Code Generation for Synchronous Control Asynchronous Dataflow Architectures

Dissertation

vom Fachbereich Informatik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Dissertation

von

Anoop Bhagyanath

Datum der wissenschaftlichen Aussprache	31.01.2020
Dekan	Prof. Dr. Jens Schmitt
Gutachter	Prof. Dr. Klaus Schneider
	Prof. Dr. Andreas Koch

D386

Abstract

Scaling up conventional processor architectures cannot translate the ever-increasing number of transistors into comparable application performance. Although the trend is to shift from single-core to multi-core architectures, utilizing these multiple cores is not a trivial task for many applications due to thread synchronization and weak memory consistency issues. This is especially true for applications in real-time embedded systems since timing analysis becomes more complicated due to contention on shared resources. One inherent reason for the limited use of instruction-level parallelism (ILP) by conventional processors is the use of registers. Therefore, some recent processors bypass register usage by directly communicating values from producer processing units to consumer processing units. In widely used superscalar processors, this direct instruction communication is organized by hardware at runtime, adversely affecting its scalability. The exposed datapath architectures provide a scalable alternative by allowing compilers to move values directly from output ports to the input ports of processing units. Though exposed datapath architectures have already been studied in great detail, they still use registers for executing programs, thus limiting the amount of ILP they can exploit. This limitation stems from a drawback in their execution paradigm, code generator, or both.

This thesis considers a novel exposed datapath architecture named Synchronous Control Asynchronous Dataflow (SCAD) that follows a hybrid control-flow dataflow execution paradigm. The SCAD architecture employs first-in-first-out (FIFO) buffers at the output and input ports of processing units. It is programmed by move instructions that transport values from the head of output buffers to the tail of input buffers. Thus, direct instruction communication is facilitated by the architecture. The processing unit triggers the execution of an operation when operand values are available at the heads of its input buffers. We propose a code generation technique for SCAD processors inspired by classical queue machines that completely eliminates the use of registers. On this basis, we first generate optimal code by using satisfiability (SAT) solvers after establishing that optimal code generation is hard. Heuristics based on a novel buffer interference analysis are then developed to compile larger programs. The experimental results demonstrate the efficacy of the execution paradigm of SCAD using our queue-oriented code generation technique.

Acknowledgement

I will always be grateful to Prof. Klaus Schneider for his exemplary guidance. I have thoroughly enjoyed all the fruitful discussions we have had over the course of my journey as a doctoral student. Under your guidance, I have learned to effectively approach problems by focusing on the important aspects and see that the details converge naturally. This also helped me recognize and appreciate the elegance of different solutions. Moreover, I have gained important insights while assisting you with teaching, which will be really useful during my academic career.

I sincerely thank Prof. Andreas Koch, who agreed to take time out of his hectic schedule to review this work. Prof. Pascal Schweitzer and Prof. Karsten Berns were kind to ensure the smooth conduct of my PhD defense as examiner and chair, respectively. I am also thankful for their valuable advice on pursuing a career in academics.

I would also like to thank my colleagues for a healthy atmosphere in the group to share and discuss each other's work. Let me take this opportunity to also thank students I have worked with for bringing fresh perspectives to my research efforts.

Most importantly, I express my deepest gratitude to my family and friends for giving meaning to my life, though I know they realize even otherwise that I am thankful.

Anoop Bhagyanath

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AST	Abstract syntax tree
CDB	Common data bus
CU	Control unit
DAG	Directed acyclic graph
DO-SCAD	Dynamically ordered SCAD
DPDI	Dynamic placement dynamic issue
DTN	Data transport network
FIFO	First-in-first-out
ILP	Instruction-level parallelism
LSU	Load-store unit
MIB	Move instruction bus
NP	Non-deterministic polynomial time (complexity class)
PU	Processing unit
RISC	Reduced instruction set computer
SAT	Satisfiability
SCAD	Synchronous control asynchronous dataflow
SMT	Satisfiability modulo theories
SO-SCAD	Statically ordered SCAD
SPDI	Static placement dynamic issue
SPSI	Static placement static issue
SSA	Static single assignment
SSI	Static single information
TTA	Transport triggered architecture
VLIW	Very long instruction word

Chapter 1

Introduction

Contents

1.1. Motivation

The miniaturization of microelectronics is resulting in an ever increasing number of transistors in processors. However, simply scaling up conventional processor architectures will not render a proportional improvement in application performance [AHKB00]. Due to this *processor scaling wall*, the past decade has seen a shift from single-core to multi-core architectures. However, due to the required thread synchronization [Lee06] and weak memory consistency issues [Mosb93; AdGh96; StNu04], in many applications, it is not a trivial task to extract enough thread-level parallelism to take advantage of multiple cores. The effective use of multiple cores is even more difficult in embedded systems since, for a majority of embedded system applications, one has to guarantee strict requirements in timing in addition to the correctness of multithreaded programs. Timing analysis becomes more complicated for multi-core architectures due to contention on shared resources by co-running threads [FGQF12; NoPa12; RGGQ12]. An alternative for improving the *application performance* is to still increase the use of instruction-level parallelism (ILP) [JoWa89; Wall91a; RaFi93] contained in the programs.

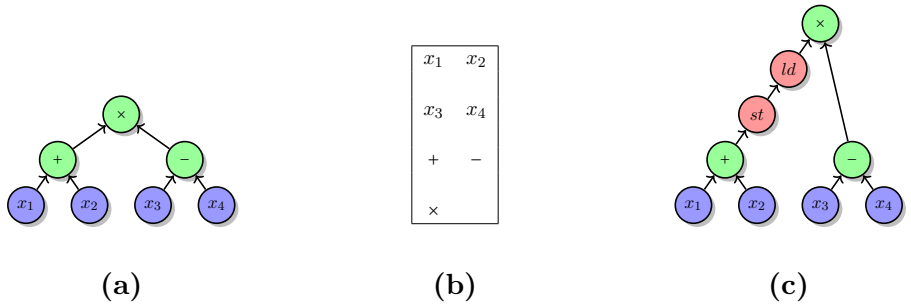
The execution paradigms of processor architectures are traditionally viewed from the perspective of control-flow and dataflow computing models. In the *control-flow model* [VonN45], the program is a linear sequence of instructions addressed by a program counter. Each instruction accesses its operands from and writes its result to an updateable memory. Typically, the control-flow model of computation expresses no parallelism. At the opposite extreme is the *dataflow model* [KaMi66; Denn74; Kahn74] where programs are dataflow

graphs, and instruction execution is driven by the arrival of data tokens. The intermediate results are directly communicated from producer instructions to consumer instructions and are not stored in a shared memory. However, though offering an elegant concurrent model of computation, dataflow processors fell out of favour due to their inefficient implementations and their inability to effectively manage a shared memory to support imperative programming languages (see Chapter 6 for more details).

Hence, commercially successful processors are based on the control-flow model. Since the model itself does not express any parallelism, the processors and compilers use various techniques to extract ILP [Toma67; FERN84; Fish81; MLCH92; Lam88; Rau94; LaHw95]. However, these techniques face unavoidable limits on their further scalability [AHKB00]. One inherent reason for these limitations is the *use of registers* to hold intermediate values: Most current processor architectures are so-called load-store architectures where only load and store instructions have access to the main memory while all other instructions use registers as operands and target addresses. This is because the execution time of memory accesses did not improve as fast as that of other instructions (*memory wall* [WuMc95; McKe04]), so that the number of memory accesses had to be limited as much as possible. A simple way to reduce them is to load values into local memories like registers and to work on local copies as long as possible. While the introduction of registers was a good idea for sequential processors, it now imposes limits for the use of ILP.

For example, consider the expression tree shown in Figure 1.1a. Using 4 registers and 4 read and write ports in the register file, one can evaluate the expression in only 3 steps by firing all instructions in each level in one step. However, if only 2 write ports are available in the register file, at most only 2 independent instructions can be fired in one step. It then takes 4 steps to evaluate the expression as shown in Figure 1.1b, irrespective of the number of processing units. By the Sethi-Ullmann algorithm [SeUl70], at least 3 registers are required to evaluate the expression tree if no load/store instructions shall be used. If only 2 registers would be available, one has to insert spill code in that the obtained result of $x_1 + x_2$ is stored in memory and loaded in a register after having evaluated $x_3 - x_4$ as shown in Figure 1.1c. It now takes 5 steps to evaluate the program (since there are 5 levels in the obtained syntax tree), irrespective of the number of processing units. More memory accesses not only reduce ILP but also adversely affect the timing-predictability of applications [WGRS09] which is an important metric in real-time embedded systems.

Hence, the *limited number of registers and register file ports* limits the use of ILP. Increasing the number of registers is however difficult since this number is directly encoded in the instruction sets. Changing it requires corresponding changes in machines, compilers, and even operating systems. Also, increasing the number of processing units and the number of ports in a register file quickly leads to a bottleneck in wiring these on the chip [ZyKo98; RDKM00]. For the latter reason, clustered architectures have been introduced where processing units can only access predefined register clusters. Moreover, *additional cycles are incurred* for writing/reading a value to/from register compared to



(a) **(b)** **(c)**

Figure 1.1.: *(a) An example expression tree, (b) the corresponding assembler code and (c) the same expression tree with spill code*

directly communicating the value from a producer processing unit to consumer processing units.

Therefore, recent processors somehow try to bypass register usage by communicating values directly from the producer processing units to consumer processing units. This is often referred to as *direct instruction communication* or direct data routing. Though based on the control-flow computing model, these processors gradually adopt the dataflow computing model since values are produced, directly communicated, consumed, and are not overwritten using a shared namespace (register). Most current processors follow a *hybrid control-flow dataflow model of computation* [YAJE14]. In widely used superscalar processors [John91; SmSo95], instructions in reservation station are executed out-of-order according to Tomasulo's algorithm [Toma67]. Instructions that occur simultaneously in reservation stations communicate their results directly with one another, while others communicate via registers. This direct instruction communication is completely controlled by the processor. The processor tracks data dependencies of instructions to add appropriate entries in the reservation station for the direct communication of values. Moreover, allocating instructions to processing units or instruction placement is also determined at runtime. These factors limit the scalability of these machines to larger numbers of processing units (see Chapter 6 for more details).

The *exposed datapath architectures* [Corp94; Corp99; WTSS97; BKMD04; MCCV06; HSMC11; GoHS11; VSGG10; WSCH15] propose an interesting scalable alternative by allowing the compiler to move values directly from producer processing units to consumer processing units. This way, direct instruction communication is simply offered by the processor, and it is the responsibility of the compiler to control and utilize the same. These architectures often provide a large number of processing units. They can use a compiler determined instruction placement to mitigate communication delays, which are becoming increasingly dominant in many-core processors [AHKB00; HoYS98]. Though exposed datapath architectures have already been studied in great detail, we ***observe*** that these architectures still use registers to execute programs, thus limiting the amount of ILP they can make use of. This limitation stems from either a drawback in their execution paradigm or the

code generator or both (see Chapter 6 for a review of related architectures).

1.2. Contributions

We propose a novel exposed datapath architecture based on a hybrid control-flow dataflow execution paradigm to facilitate direct instruction communication. We moreover recommend an associated code generator that *completely avoids the use of registers*. The main contributions are therefore:

- ***SCAD:*** The *Synchronous Control Asynchronous Dataflow* architecture is an exposed datapath hybrid control-flow dataflow architecture that uses first-in first-out (FIFO) buffers, i.e., queues, at output and input ports of processing units. SCAD is programmed by *move instructions* that transport values from the heads of output buffers to the tails of input buffers (exposed datapath). We also study two close variants of SCAD: statically ordered ***SO-SCAD*** and dynamically ordered ***DO-SCAD***, to compare it with the classic register and dataflow architectures.

- ***Code Generation:*** It is observed that SCAD code generation inspired from classical *queue machines*, in contrast to register machines, not only exploits more ILP but also lends naturally to direct instruction communication thereby *eliminating the need to use registers*. However, with limited processing units, the content of output buffers will need to be rotated to access values which are not at their heads. This leads to computational and transportation overheads.

- ***Optimal Code Generation:*** We prove that it is a *hard problem* to compile a given program to optimal (overhead-free) move code for a given SCAD machine. Consequently, boolean constraints are formulated for optimal code generation so that *satisfiability (SAT)* solvers can be used to generate optimal code.

- ***Heuristics for Code Generation:*** Since optimal code generation using SAT solvers is only feasible for small programs, we developed a heuristic for SCAD code generation. To that end, instructions are assigned to processing units using a novel *buffer interference analysis* so that overhead-free SCAD code is generated by considering instructions in program order.

1.3. Outline

The rest of the thesis is organized as follows: Chapters 2,3,4 and 5 discuss core contributions of this thesis in the order listed above. Experiments in Chapter 4 reveal that SCAD finds an essential balance between hardware complexity and compiler flexibility to implement direct instruction communication effectively. Further experimental results in Chapters 4 and 5 show the efficacy of our code generation technique for SCAD compared to that based on traditional code

generation for register architectures. Chapter 6 reviews execution paradigms of and compilation for other architectures concerning the use of ILP. Conclusions and future work are discussed in Chapter 7.

