a SCAD machine are equally useful in both cases when only considering memory optimality. However, the different forms of processor memory come with different trade-offs. On the one hand, as shown in Lemma 5, memory optimal SCAD code obtained from memory optimal register code may contain queue overhead, bounded by $O(n \cdot l)$ where $l$ is the depth of the buffer in the SCAD machine. On the other hand, the circuit size of buffers grows only linearly with the size of the buffers. In a register machine with $l$ registers, although there is no queue overhead, the circuit complexity grows with $O(l^2)$. Therefore, when only considering memory optimality (ignoring the runtime queue overhead in SCAD), the SCAD machine scales better than the register machine. In fact, ignoring runtime overhead that may occur in the SCAD machine is justified, since it has been shown that certain programs (in particular expression trees) can be compiled without any runtime overhead even on a single PU SCAD machine as long as there is sufficient space in buffers [6]. More importantly, with multiple PUs in a SCAD machine, there exist more ways to reduce runtime overhead. The depth of individual buffers $l$ decreases as the buffer space is distributed among multiple PUs. Thus, there exists a trade-off between the amount of runtime overhead and the number of PUs in a SCAD machine. Moreover, the recent trend in processor architectures is to increase the number of cores or processing units (many-core architectures) to improve parallel execution of programs. We now show by experiments that a SCAD machine with enough PUs to avoid any runtime overhead and sufficient buffer space to avoid memory accesses offers impressive performance benefits over a SCAD machine using enough registers to avoid memory accesses.

To generate input programs for experiments, we 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. Each basic block is
compiled in two ways to generate move code: (1) Bypassing-based – we use Microsoft’s Z3 SMT solver (as described in [8]) to schedule the basic block on SCAD such that a minimal number of PUs are used that are required to eliminate any queue overhead and minimal time is needed to execute the basic block. Clearly, the resulting move code is executed on a SCAD machine without any registers and all values are moved directly between PUs. Furthermore, minimum buffer size is used so that all intermediate values in the basic block are accommodated in buffers and no load/store instructions are needed for intermediate values. (2) Register-based – we first allocate basic block variables to a minimum number of registers using the well-known Chaitin-Briggs heuristic [9], [11] (that yields nearly optimal results) such that all intermediate values are mapped to registers. Then, the SMT solver is used to schedule the basic block on SCAD with the same number of PUs (as previously determined) minimizing the execution time of the basic block. Clearly, the resulting move code is executed on a SCAD machine containing registers. All values are written to and read from registers. Note that in both cases load and store instructions are only used to load source variables (leaf nodes of DAGs) and store final results (root nodes of DAGs). In a SCAD machine without registers, all intermediate values are accommodated in minimum sized buffers while in a SCAD machine with registers, a minimal number of registers is used for the same. A cycle accurate SCAD simulator\(^1\) is used to execute the move programs. We assume unit latencies for PUs, data transport, memory and register accesses. Both memory and register files have a single read/write port. As mentioned in [8], with a timeout of 60 seconds, DAG sizes up to 12 nodes were successfully compiled by the SMT solver on an 2x Intel Xeon CPU X5450 (4x3.0GHz) 64bit computer with 32 GB RAM running the Ubuntu 16.04 operating system.

The average time taken to execute basic blocks of different sizes (number of nodes) is shown in Figure 8. Measurements are shown for both SCAD machine with and without registers. As expected, the execution time of register-based move programs increases more steeply compared to that of bypassing-based move programs, as the basic block size increases. This is because with more number of intermediate values in the basic block, there are more intermediate values that are written to and read from registers when executing register-based move program. Instead, these intermediate values are directly transported between PUs when executing bypassing-based move code. Figure 9 that shows the average number of data transports by the register file (in SCAD machine with registers) and by each PU (in SCAD machine without registers), to execute basic blocks of different sizes. Recall that both bypassing-based and register-based move programs have the same number of load and store instructions, and therefore, also the same number of data transports (communication) by the load-store unit. In executing the register-based move program, the remaining data communication (shown by the Reg comm line in Figure 9) happens via the limited number of ports in register file (a single port in our experimental set up). While executing the bypassing-based move program, the remaining data communication (shown for each PU by the PU comm line in Figure 9) happens via ports of all PUs in the SCAD machine. Since every PU has dedicated input and output ports, the data communication finds enough bandwidth even for larger programs. This is especially true for the advent of manycore architectures. However, executing register-based move code for larger programs soon leads to a bottleneck in accessing a large set of data via a limited number of ports in register files. This shows that bypassing-based move code enables a more distributed computing and communication that is scalable to larger programs exhibiting more parallelism.

