Complexity Analysis of Code Generation for the SCAD Machine

Markus Anders

October 26, 2017

Bachelor Thesis

Technische Universität Kaiserslautern
Department of Computer Science

First reviewer	Prof. Dr. Klaus Schneider
Second reviewer	M. Sc. Anoop Bhagyanath

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kaiserslautern, den 26. Oktober 2017

Markus Anders

Zusammenfassung
Synchronous Control Asynchronous Dataflow (SCAD) Architekturen sind eine neue Art von Exposed Datapath Architekturen die FIFO Puffer an allen Ein- und Ausgängen der Processing Units (PU) verwenden. Alle PUs und Datenleitungen sind dem Compiler bekannt und es ist dessen Aufgabe, diese optimal auszunutzen. Da SCAD Maschinen nicht notwendigerweise Register enthalten und für das Verschieben von Werten zwischen den FIFO Puffern zusätzliche Kosten entstehen können, unterscheiden sich die Codegenerierungsprobleme für SCAD Maschinen von traditionellen Problemen für Registermaschinen. In dieser Arbeit wird die Komplexität für einige dieser Codegenerierungsprobleme analysiert. Für den Fall, dass die Größe der FIFO Puffer limitiert ist, wird eine Relation zu Problemen für Registermaschinen gezeigt. Diese Relation genügt um anschließend NP-Vollständigkeit für das SCAD Problem zu folgern. Im Weiteren wird die Größe der FIFO Puffer nicht limitiert und die Zusatzkosten für FIFO Puffer betrachtet. Für dieses Problem kann dann das Knoteneinfärbungsproblem für ungerichtete Graphen reduziert werden.

Abstract
Synchronous Control Asynchronous Dataflow (SCAD) architectures are a new kind of exposed datapath architectures. SCAD uses FIFO buffers at each input and output of its processing units (PU). All of the PUs and datapaths are exposed to the compiler. It is the compilers task to fully utilize them. The resulting code generation problems for SCAD machines differ from the ones in register machines as there may be no registers. Moreover, additional overhead in moving values between FIFO buffers may occur. In this thesis, the complexities for some of these new code generation problems are analyzed. First, the case of bounded buffers and memory optimal compilation of basic blocks is considered. The problems for SCAD machines are closely related to similar register machine problems and thereby also shown to be NP-complete. Next, unbounded buffers are assumed and the overhead created by the FIFO buffers is considered. It is shown that this problem is NP-hard by a reduction of graph coloring.

Contents

1 Introduction

1.1 Motivation

The traditional von-Neumann architectures follow a simple control model that fetches instructions sequentially according to a program counter. While using several cores enables processors to leverage thread-level parallelism (TLP) explicitly expressed by the programmer, this approach can not fully exploit existing parallelism on the instruction-level present in many applications [1]. In contrast, operations in dataflow architectures are not triggered by a program counter, but by the availability of their operands. Thus they do not use a sequential execution style and are in principle able to maximize instruction-level parallelism (ILP)[1]. There are however significant challenges when implementing dataflow architectures. They are either not able to exploit significant amounts of parallelism in the applications or have to deal with tag matching problems. Tag matching severely decreases computing performance and power efficiency while increasing chip size. Moreover, traditional imperative programming languages do not map well to dataflow architectures [1].

As both the von-Neumann and dataflow model have deficiencies, a tradeoff between the two seems desirable. There are many hybrid dataflow/von-Neumann architectures that try to find a good balance between the inefficiencies of dataflow architectures and the limitations of the von-Neumann execution model [1]. The presented architectures are categorized based on the use of a traditional von-Neumman program counter, use of central registers or destructive reads and the use of very long instruction words (VLIW). The role of compiler and processor in exploiting ILP is explored as well.

WaveScalar [2] is a dataflow architecture with the ability to use von-Neumann memory semantics. Instructions are grouped into large blocks called waves. These are connected acyclic portions of the control flow graph. Data elements are identified with a wave number which increases whenever they enter a new wave. Using this number, memory accesses can be ordered accordingly to achieve the desired semantics to compile imperative languages to the architecture easily. It is however not required to use this ordering mechanism. WaveScalar thus neither has a program counter nor employs a central register file. While being able to more easily compile traditional programming languages to the architecture does seem like a significant improvement over pure dataflow architecture, there are still similar

tag matching issues.

Superscalar architectures differ from the von-Neumann model in using dynamic scheduling as introduced by the Tomasulo algorithm [3] that allows out-of-order execution of instructions in a sequential program. They use a program counter, have a central register file and do not employ VLIW. The processor is responsible for the out-of-order execution of the instructions. Data dependencies among instructions are dynamically analyzed at runtime in a fixed instruction window. Instructions are then scheduled to different pipelines that work in parallel. In the commit stage, results are written back to the register file in program order again. There is no parallel write-back to registers. The commit stage can thus bottleneck the system by giving back pressure to the rest of the pipelines. Instruction scheduling at runtime also increases chip size and power consumption [4].

Very long instruction word (VLIW) architectures [5] avoid these drawbacks by enabling parallel write-back to the register file. Furthermore, instructions that can run in parallel are determined at compile-time. These are packed into a single very long instruction word. VLIW architectures also use a program counter and a central register file. All instructions have to commit their results back to a register. Thus, the number of registers limits the amount of instructions that can be issued in a single VLIW. Moreover, the register file can quickly become a hot spot [6] that in turn limits scalability of the amount of instructions present in the processor. Increasing the number of registers may also mean a change in the instruction set itself, which can be disruptive to all aspects of a computer system [7].

The RAW microprocessor is structured as a 2D mesh of tiles [8]. Each tile contains ALUs, a portion of the on-chip memory, a portion of the registers and is directly connected to its neighbours. Due to this structure, it has non-uniform access times to registers. The compiler places instructions onto the tiles which makes instruction scheduling both a spatial and temporal problem. However, only a portion of registers actually has to be wired to each ALU reducing register file pressure. The architecture uses a program counter.

The TRIPS architecture [7] reduces the use of registers with the concept of hyperblocks. Hyperblocks consist of up to 128 instructions that are grouped together and mapped to execution units by the compiler. Data dependencies among these instructions are encoded explicitly into the instruction set and are thus determined by the compiler. These hyperblocks are then executed with direct instruction communication bypassing the use of registers. How-

ever, the problem is thereby only shifted to the block level: after a hyperblock is executed its results need to be committed to the central register file. At the block level TRIPS also uses a program counter.

To avoid dynamic data dependency checks inside the processor and to bypass the use of a central register file, Transport-Triggered Architectures (TTA) [9] expose all datapaths and function units (FU) to the compiler. The compiler issues move instructions between the inputs and outputs of the FUs instead of operations. When an operand is moved to a specific trigger input of the FU, an operation is triggered and the results are calculated. This enables compilers to directly communicate intermediate results between function units rather than using the central register file. In order to issue moves, TTA uses a program counter.

The Synchronous Control Asynchronous Datapath (SCAD) architectures [4] try to incorporate ideas of TTAs and dataflow architectures by replacing registers at the inputs and outputs of a function unit with FIFO buffers. Similar to dataflow architectures, values of buffers are consumed when read. Move instructions between the buffers are registered synchronously but the actual data moves and instruction execution can be later and asynchronous. For more details, see Section 2.4. This enables - among other advantages - variable execution times of operations without the need to expose these to the compiler. It has been shown that new approaches to compilation can fully exploit instruction-level parallelism in these architectures without the use of a register file [10]. The SCAD architectures discussed in this thesis use program counters. However, it has been shown in [11] that a few modifications make it possible to drop them in dynamically ordered SCAD (DO-SCAD) architectures. These need tag matching hardware, though.

Traditional code generation problems that focus on the efficient usage of registers are known to be NP-complete [12, 13, 14]. However, since registers may not exist in SCAD architectures, these results do not directly apply to them. There are also NP-completeness results for general problems concerning partitioning and scheduling parallel programs to multiprocessors given in [15] that are referenced for exposed datapath architectures like RAW [8]. How to reduce them onto the more limited problem of SCAD code generation does however not seem to be obvious. Especially since there do not seem to be direct applications of existing algorithms to SCAD code generation, knowing complexity results or lower bounds for the specific subproblems would alleviate the risk of overlooking simple, efficient and optimal algorithms.

1.2 Contribution

Space in two-address register machines is related to the buffer space in SCAD machines with bounded buffers. The minimal amount of memory operations in the compiled program is considered for these relations. Polynomial time transformations from memory optimal one-register code to memory optimal SCAD machine code are described that imply NP-completeness of the problem for SCAD machines with bounded buffers.

Graph coloring is reduced to queue optimal compilation of programs with control flow on SCAD machines with unbounded buffers. Queue optimal means that no additional operations are added that only move tokens from input buffers to output buffers. This shows NP-hardness of the stated queue optimal code generation problem for SCAD machines.