Chapter 2

Execution Paradigms

Contents

In this chapter, we present a novel processor architecture: the Synchronous Control Asynchronous Dataflow (SCAD) architecture [BhJS16]. SCAD is an exposed datapath architecture that uses a blend of control-flow and dataflow computational models for executing programs. We also discuss two variants of SCAD [BhSc17a] that differ in hardware and compiler complexity: First, statically ordered SCAD (SO-SCAD) that tends more towards the control-flow model in its execution style compared to SCAD. Second, the dynamically ordered SCAD (DO-SCAD) that adopts a more dataflow like execution style compared to SCAD.

2.1. SCAD

2.1.1. Organization

The organization of processing units in a SCAD architecture is shown in Figure 2.1. Each processing unit (PU) has queues or first-in-first-out (FIFO) buffers at its input and output ports. Input and output buffers are connected to two interconnection networks: There is the *move-instruction bus (MIB)* (given in red color) which is used to synchronously send values from the control unit to the PUs, and the *data transport network (DTN)* (given in green color) which is

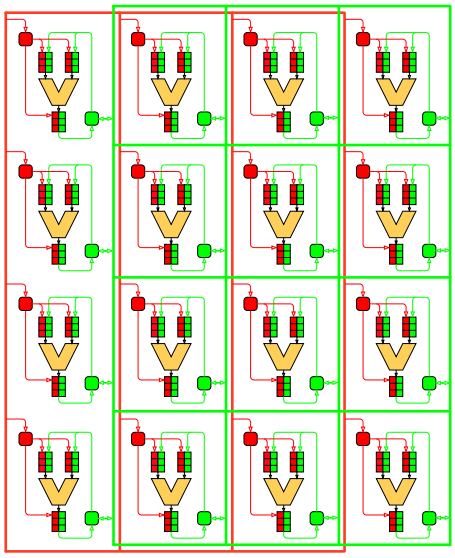


Figure 2.1.: *Execution framework of a SCAD processor that use a 2D mesh network as data transport network*

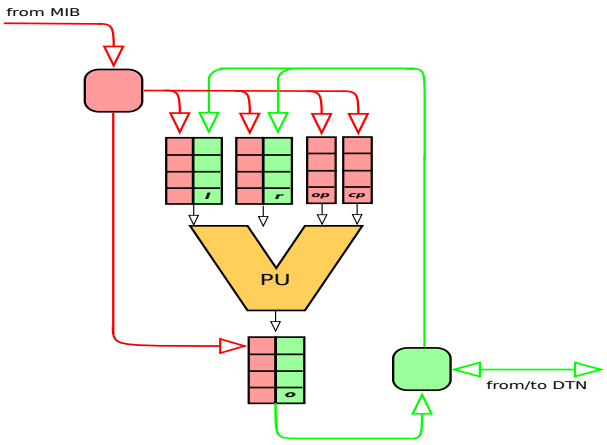


Figure 2.2.: *A processing unit in a SCAD machine*

used by the PUs to asynchronously send values to each other whenever these are available.

A processing unit in a SCAD machine is shown in Figure 2.2. Note that besides input buffers l and r to store left and right operands of a binary operation, there is an opcode input buffer op and a copies input buffer cp used to hold the number copies of the result to be produced in the output buffer o. The opcode and copies buffers of PUs are not shown in Figure 2.1 to avoid cluttering. The operand input buffers and output buffers hold pairs (adr, val) of entries. For an input buffer, adr is the address of the output buffer of the PU that produced or that will produce the value val. An entry $(adr, \perp)$ with the special value $\perp$ is used to indicate that the required value is not yet available and will later

be sent from the output buffer *adr*. Similarly, for an output buffer, *adr* is the address of the input buffer of the PU that will consume the value *val*. An entry $(adr, \perp)$ with the special value $\perp$ is used to indicate that the required value is not yet available and will later be produced by the PU and can then be sent to the input buffer *adr*.

SCAD is programmed by a sequence of *move instructions* $src \to tgt$ whose semantics is to move a value from the head of output buffer *src* to the tail of input buffer *tgt*. Although a 2D mesh network is used as DTN in Figure 2.1, any interconnection network ranging from a simple set of buses and sockets to more complex parallel networks such as Omega, Banyan, Beneš networks, etc. can be used as DTN. For instance, Figure 2.3 shows PUs interconnected using a fat binary tree. Similarly, a PU in a SCAD architecture may implement any function with an arbitrary number of inputs and outputs. These properties recommend SCAD as an interesting candidate for application-specific processors. However, we do not explore co-design possibilities with SCAD in this thesis, and instead, we simply assume PUs capable of executing standard binary operations.

2.1.2. Execution of a move program

The execution of a move program works as follows: Using the program counter, the *control unit* (CU) will fetch the next move instruction $src \to tgt$ from the instruction memory and will broadcast it via the MIB to all PUs. The input buffer with address *tgt* will add the entry $(src, \perp)$ to its tail, and the output buffer with address *src* will add the entry $(tgt, \perp)$ to its tail. If one of the two buffers should be full, it will signal this via a feedback signal `fullBuffer` to the control unit. Then the other buffer will also not store the entry, and the control unit will resend the move instruction $src \to tgt$ in the next cycle (it is stalled at this point of time). The data transport related to a move instruction $src \to tgt$ is deferred to a later time when the data is available. Therefore, the addresses in move instructions are registered synchronously in program order while the flow of data proceeds asynchronously: synchronous control asynchronous dataflow. Also, note that all move instructions are stored in buffers in the order in which they were issued by the control unit, i.e., in program (control-flow) order. To see in more detail how a move program is executed, let us consider the behaviors of the PUs and their input and output buffers.

If a processing unit will find entries (adr_1, x_1),...,(adr_m, x_m) with $x_i \neq \perp$ at the heads of its m input buffers and there is free space in its n output buffers, it can react and will consume entries (adr_1, x_1), ..., (adr_m, x_m) to produce new result values $y_1 := f_1(x_1, \ldots, x_m)$, ..., $y_n := f_n(x_1, \ldots, x_m)$ where f_1, ..., f_n are the functions associated with that PU (assuming that number of copies of each result to produce is 1). Each output value y_i is then stored in that entry $(tgt, \perp)$ of output buffer number i that is closest to the head of the output buffer, i.e., that entry is replaced with (tgt, y_i). If there should be no such entry, then a new entry $(\perp, y_i)$ is placed at the tail of the output buffer i, and the next target address for this output buffer will be stored in this entry.

Note that it is possible that the result value has been computed before a move instruction has been issued by the control unit to move it to another place.

The output buffers are responsible for the final transport of data by sending messages between PUs over the DTN. Such a message (src, tgt, val) consists of the address of the sending output buffer src, the address of the input target buffer tgt, and the value val that is transported by the message. A message (src, tgt, val) is created when the output buffer with address src has a completed entry (tgt, val) as its head. This message is then sent to the input buffer tgt via the DTN. When it finally reaches the input buffer tgt, the input buffer will replace the entry $(src, \perp)$ closest to its head with (src, val), which may trigger a new firing of its PU. Additionally, the output buffers snoop the MIB for receiving new target addresses for their values. If output buffer src will see the move instruction $src \to tgt$ on the MIB, it will check whether it contains an entry $(\perp, y)$. If so, it will replace the one closest to its head with (tgt, y). Otherwise, it will create a new tail $(tgt, \perp)$, if there is still space available. Otherwise, it will signal `fullBuffer` to the control unit, which then has to stall and resend the move instruction later. The input buffers also always snoop the two interconnection networks, i.e., the MIB and the DTN. As explained above, address entries $(src, \perp)$ are put in order in the input buffer tgt whenever a move instruction $src \to tgt$ is seen on the MIB, and an available entry $(src, \perp)$ is completed with the value val when a message (src, tgt, val) arrives. Note that values from different output buffers reorder appropriately in an input buffer based on the order of output buffer addresses registered in the input buffer. However, the DTN must maintain the ordering of values sent from a particular output buffer to a particular input buffer.

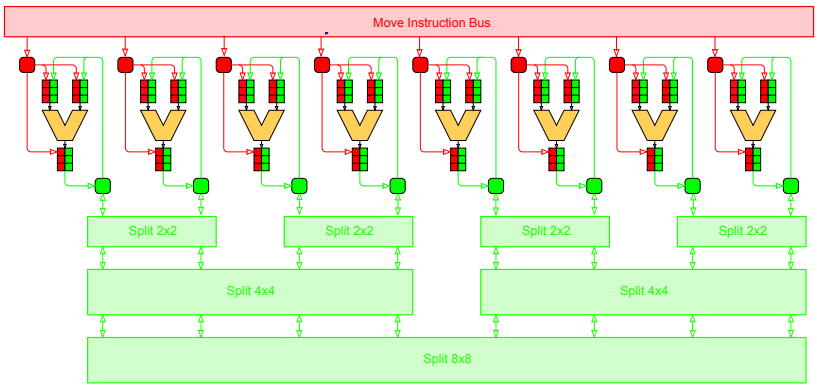


Figure 2.3.: *Execution framework of a SCAD processor that use a fat binary tree as data transport network*

Clearly, there is at least one *store unit* (SU) with two input buffers, one for the memory addresses and another one for the values to be stored at the corresponding addresses. There is no output buffer. Instead, the SU stores the values in the order specified by the input buffers (in the program order) to the main memory. Clearly, there is also at least one *load unit* (LU) that has just one input buffer for the addresses and an output buffer for the values

loaded from memory. They will be sent through the DTN similar to output values of other PUs, and whether the SU and the LU have to be synchronized depends on a chosen weak memory model [Mosb93; AdGh96; StNu04]. In the rest of this thesis, we assume a combined load-store unit (LSU) as shown in Figure 2.4. The left operand buffer l holds memory addresses, the right operand buffer r holds values to be stored in case of store operations, the opcode buffer op holds *load* or *store* opcode, and the copies buffer cp holds the number of copies of loaded values to be produced in output buffer o in case of load operations.

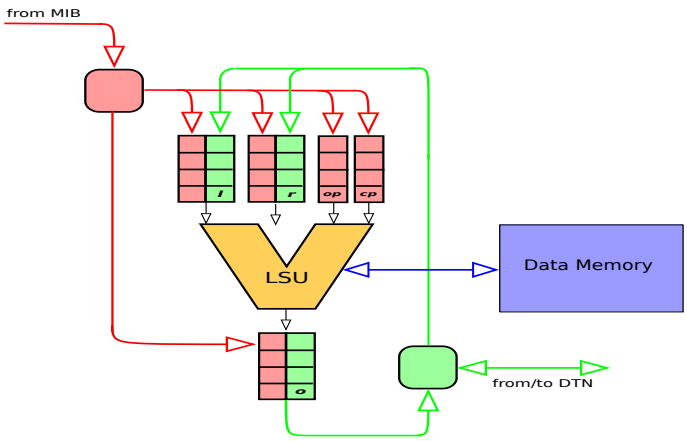


Figure 2.4.: *The load-store unit in a SCAD machine*

Move instructions of the form $imm \rightarrow tgt$ are also necessary to move any *immediate value* imm to any input buffer tgt. That is, an entry $(\top, imm)$ is enqueued to the tail of input buffer tgt by MIB, where $\top$ is simply a placeholder in the address lane, which indicates that the particular slot in the address lane is occupied. Any further address entries will have to be enqueued behind this slot. Of course, multiple independent move instructions can be broadcast simultaneously by using multiple lanes in MIB. Every sub-sequence of move instructions where no buffer address occurs more than once is a bundle that could be broadcast at once without affecting the correctness of the move code.

Note that we have not included register files or any local storage other than buffers in the SCAD architecture, although it is possible to use them just like any other PU. In Chapter 3, we motivate a code generation technique for SCAD that does not require any local memory other than the buffers. In other words, we utilize the direct instruction communication facilitated by the SCAD architecture to the fullest.

2.1.3. Control flow in SCAD

The control flow is implemented in SCAD using move instructions whose target is the control unit. The control unit (CU) maintains the program counter pc that is used to address the instruction memory. As long as a valid address is available in pc, the corresponding move instruction is fetched and issued.

The CU is responsible for both conditional and unconditional branches. *Unconditional branches* are simply encoded as move instructions to the special address *pc*. That is, the move instruction $adr \rightarrow pc$ sets the program counter in CU to *adr*, and the CU fetches the instructions starting from address *adr* in subsequent cycles.

Conditional branches are handled by input buffers of the CU. To this end, the CU is equipped with three input buffers as shown in Figure 2.5: $cu@c$ is used to hold the branch condition, $cu@then$ holds the 'then' address (or branch target address), and $cu@else$ is used to hold the 'else' address. Conditional branching is implemented as follows: First, the branch target address is moved to $cu@then$. If the instruction $adr \rightarrow cu@then$ is fetched by CU, it immediately moves the branch target address *adr* to $cu@then$. Next, issue move instructions to compute the branch condition. Finally, issue a move instruction, from the output buffer where the branch condition will be produced, to $cu@c$. If this instruction is fetched by the CU, it destroys its local *pc* and automatically moves $pc + 1$ to $cu@else$. Since a valid address is now not available in *pc*, the CU ***stalls*** or stops fetching further instructions. Once the branch condition is computed and transported to $cu@c$, the heads of all input buffers of the CU will have valid entries. This triggers the CU, and a new local *pc* is defined appropriately. Note that both buffers $cu@then$ and $cu@else$ are populated by the CU itself (see Figure 2.5). Furthermore, since the CU stalls on each conditional branch until the branch outcome is known, a single entry is sufficient in all its input buffers. Consequently, the buffer $cu@c$ does not require an address field.