The impact of the number of intermediate values on the execution times is more apparent in Figure 10. Figure 10 shows the average time taken to execute 12-node basic blocks of different levels. As the number of levels increases, the number of intermediate values also increases for basic blocks of the same size, thus leading to more reads and writes of registers during register-based move program execution. Therefore, register-based move programs take longer time to execute compared to bypassing-based move programs with increasing the number of levels of the basic block. Figure 9 shows the average number of data transports by the register

\(^1\)http://es.cs.uni-kl.de/tools/teaching/ScadSim.html
V. RELATED WORK

The RAW machine [16] consists of processor cores (every core containing a register file) that are arranged in a 2D tiled architecture with routers between them. The compiler statically schedules programs to run across all processor cores. Wavescalar [22] and TRIPS [10] are based on the explicit dataflow graph execution (EDGE) paradigm. Both fetch blocks of instructions and execute them on an array of PUs with buffers at their inputs and outputs. While TRIPS compilation [18] relies on register accesses to execute instruction blocks, Wavescalar does not use registers at all. However, Wavescalar requires complex tag-matching hardware. Compilation for both TRIPS [18] and Wavescalar [26] is based on heuristics. Similar to SCAD, transport-triggered architectures (TTAs) [12] are programmed by move instructions where computation is performed as a side effect whenever new inputs arrive at a PU. Outputs of PUs are connected to the inputs using an interconnection network. The compiler is responsible not only for scheduling these moves, but also for bundling independent moves where each bundle is executed by the hardware in one step. An optimal code generator for TTAs based on linear integer programming is presented in [3]. Many heuristics are also proposed for generating move code for TTAs. Though these code generators bypass the use of registers, TTAs still need register files to store some intermediate results since PUs in TTA have registers instead of buffers at their input and output ports. The MOVE-Pro architecture [15] improves the freedom of bypassing by employing multiple registers at the outputs of PUs along with other optimizations to the TTA architecture. The statically scheduled Mill architecture\(^2\) uses a fixed-length FIFO (called Belt) to store operands and results of execution and this way completely removes the general purpose register file. Old results are pushed out when the new ones are enqueued. However, unlike the buffers in a SCAD machine, the PUs in the Mill CPU may directly read operand values from any location in the Belt. The execution paradigm of the Synchronous Transfer Architecture (STA) [27] is closer to SCAD with buffers at the outputs of PUs and a configurable interconnect that allows PUs to receive operands for execution from these output buffers. Both operations to be executed on PUs and the interconnect configurations are encoded in instructions and execute a static schedule of instruction bundles (like in standard VLIW). Optimal code generation based on integer linear programming, proposed for STAs [14] bypasses register usage only within adjacent instruction bundles. Flexcore [23], a 91-bit native instruction set architecture, encodes both the operations to be executed on individual PUs and connections enabled by flexible network that connects PUs. Similarly, in the explicit datapath wide single instruction multiple data (SIMD) architecture [24], a set of PUs are arranged in a circular layout where each unit is connected to its left and right neighbors. It is programmed using very long instructions that encode for each PU, the source and destination of its operands. Although compilation for both Flexcore [19] and explicit datapath wide SIMD [21] tries to maximize bypassing, registers are still required while our queue-based code generation for SCAD completely eliminates the use of registers.

VI. CONCLUSIONS

We described a strong relationship between machines based on local registers and others based on local buffers like SCAD. For both machines’ programs, spill code has to be added to temporarily store intermediate results in the main memory if these cannot be kept in the processors. We analyzed the relationship between the code generation problems where the amount of memory operations is to be minimized. We found polynomial-time transformations between register-based and buffer-based machine code with the same amount of memory operations. As the register code problem was shown to be NP-complete, this also implies the NP-completeness for the SCAD code generation problem.

\(^2\)https://millcomputing.com/technology/docs/belt
Furthermore, this shows a one-to-one correspondence between registers and entries in the input and output buffers of PUs. Hence, known algorithms and heuristics for register-based code generation can be directly applied to SCAD machines using the transformation given in this paper. However, the transformation may introduce queue overhead proportional to the length of the buffers. Because buffers are queues, their values are not arbitrarily accessible and hence overhead to access certain values can become necessary. Reducing this queue overhead is not solved by the presented method. Programs generated solely by transforming register code will also not make full use of the parallel computing capabilities of SCAD machines.

Finally, the conducted experiments show that SCAD machines using buffers are superior to machines using a central register file, if queue overhead is eliminated through a sufficient amount of PUs and optimal compilation. As shown, both will need the same amount of memory operations. However, using buffers and the direct communication between PUs will lead to less data moves and less execution time.

REFERENCES