1.3 Outline

Next, preliminaries are cited in Section 2 starting with basic definitions for graph and code generation problems. Necessary models and definitions for register machines and SCAD machines are also introduced. Section 3 deals with the aforementioned relation between register machines and bounded buffer SCAD machine code generation problems. Then, Section 4 continues by reducing graph coloring to the unbounded buffer SCAD machine code generation problem. The last section summarizes the conclusions for each reduction and outlines possible future work. The main contributions of this thesis are in Section 3 and Section 4.

2 Preliminaries

First, basic graph notations and the well-known coloring problem are needed for the reduction for SCAD machines with unbounded buffers. As the reduction for bounded buffers is based on optimal code generation for register machines, fundamental notations and results for code generation are cited. Instruction sets for register machines and related complexity results are given. Finally, SCAD machines and different kinds of code generation problems for SCAD machines are discussed in this chapter.

2.1 Graphs

Only the graph coloring problem for undirected graphs is considered in this thesis. Therefore, all graphs $G = (V, E)$ will denote an undirected graph G with a set of vertices V and edges E. As all edges are undirected, they will be considered to be sets of size 2 so $E \subseteq \{e \in 2^V \text{ with } |e| = 2\}$. The well-known graph coloring definitions of [13] are used for the purposes of this thesis.

Problem 1 (GRAPH k-COLORABILITY).
INSTANCE: Graph $G = (V, E)$, positive integer k
QUESTION: Does there exist a function $\phi : V \to \{1, 2, ..., k\}$ such that $\phi(u) \neq \phi(v)$ whenever $\{u, v\} \in E$?

As given in [13] the following lemma is stated without proof.

Lemma 2.1. *GRAPH k-COLORABILITY is NP-complete.*

This also holds for any fixed $k \geq 3$.

2.2 Code Generation

The code generation problems that are described later will mostly use basic blocks as their input. Two means of representing a basic block are used interchangeably. Three-address code in static single assignment form (SSA) is used as shown in Example 1. A basic block is said to be in static single assignment (SSA) form if every variable is only assigned once. In this thesis all basic blocks are considered to be in SSA form. Variables that do not appear on a left hand side of any assignment are called load variables. These will have to be loaded from memory by the target machine. As shown in [14] basic blocks in SSA form can be transformed into an expression directed

$$
\begin{aligned}
i_1 &= x_1 + x_2 \\
y_1 &= i_1 \times x_2 \\
y_2 &= i_1 + x_2
\end{aligned}
$$

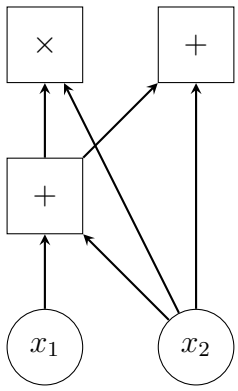


Figure 1: Example code and the corresponding expression DAG.

acyclic graph (expression DAG) in which the vertices represent variables and edges dependencies. Load variables are the leafs of expression DAGs.

Example 1. See Figure 11 for the three-address code and corresponding expression DAG. The example is taken from [4]. Note that each subresult becomes an inner node, each endresult a root and each load variable a leaf of the expression dag.

Programs with control flow will be necessary as input for the SCAD code generation problem with unbounded buffers. To simplify matters it will however be assumed that these only consist of basic blocks with control flow between them. Definitions for branching or looping constructs will not be added to the three-address code. The control flow will be given as a graph with directed edges between the basic blocks. Edges indicate possible control flow similarly to [12]. It will be assumed that code generators may only generate code for each basic block once. No basic blocks may be added or removed from the program.

2.3 Register Machines

For register machines, the model presented in [14] is used. The code generation problems are assumed to produce code for a two-address one-register machine using one of the instruction sets given in Figure 2. The non-commuative instruction set only has the memory operation with the memory address used in the right operand. The commutative set can use it in both the left and

Non-commutative

(1) $r \leftarrow m$ (load)
(2) $m \leftarrow r$ (store)
(3) $r \leftarrow r$ op m (memop 1)

Commutative

(1) $r \leftarrow m$ (load)
(2) $m \leftarrow r$ (store)
(3) $r \leftarrow r$ op m (memop 1)
(4) $r \leftarrow m$ op r (memop 2)

Commutative Multi-Register

(1) $r_i \leftarrow m$ (load)
(2) $m \leftarrow r_i$ (store)
(3) $r_i \leftarrow r_i$ op m (memop 1)
(4) $r_i \leftarrow m$ op r_i (memop 2)
(5) $r_i \leftarrow r_j$ (assign)
(6) $r_i \leftarrow r_i$ op r_j (regop 1)
(7) $r_i \leftarrow r_j$ op r_i (regop 2)

Figure 2: Instruction sets for register machines.

the right operand.

Example 2. See Figure 3 for the example register code. x_1 and x_2 are initially stored in memory. $M(x_1)$ and $M(x_2)$ refer to their initial addresses in memory. $M(i_1)$, $M(y_1)$ and $M(y_2)$ are some other memory addresses used to save the respective variables. The given code is non-commutative.

The optimal code generation (OCG) problem is defined as follows:

Problem 2 (OCG ONE-REGISTER).
INSTANCE: DAG D
QUESTION: What is the shortest machine program M that evaluates and stores all roots of D?

In [14], it is shown that Problem 2 is NP-complete for the non-commutative instruction set. The following lemma is therefore given without proof.

Lemma 2.2. *Problem 2 for the non-commutative instruction set is NP-complete.*

$i_1 = x_1 + x_2$	$r \leftarrow M(x_1)$ (load x_1)
$y_1 = i_1 \times x_2$	$r \leftarrow r + M(x_2)$ (calculate i_1)
$y_2 = i_1 + x_2$	$M(i_1) \leftarrow r$ (save i_1)
	$r \leftarrow r \times M(x_2)$ (calculate y_1)
	$M(y_1) \leftarrow r$ (save y_1)
	$r \leftarrow M(i_1)$ (load i_1)
	$r \leftarrow r + M(x_2)$ (calculate y_2)
	$M(y_2) \leftarrow r$ (save y_2)

Figure 3: Example code and corresponding one-register machine code.

Additionally, exponential-time algorithms for both the commutative and non-commutative case are given in [14] for any number of registers. Only the amount of sharing between variables dictates the exponential factor of their runtime.

2.4 SCAD Machines

First, the basic structure and mode of operation of SCAD machines are described. Then, machine code and code generation problems for SCAD machines are given. The descriptions are based on the ones given in [4].

Basic Structure. A SCAD (synchronous control asynchronous dataflow) architecture basically consists of a control unit (CU) and processing units (PU) that are interconnected by a synchronous move instruction bus (MIB) and a data transport network (DTN). The synchronous move instruction bus is used to send move instructions from the control unit to the processing units. The data transport network is used to transport values between the processing units.

Processing Unit I/O. A processing unit - shown in orange in Figure 4 - can have several input and output ports with a unique address each. Every port also has a FIFO buffer behind it that buffers the values going to the port. It reads values from its input buffers and writes them to the output buffers. A buffer entry is a tuple (adr, val). For input buffers, adr is the address of the output buffer that produces value val. If val has not been delivered yet the special symbol $\perp$ is used to denote this. For output buffers,

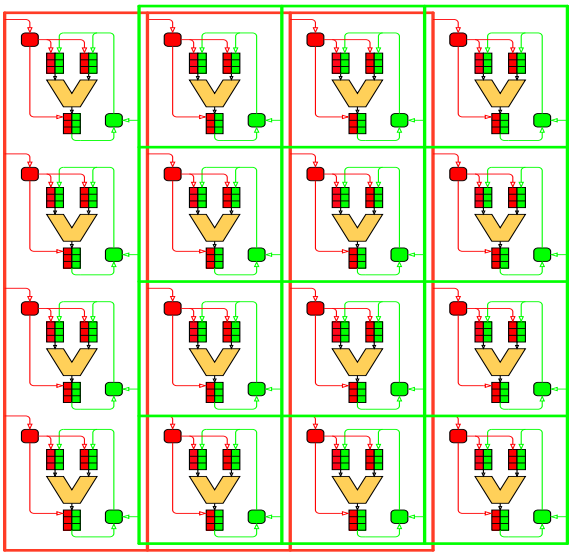


Figure 4: Architecture of a SCAD processor taken from [4].

adr denotes the input buffer to which *val* will be sent and $\perp$ indicates a value that has not been produced yet. In Figure 4 red cells indicate *adr* fields and green cells *val* fields. Buffers of processing units can either be bounded by a natural number n or unbounded. In the following, this will be referred to as "bounded buffer SCAD machine" and "SCAD machine with bounded buffers" interchangeably.

Move Instruction Bus (MIB). The move-instruction bus - shown in red in Figure 4 - is used to send values from the control unit to the processing units. The processing units continously snoop the MIB for instructions (*src*, *tgt*). If an output buffer has address *src*, it will add (*tgt*, $\perp$) to its buffer. Similarly for input buffers with address *tgt*, (*src*, $\perp$) is added to the buffer. If any of these buffers is full, a feedback signal $fullBuffer$ is given, no buffer is written and the CU is stalled. Thus, the instruction will be repeated until a buffer entry is freed and the move instruction is sucessful.