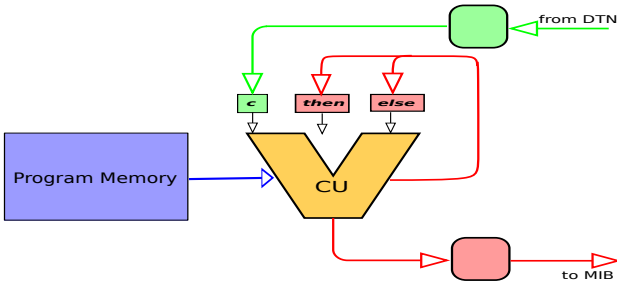


Figure 2.5.: *The control unit in a SCAD machine*

For the experimental results in this thesis, we use a SCAD simulator that stalls on encountering branches. As already mentioned, this thesis's main focus is to study and evaluate improvements in performance by executing programs in SCAD by direct instruction communication (avoiding the use of registers). Nevertheless, to exploit instruction-level parallelism (ILP) across control flow boundaries, SCAD may either utilize branch prediction for speculative execution similar to dynamically scheduled superscalar machines or predicated execution similar to statically scheduled VLIW processors. In the following sections, we describe how branch prediction and predicated execution may be implemented in SCAD.

Branch prediction

In superscalar processors [John91; SmSo95], instructions are ordered in program order in the reorder (FIFO) buffer. This allows the processor to flush all entries in different data structures (reservation station, forward reference table, and reorder buffer) when the branch instruction is at the head of the reorder buffer, and the predicted branch outcome was incorrect. Instead of a single reorder buffer, we have multiple input and output FIFO buffers in SCAD processors that are populated in program order. When a branch condition outcome is predicted to speculatively execute move instructions along the predicted control flow path, the CU in SCAD will signal `speculativeExecution` to all PUs and LSU so that they can 'mark' the current tail of their input and output buffers. This indicates for each buffer that all entries that follow the marked entry are under speculation. Similarly, when the actual branch condition outcome is computed by a PU and transported to the CU, the CU will signal `predictionStatus` to all PUs and LSU, so that they can either 'unmark' the last marked entry in their buffers in case the prediction was correct or flush all entries that follow the last marked entry in their buffers in case of a wrong prediction. To that end, the CU in SCAD has an additional input buffer $cu@p$ to hold predicted branch outcomes as shown in Figure 2.6, and the input buffers have multiple entries to support a speculation depth greater than one. The maximal speculation depth is then determined by the size of input buffers of the CU. Also, note that the input buffer $cu@c$ that holds actual branch outcomes must now have an address lane since the actual branch condition outcomes for branches currently under speculation are computed in dataflow order (and therefore not necessarily in program order) before they are transported to the CU. Either static branch prediction or dynamic branch prediction may be used (in this case, we assume in Figure 2.6 that the branch predictor hardware is in the control unit).

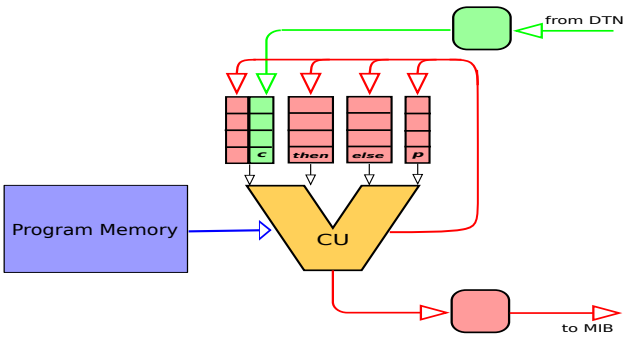


Figure 2.6.: *The control unit in a SCAD machine with branch prediction*

For the correctness of the branch prediction scheme, it is important that broadcasting of the `predictionStatus` signals by the CU must be in the same order as the broadcasting of the corresponding `speculativeExecution` signals. This is naturally so because the CU's input buffers behave similarly to input buffers of any PU: the CU will also always consume values from the head of its input

buffers, and these buffers are filled only from the tail. For conditional branching, the CU will first enqueue corresponding entries to the tail of its input buffers, then set the program counter appropriately based on predicted branch outcome, and finally broadcast the `speculativeExecution` signal to all PUs and the LSU. If values are available at the heads of all its input buffers, the CU will consume these values, compare actual and predicted branch outcomes, and then broadcast the appropriate `predictionStatus` signal to all PUs and the LSU. Of course, if the branch prediction was incorrect, all entries in the CU's input buffers are flushed, and the program counter is reset appropriately based on the actual branch outcome. Therefore, the comparison of actual and predicted branch outcomes and the subsequent `predictionStatus` signaling is always performed in the order of prediction of corresponding branch outcomes and subsequent `speculativeExecution` signaling.

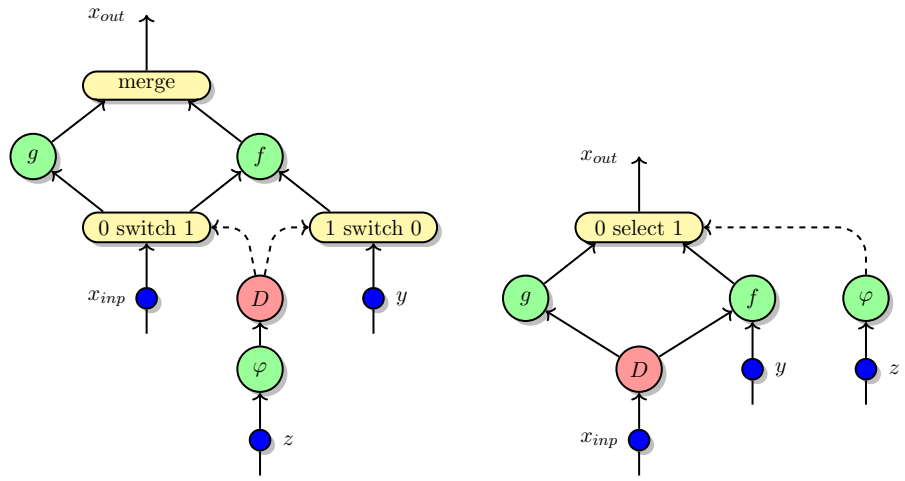
Note that not only the buffers of PUs and the LSU but also the routers in the DTN must 'mark' speculative execution so that all appropriate values in the SCAD machine are flushed or discarded in the event of a wrong branch prediction. However, there is an important difference between the behavior of PU buffers and LSU buffers in SCAD in the context of speculative execution. When an entry under speculation reaches the head of input buffers of the LSU, the LSU has to stall if the corresponding operation is a memory write operation so that no memory updates are carried out until the corresponding branch is evaluated. On the other hand, PUs and the DTN can continue operating normally until a wrong branch prediction is signaled (via the `predictionStatus` signal). That is, PUs can consume available values from input buffer heads (even if these values are under speculation), compute and enqueue the result to the tail of its output buffer. Similarly, the DTN can transport available result values (even if these values are under speculation) from the heads of output buffers to appropriate input buffers. This is possible since the values are not stored in a shared namespace (register or memory) that cannot be rolled back in case of wrong predictions. Instead, they are produced and directly communicated for consumption by PUs. This, in fact, enables speculative execution in SCAD processors. Clearly, PUs, DTN routers, and the LSU must maintain a count of the number of consumed speculation marks, so that wrong entries are not 'unmarked' on receiving next `predictionStatus` signals informing correct predictions.

Predicated execution

For predicated execution, the control flow is translated to dataflow, and the resulting dataflow graph is executed. Nodes of a dataflow graph are associated with operations that are applied to operands sent to these nodes along the edges between the nodes. The control flow of programs is displayed in the dataflow graphs using special nodes such as *switch* and *select* nodes, and it is not a trivial task to implement these nodes as processing units in SCAD. Consider the conditional statement:

$$if\ \varphi(z)\ \ then\ \{x = f(x, y);\}\ else\ \{x = g(x);\}$$

The assignment $x = f(x, y)$ is executed if the boolean expression $\varphi(z)$ evaluates to true. Otherwise, the assignment $x = g(x)$ is executed. Depending on when the boolean condition is checked, there are two ways to translate the above conditional statement. Dataflow graphs shown in Figures 2.7a and 2.7b both implement the above conditional statement, where x_{inp} is the initial value of x.



(a) *switch and merge nodes* **(b)** *select node*

Figure 2.7.: *Dataflow graphs for conditional statement*

In the dataflow graph shown in Figure 2.7a, special nodes *switch* and *merge* are used to implement the conditional branching. The *switch* node has two inputs: a boolean input c (shown by a dashed edge) and in. It has two outputs : out_0 and out_1. If the boolean input c is true (respectively false), the value in the other input in is forwarded to out_1 (respectively out_0). To execute the conditional statement, the *switch* node directs the input value x_{inp} to one of the two paths depending on the outcome of the boolean expression $\varphi(z)$. This triggers the firing of nodes in the selected path while nodes in the other path are not executed. Finally, the *merge* node fires when the value is available in any one of its two inputs and forwards this value to the output. In the dataflow graph shown in Figure 2.7b, a special node *select* is used. The *select* node has one output and three inputs: a boolean input c (shown by a dashed edge), in_0 and in_1. If the boolean input c is true (respectively false), the value in in_1 (respectively in_0) is forwarded to the output. To execute the conditional statement, the *select* node selects the value computed by one of the two paths depending on the outcome of boolean expression $\varphi(z)$. Unlike the *switch* based dataflow graph which first evaluates the branch condition, the *select* based dataflow graph first executes both paths irrespective of the branch condition outcome and then selects the computed result from the appropriate path depending on the branch condition outcome.

Consider the dataflow graph in Figure 2.7a. To implement the same in

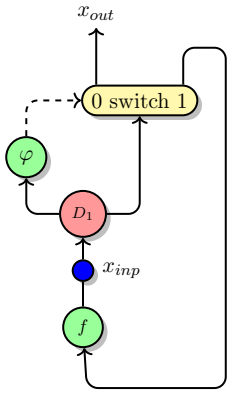


Figure 2.8.: *Dataflow graph for while loop statement using switch node*

SCAD, the SCAD compiler must generate move code for computations in both 'if' and 'else' paths. However, at runtime, only move instructions registered for computation of one path will find values to transport. To ensure the correct execution, it is necessary that data transports registered for the computation of the other path are flushed or discarded. In other words, the *switch* node can be implemented as a PU in the SCAD processor only with additional provisions. We do not explore this possibility in this thesis. Notice that the implementation of the conditional statement by the *switch* based dataflow graph is similar to conditional branching by stalling the CU, where the CU stalls on encountering a branch move instruction until the branch condition outcome is computed. This is a sequential computation of the conditional statement. On the other hand, the *select* based dataflow graph performs a parallel computation of the conditional statement but requires more hardware resources. We observe that it is possible to implement the *select* node as a PU in the SCAD processor. To this end, the PU will have three inputs, c, in_0 and in_1, corresponding to the inputs of the *select* node and a fourth input to hold the number of copies to produce. If the boolean input is available, the *select* node in the dataflow graph will forward the appropriate input value, if available, to its output irrespective of the availability of other input value. However, the *select* PU in a SCAD processor must wait for values to be available at all its input buffer heads for consumption, then produce copies of the appropriate input value in the output buffer and discard the other input value. It is important that the PUs in a SCAD processor consume all input values registered to execute an operation to avoid any stray values (values not accounted for).

The dataflow graph for loop statements must be constructed using *switch* nodes to ensure the termination of the loop. For example, consider the while statement:

$$while\ \varphi(x)\ \ \{\ x = f(x);\ \}$$

It may be expressed as

$$if\ \varphi(x)\ \ then\ \{do\ \{x = f(x);\}\ while\ \varphi(x);\}\ else\ \{\}$$

If the above conditional statement is translated to a dataflow graph using only *select* node, it will wait forever for the computation in the 'then' branch to terminate. Therefore, the *switch* node must be used as shown in Figure 2.8 so that the loop body is executed only after the condition check. Since the *switch* node cannot be implemented as a PU in SCAD without additional support, the SCAD processors must rely on branch prediction to exploit ILP across loop boundaries. However, various techniques such as loop unrolling and hyperblock formation [MLCH92] to cover conditional statements are applicable in SCAD.

2.1.4. Remarks on the execution paradigm

The program counter of the CU steers the control flow of the program execution in SCAD. In this respect, SCAD adheres to the control-flow computing model. However, intermediate results are produced and communicated directly for consumption instead of using a shared namespace for storing these values. In this respect, SCAD adheres to the dataflow computing model. This way, SCAD employs a *hybrid control-flow dataflow model of computation*. A code generator for SCAD must compile a source code program to a sequence of move instructions that are registered in the execution framework of SCAD by the control unit. Of course, to decide about the communication of values between PUs, the SCAD compiler must know which values are produced by the PUs. To this end, it must allocate instructions to PUs (instruction placement). The actual firing of PUs (instruction issue) is determined at runtime when appropriate input (operand) values are available. From this point of view, SCAD is categorized as an *exposed datapath* architecture that presents a *static placement dynamic issue (SPDI)* scheduling model. See Chapter 6 for a review of other architectures based on the above characterizations.

Hardware complexity

The intermediate results from executions of PUs in SCAD are stored in their output buffers. Both address and value lanes in each output buffer are *'pure' FIFO buffers*. The MIB and PU always add address and value entries, respectively, to the tail of respective lanes in any output buffer. The DTN always removes these entries from the head of respective lanes. Since the number of entries in output FIFO buffers scale better compared to registers in a register file, the output buffers in SCAD are capable of holding more intermediate results. Moreover, with a central register file, multiple instructions will have to use a limited number of ports to write their execution results. When increasing the number of ports, the area and access times of registers increase at a rate of n^3 and $n^{3/2}$ [RDKM00], while power dissipation increases super-linearly at the rate of n^2 to n^3 [ZyKo98]. In contrast, every PU in SCAD has its own output buffer.

Notice that input buffers in SCAD are *more difficult buffers*. The address lane in an input buffer is still a FIFO buffer since address entries are always populated from the tail by the MIB and are removed from the head by the

PU. On the other hand, though entries in the value lane are removed from the head by PU, the values delivered via the DTN are not simply added to the tail. Instead, the value val in a DTN message (src, tgt, val) must find its place in the value lane adjacent to the address entry src closest to the head of the address lane. This is necessary so that the correct operand values occur at the input buffer heads of a PU for executing the next operation. In register machines that execute typical reduced instruction set computer (RISC) instructions, operand values are encoded by register names. Again, with a central register file, multiple instructions will have to use a limited number of ports to read their operand values, while every PU in SCAD has its own input buffers. In dataflow machines, a token-matching hardware [GuKW85] is required to find the operand values that refer to the same operation. To match n left operand values with n corresponding right operand values, a total of $n!$ comparisons are needed in the worst case. This is even assuming that all corresponding operand values are already available for comparison. In reality, some of these operand values may not yet be computed. In SCAD input buffers, only n comparisons are needed in the worst case to determine the appropriate slot for an incoming operand value. Furthermore, the synchronous registration of move instructions guarantees that any incoming value will have a slot reserved in the respective input buffer.

Compiler flexibility

The values can be transported only from head of output buffers to the tail of input buffers. Therefore, to move values from output buffers to input buffers, the SCAD compiler must know the order of values that occur in these buffers. To this end, the compiler must determine an order of instructions executed by each PU, which in turn determines the order of result values in the output buffer and the operand values in the input buffers of each PU. This means that depending on the ordering of values, some value to transport to an input buffer might not be currently found at head of the output buffer of the PU that produces this value. In this case, the compiler will have to rotate the current values at the head of the output buffer (via some input buffers) to access the relevant value for transportation. The rotation of values is necessary to avoid storing these values temporarily in a shared namespace (register file), i.e., equivalently to execute the whole program by direct instruction communication avoiding the use of registers.

The rotation of values incurs not only additional moves to input buffers, but also additional computational overhead since they need to be then copied by PUs to output buffers for later transportation (more details are given in Chapter 3). Clearly, this transportation and computational overhead are not desirable due to its adverse impact on performance. Therefore, the compiler must try to minimize the overhead by appropriately ordering instructions on PUs. With more PUs and thus more buffers, restrictions to access values lessen, consequently requiring lesser overhead. Nevertheless, the ordering constraint restricts the compiler in freely allocating instructions to PUs to maximize the use of ILP. However, we will see in experimental results (in Section 4.3) that it

is not difficult to avoid the overhead in SCAD. In other words, utilizing *direct instruction communication* comes at a reasonable cost in the SCAD execution paradigm. Furthermore, the compromise in the use of ILP due to the ordering constraint is negligible. By an experimental comparison of SCAD with its variant architectures (discussed next), we show that the aforementioned qualities can be attributed to the right blend of control-flow and dataflow computing characteristics in SCAD. In the following sections, we introduce subtle variants of SCAD: the dynamically ordered SCAD (DO-SCAD) with more dataflow computing characteristics, where the compiler is completely free to allocate instructions to maximize the use of ILP, but at the cost of considerably more complex hardware compared to SCAD; and the statically ordered SCAD (SO-SCAD) with more control-flow computing characteristics, where hardware complexity is lesser, but the compiler is even more restricted in effectively exploiting ILP compared to SCAD.

2.2. Dynamically Ordered SCAD

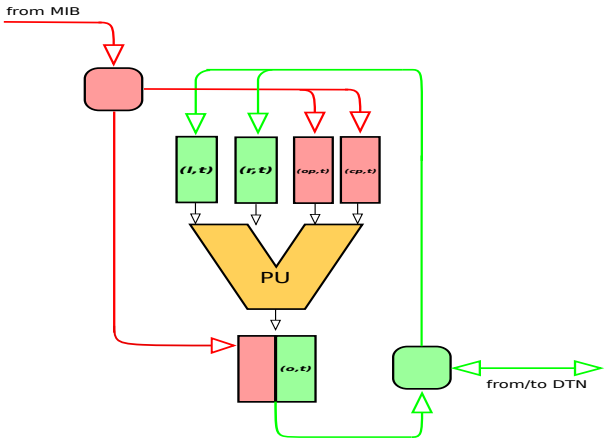


Figure 2.9.: *A processing unit in a dynamically ordered SCAD machine*

In SCAD, output buffer addresses registered in program order in the address lane in input buffers are used at runtime to reorder values that arrive at PU inputs via the DTN. This way, it is ensured that the correct operands for executing the next operation occur at the heads of the input buffers of the PU. Figure 2.9 shows a PU in a dynamically ordered SCAD (DO-SCAD) machine. Instead of FIFO buffers, there is a *pool of entries* at PU inputs and outputs. Unlike the ordering used in SCAD, tag matching is used by PUs in DO-SCAD architectures to find the correct operands to execute an operation. For simplicity, we assume that all tag values are determined at compile time. A PU's input pool holds pairs (val, t) of entries, where val is the operand value and t is the tag associated with that operand. In the PU's outputs, there are two kinds of tuples: (1) (val, t) where val is the result of an execution of an operation, and t is the tag associated with its operands,

and (2) (adr, t_r, t_o) where adr is the address of an input of the destination PU to which the value val associated with operand tag t_o must be transported and t_r is the tag associated with and transported along with val for operand matching in the destination PU. Clearly, the move instructions must now be augmented with operand and result tags as follows: $src\{t_o\} \rightarrow tgt\{t_r\}$. For example, consider two instructions i and j whose operands are associated with tags t_i and t_j, respectively.

$$x_{\mathsf{tgt}(i)} = x_{\mathsf{srcL}(i)} \odot_i x_{\mathsf{srcR}(i)}$$
$$x_{\mathsf{tgt}(j)} = x_{\mathsf{tgt}(i)} \odot_j x_{\mathsf{srcR}(j)}$$

Note that the left operand of instruction j is the target of instruction i ($x_{\mathsf{tgt}(i)}$). Assume that instructions i and j are assigned to PUs m and n, respectively. Then, the move instruction $u_m@o\{t_i\} \rightarrow u_n@l\{t_j\}$ transports the target of instruction i from the output of PU m to the left input of PU n to execute instruction j. The execution of a move program now proceeds as follows: the move instruction $src\{t_o\} \rightarrow tgt\{t_r\}$ is broadcast via the MIB to all PUs. The PU output with address src will add the entry (tgt, t_r, t_o) to its pool of entries. A PU can fire if it finds operand values with matching tags for execution. The result of an execution along with an operand tag is added to the output pool of tuples (val, t_o). At the PU's outputs, the operand tag in the pool of tuples (adr, t_r, t_o) is matched with that in the pool of tuples (val, t_o). If a match is found, the value val and result tag t_r are made available to the DTN for transporting to the PU input with address adr.

To avoid overhead (additional rotation of values), the SCAD compiler has to carefully allocate and decide about the order of instructions on the PUs, so that the order of values dequeued from the output buffers and enqueued to the input buffers concur. This reduces the SCAD compiler's flexibility to allocate instructions to PUs to utilize all ILP contained in a program. In DO-SCAD, matching tags at the PU output ensures that correct values are sent from the output pool to input pools, and matching tags at inputs ensures that the correct operand values are consumed by the PU for execution. So there is no notion of any compiler determined ordering of values in DO-SCAD. Therefore, the DO-SCAD compiler has the complete freedom to allocate instructions to PUs to utilize maximal ILP. However, this comes at the cost of considerably more complex hardware that is needed to *accommodate and match tags*. Moreover, DO-SCAD architectures face the same memory ordering problem that dataflow computers suffer from.

2.3. Statically Ordered SCAD

In a statically ordered SCAD (SO-SCAD) architecture, the PUs only rely on the order of arrival of values at the PU inputs to identify the operands for executing the next operation. Figure 2.10 shows a PU in a SO-SCAD machine. Note that there are no address lanes in the input buffers. The input and output values of PUs now reside in *'pure' FIFO buffers* unlike the more

difficult input buffers of the SCAD architecture, thus reducing the hardware complexity and simplifying the execution of a move program. When a move instruction $src \rightarrow tgt$ is broadcast via the MIB to all PUs, only the output buffer with address src will add the entry $(tgt, \perp)$ to its tail. Like SCAD, the result of a PU's execution is added to the value slot in the entry $(adr, \perp)$ closest to the head of its output buffer. The DTN snoops the head of output buffers for transporting values to the addressed input buffers. However, the transported values are then simply enqueued to the respective input buffers' tail.

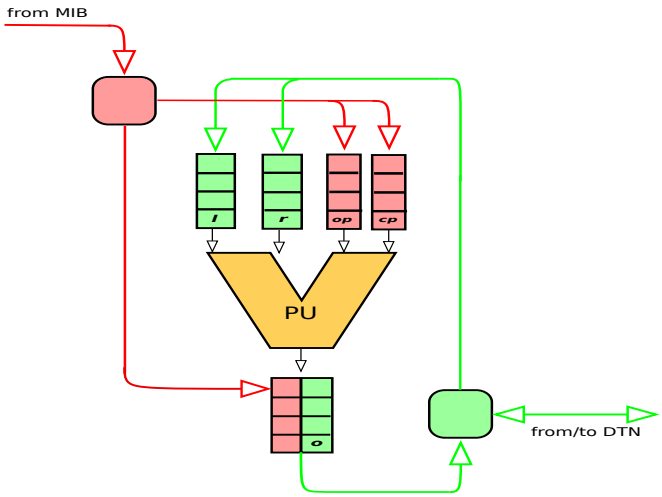


Figure 2.10.: *A processing unit in a statically ordered SCAD machine*

In SCAD, a compiler determined order of operand values (implied by order of instructions on PUs) is enforced locally at each input buffer in each PU at runtime. This ensures that irrespective of when these values are produced and transported, correct operand values always occur at input buffer heads for executing the next operation. In other words, the runtime reordering provision in input buffers allows the execution in SCAD to tolerate a variable PU and DTN latency. The ordering of instructions is determined at compile-time and enforced at runtime in SCAD, making it a *hybrid ordered* architecture. In DO-SCAD, ordering is both determined and enforced at runtime, thus *dynamically ordered*. In SO-SCAD PUs, the next operation to execute is decided simply by order of arrival of values at PU inputs, which in turn depend on when these values are produced and transported from some output buffers. To ensure the correct order of arrival of values at PU inputs, it is necessary that the SO-SCAD compiler not only determines the order of instructions on PUs but also enforces this order in some way (discussed later). Therefore, ordering is both determined and enforced at compile time in SO-SCAD, thus *statically ordered*. Notice that the SO-SCAD execution paradigm resembles the classic control-flow (or register) architectures where the compiler encodes operands and results of an instruction by register addresses. Similarly, the DO-SCAD execution paradigm resembles classic dataflow architectures.

There are two apparent ways in which the SO-SCAD compiler can enforce the ordering: (1) The instruction issue (firing of PUs) times can be statically determined, and the control unit can trigger firings of PUs at these predetermined times to ensure that the produced values are transported and arrive in the expected order at each input buffer. To that end, the latency of PUs and DTN must be exposed so that the compiler can derive a correct static schedule similar to VLIW compilers [FERN84; Gros00]. The main drawback is that a statically determined instruction issue inhibits execution in dataflow order (thus restricting the use of ILP) and cannot adapt to a variable latency of PUs and DTNs. Furthermore, it is considerably more difficult to avoid overhead (due to the need to rotate values) in a SO-SCAD machine with static instruction issue (see experimental results in Section 4.3) compared to a SCAD machine. (2) Alternatively, the compiler may determine that all values arriving at the same input buffer are produced in the expected order in the same output buffer (i.e., by the same PU). Since the DTN always transports values from heads of output buffers, it is guaranteed that the operand values destined to the same input buffer will arrive in the expected order. Although this way of enforcing an order seems like a good approach at first, the compiler is so restricted in allocating and ordering instructions on PUs that it often requires as many PUs as the number of instructions to avoid an additional rotation of values or overhead (see experimental results in Section 4.3).