Data Transport Network (DTN). A message in the data transport network - shown in green in Figure 4 - consists of (*src*, *tgt*, *val*) where *src* is the source output buffer where *val* was produced and *tgt* is the target input buffer the message is being sent to. When an output buffer *src* sees (*tgt*, *val*)

with $val \neq \perp$ at its head, it will produce the message (src, tgt, val). In the input buffer tgt the entry (src, $\perp$) closest to its head will be replaced with (src, val).

Operation Firing. Assume a PU with n inputs and m outputs has values $x_1, ..., x_n$ at the heads of its respective input buffers that are not $\perp$. If each of the m output buffers has enough free space the corresponding operation will fire and produce output values $f_1(x_1, ..., x_n)$, ..., $f_m(x_1, ..., x_n)$ for every output buffer. Output values overwrite (adr , $\perp$) closest to the head of their respective output buffer with (adr, $f_j(x_1, ..., x_n)$) where adr can either be an address or $\perp$.

Load-Store Unit (LSU). The load store unit is a special processing unit that handles memory operations. It has inputs opcode opc, value val, address adr and number of copies cps. There are opcodes $store$ and $load$: $load$ will load the value at memory address adr and output it cps times to the output buffer of the LSU. It does not use val. $store$ will store val at memory address adr and does not use cps. In this thesis two different kinds of of stores will be used. First, there is the consuming store $store_c$ that consumes val and produces nothing to out. Furthermore, there is the preserving store $store_p$ that will additionally output val to out.

Universal Processing Units. In this thesis, a simplified architecture is assumed in which each PU - apart from the LSU - has the same operation firing function. Every PU has four inputs and one output. The inputs are left operand opl, right operand opr, opcode opc and amount of copies cps. opc is the opcode determining the type of binary operation. opl and opr are left and right operands of this operation. cps is the amount of copies that are to be produced to the output buffer of the PU. Note that a universal PU and LSU can be combined into a single PU.

Move Code. Machine code for SCAD machines consists only of simple move instructions like $source \rightarrow target$. This will initiate a data move from $source$ to $target$ following the semantics explained previously.

Queue Overhead. Figure 5 shows the basic structure of a queue machine. Queue machines follow a similar execution pattern as SCAD PUs and thus code generation for queue machines has been studied as a method for SCAD code generation in [4]. As shown there it is often necessary for queue machines to shuffle values of the queue using special queue operations dup and

Instruction	Description
load adr, n	Load data at memory address *adr* and enqueue n copies of it
store adr	Store value from the head of the queue at *adr*
opcode n	Dequeue necessary operands and enqueue n copies of the resulting value
swap	Dequeue two operands and enqueue them in reversed order
dup n	Dequeue one operand and enqueue it n times

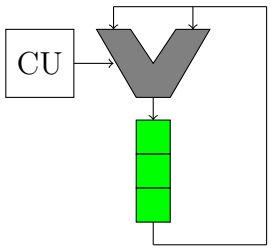


Figure 5: Structure of a queue machine and its corresponding instructions. Based on the one given in [4].

swap. *dup* duplicates a value a given n times and *swap* dequeues two values and enqueues them in reversed order again. In [4] it was shown that all queue machine programs can be translated to unbounded buffer SCAD machine programs. These do not need any spill code. Queue machines are known to be Turing complete [16]. Turing completeness for SCAD machines with unbounded buffers not needing spill code therefore also follows immediately. While it has been shown in [4] that when directly transforming queue machine code to SCAD code, some *dup* and *swap* operations may become unnecessary. It is also shown however, that not all of them can be alleviated. Clearly it is not desired to add unnecessary operations as this results in longer and potentially slower code. In the optimal code generation problems queue overhead will therefore be considered. When dealing with a multiple input PU dup_x will refer to the *dup* operation that duplicates the value of input buffer x. Using this definition, no input buffer determining the target input buffer of the duplicate is necessary. In a universal PU *dup* and *swap* operations are always assumed to exist. They are however limited to the *opl* and *opr* buffer. Being able to duplicate *cps* or even *opc* is not considered.

Code Generation with Unbounded Buffers. In SCAD machines with unbounded buffers all values of a program can be stored within the processor. No spill code is necessary. Since memory access is usually very expensive, it will be assumed that the minimal amount of memory accesses is used. This means one load operation per load variable and no stores. Overhead can then occur in the form of queue operations *dup* and *swap* only. The code generation problem will thus assume optimal compilation in terms of queue operations. The reduction for this problem will make use of control flow. Therefore, the input to the problem are programs with control flow instead of basic blocks.

Problem 3 (k-PU QUEUE OPTIMAL COMPILATION).
INSTANCE: Program P, positive integer k
QUESTION: Is it possible to compile P without *dup*, *swap*, *store* and only one memory operation per load variable on k PUs with unbounded buffers?

While this thesis uses this version of the problem for the sake of the reduction, other variants should be considered. An optimizing variant actually finding a schedule with minimal queue overhead for P has obvious practical use. This problem is however at least as hard. Using the same input, a minimal schedule with no *dup* and *swap* would answer the decision variant with yes and one with at least a single *dup* and *swap* with no. While memory access is certainly expensive it would still make sense to consider both memory overhead and queue overhead at the same time in a weighted fashion. This is however at least as hard again, since weighing memory operations with 0 would yield the answer to the decision problem once more. Also finding the minimal amount of PUs necessary to compile the program without overhead could be useful. Again, this is at least as hard as the decision problem. If the number of PUs necessary is greater than the one considered by the decision problem, it is clearly not possible to compile the program without queue overhead. If not, it is possible. So if NP-hardness for Problem 3 can be shown, NP-hardness for all of the stated variants can be concluded. The decision variant limited to basic blocks has been mapped to a SAT encoding in [4] proofing membership in NP.

Code Generation with Bounded Buffers. In the case where all buffers are bounded, space in the processor is limited. As in register machines with a limited amount of registers, values may have to be spilled to main memory.

It is thus useful to look at memory optimal compilation that minimizes the amount of spills that have to be done.

Problem 4 (k-PU n-BOUNDED MEMORY OPTIMAL COMPILATION).
INSTANCE: Expression DAG D, positive integer k, positive integer n
QUESTION: Which move code M evaluates and stores all values of D with the least amount of memory operations on k PUs in which all buffers are bounded by n?

For the sake of the reduction in Section 3 the stated problem variant does not consider queue operations at all.

3 Code Generation with Bounded Buffers

In this section, the optimal code generation problem for non-commutative one-register machines (Problem 2) will be reduced to memory optimal code generation for a 1-PU universal SCAD architecture that is restricted to buffer sizes 1 in all the buffers (Problem 4). The described SCAD architecture is shown in Figure 6. Note that LSU and PU are combined into a single unit and buffers only have size 1. The machine supports preserving and consuming stores. *opl* acts as the *val* buffer and *opr* acts as the *adr* buffer. The different names for the buffers are used interchangeably. Using this order, the SCAD machine will become equivalent to the non-commutative version of the one-register machine. This will become apparent in the following discussion.

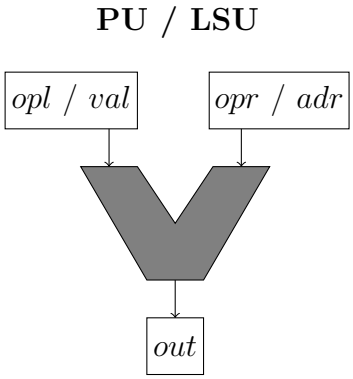
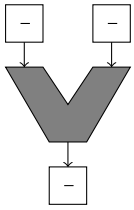


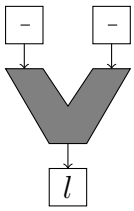
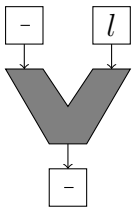
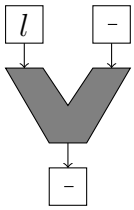
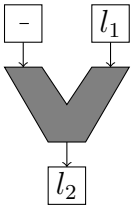
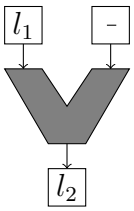
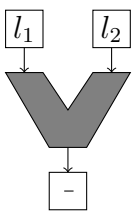
Figure 6: Combined LSU / PU and their respective buffers of the restricted SCAD architecture. *opc* and *cps* buffers are omitted. All buffers have size 1.