Chapter 3

Code Generation Techniques for SCAD

Contents

In this chapter, we explain the underlying principles for code generation for SCAD architectures. We discuss two contrasting ways of generating move code for SCAD machines: register oriented and queue oriented code generation, explaining why queue oriented code generation is more adequate for SCAD architectures [BhJS16]. To simplify explanations, we consider move code generation for a universal SCAD machine, which is defined as follows:

Definition 3.1 ⟨ Universal SCAD Machine ⟩
A universal SCAD machine is a SCAD machine with a single universal processing unit capable of executing memory accesses, standard binary and unary operations. It has one output buffer ‘ o’ to store the result of each operation and four input buffers: ‘ l’ to store the first (left) operand, ‘ r’ to store the second (right) operand, ‘ op’ to store the operation to be executed, and ‘ cp’ to store the number of copies of the result to be added to the output buffer.

3.1. Register Oriented Code Generation

Recall that it is possible to include register files with any arbitrary number of registers, input ports (to read and/or write registers), and output ports (to access read registers) in the SCAD architecture. A register file's input ports will receive immediate data (addresses of registers to be read or written) from the control unit via the MIB. The output ports send data (register content) to other processing units (PUs) via the DTN. Code generation for register machines is an extensively researched field, so that already established efficient methods may be used to compile programs to a sequence of typical reduced instruction set computer (RISC) instructions where operands and results of an instruction are register addresses. A register oriented code generator for SCAD may first translate these RISC instructions to corresponding move instructions for a SCAD machine and then analyze the resulting sequence of move instructions for bypassing as many reads and writes of registers as possible.

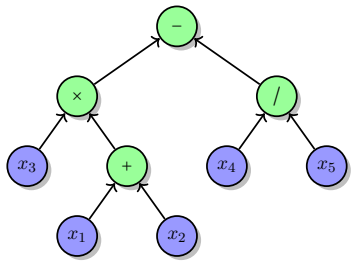
expression tree	register program	SCAD program
	$r_0 \leftarrow x_3$ $r_1 \leftarrow x_1$ $r_2 \leftarrow x_2$ $r_1 \leftarrow r_1 + r_2$ $r_0 \leftarrow r_0 * r_1$ $r_1 \leftarrow x_4$ $r_2 \leftarrow x_5$ $r_1 \leftarrow r_1/r_2$ $r_0 \leftarrow r_0 - r_1$	$adr(x_3) \rightarrow l$; $load \rightarrow op$; $1 \rightarrow cp$; $o \rightarrow r_0$; $adr(x_1) \rightarrow l$; $load \rightarrow op$; $1 \rightarrow cp$; $adr(x_2) \rightarrow l$; $load \rightarrow op$; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; $add \rightarrow op$; $1 \rightarrow cp$; $r_0 \rightarrow l$; $o \rightarrow r$; $mul \rightarrow op$; $1 \rightarrow cp$; $o \rightarrow r_0$; $adr(x_4) \rightarrow l$; $load \rightarrow op$; $1 \rightarrow cp$; $adr(x_5) \rightarrow l$; $load \rightarrow op$; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; $div \rightarrow op$; $1 \rightarrow cp$; $r_0 \rightarrow l$; $o \rightarrow r$; $sub \rightarrow op$; $1 \rightarrow cp$;

Figure 3.1.: *An expression tree with its register program, and corresponding SCAD program*

For example, consider the expression tree shown in Figure 3.1. According to the Sethi-Ullmann algorithm [SeUl70], at least 3 registers are required to evaluate the expression tree without storing intermediate results to the main memory. This is achieved by ordering the nodes by a depth-first traversal (post-order traversal) of the tree and then assigning registers to nodes. The resulting RISC program is shown in Figure 3.1. It is straightforward to translate each RISC instruction to a set of move instructions for the execution on a universal SCAD machine so that after the execution of, the RISC instruction in a traditional register machine, and the set of move instructions in the universal SCAD machine, the content of registers are the same in both machines. Figure 3.1 shows a SCAD program for a universal SCAD machine that bypasses all accesses to registers r_1 and r_2. It is understood as follows: After loading x_3 to register r_0, $x_1 + x_2$ may be evaluated without using registers r_1 or r_2. This is because after loading x_1 and x_2 from the main memory, these

values are available in that order in the output buffer o of the universal SCAD machine for transportation to input buffers l and r, respectively, for computing $x_1 + x_2$. The $*$ operation will now find its left operand x_3 in register r_0 and its right operand $x_1 + x_2$ at the head of output buffer o. After storing the multiplication result $x_3 * (x_1 + x_2)$ in register r_0, again similarly the use of r_1 and r_2 may be avoided in evaluating x_4/x_5. Finally, the subtraction operation will find its left operand $x_3 * (x_1 + x_2)$ in register r_0 and its right operand at the head of output buffer o.

Note that it is not possible to bypass all register accesses this way in general. In the above example, x_3 and the result of $*$ must be stored in register r_0 (or rotated incurring overhead), so that + and / can respectively be performed by direct communication of corresponding operand values from the output buffer to the input buffers. Clearly, the reason for this restriction is the ordering of the instructions by the *depth-first traversal* that was motivated by the reuse of registers in the first place. Furthermore, the depth-first ordering of instructions limits the use of ILP offered by programs. All instructions in each level of an expression tree are independent, and the maximal ILP is used when instructions are executed level-wise when computing expression trees.

3.2. Queue Oriented Code Generation

A queue machine [Voll70; FeEr81] reads operands for executing an operation from the head of a queue and adds the results to the tail of that queue. The architecture of a queue machine is shown in Figure 3.2.

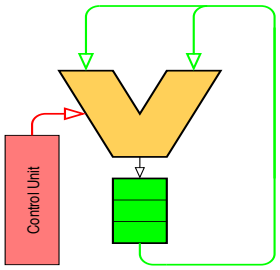


Figure 3.2.: *Architecture of a queue machine*

3.2.1. Code generation for queue machines

A queue program to evaluate an expression tree is generated by a breadth-first traversal of the tree [FeEr81]. A consistent left to right or right to left traversal ensures that the operands required to execute operations at one level are available in the queue in the correct order. The queue program for the expression tree and the contents of the queue after executing each instruction of that program is shown in Figure 3.3. A list of queue instructions is listed in Table 3.1.

queue instruction	description
load adr,n	Load data from memory address *adr* and add *n* copies of the loaded value to the tail of the queue.
store adr	Store the value from the head of the queue to the memory address *adr*.
opcode n	Dequeue necessary operands from the head of the queue to execute the operation *opcode* and add *n* copies of the result to the tail of the queue.
swap	Dequeue two operands from the head of the queue, *swap* them, and add them to the tail of the queue.
dup n	Dequeue one operand from the head of the queue, and add *n* copies of it to the tail of the queue.
goto PC, L	Unconditional Branch: Transfer the control from *PC* to *PC+L*
ifGoto PC, L	Conditional Branch: Transfer the control from *PC* to *PC+L* if the head of the queue holds else *PC+1*

Table 3.1.: *List of queue instructions*

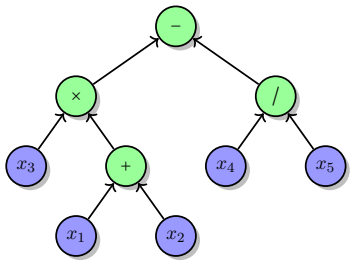
expression tree	queue program	queue content
	*load* x_1,1	$[x_1]$
	load x_2,1	$[x_1, x_2]$
	load x_3,1	$[x_1, x_2, x_3]$
	add 1	$[x_3, x_1 + x_2]$
	load x_4,1	$[x_3, x_1 + x_2, x_4]$
	load x_5,1	$[x_3, x_1 + x_2, x_4, x_5]$
	mul 1	$[x_4, x_5, x_3 * (x_1 + x_2)]$
	div 1	$[x_3 * (x_1 + x_2), x_4/x_5]$
	sub 1	$[x_3 * (x_1 + x_2) - (x_4/x_5)]$

Figure 3.3.: *An expression tree with its queue program and the content of the queue after executing each instruction*

Basic blocks of programs are often represented by directed acyclic graphs (DAGs). Generating a queue program for an expression *tree* is easy since an expression tree is by definition a level-planar graph [ScLY02]. However, generating queue programs for general expression DAGs involves first converting the DAG into a *level-planar graph* and then performing a breadth-first traversal of the graph [ScLY02] as shown in Figure 3.4: The given expression DAG is first levelized, which means that operations must only refer to operands at the same level. This can be easily achieved by introducing *dup* operations, which take a value from the head of the queue and add some copies of it to the tail of the queue. Then, the graph is planarized. This means that crossing edges are removed by inserting *swap* operations that take two values from the head of the queue and add them in exchanged order to the tail of the queue. One

can sometimes avoid the introduction of *swap* operations by suitable ordering of input or output nodes, but not in general. Finally, another levelization is usually required since the *swap* operations may be placed at new levels.

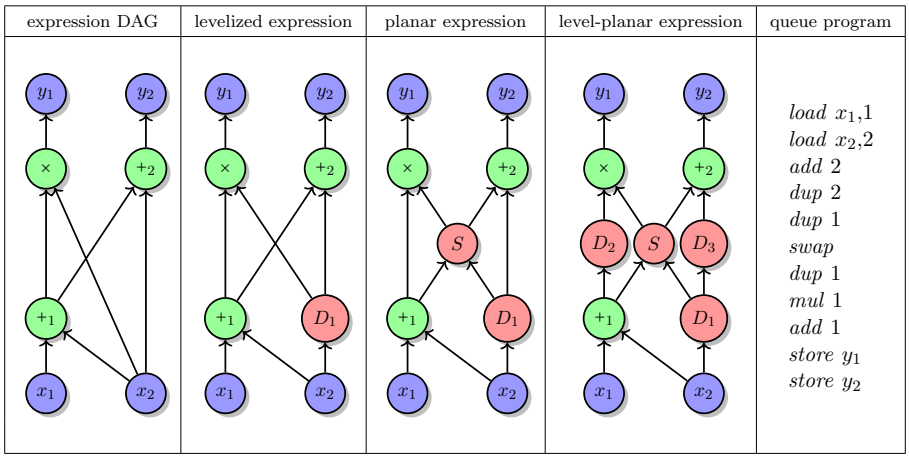


Figure 3.4.: *An expression DAG with its levelized version, its planarized version, the final level-planar expression DAG, and the obtained queue program*

3.2.2. SCAD code from queue code

queue instruction	corresponding SCAD move instructions			
load adr, n	$adr \rightarrow l$;		$load \rightarrow op$;	$n \rightarrow cp$;
store adr	$adr \rightarrow l$;	$o \rightarrow r$;	$store \rightarrow op$;	
opcode n	$o \rightarrow l$;	$o \rightarrow r$;	$opcode \rightarrow op$;	$n \rightarrow cp$;
swap	$o \rightarrow l$;	$o \rightarrow r$;	$swap \rightarrow op$;	
dup n	$o \rightarrow l$;		$dup \rightarrow op$;	$n \rightarrow cp$;
goto PC, L	$PC \rightarrow l$;		$goto \rightarrow op$;	$L \rightarrow cp$;
ifGoto PC, L	$PC \rightarrow l$;	$o \rightarrow r$;	$ifGoto \rightarrow op$;	$L \rightarrow cp$;

Table 3.2.: *Mapping queue machine instructions to move instructions of a universal SCAD machine*

To generate SCAD code from a given queue code, we map each queue instruction to a sequence of move instructions for the universal SCAD machine as listed in Table 3.2 ([BhJS16]). Thus, the following theorem is stated.

Theorem 3.1 (Queue Simulation) *A queue machine can be simulated by a universal SCAD machine (consequently by any SCAD machine with multiple processing units).*

Proof See Table 3.2. Note that the contents of the queue in the queue machine and the output buffer in the universal SCAD machine will be identical after each execution of each queue instruction on the queue machine and the corresponding move instructions on the universal SCAD machine, respectively. It is not difficult to adapt the mapping for a SCAD machine with multiple processing units, given an assignment of each queue instruction to that processing unit in the SCAD machine which executes the queue instruction. ■

Figure 3.5 shows the move code program for the universal SCAD machine for the expression tree, obtained by translation from the queue program according to the mapping in Table 3.2. Clearly, SCAD programs obtained by translation of queue programs do not need to use registers. In other words, all intermediate results are communicated or transported directly from the output buffers to the input buffers of processing units. This is enabled by the ordering of nodes in the expression tree by a *breadth-first traversal* in contrast to the depth-first ordering used in register oriented code generation. Importantly, the breadth-first ordering allows full use of ILP since independent instructions are ordered consecutively. However, as we have seen, overhead in terms of *dup* and *swap* operations might be required to compile basic blocks or expression DAGs. The use of *dup* and *swap* overhead not only incur additional computation but also lead to additional move instructions (transportation overhead) in SCAD.