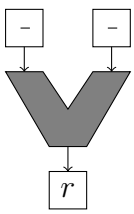
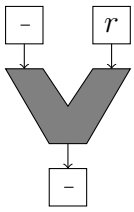
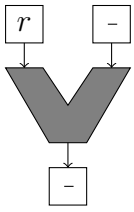
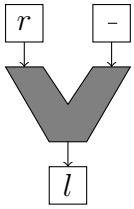
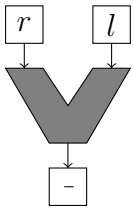
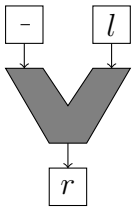
The basic idea of the reduction is to define polynomial time transformation methods from move instruction code to register machine code and vice-versa that need the same amount of memory operations in both. Thus, a polynomial time memory optimal code generator for one could be chained to the transformation method to give a polynomial time memory optimal code generator for the other. If one of them is NP-complete, the same would follow for the other immediately.

3.1 SCAD Code to Register Code

In order to reduce memory accesses, it is necessary to keep previously computed values somewhere in the processor. There are three places inside the SCAD machine where values can potentially be stored. In the one-register machine there is only one. It is however easy to see that the SCAD machine can not perform any actions that make it save memory accesses compared to the commutative one-register machine. For that it would have to keep one value that is not directly loaded from memory in the processor and be still able to perform a computation independent of this value. As there are not many cases they will be enumerated in the following table. Two different types of values will be considered. l (load) will denote a value that was directly loaded from memory while r (register) means that the value is the result of some binary operation. Because r is the result of a binary operation, it will be stored in the register in the one-register machine. l values may in some cases not be stored in a register, as operands may come directly from memory in the one-register machine. Therefore, the difference between l and r values will be of utmost importance in the following discussion. _ means that no value is currently present in the buffer. Computations with immediate operands will not be considered for the sake of simplicity. Note that doing calculations with immediate operands can not contribute to the desired results of the given problem as they only consist of operations on load variables and subresults. The only case where immediate operands are loaded into *adr* is when they are used directly as an address for memory operations. It will also not be considered that load variables or subresults are moved into the *cps* or *opc* buffer.

#	State	Description
(1)		No value is currently present in the machine. The only possible action is that a load using *adr* with an immediate operand produces a l value to *out*. Possible successor states: 2

#	State	Description
(2)		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 possiblities: l is moved to *opl* or *opr*. Possible successor states: 3, 4
(3)		As *adr* is not free another load variable l can not be loaded. l can be moved away to *out* using a dup_{opr} operation. Possible successor states: 2
(4)		As *adr* and *out* are free another variable can be loaded to *out*. Alternatively, l can be moved to *out* or stored. The store can be consuming or preserving. Possible successor states: 1, 2, 6
(5)		The only non-stalling option is to move l_2 to *opl*. Possible successor states: 7
(6)		The only non-stalling options are to move l_2 to *opr* or to destructively store l_1. Possible successor states: 2, 7
(7)		Both l_1 and l_2 can be moved to *out*. A store can not occur as *adr* is not free. Alternatively, a binary operation can be performed producing r to *out*. Note that the result of the binary operation has a different type r from the load variables. State 2 is therefore not a possible successor. Possible successor states: 5, 6, 8

#	State	Description
(8)		The only non-stalling option is to move r to *opl* or *opr*. Possible successor states: 9, 10
(9)		As *adr* is not free, another variable l cannot be loaded. r can only be moved to out. Possible successor states: 8
(10)		As *adr* and *out* are free, another variable l can be loaded. Alternatively, r can moved to *out* or stored. Possible successor states: 1, 8, 11
(11)		l can either be moved to *opl* or r can be stored destructively. Possible successor states: 2, 12
(12)		A binary operation can be performed producing a r value to *out*. Alternatively, r can be moved to *out*. l can be moved to *out* as well. Possible successor states: 8, 10, 11, 13
(13)		r can be moved to *opl*. l cannot be moved or stored as *adr* and *out* are both not free. Possible successor states: 12

As can be seen from the possible states, all operations incorporate at most one previously computed r value as 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 consuming or preserving. In register machines, values are always stored in the preserving fashion, but overwriting it afterwards without using it is equivalent to the consuming 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 $r \leftarrow r \odot_i m$ operation of the one-register machine. Note that r cannot be moved to *opr* while an l is present in the machine - and no variable can be loaded when r is in *opr* as this is also the *adr* field. This prevents the SCAD machine from performing an action that is equivalent to $r \leftarrow m \odot_i r$.

The main idea of the transformation is to read the SCAD move code sequentially and to track the origins and motions of tokens through the buffers. In register machines, data does not move through the machine but is accessed directly at any given point. Therefore, when an operation is executed or data is stored, the tracking information is used to determine where it originally came from to produce the equivalent register code. The resulting register machine code is called P. To track the tokens, auxiliary information S is kept about the SCAD machine state. S contains a type t and an address a for every buffer of the SCAD architecture accessed with the buffer name. The type t can be *immediate*, *load* or *register* to determine the origin of the value in the token. Address a is used to store the memory address of load variables. a is also used to store the immediate value of opcodes placed in *opc*.

First, all fields in S are considered to be empty denoted with $\_$. To simplify the description, a help function $clear(variable_1, variable_2, ...)$ is defined that sets the t and a parameter of every $variable_i$ to $\_$. For delayed moves, meaning that a value has not yet arrived when the move is issued, a move buffer M is added. M can contain either an instruction or $\_$. An overview over the necessary auxiliary data is given in Figure 7.

As a simplifying assumption, it is assumed that all *cps* buffers are filled with 1 when necessary as there is no other possibility for the amount of copies anyway. Additionally, it is assumed that no values are overwritten with the assignments on S: our simulation model always executes and moves data as soon as possible without delay. In an actual SCAD machine with buffer sizes restricted to 1, a move instruction that cannot be added as the head of the

Move Buffer M

Name	Description
M	Latest move instruction with *out* as source

SCAD State S

Name	Description
S.opl	left PU input
S.opr	right PU input
S.opc	PU opcode (store, load, dup or $\odot_i$)
S.out	PU output

For each entry there is a t (type) and a (address) field.

Figure 7: Overview of the structure of auxiliary information needed in the transformation method.

queue to the processing units would obviously stall as there is no free *adr* field in the buffers - the target address could not be written to the specific input buffer as it is full. However, S simulates immediate execution. There is no further operation or data transfer to make - the program would deadlock.

The overall method works as follows.

1. If there is another move instruction, read it, apply it to the auxiliary data and continue. If not, terminate.
2. Exhaustively do symbolic data moves and operation execution on the auxiliary data and go back to 1.

How to apply move instructions, move data and execute operations is detailed in the next sections.

3.1.1 Read Instructions from Input

When move instructions are read, they are either buffered in M or in the case of immediate operands directly added to the state S.

$out \rightarrow dest$ Where *dest* is either *opl* or *opr*. The move instruction is added to M.

$M := out \rightarrow dest$

M cannot have an entry as otherwise the original SCAD program would deadlock.

$m \rightarrow opr$ With m being an immediate operand and opr acting as the address field. Something will be stored or loaded at address m. The memory address will be needed when the register code is created. It is therefore saved in a.

$S.opr.t := \mathrm{immediate}$
$S.opr.a := m$

$op \rightarrow opc$ Where op is $store_c$, $store_p$, $load$, dup or $\odot_i$. This determines the next operation.

$S.opc.t := \mathrm{immediate}$
$S.opc.a := op$

3.1.2 Symbolic Data Moves and Operation Firing

After instructions and values have been added to S and M, symbolic data moves and operations are performed exhaustively to simulate execution of the program on a SCAD machine.

Data move $out \rightarrow dest$ In this case $M = out \rightarrow dest$ where $dest$ is opl or opr. A value is being moved from out to $dest$. The data move is performed as soon as $S.out.t \neq \_$ and $S.dest.t = \_$. So there is a value present in the output buffer and a free slot in the destination buffer. Notice that $dest$ can never contain a value as otherwise the move instruction could not have been scheduled, it would have stalled. In order to perform the move, the values in S are simply copied to the other buffer.

$S.dest.t := S.out.t\,,$
$S.dest.a := S.out.a\,,$
$\mathrm{clear}\,(S.out)$

If all necessary inputs are provided and there is free space in the output buffer the respective tokens are consumed and output tokens produced.

Binary operation $\odot_i$ As soon as all input buffers of the PU are full, $S.opc.a = \odot_i$ and $S.out.t = \_$ a binary operation is fired. As shown previously $S.opr.t$ is always a *load* variable and $S.opl.t$ can be either a *load* or *register* variable. Thus, the following code is appended to P. If $S.opl.t = load$, append

$\mathrm{r} \leftarrow S.opl.a$

to P. In any case, then append

$\mathrm{r} \leftarrow \mathrm{r}\ S.opc.a\ S.opr.a$

to P. If $S.opl.t = register$, its value is already in the register and thus does not have to be loaded. Note that the memory addresses of the operands were stored in their address a fields in S. The type of operation was stored in $S.opc.a$. After producing the code, S has to be cleaned up to represent the new state. All input buffers were consumed. As the token that is produced was computed, the label declares it as register content.