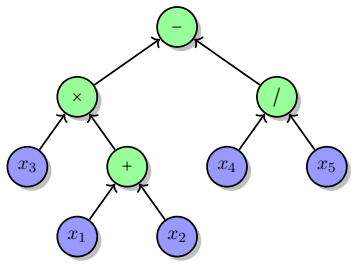
expression tree	queue program	SCAD program
	*load* x_1,1 *load* x_2,1 *load* x_3,1 *add* 1 *load* x_4,1 *load* x_5,1 *mul* 1 *div* 1 *sub* 1	$adr(x_1) \rightarrow l$; *load*$\rightarrow$ *op*; $1 \rightarrow cp$; $adr(x_2) \rightarrow l$; *load*$\rightarrow$ *op*; $1 \rightarrow cp$; $adr(x_3) \rightarrow l$; *load*$\rightarrow$ *op*; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; *add*$\rightarrow$ *op*; $1 \rightarrow cp$; $adr(x_4) \rightarrow l$; *load*$\rightarrow$ *op*; $1 \rightarrow cp$; $adr(x_5) \rightarrow l$; *load*$\rightarrow$ *op*; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; *mul*$\rightarrow$ *op*; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; *div*$\rightarrow$ *op*; $1 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow r$; *sub*$\rightarrow$ *op*; $1 \rightarrow cp$;

Figure 3.5.: *An expression tree with its queue program, and corresponding SCAD program*

3.2.3. Overhead

Clearly, *dup* and *swap* overheads adversely affect the performance. However, since there are multiple buffers in a SCAD machine, it often requires lesser overhead compared to a queue machine that only has one central queue.

Theorem 3.2 (Overhead) *For any queue program, there is a corresponding SCAD program with the same number of dup and swap in-*

structions. However, there are SCAD programs without dup and swap instructions where the queue machine requires dup and swap instructions.

Proof Given any queue program, the corresponding SCAD program generated by the mapping given in Table 3.2 will have the same number of *dup* and *swap* instructions. The converse is however not true. For the simple basic block $y_1 = x_1 - x_2; y_2 = x_1/x_2$, the expression DAG and its level-planar version are shown in Figure 3.6. As can be seen, the expression DAG is not planar and therefore, we have to introduce additional *dup* and *swap* instructions for the queue machine. This is required since all operands have to be brought into a *total* order in the queue machine since it has only one queue. After loading, duplicating, and swapping, the content of the queue is $[x_1, x_2, x_1, x_2]$ so that the two binary operations can read their operands in the right order to compute $x_1 - x_2$ and x_1/x_2. For the universal SCAD machine, we can do without *dup* and *swap* instructions: To this end, we first load two copies of x_1 and then two copies of x_2, so that the content of the output buffer o is $[x_1, x_1, x_2, x_2]$ after the first two lines. We then move the two copies of x_1 at head of output buffer o to input buffer l so that l holds values $[x_1, x_1]$. The remaining two copies of x_2, which are now at the head of output buffer o, are moved to the input buffer r so that r holds values $[x_2, x_2]$. Finally, opcodes *sub* and *div* are moved to the *op* buffer so that the corresponding results are produced in the output buffer for storing.

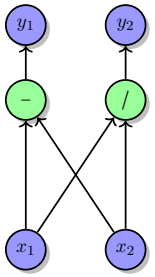
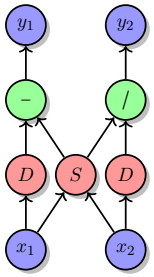
expression DAG	level-planar DAG	queue program	SCAD program
		*load* x_1,2 *load* x_2,2 *dup* 1 *swap* *dup* 1 *sub* 1 *div* 1 *store* y_1 *store* y_2	$adr(x_1) \rightarrow l$; *load*$\rightarrow$ *op*; $2 \rightarrow cp$; $adr(x_2) \rightarrow l$; *load*$\rightarrow$ *op*; $2 \rightarrow cp$; $o \rightarrow l$; $o \rightarrow l$; $o \rightarrow r$; $o \rightarrow r$; *sub* $\rightarrow$ *op*; $1 \rightarrow cp$; *div* $\rightarrow$ *op*; $1 \rightarrow cp$; $adr(y_1) \rightarrow l$; $o \rightarrow r$; *store* $\rightarrow$ *op*; $adr(y_2) \rightarrow l$; $o \rightarrow r$; *store* $\rightarrow$ *op*;

Figure 3.6.: *A given expression DAG with its planarized version, the obtained queue program, and a SCAD program without swap and dup instructions*

■

Since there are two input buffers and one output buffer in the universal SCAD machine, there are fewer restrictions in accessing values compared to the queue machine with only one central queue. If we add more processing units to a SCAD processor, the more number of input and output buffers further reduces

the restrictions for accessing values. Consequently, there is a lesser need for *dup* and *swap* instructions. Consider the expression DAG given in Figure 3.7. For the same expression DAG, both the queue machine and the universal SCAD machine require *dup* and/or *swap* instructions as shown in Figure 3.4. This is because both x_2 and the result of $+_1$ occur in the output buffer of the single universal processing unit in the universal SCAD machine, necessitating *dup* and *swap* overhead to execute $\times$ and $+_2$. On a SCAD machine with one processing unit (u) and one load-store unit (lsu), x_2 will be loaded to the output buffer of lsu, and the result of $+_1$ will be produced in the output buffer of u. This avoids the need for any overhead to compute the expression DAG. The resulting move program is shown in Figure 3.7. In the move program in Figure 3.7, we use the following notations to denote different buffer addresses: The addresses of left and right operand buffers of any processing unit u are denoted by $u@l$ and $u@r$, respectively. The output buffer address is denoted by $u@o$. The addresses of the opcode and the copies buffers are denoted by $u@op$ and $u@cp$, respectively. The corresponding buffer addresses of LSU are denoted by $lsu@l$, $lsu@r$, $lsu@o$, $lsu@op$ and $lsu@cp$, respectively. We use the above notation in the rest of this thesis.

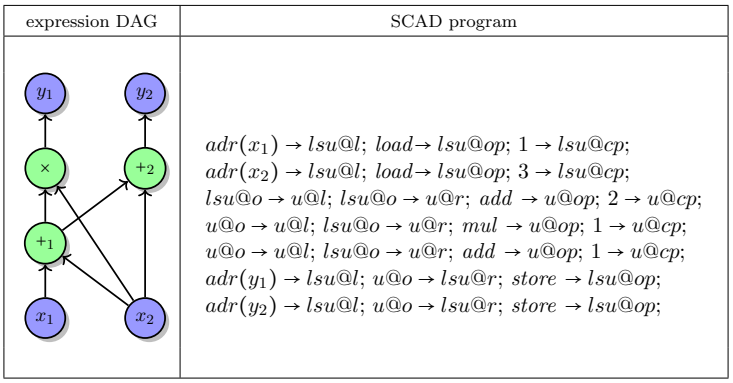


Figure 3.7.: *An expression DAG with a move cod program without dup and swap instructions for a SCAD machine with one load-store unit (lsu) and one processing unit (u)*

Chapter 4

Optimal Code Generation

Contents

Increasing numbers of *dup* and *swap* operations not only degrade the performance but also increase the code size and power consumption of SCAD machines. Therefore, it is desirable to obtain optimal code for a given program and a given SCAD machine, where optimal refers to a minimal number of overhead operations. We have seen that with more processing units, lesser overhead is necessary. In Section 4.1, we prove that it is an NP-hard problem to determine for a given program, the minimal number of processing units required in a SCAD machine to execute the program without any overhead [Ande17]. We then map the decision version of the problem to an equivalent satisfiability (SAT) problem in Section 4.2 [BhSc16; BhSc17; BhSc17a]. Besides constraints for the resource-constrained (minimize the number of processing units) optimal code generation, we also formulate optional boolean constraints in Section 4.2.3 to optimize the execution time for time-constrained optimal code generation.

4.1. Complexity Analysis

Recall that *dup* and *swap* operations are used to rotate values at output buffer heads to access other relevant values from these output buffers. The alternative is to store values at the output buffer heads to memory and load back these values after accessing other relevant values. However, memory accesses must also be avoided since they are expensive. Therefore, for complexity analysis, we limit memory accesses to just one memory read to load the initial value of variables in the program.

Theorem 4.1 (NP-hardness) *Given a program $\mathcal{P}$ and a SCAD machine $\mathcal{S}$ with one load-store unit and p processing units, it is an NP-hard problem to determine if it is possible to schedule the program $\mathcal{P}$ on the SCAD machine $\mathcal{S}$ so that:*

- *memory accesses are minimized (to one read per load variable)*
- *overhead (dup or swap) operations are not used.*

Proof We prove the NP-hardness by a reduction from the *graph coloring* problem that is known to be NP-complete [Karp72]. The decision version of the graph coloring problem is stated as follows: Given p colors $\{c_1, \ldots, c_p\}$ and an undirected graph $G = (V, E)$ with n vertices $V := \{v_1, \ldots, v_n\}$ and m edges $E := \{e_1, \ldots, e_m\}$, determine if there exist an assignment of colors to vertices, $Col : V \longrightarrow \{c_1, \ldots, c_p\}$, so that no two adjacent vertices share the same color, i.e., $\forall_{e_i:=(v_j,v_k)} Col(v_j) \neq Col(v_k)$.

Reduction: For the graph G, we construct a program $\mathcal{P}$ whose control flow is shown in Figure 4.1. $\mathcal{P}$ contains basic blocks $B := \{B_1, \ldots, B_m\}$, B_L and B_R. Node A is only used to generate control flows to the set of basic blocks B, each of which branches to B_L or B_R. Basic blocks B write to vertex variables $v_1, \ldots, v_n$, while B_L and B_R read vertex variables. The basic idea of the reduction is that the processing unit (PU) q that produces a vertex variable v_i in the schedule of program $\mathcal{P}$ on the SCAD machine $\mathcal{S}$ corresponds to an assignment of color c_q to vertex v_i in the graph G.

Modeling edges: The m edges of the graph G are modeled by m basic blocks $B := \{B_1, \ldots, B_m\}$. Each basic block $B_i \in B$ corresponds to an edge $e_i := (v_j, v_k)$. Since v_j and v_k must be assigned different colors, we must enforce that the corresponding vertex variables are produced by different PUs. This is achieved by constructing each basic block B_i as follows:

$$B_i : \quad \begin{array}{rcl} v_j & = & l_i \odot l_i' \\ v_k & = & l_i' \odot l_i \end{array}$$

where $\odot$ denotes some binary operation, and l_i and l_i' are some load variables unique to the basic block B_i whose values must be loaded from the main

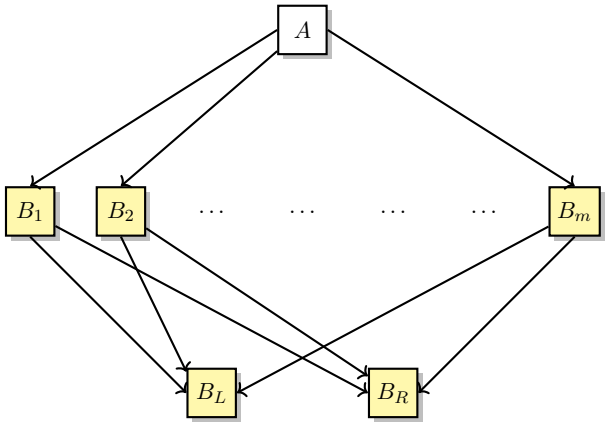


Figure 4.1.: *Control flow graph of the constructed program $\mathcal{P}$*

memory. Since only one memory read is allowed per load variable, l_i and $l_i^{'}$ must only be read once by the load-store unit (LSU) into its output buffer, either in order $l_i \prec l_i^{'}$ or in order $l_i^{'} \prec l_i$. Assume the order $l_i \prec l_i^{'}$ so that the content of the LSU output buffer is $[l_i, l_i, l_i^{'}, l_i^{'}]$ (head of the buffer at left and tail at right). See Figure 4.2. Now assume that vertex variables v_j and v_k are produced by the same PU in that order, i.e., $v_j \prec v_k$. Then, l_i (respectively, $l_i^{'}$) must be moved to the left input buffer (respectively, the right input buffer) of the PU before moving $l_i^{'}$ (respectively, l_i) to the same input buffer. The copy of l_i at the head of the LSU output buffer can be moved to the left input buffer of the PU. After this move, the content of the LSU output buffer is $[l_i, l_i^{'}, l_i^{'}]$, with again a copy of l_i at its head. However, we now need to access $l_i^{'}$ to move it to the right input buffer of the PU before moving the copy of l_i to the same buffer. Therefore, we have to move the copy of l_i from the head of the LSU output buffer to some other buffer to access $l_i^{'}$. This will incur an overhead operation. The same is the case for variable orderings $l_i^{'} \prec l_i$ and $v_k \prec v_j$. Since it is required that the schedule of the program $\mathcal{P}$ on the SCAD machine $\mathcal{S}$ does not use any overhead operation, the vertex variables v_j and v_k must be produced by different PUs.