$\mathrm{clear}(S.opl, S.opr, S.opc)$
$S.out.t := \mathrm{register}$

dup_x **execution** Where x can either be *opl* or *opr*. As soon as the necessary input buffer of the PU is full, $S.opc = dup_x$ and $S.out.t = \_$, the transformation continues as follows. Note that the other operand does not need to be filled for the operation to execute. This instruction pushes the value of x to the output buffer.

$S.out.t := S.x.t,$
$S.out.a := S.x.a,$
$\mathrm{clear}(S.x)$

A *dup* operation obviously does not change the type of the token as this only moves the value to a different buffer. The type and address is therefore only copied to a different buffer as was the case for symbolic data moves.

Store As soon as the *val* (*opl*) and *adr* (*opr*) buffers are filled and $S.opc.a = store_c$ or $S.opc.a = store_p$ a store occurs. The memory address was stored in $S.opr.a$. From the previous discussion it is known that only

stores of *register* values need to be considered for optimal SCAD programs. So the following is added to P:

$S.opr.a \leftarrow \mathrm{r}$

If the store is preserving *out* needs to be clear and the fields of *val* are copied as the token is produced to *out*.

$S.out.t := S.opl.t\,,$
$S.out.a := S.opl.a\,,$

Afterwards the entries in S need to be cleared again.

$\mathrm{clear}\,(S.opc\,,\ S.opr\,,\ S.opl\,)$

Load As soon as the $lsu.adr$ buffer is filled and $S.lsu.opc.v = load$, a load occurs. As this token was obtained directly from memory the type will declare it as a load variable.

$S.lsu.out.t := \mathrm{load}$
$S.lsu.out.a := S.opr.a$
$\mathrm{clear}\,(S.opc\,,\ S.opr\,)$

The actual load will be added to P later.

When an operation is executed in the PU the origin of the tokens are used to determine the proper register machine code. Symbolic data moves only copy the information about tokens to different locations. Using a *dup* only resembles a symbolic data move. Only the limited amount of states concerning load and register variables can be reached as shown previously making it possible to obtain equivalent machine code for a non-commutative one-register machine with the same amount of memory operations.

3.2 Register Code to SCAD Code

Given a memory optimal register program P the following method will produce SCAD move code S with the same amount of memory operations. Operations on immediate operands 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 as consecutive stores are obviously not optimal by simply looking one instruction ahead and checking whether it overwrites r or not.

$r \leftarrow m$ If there was content in the register before, it cannot be used anymore. However, in an optimal program this had to be stored before as otherwise there would have been unnecessary memory operations. The following is added to S:

$\mathrm{load} \rightarrow \mathrm{opc}$
$m \rightarrow \mathrm{opr}$
$\mathrm{out} \rightarrow \mathrm{opl}$

The register token is moved to *opl* as this is assumed by the following operations.

$m \leftarrow r$ There must be a register token in the PU as otherwise the register code would attempt to store an empty register which would not be optimal. The register token is in *opl*. If r will be overwritten in the next instruction a consuming store is used in S:

$\mathit{store}_c \rightarrow \mathrm{opc}$
$m \rightarrow \mathrm{opr}$

If r is reused in the next instruction a preserving store is used in S and the register token moved back to *opl*:

$\mathit{store}_p \rightarrow \mathrm{opc}$
$m \rightarrow \mathrm{opr}$
$\mathrm{out} \rightarrow \mathrm{opl}$

$r \leftarrow r \odot_i m$ Again the register token is in *opl*. If there were no register token the operation would be calculated on an empty register which is undefined behaviour. Therefore, it would at least not be optimal.

$\mathit{load} \rightarrow \mathrm{opc}\ (\mathrm{load}\ m\ \mathrm{to}\ \mathrm{opr})$
$m \rightarrow \mathrm{opr}$
$\mathrm{out} \rightarrow \mathrm{opr}$
$\odot_i \rightarrow \mathrm{opc}\ (\mathrm{perform}\ \odot_i)$
$\mathrm{out} \rightarrow \mathrm{opl}$

3.3 Reduction Complexity and Optimality

For our discussion it is important to note that both directions are polynomial time transformations.

Lemma 3.1. *The given transformation of SCAD code to register code has linear runtime.*

Proof. Step 1 of the transformation reads an instruction. Depending on the instruction it will either be added to M or replace values in S. Both can obiously done in $\mathcal{O}(1)$. As this is done for all move instructions, the overall cost for all applications of Step 1 is $\mathcal{O}(n)$.
In Step 2 data is moved and operations are executed exhaustively. The cost for moving data is always $\mathcal{O}(1)$ as a single move just replaces a constant amount of values of S. All operation executions read and write a constant amount of values from S thus are in $\mathcal{O}(1)$ as well. As any execution of Step 1 can only cause a constant amount of applications of Step 2 the overall cost of the transformation is bounded by $\mathcal{O}(n)$ which is polynomial cost. □

Lemma 3.2. *The given transformation of register code to SCAD code has linear runtime.*

Proof. For every register command at most 5 move instructions are added to the SCAD code. Thus, the transformation has linear runtime in the size of the original program. □

The measure of optimality in SCAD is the amount of loads and stores while in one-register machines it is the total amount of operations. However, all operations in the one-register machine are memory operations. For all the translations these measures coincide which can be easily seen from the constructions.

Lemma 3.3. *The register code P resulting from the presented transformation of optimal SCAD code S is optimal.*

Proof. Proof by contradiction. Assume that P with n operations produced from S with n memory operations is not optimal. Then there exists equivalent register code P' with $n' < n$ operations. By the construction given before, this can be translated to SCAD code S' with n' memory operations. This contradicts the assumption that S is optimal. □

3.4 Implications and Further Remarks

The following theorems and corollaries assume the architecture given earlier in this section.

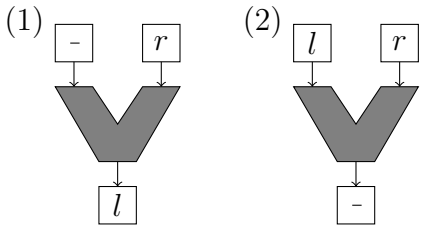


Figure 8: Possible SCAD states added by enabling both *opl* and *opr* to become *adr*.

Theorem 3.4. *Problem 4 for $k = 1$ and $n = 1$ is NP-complete.*

Proof. Since Problem 2 is in NP chaining the given polynomial time transformation to an algorithm A solving Problem 2 in NP will yield an algorithm solving Problem 4 for $k = 1$ and $n = 1$ that is in NP. Thus, Problem 4 for $k = 1$ and $n = 1$ is in NP. Now assume there exists an algorithm A' solving Problem 4 in polynomial time. If this was the case, chaining the polynomial time transformation to A' will yield a polynomial time algorithm for Problem 2. Therefore, as Problem 2 is NP-hard so is Problem 4. □

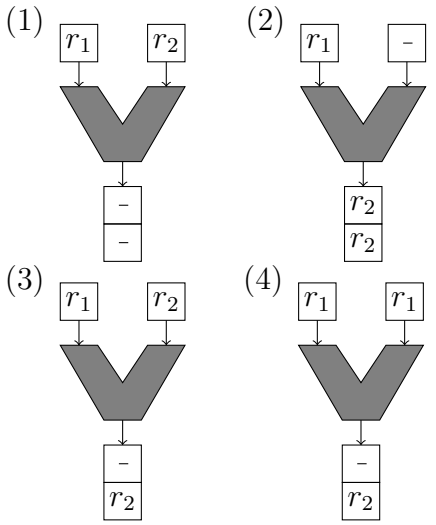
Clearly, NP-hardness for the general case follows immediately.

Corollary 3.5. *Problem 4 is NP-hard.*

Some parameters of the architecture and problem may be changed yielding different results.

Commutative One-Register. The instruction set may be changed to the commutative one-register instruction set shown in Figure 2 and 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 *opl* and *opr* buffers to become *adr* will make it possible to move r to *opr* and still load a variable that is then put into *opl*. States shown in Figure 8 are thus added to the ones in the non-commutative case.

Split LSU / PU. If PU and LSU are split into separate units with the preserving store still enabled the SCAD machine will be able to save memory



g

Figure 9: Some of the states possible in an *out* buffer of size 2.

operations compared to the register machine. It can put a calculated value r_1 into the *val* buffer of the LSU, load two unrelated load variables, do a computation on these producing r_2, store r_1 with a preserving store and then compute $r_3 = r_1 \odot_i r_2$ without ever having to reload any subresult. This takes at least two registers in a register machine. However, the register machine can obviously produce r_3 without a store and the SCAD machine can not. So these are not equal in terms of necessary memory operations as well.

Consuming Store. In case of the split PU and LSU one could also consider removing the possibility of preserving stores. It then becomes impossible to reuse values that are being stored. It is obviously not possible to duplicate values to two tokens as the *out* buffer has only one entry. So the single token of a value to be stored is moved to the LSU. There, it is consumed and stored. As the register machine can reuse the stored value without loading it again, it can save memory operations compared to the SCAD machine.

Increasing Buffer Sizes. Consider the commutative case and a combined PU/LSU unit again but with larger buffers. For simplicity assume that only the *out* buffer is elongated. It is however easy to see that it does not matter which buffer is elongated as long as the *out* buffer has at least size 2 to

enable duplications. Let n be the size of the *out* buffer. It can then be shown that this is equal to a commutative two-address n-register machine with an instruction set as given in Figure 2 concerning memory operations. For each additional entry in the *out* buffer a r value can be stored independently from computations involving load variables and another r value. In n registers with a commutative instruction set there can be n independent computed r values which 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 9. Having 4 values in the 4 possible buffer positions would clearly stall the machine. This is equivalent to one register being overwritten and the other one not by the operation in the register machine. Figure 9 only shows a selection of possible states but note that a third r value differing from the others can not 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 was 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.

Increasing PU Count. Only one of the PUs is the combined PU/LSU to keep memory consistency. However, apart from this the discussion from before 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.

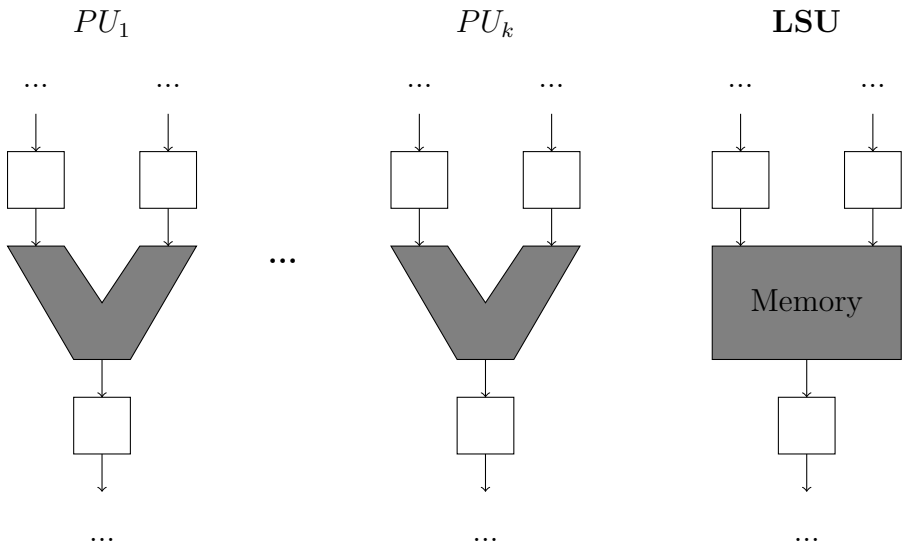


Figure 10: SCAD architecture with k universal processing units, unrestricted buffers and LSU used in the reduction.

4 Code Generation with Unbounded Buffers

In this section, the graph coloring problem (Problem 1) is reduced to queue optimal code generation for SCAD machines with unbounded buffers (Problem 3). To reiterate, queue optimal means that neither *dup* nor *swap* operations are used in the program.

4.1 Implications of Control Flow

One main observation needed for the reduction is that if several blocks B_1, B_2, ... B_m have control flow into the same basic block C, variables that need to be carried over and are accessed in C need to be in the same place leaving any previous basic block, if no additional copies or moves are allowed. If not, C would have to dynamically determine where the variables were placed obviously producing overhead. In the case of register machines, this means that the variables have to be stored in the same registers leaving basic blocks B_1, ..., B_m. In SCAD machines, this means that the variables have to be in the same buffer at the same position. However, this does not necessarily have to be an output buffer - the values could have been produced by different

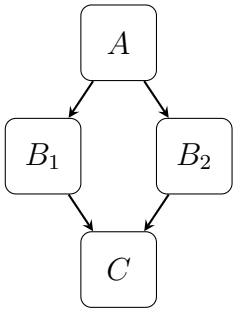


Figure 11: Example control flow 1.

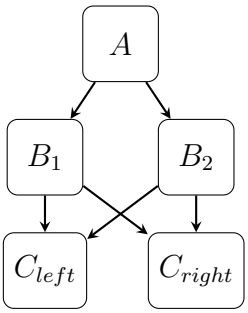


Figure 12: Example control flow 2.

PUs in different basic blocks but then moved to the same input buffer before entering C.

Example 3. Consider the control flow between basic blocks A, B_1, B_2 and C as shown in Figure 11. Now assume that B_1 and B_2 both write to variable x, for example $x = 1$ in B_1 and $x = 2$ in B_2. C will read x in the left operand like $y = x + 1$. It is then possible in an optimal program for B_1 and B_2 to produce x on different PUs: as y is produced on a specific PU i, x could be moved to the left input buffer of i in B_1 and B_2 before entering C.

The following reduction will however use a trick to enforce that variables have to be produced by the same PU instead of just being placed in the same buffer.

Example 4. Consider the control flow between basic blocks A, B_1, B_2, C_{left} and C_{right} as shown in Figure 12. Again, assume that B_1 and B_2 both write

to variable x, for example $x = 1$ in B_1 and $x = 2$ in B_2. However, now C_{left} will read x in the left operand like $y = x + 1$ and C_{right} in the right operand like $y = 1 + x$. Moving x to a left or right input buffer before entering C_{left} or C_{right} will now lead to necessary overhead in one of the two: if it is moved to a left input buffer, C_{right} will need to transfer it to a right one. In a *dup* and *swap* free program, x therefore has to be produced on the same PU in both B_1 and B_2.

4.2 Reduction

Given a graph $G = (V, E)$ and a number of colors k, program P is constructed as follows: It is assumed that $|V| = n$ and $|E| = m$. Furthermore, vertices are labeled v_1, v_2, ..., v_n and edges e_1, ..., e_m. Program P will have $m + 3$ basic blocks A, B_1, ..., B_m, C_{left}, C_{right}. The SCAD architecture will have k processing units. C_{left} and C_{right} read variables v_1, ..., v_n and all other basic blocks B_1, ..., B_m write to v_1, ..., v_n. A is only used to produce control flow to all B_i. It is assumed that there is control flow from A to all B_j, and from all B_j to C_{left} and C_{right} for $j \in \{1, ..., m\}$.

The basic idea of the reduction is that the PU on which vertex variables are produced correspond to the colors. They cannot be moved in B_1, ..., B_m as C_{left} needs all of them in the left input buffer and C_{right} in the right one. So if they were in an input buffer in one of C_{left} and C_{right} an unnecessary dup or swap would be needed to switch them as was also discussed earlier in Example 4. If in any of B_j they were produced on PU k_1 but then needed in k_2 before handing control to C_{left} or C_{right}, overhead would occur because they would have to go through an input buffer. The control flow is shown in Figure 13.
First, the basic blocks C_{left} and C_{right} are constructed.

Basic block C_{left}: This basic block simply reads all the variables in the left operand position.

$l_1 = v_1 \odot \_$
$l_2 = v_2 \odot \_$
$(\ldots)$
$l_n = v_n \odot \_$

Basic block C_{right}: This basic block simply reads all the variables in the right operand position.

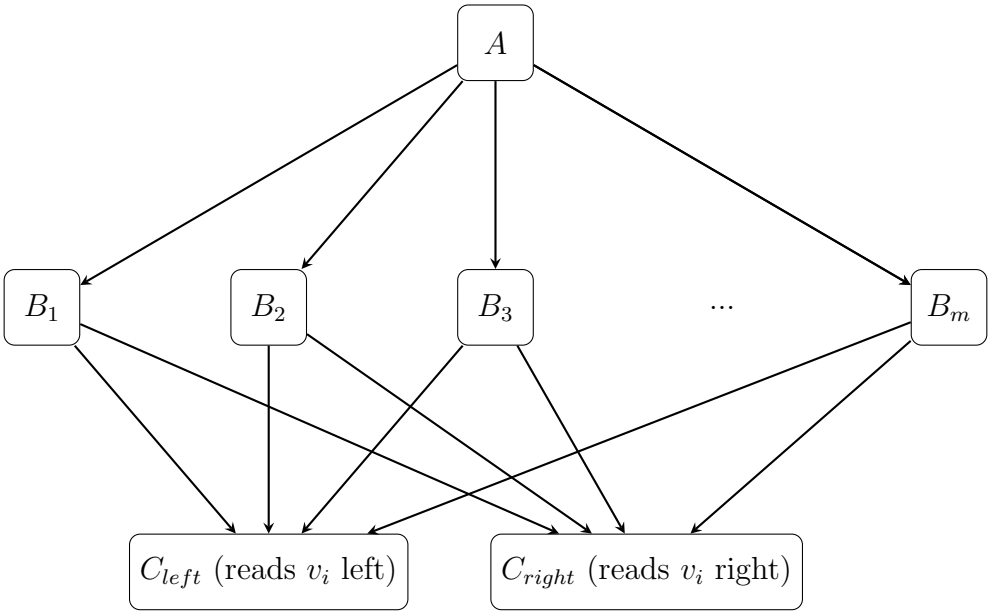


Figure 13: Control flow between basic blocks of the construction.