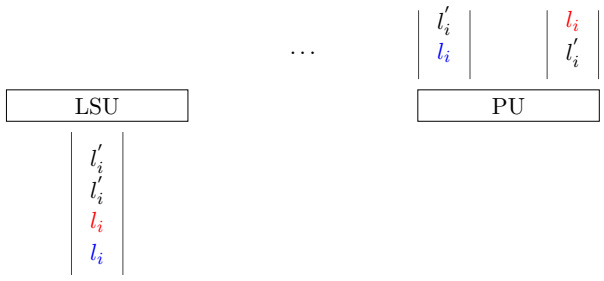


Figure 4.2.: *Failed attempt to schedule basic block B_i without overhead*

Modeling unique color assignment: Note that if multiple edges are incident on the same vertex v_i in the graph G, multiple basic blocks in the set of basic

blocks B in the program $\mathcal{P}$ will write to the corresponding vertex variable v_i. Since any vertex v_i must be assigned one and only one color, we must enforce that any vertex variable v_i is produced by one and only one PU in all basic blocks that write to v_i. The unique assignment of color to vertices in the graph G is modeled by the basic blocks B_L and B_R in the program $\mathcal{P}$. B_L reads all vertex variables as left operands, and B_R reads all vertex variables as right operands:

$$B_L: \begin{array}{c} \_ = v_1 \odot \_ \\ \vdots \\ \_ = v_n \odot \_ \end{array} \qquad B_R: \begin{array}{c} \_ = \_ \odot v_1 \\ \vdots \\ \_ = \_ \odot v_n \end{array}$$

where $\_$ denotes some immediate value when appearing as operand of an instruction and some variable (different from vertex variables) when appearing as the target of an instruction.

Multiple basic blocks in B may write to any vertex variable v_i that is then read in B_L and B_R. Since basic blocks B branch to B_L (or B_R), any v_i must reside in a *statically determined buffer* at the exit of basic blocks B so that it can be accessed in B_L (or B_R) from this buffer. Note that v_i must be moved to the left input buffer of some PU in the basic block B_L and to the right input buffer of some PU in B_R. This enforces that the variable v_i must reside in a statically determined *output buffer* (say *out*) at the exit of basic blocks B. If v_i resides in a left (respectively right) input buffer, then it must be rotated to the right (respectively left) input buffer of the PU executing instruction $\_ = \_ \odot v_i$ (respectively $\_ = v_i \odot \_$) in basic block B_R (respectively B_L). This will incur an overhead operation contradicting our requirement of an overhead-free schedule of program $\mathcal{P}$ on SCAD machine $\mathcal{S}$. Finally, in any basic block in B, v_i must be produced by the *same PU* whose output buffer is *out*. If v_i is produced by a different PU in any basic block $B_j \in B$, it must be moved to the output buffer *out* incurring an overhead operation. Putting together the above arguments, the construction of basic blocks B_L and B_R enforces that in the overhead-free schedule of program $\mathcal{P}$ on SCAD machine $\mathcal{S}$, any vertex variable v_i is produced by one and only one PU, thus modeling a unique assignment of colors to vertices in the graph G.

Solution: Clearly, every optimal (overhead-free) schedule of program $\mathcal{P}$ on the SCAD machine $\mathcal{S}$ corresponds to a valid coloring of the graph G in that $Col(v_i) = c_q$ for any vertex v_i where PU q is the unique PU that produces the vertex variable v_i in the optimal schedule. Similarly, any valid coloring of the graph G corresponds to an optimal schedule of program $\mathcal{P}$ on the SCAD machine $\mathcal{S}$ in that any vertex variable v_i is produced by PU q where $Col(v_i) = c_q$. The resulting schedule will be optimal: A valid coloring will assign different colors to all adjacent vertices in the graph G. This means that in the resulting schedule of $\mathcal{P}$ on $\mathcal{S}$, in any basic block $B_i \in B$ corresponding to edge $e_i := (v_j, v_k)$, the vertex variables v_j and v_k will be produced by different PUs. This allows all basic blocks B in program $\mathcal{P}$ to be compiled without any overhead. Furthermore, since any vertex is assigned a unique color in the graph G, any vertex variable is produced by a unique PU in program $\mathcal{P}$. This means that

the basic blocks B_L and B_R can also be compiled without any overhead since vertex variables can be moved directly from the output buffers of the respective assigned PUs to the appropriate input buffers in B_L and B_R. ■

4.2. Mapping to SAT

To formulate the problem to generate code without overhead in propositional logic, we assume that the following is given:

- a basic block (DAG) in the form of three-address code in static single assignment (SSA) form [AlWZ88; CFRW91; RoWZ88], i.e.,

$$\begin{array}{lcl} x_{\mathsf{tgt}(0)} & = & x_{\mathsf{srcL}(0)} \quad \odot_0 \quad x_{\mathsf{srcR}(0)} \\ & \vdots & \\ x_{\mathsf{tgt}(\ell-1)} & = & x_{\mathsf{srcL}(\ell-1)} \quad \odot_{\ell-1} \quad x_{\mathsf{srcR}(\ell-1)} \end{array}$$

  for some variables $\mathcal{V} \coloneqq \{x_0, \ldots, x_{n-1}\}$, where $\odot_i$ denotes some binary operation.

- a SCAD machine with one load-store unit and p processing units that may execute any binary operation.

The problem is to determine if the basic block can be executed on the SCAD machine without any dup and/or swap overhead. If so, determine the schedule of the basic block on the PUs of the SCAD machine.
In SSA form, every variable x_i occurs at most once as the left-hand side in the three-address code, but it may occur several times on the right-hand side. This defines three different kinds of variables:

- target variables $\mathcal{V}_{\mathsf{tgt}}$ are those that occur on the left-hand sides.
- source variables $\mathcal{V}_{\mathsf{src}}$ are those that occur on the right-hand sides.
- load variables $\mathcal{V}_{\mathsf{ld}}$ are those that only occur on the right-hand sides $\mathcal{V}_{\mathsf{ld}} \coloneqq \mathcal{V}_{\mathsf{src}} \setminus \mathcal{V}_{\mathsf{tgt}}$

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

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

- level 0 are all instructions that only read variables $\mathcal{V}_{\mathsf{ld}}$
  $\rightsquigarrow$ their target variables $\mathcal{V}^0_{\mathsf{def}}$ are written by this level
- level $j+1$ are all instructions that only read variables $\mathcal{V}_{\mathsf{ld}} \cup \bigcup_{i=0}^{j} \mathcal{V}^j_{\mathsf{def}}$ and where at least one variable of $\mathcal{V}^j_{\mathsf{def}}$ is read

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

A *schedule* of the basic block on the SCAD machine comprises an assignment of variables to units (PU/LSU) of the SCAD machine and an ordering of the variables on these units. The assignment determines which instructions of the basic black are executed by each unit, and the ordering determines the order in which each unit executes these instructions. The schedule is an *optimal schedule* if move code can be generated with the given assignment and ordering, to execute the basic block on the SCAD machine.

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

Assuming that PU 0 is the load-store unit in the given SCAD machine with p processing units $\{1, \ldots, p\}$, we define the following relation to determine the assignment of variables to PUs in the SCAD machine:

- ***PU assignment relation*** $\alpha_{i,j}$ for $x_i \in \mathcal{V}$ and $j \in \{0, \ldots, p\}$
  - $\alpha_{i,j}$ means that $x_i \in \mathcal{V}$ is produced by PU j
  - this determines the instructions of the basic block executed by PU j

Every variable has to be produced by one and only one PU, as expressed in the ***unique PU assignment constraint*** $\mathcal{C}_1$ in 4.1. The first part of the constraint asserts that any variable x_i is assigned to at least one PU and the second part asserts that if x_i is assigned to any PU k, then it is not assigned to any other PU $j \neq k$.

$$\mathcal{C}_1 :\Leftrightarrow \left(\bigwedge_{x_i \in \mathcal{V}} \bigvee_{k=0}^{p} \alpha_{i,k} \right) \wedge \left(\bigwedge_{x_i \in \mathcal{V}} \bigwedge_{k=0}^{p} \alpha_{i,k} \Rightarrow \bigwedge_{j=0, j \neq k}^{p} \neg \alpha_{i,j} \right) \tag{4.1}$$

Furthermore, as we determine that all $x_i \in \mathcal{V}_{\mathsf{ld}}$ are assigned to PU 0,

- we replace for all $x_i \in \mathcal{V}_{\mathsf{ld}}$ all $\alpha_{i,0}$ with true
- we replace for all $x_i \notin \mathcal{V}_{\mathsf{ld}}$ all $\alpha_{i,0}$ with false
- we replace for all $x_i \in \mathcal{V}_{\mathsf{ld}}$ and $j > 0$ all $\alpha_{i,j}$ with false

$\rightsquigarrow$ only $\alpha_{i,j}$ remain where $x_i \notin \mathcal{V}_{\mathsf{ld}}$ and $j > 0$

4.2.1. Buffer constraints

To determine an ordering of assigned variables on respective PUs (i.e., an order of instructions executed by each PU), we introduce a *strict partial order relation* $\prec$ called the variable order such that:

- ***variable order relation*** $x_i \prec x_j$ for $x_i, x_j \in \mathcal{V}$

– restrictions of $\prec$ to an output buffer establishes a total order among variables produced in that output buffer.

The constraints of the ordering of variables in output and input buffers are then formulated using the variable order relation. To better comprehend the formulation of buffer constraints, we first describe a preliminary approach referred to as the production order approach before describing the final buffer constraints in the consumption order approach. The exact meaning of the variable order $\prec$ differs subtly in both approaches. Irrespective of that, the following constraints are imposed on the variable order relation.

First, we demand that the variable order $\prec$ is a strict partial order. Therefore, $\prec$ is both ***transitive*** (***constraint*** $\mathcal{C}_2$ in 4.2) and ***irreflexive*** (***constraint*** $\mathcal{C}_3$ in 4.3). Both together imply that $\prec$ is acyclic as expressed in 4.4.

$$\mathcal{C}_2 :\Leftrightarrow \bigwedge_{x_i,x_j,x_k \in \mathcal{V}} x_i \prec x_j \wedge x_j \prec x_k \Rightarrow x_i \prec x_k \tag{4.2}$$

$$\mathcal{C}_3 :\Leftrightarrow \bigwedge_{x_i \in \mathcal{V}} \neg x_i \prec x_i \tag{4.3}$$

$$\mathcal{C}_2 \wedge \mathcal{C}_3 \Rightarrow \bigwedge_{x_i,x_j \in \mathcal{V}} x_i \prec x_j \Rightarrow \neg x_j \prec x_i \tag{4.4}$$

Second, we demand that the total order among variables produced in the same output buffer respect data dependencies in the basic block. To express that two variables x_i and x_j stem from the same output buffer (i.e., are produced by the same PU), we introduce the following notation:

$$\beta_{i,j} :\Leftrightarrow \bigvee_{k=0}^{p} \alpha_{i,k} \wedge \alpha_{j,k} \tag{4.5}$$

The data dependency constraint $\mathcal{C}_4$ is now formulated as shown in 4.6, where $x_i \prec_d x_j$ means that x_j is data dependent (read-after-write dependent) on x_i. If $x_{\mathsf{tgt}(i)}$ and $x_{\mathsf{tgt}(j)}$ are produced by the same PU in that order (i.e., $x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)}$, then $x_{\mathsf{tgt}(i)}$ must not be data dependent on $x_{\mathsf{tgt}(j)}$. Similarly, for the order $x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)}$.

$$\mathcal{C}_4 :\Leftrightarrow \bigwedge_{i,j=0}^{\ell-1} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \Rightarrow \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \Rightarrow \neg x_{\mathsf{tgt}(j)} \prec_d x_{\mathsf{tgt}(i)} \\ \wedge \\ x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \Rightarrow \neg x_{\mathsf{tgt}(i)} \prec_d x_{\mathsf{tgt}(j)} \end{array} \right) \tag{4.6}$$

The data dependency relation $\prec_d$ may be either computed offline, or computed by the SAT solver by formulating the relation given by 4.7 as a strict partial order relation with transitive and irreflexive constraints in 4.8 and 4.9, respectively.

$$\bigwedge_{i=0}^{\ell-1} x_{\mathsf{tgt}(i)} \prec_d x_{\mathsf{srcL}(i)} \wedge x_{\mathsf{tgt}(i)} \prec_d x_{\mathsf{srcR}(i)} \tag{4.7}$$

$$\bigwedge_{x_i,x_j,x_k \in \mathcal{V}} x_i \prec_d x_j \wedge x_j \prec_d x_k \Rightarrow x_i \prec_d x_k \tag{4.8}$$

$$\bigwedge_{x_i \in \mathcal{V}} \neg x_i \prec_d x_i \tag{4.9}$$

Production order approach

In this preliminary approach, the variable order $\prec$ is the *production order*. That is, $x_i \prec x_j$ means that the same PU produces x_i and x_j in that order. Alternatively, $\prec$ is the *consumption order enforced by the output buffers*. In other words, all copies of x_i must be consumed before consuming any copy of x_j. Clearly, a total variable ordering must exist for those variables that are at some time in the same output buffer.