$l_1 = \_ \odot v_1$
$l_2 = \_ \odot v_2$
$(\dots)$
$l_n = \_ \odot v_n$

All other blocks write to v_1, ..., v_n and flow into these two blocks. The interface provides that all vertex variables are produced by the same processing unit in the same order in any of the other basic blocks, as they can not be moved to input buffers before going into C_{left} and C_{right}. This is due to the fact that they need the variables in different buffers (left and right) and that the variables always need to be in the same place from all the basic blocks. Thus, the production order and place is always the same.
Now the edges need to be modelled.

Basic Block B_i: Every basic block B_i will correspond to an edge $e_i := (v_a, v_b)$. The edge is between v_a and v_b which both correspond to a variable respectively. v_a and v_b need to be colored differently as they are incident to the same edge. This is enforced with two load variables.

$v_a = a \odot b$

$v_b = b \odot a$
$v_j = \_ \odot \_ \ (\text{for all } b \neq j \neq a)$

Where a and b are some load variables. v_a and v_b can not be produced on the same processing unit in a *dup* and *swap* free program. This will be explained in Lemma 4.3 and Figure 4.3. _ represents an immediate operand. The rest of the vertex variables therefore only depend on immediate operands, so they can be scheduled as pleased to fit the order necessary by the other basic blocks.

So there is a basic block that prevents two vertices v_a and v_b to be produced by the same PU for every edge. From the control flow it is known that all occurences of a vertex v_a in the basic blocks need to be produced on the same processing unit i. Thus, the vertex v_a can be colored with color i which must be different from all its neighbours. If it were not, there would be a basic block j in which the two lines given in the construction would be scheduled on the same PU, which is not optimally schedulable. If v_a were colored with different colors, a problem in the control flow would occur: it would have to be moved through an input buffer at some point before actual usage.

Example 5. Graph G shown in Figure 14 will be used as an example for the reduction. The graph has 4 vertices v_1, ..., v_4 and 4 edges e_1, ..., e_4. For every edge a basic block is constructed.

$\mathbf{B_1}$ with $e_1 = (v_1, v_2)$	$\mathbf{B_2}$ with $e_2 = (v_1, v_3)$	$\mathbf{B_3}$ with $e_3 = (v_1, v_4)$	$\mathbf{B_4}$ with $e_4 = (v_2, v_3)$
$v_1 = \text{a} \odot \text{b}$	$v_1 = \text{a} \odot \text{b}$	$v_1 = \text{a} \odot \text{b}$	$v_1 = \_ \odot \_$
$v_2 = \text{b} \odot \text{a}$	$v_2 = \_ \odot \_$	$v_2 = \_ \odot \_$	$v_2 = \text{a} \odot \text{b}$
$v_3 = \_ \odot \_$	$v_3 = \text{b} \odot \text{a}$	$v_3 = \_ \odot \_$	$v_3 = \text{b} \odot \text{a}$
$v_4 = \_ \odot \_$	$v_4 = \_ \odot \_$	$v_4 = \text{b} \odot \text{a}$	$v_4 = \_ \odot \_$

Next, the C_{left} and C_{right} blocks need to be added.

$\mathbf{C_{left}}$	$\mathbf{C_{right}}$
$l_1 = v_1 \odot \_$	$l_1 = \_ \odot v_1$
$l_2 = v_2 \odot \_$	$l_2 = \_ \odot v_2$
$l_3 = v_3 \odot \_$	$l_3 = \_ \odot v_3$
$l_4 = v_4 \odot \_$	$l_4 = \_ \odot v_4$

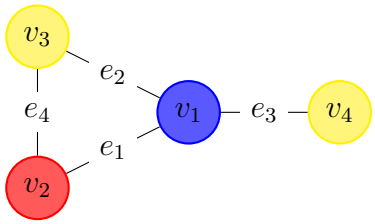


Figure 14: Graph G colored with 3 colors.

	PU_1	PU_2	PU_3
$\mathbf{B_1}$:	$v_1 = \text{a} \odot \text{b}$	$v_2 = \text{b} \odot \text{a}$	$v_3 = \_ \odot \_$
			$v_4 = \_ \odot \_$
$\mathbf{B_2}$:	$v_1 = \text{a} \odot \text{b}$	$v_2 = \_ \odot \_$	$v_3 = \text{b} \odot \text{a}$
			$v_4 = \_ \odot \_$
$\mathbf{B_3}$:	$v_1 = \text{a} \odot \text{b}$	$v_2 = \_ \odot \_$	$v_3 = \_ \odot \_$
			$v_4 = \text{b} \odot \text{a}$
$\mathbf{B_4}$:	$v_1 = \_ \odot \_$	$v_2 = \text{a} \odot \text{b}$	$v_3 = \text{b} \odot \text{a}$
			$v_4 = \_ \odot \_$

$\mathbf{C_{left}}$:

PU_1	PU_2	PU_3
$l_1 = v_1 \odot \_$		
$l_2 = v_2 \odot \_$		
$l_3 = v_3 \odot \_$		
$l_4 = v_4 \odot \_$		

And likewise for $\mathbf{C_{right}}$.

Figure 15: A valid schedule for the corresponding program of G on 3 processing units.

As a coloring with 3 colors is desired, the program will be scheduled on 3 processing units. A valid schedule can be found by simply following the colors as shown. The processing units are given colors arbitrarily: blue to PU_1, red to PU_2 and yellow to PU_3. Thus all writes of v_1 will be done on PU_1 corresponding to the blue color of the vertex. If the same is done for all other vertices and the vertices are simply written and read in the same order in every basic block, a valid schedule naturally arises from the valid coloring as shown in Figure 15. If the coloring was not valid, the schedule would not be either: there would be a basic block where the lines reading a and b would be on the same PU.

4.3 Correctness and Reduction Complexity

In this section, P will refer to a program resulting from the given reduction. v_i and e_i correspond to the variables, vertices and edges.

Lemma 4.1. *In a queue optimal move instruction program, v_i has to be stored in an output buffer PU k before being read in C_{left} and C_{right} for all $i \in \{1, ..., n\}$.*

Proof. Proof by contradiction. Assume v_i is stored in an input buffer. Without loss of generality, it is assumed to be a left input buffer. C_{right} needs v_i in the right input buffer. Thus, a *dup* operation is needed to produce v_i in an output buffer in order to move it to the required right input buffer afterwards. This is a contradiction to the assumption that the program is *dup*-free. □

Lemma 4.2. *In a queue optimal move instruction program, v_i has to be produced on the same PU k for all basic blocks B_j for all $i \in \{1, ..., n\}, j \in \{1, ..., m\}$.*

Proof. Proof by contradiction. Assume there are basic blocks B_{j_1} and B_{j_2} on which v_i is produced on different processing units k_1 and k_2. By the previous lemma it is known that v_i is stored in an output buffer before being read in C_{left} and C_{right}. Without loss of generality, it is assumed that it is the output buffer of k_1. Since the interface must be the same for all basic blocks, v_i has to be moved from the output buffer of k_2 to k_1 in the basic block B_{j_2}. This needs an additional *dup* operation as moves are only possible from output buffers to input buffers and v_i has to be moved back to an output buffer afterwards. This is a contradiction to the assumption that the program is queue optimal and therefore free of *dup* operations. □

Lemma 4.3 ("⇐"). *Every queue optimal schedule for P corresponds to a valid coloring ϕ.*

Proof. It is known from the previous lemmata that every v_i occurence in any basic block is produced on the same processing unit p. Thus $\phi(v_i) := p$ is well defined - it defines a single color $p \in \{1, ..., k\}$ for every vertex. It remains to show that $\forall \{v_1, v_2\} \in E : \phi(v_1) \neq \phi(v_2)$ holds.
Proof by contradiction: Assume $e_i = \{v_1, v_2\} \in E$ and $\phi(v_i) = \phi(v_j)$ hold for some v_i and v_j. For every edge there is a corresponding basic block. It

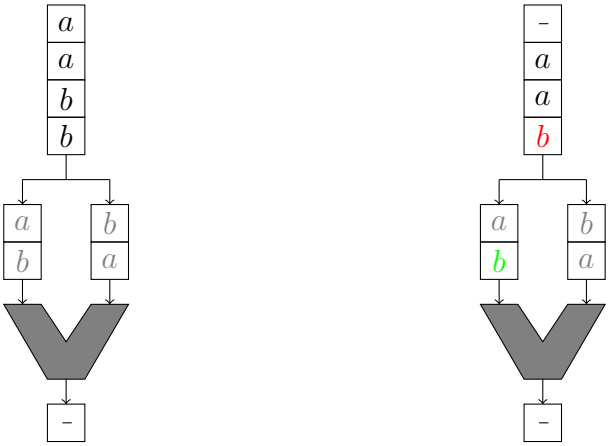


Figure 16: Failed attempt to execute the code used in the reduction on a single PU.

is assumed that B_i corresponds to e_i. In B_i, it is defined that $v_1 = a \odot b$ and $v_2 = b \odot a$ hold with a and b being load variables. $\phi(v_i) = \phi(v_j)$ implies that these lines are assigned to the same processing unit. However, it is not possible to do this without *dup* or *swap*. To show this refer to Figure 4.3. Note that choosing any different ordering for the loads inside the LSU or the lines inside the PU has no effect on the argument. As only one load per load variable in a basic block is allowed and no dups are possible all necessary copies of load variables have to be created inside the LSU. The situation shown in the upper section of the figure is thus created in the output buffer of the LSU. This has to be transformed into the situation shown in gray in the lower section inside the PU for the code of the reduction. However while the first load variable b shown in green can be successfully moved into the correct position the second one obviously can not without a *dup* or *swap* to move the a inside that buffer first. This is a contradiction to the assumption that P is a queue optimal schedule. □

Lemma 4.4 ("⇒")**.** *Every valid coloring ϕ corresponds to a queue optimal schedule P.*

Proof. The coloring ϕ assigns a color $\phi(v_i) \in \{1, ..., k\}$ to each vertex v_i. This defines the producing PU for variable v_i. Variables inside all basic blocks will be totally ordered to some total order $<$. So every basic block B_j produces all v_i in the same buffer position making optimal compilation of C_{left} and

C_{right} trivial. Since all variables except for $\{v_1, v_2\} = e_j$ produced by B_j can be placed arbitrarily as they are only depending on immediate operands it suffices to show that v_1 and v_2 can be produced without *dup* and *swap* operations. But since v_1 and v_2 are adjacent, $\phi(v_1) \neq \phi(v_2)$ holds which is also the producing PU for both implying that $v_1 = a \odot b$ and $v_2 = b \odot a$ are not computed on the same PU. Thus, any B_j can be compiled without *dup* and *swap* operations. □

Lemma 4.5. *The instance of queue optimal SCAD compilation resulting from the given construction is polynomial in the size of the graph coloring instance.*

Proof. The size of the graph coloring instance is measured in the size of graph $|G| = |(V, E)| = |V| + |E|$. For every edge a basic block with the size of $|V|$ is constructed. The rest of the construction only adds linear factors to this cost, meaning that the size is in $\mathcal{O}(|V| \times |E|)$ which is obviously polynomial in the size of the original instance. □

4.4 Implications

The complexity result for Problem 3 can now be given. There is an if and only if relation relating queue optimal schedules for P to valid schedules. Thus, there exists a queue optimal schedule for the constructed program if and only if a valid graph coloring exists for the graph. Thus, deciding if there exists a queue optimal schedule for any P is at least as hard as graph coloring.

Theorem 4.6. *Problem 3 is NP-hard.*

Proof. The statement follows immediately from the lemmata in this section and Lemma 2.1. □

Note that because graph coloring is also NP-complete for any fixed $k \geq 3$, so is Problem 3.

Corollary 4.7. *Problem 3 is NP-hard for any fixed $k \geq 3$.*

5 Conclusions and Future Work

The complexity analysis for SCAD code generation was split into bounded and unbounded buffers for the input and output buffers. The conclusions will be structured in the same manner followed by a summary of open problems and possible future endeavours.

5.1 Bounded Buffers

In the case where there is potentially not enough space in the processor to keep track of all necessary data, we described a strong relation between register machines and SCAD machines. Spill code has to be added in both architectures to generate the desired code as some results can not be contained in the processor and thus have to be stored in memory. This is even true when restricting programs to SSA basic blocks. We analyzed the relationship between optimizing the amount of memory operations for both architectures.

The outlined polynomial time methods transform one-register machine code to equivalent SCAD machine code and vice versa with the same amount of memory operations. As the register code problem was shown to be NP-complete[14], NP-completeness for the SCAD problem followed immediately. By extending the transformation to multiple registers and more buffer space, NP-completeness for the general problem followed.

Furthermore, this shows that space in registers is directly related to space in input and output buffers of SCAD machines. So only considering the memory optimality, the known algorithms and heuristics for register machines can be directly applied to SCAD machines by using the given transformation method. However, the transformation may introduce overhead proportional to the length of the buffers. Because buffers are queues, values in them are not arbitrarily accessible and hence overhead to access certain values may be necessary. Reducing this queue overhead is not solved by the presented method.

5.2 Unbounded Buffers

With the assumption of unbounded buffers, we could not find a simple bidirectional relation to known register machine problems as in the previous case. As buffers are unbounded, spill code is not necessary. The only overhead that

is left is the queue overhead. We reduced graph coloring to deciding whether a program can be compiled on a given SCAD architecture without additional queue operations. However, branching was used in the reduction. So only NP-hardness for programs with control flow followed. A complexity result for the restricted problem on SSA basic blocks was unfortunately not found. We also concluded that the decision problem can be reduced to optimizing the amount of queue operations or deciding the necessary amount of processing units for optimal compilation. Hence, NP-hardness for these problems followed as well.

In register machines, the register allocation problem considers the case where there are enough registers to store results. It is known to be strongly related to graph coloring and NP-hard for the general case and using non-SSA basic blocks. This shows that there is a relation between the register allocation problem and the unbounded buffer code generation problems in SCAD. However, the relation only seems to be elegant in one direction. The graph coloring instances are reduced to very special instances in SCAD and a reverse reduction was not found. We are not aware of a method that uses existing register allocation algorithms or heuristics to solve a similar problem on SCAD architectures.

In register allocation the interferences between variables can be easily computed by dataflow analysis while there are hints that this can not be done in the same manner for SCAD. We showed that the same variable can be produced by different processing units in different basic blocks and then simply moved to the same place without introducing any overhead. So processing unit assignment of a variable does not directly correspond to register assignment.

5.3 Future Work

A complexity result for the complexity of generating queue optimal code for SSA basic blocks has not been found yet. A polynomial time algorithm for this problem could significantly increase performance and quality of algorithms and heuristics for the general problems. Thus, finding a polynomial time algorithm or NP-hardness proof will be of great interest.

As the general case for unbounded buffers was shown to be NP-hard and no reverse reduction was found in order to use known methods for register allocation, the next step is to find and to evaluate algorithms and heuristics that can handle programs of practical sizes with acceptable performance and

quality. The proposed usage of existing register methods for memory optimal SCAD code generation is also yet to be precisely implemented and evaluated. Whether the produced queue overhead may be bearable in real-world applications should be explored in future work. The presented transformation method also does not consider the queue overhead at all. Thus, it is an inherently naive approach to the problem and further improvements should be considered. Reducing the overhead also ties in with the aforementioned open problem discussed in this section.

List of Figures

References

[1] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-Neumann architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 1489–1509, June 2014.

[2] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Microarchitecture (MICRO)*, (San Diego, California, USA), pp. 291–302, IEEE Computer Society, 2003.

[3] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[4] A. Bhagyanath and K. Schneider, "Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers," in *Formal Methods and Models for Codesign (MEMOCODE)* (E. Leonard

and K. Schneider, eds.), (Kanpur, India), pp. 143–152, IEEE Computer Society, 2016.

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

[6] M. Özsoy, Y. Koçberber, M. Kayaalp, and O. Ergin, "Dynamic register file partitioning in superscalar microprocessors for energy efficiency," in *International Conference on Computer Design (ICCD)*, (Amsterdam, The Netherlands), pp. 515–520, IEEE Computer Society, 2010.

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

[8] W. Lee, 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 *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (D. Bhandarkar and A. Agarwal, eds.), (San Jose, California, USA), pp. 46–57, ACM, 1998.

[9] H. Corporaal, "TTAs: Missing the ILP complexity wall," *Journal of Systems Architecture*, vol. 45, pp. 949–973, June 1999.

[10] A. Bhagyanath and K. Schneider, "Exploring the potential of instruction-level parallelism of exposed datapath architectures with buffered processing units," in *Application of Concurrency to System Design (ACSD)* (A. Legay and K. Schneider, eds.), (Zaragoza, Spain), pp. 106–115, IEEE Computer Society, 2017.

[11] A. Bhagyanath and K. Schneider, "Exploring different execution paradigms in exposed datapath architectures with buffered processing units," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, (Samos, Greece), IEEE Computer Society, 2017.

[12] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or

revisiting register allocation: Why and how," in *Languages and Compilers for Parallel Computing (LCPC)* (G. Almási, C. Caşcaval, and P. Wu, eds.), vol. 4382 of *LNCS*, (New Orleans, Louisiana, USA), pp. 283–298, Springer, 2007.

[13] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[14] A. Aho, S. Johnson, and J. Ullman, "Code generation with common subexpressions," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 146–160, 1977.

[15] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, Massachusetts, USA: The MIT Press, 1 ed., 1989.

[16] R. Vollmar, "Über einen Automaten mit Pufferspeicherung," *Computing*, vol. 5, no. 1, pp. 57–70, 1970.