Consider now any pair of instructions $x_{\mathsf{tgt}(i)} = x_{\mathsf{srcL}(i)} \odot_i x_{\mathsf{srcR}(i)}$ and $x_{\mathsf{tgt}(j)} = x_{\mathsf{srcL}(j)} \odot_j x_{\mathsf{srcR}(j)}$ of the basic block. If the instructions are executed on different PUs, it is possible to move their operands to the corresponding input buffers irrespective of ordering of operands in some output buffers. Assume that the instructions are executed on the same PU k. Then either $x_{\mathsf{tgt}(i)}$ must precede $x_{\mathsf{tgt}(j)}$ in the production order (i.e., $x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)}$) or vice versa. If $x_{\mathsf{tgt}(i)}$ (respectively $x_{\mathsf{tgt}(j)}$) is produced before $x_{\mathsf{tgt}(j)}$ (respectively $x_{\mathsf{tgt}(i)}$), then we should be able to move the operand $x_{\mathsf{srcL}(i)}$ (respectively $x_{\mathsf{srcL}(j)}$) before the operand $x_{\mathsf{srcL}(j)}$ (respectively $x_{\mathsf{srcL}(i)}$), to the left input buffer of PU k. Therefore, if both left operands $x_{\mathsf{srcL}(i)}$ and $x_{\mathsf{srcL}(j)}$ are produced by the same PU, we must enforce the ordering $x_{\mathsf{srcL}(i)} \preceq x_{\mathsf{srcL}(j)}$ (respectively $x_{\mathsf{srcL}(j)} \preceq x_{\mathsf{srcL}(i)}$). Otherwise, one will need to use an overhead instruction to access the appropriate operand value. Similar arguments apply for the right operands. This is expressed in constraint 4.10. Clearly, if $x_{\mathsf{tgt}(i)}$ precedes $x_{\mathsf{tgt}(j)}$ in the production order, then $x_{\mathsf{tgt}(j)}$ must not precede $x_{\mathsf{tgt}(i)}$ in the production order. Recall that cycles in the production order $\prec$ are avoided by the transitivity (constraint $\mathcal{C}_2$) and the irreflexivity (constraint $\mathcal{C}_3$).

$$\bigwedge_{i,j=0}^{\ell-1} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \quad \Rightarrow \quad \left(\begin{array}{c} \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \\ \wedge \\ \beta_{\mathsf{srcL}(i),\mathsf{srcL}(j)} \Rightarrow x_{\mathsf{srcL}(i)} \preceq x_{\mathsf{srcL}(j)} \\ \wedge \\ \beta_{\mathsf{srcR}(i),\mathsf{srcR}(j)} \Rightarrow x_{\mathsf{srcR}(i)} \preceq x_{\mathsf{srcR}(j)} \end{array} \right) \\ \vee \\ \left(\begin{array}{c} x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \\ \wedge \\ \beta_{\mathsf{srcL}(i),\mathsf{srcL}(j)} \Rightarrow x_{\mathsf{srcL}(j)} \preceq x_{\mathsf{srcL}(i)} \\ \wedge \\ \beta_{\mathsf{srcR}(i),\mathsf{srcR}(j)} \Rightarrow x_{\mathsf{srcR}(j)} \preceq x_{\mathsf{srcR}(i)} \end{array} \right) \end{array} \right) \tag{4.10}$$

Clearly, the buffer constraint 4.10 is *necessary*. That is, all optimal schedules must satisfy this constraint. To visualize variable orderings easily, we use drawings of input buffers displaying relative positions of variables that pass

through each input buffer at some time. See for example Figures 4.3, 4.4 and 4.5. Any directed line from x_i in input buffer k to x_j in input buffer l indicates that a copy of x_i must be moved to input buffer k before moving a copy of x_j to the input buffer l. We now distinguish two types of directed lines. The directed solid (black) line from x_i to x_j means that variables x_i and x_j are produced in that order at some time in some output buffer. So, all copies of x_i must be moved before moving any copy of x_j. In further explanations, we call these *output edges*. The meaning of output edges is already encapsulated by the variable ordering relation $\prec$, so that we may denote an output edge by $x_i \prec x_j$. Note that there exists an output edge from each copy of x_i to each copy of x_j in the input buffers. The directed dashed (blue) line from x_i to x_j means that a copy of variables x_i and x_j appears in that order at some time in that particular input buffer. So, some copy of x_i must be moved to that input buffer before moving some copy of x_j to the same input buffer. Note that it is always a unidirectional (vertical) line since it is specific to variable copies in each input buffer. We call these *input edges* denoted by $x_i \rightsquigarrow x_j$.

Now consider the input buffers of PU k shown in Figure 4.3. The variable ordering satisfies constraint 4.10 since the left operands are produced in different output buffers and the right operands are also produced in different output buffers. However, still optimal move code cannot be generated. Before moving $x_{\mathsf{srcL}(i)}$ to the left input buffer of PU k, $x_{\mathsf{srcR}(j)}$ must be moved. Before moving $x_{\mathsf{srcR}(j)}$ to the right input buffer of PU k, $x_{\mathsf{srcR}(i)}$ must be moved to the same input buffer. However, $x_{\mathsf{srcR}(i)}$ can be moved only after moving $x_{\mathsf{srcL}(j)}$. Finally, $x_{\mathsf{srcL}(j)}$ can be moved to the left input buffer of PU k only after moving $x_{\mathsf{srcL}(i)}$ to the same input buffer. Thus, optimal move code generation is blocked by the cycle $(x_{\mathsf{srcL}(i)} \rightsquigarrow x_{\mathsf{srcL}(j)} \prec x_{\mathsf{srcR}(i)} \rightsquigarrow x_{\mathsf{srcR}(j)} \prec x_{\mathsf{srcL}(i)})$. This proves that the buffer constraint 4.10 is not *sufficient*. That is, there exists non-optimal schedules that satisfy the constraint. It is not difficult to see that the buffer constraint 4.10 only filters out those non-optimal schedules that lead to cycles limited to a single input buffer as shown in Figure 4.4.

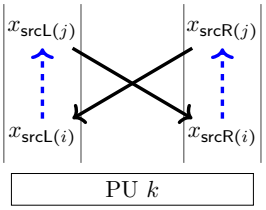


Figure 4.3.: *Ordering of variables forming a cycle spanning both left and right input buffers of PU k*

We observe that cycles in Figures 4.4 and 4.3 are special cases of a general cycle spanning all input buffers shown in Figure 4.5. Clearly, input edges $\rightsquigarrow$ alone cannot form a blocking cycle. Also output edges $\prec$ alone cannot form a cycle since we already establish that $\prec$ is acyclic by imposing transitivity (constraint 4.2) and irreflexivity (constraint 4.3). It is not difficult to further see that any cycle may be decomposed into a corresponding cycle comprising of alternating input and output edges.

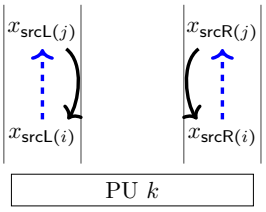


Figure 4.4.: *Ordering of variables forming a cycle in the left input buffer and a cycle in the right input buffer of PU k*

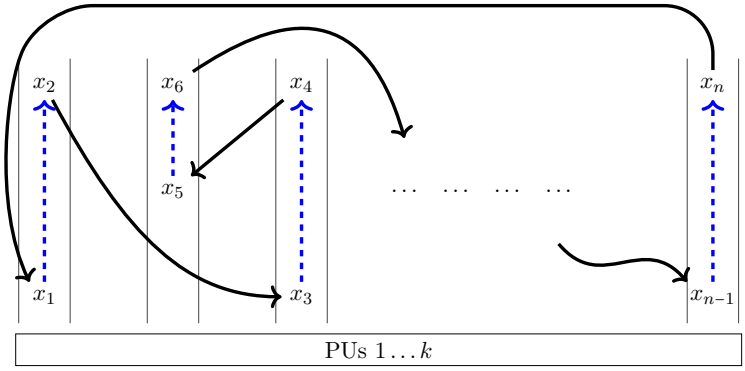


Figure 4.5.: *Ordering of variables forming a cycle spanning all input buffers*

An input edge $x_i \rightsquigarrow x_j$ means that some copy of x_i must be moved to some input buffer before moving some copy of x_j to the same input buffer. An output edge $x_j \prec x_k$ means that all copies of x_j must be moved before moving any copy of x_k. Concatenating both, $x_i \rightsquigarrow x_j \prec x_k$ means that some copy of x_i must be moved before moving any copy of x_k even if x_i and x_k are produced in different output buffers. This establishes *an order of consumption enforced by the input buffers* besides the consumption order enforced by the output buffers (i.e., the production order). Notice that similar to the consumption order enforced by the output buffers, the consumption order enforced by the input buffers must also be transitive. That is, if some copy of x_i must be consumed before consuming any copy of x_j and if some copy of x_j must be consumed before consuming any copy of x_k, then some copy of x_i must be consumed before consuming any copy of x_k. Also, the consumption order enforced by the input buffers must be acyclic in an optimal schedule (i.e., there must not exist any blocking cycles). Otherwise, if it is the case that some copy of x_i must be consumed before consuming any copy of x_j and some copy of x_j must be consumed before consuming any copy of x_i, the optimal move code generation will fail.

Consumption order approach

In this approach, the variable order relation $\prec$ is augmented to represent both the consumption order enforced by the output buffers (or the production order) and the consumption order enforced by the input buffers. In other words, if

variables x_i and x_j are produced by the same PU, $x_i \prec x_j$ means that all copies of x_i must be consumed before consuming any copy of x_j. If they are produced by different PUs, $x_i \prec x_j$ means that some copy of x_i must be consumed before consuming any copy of x_j.

Assuming that the variables x_i and x_j appear in any input buffer in that order (i.e., there exists an input edge $x_i \rightsquigarrow x_j$), we introduce the following notation to express that variable x_i precedes variable x_k in consumption order enforced by the input buffer, when x_j and x_k are produced by the same PU in that order (i.e., there exists an output edge $x_j \prec x_k$).

$$\gamma_{i,j,k} :\Leftrightarrow \beta_{j,k} \wedge x_j \prec x_k \Rightarrow x_i \prec x_k \tag{4.11}$$

Now, the final ***buffer constraint*** $\mathcal{C}_5$ that encapsulates both consumption orders is formulated as follows:

$$\mathcal{C}_5 :\Leftrightarrow \bigwedge_{i,j=0}^{\ell-1} \bigwedge_{x_k \in \mathcal{V}} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \\ \wedge \\ \gamma_{\mathsf{srcL}(i),\mathsf{srcL}(j),k} \wedge \gamma_{\mathsf{srcR}(i),\mathsf{srcR}(j),k} \end{array} \right) \\ \vee \\ \left(\begin{array}{c} x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \\ \wedge \\ \gamma_{\mathsf{srcL}(j),\mathsf{srcL}(i),k} \wedge \gamma_{\mathsf{srcR}(j),\mathsf{srcR}(i),k} \end{array} \right) \end{array} \right) \tag{4.12}$$

Finally, a conjunction of the relevant boolean constraints gives the ***final constraint formula*** $\mathcal{C}_s$ for *optimal scheduling of the given basic block on the given SCAD machine, where optimal refers to the absence of dup or swap instructions.*

$$\mathcal{C}_s :\Leftrightarrow \bigwedge_{i=1}^{5} \mathcal{C}_i \tag{4.13}$$

Necessity and Sufficiency

Since the unique PU assignment and the data dependency constraints are straightforward, we focus on the buffer (ordering) constraint. We have seen that if any schedule of a basic block on a SCAD machine results in cycles in the consumption order (i.e., do not satisfy the buffer constraint), optimal move code generation will fail. So the following lemma is stated:

Lemma 4.1 (Necessary) *All optimal schedules of a basic block on a SCAD machine must satisfy constraint $\mathcal{C}_s$. In other words, $\mathcal{C}_s$ is a necessary constraint for optimal basic block scheduling on SCAD.*

In the following, we prove the sufficiency of constraint $\mathcal{C}_s$.

Lemma 4.2 (Sufficient) *Any non-optimal schedule of a basic block on a SCAD machine must not satisfy constraint $\mathcal{C}_s$. In other words, $\mathcal{C}_s$ is a sufficient constraint for optimal basic block scheduling on SCAD.*

Proof Assume that $\prec$ is the variable ordering in a non-optimal schedule that satisfies the constraint $\mathcal{C}_s$. Since the schedule is non-optimal, it is not possible to generate move code with the ordering $\prec$ without any overhead. Therefore, all possible ways of moving values from output buffers to input buffers must reach a deadlock in that no more values can be moved from the head of any output buffer to the tail of any input buffer, thus necessitating the use of overhead instructions. We now prove that if such a deadlock is reached, the variable order relation $\prec$ must contain at least one cycle, thus not satisfying the constraint $\mathcal{C}_s$ contradicting our initial assumption. At deadlock, assume that $\{x_1, \ldots, x_m\}$ are the next operands expected by input buffers $1 \ldots m$, respectively, where $\forall_i x_i \in \mathcal{V}$. Since no more moves are possible, none of $\forall_i x_i$ is at the head of any output buffer. Then, there exists $\{y_1, \ldots, y_n\}$ where $y_i \neq x_i$ is an operand for input buffer i (whose next operand is x_i) and $\forall_i y_i$ is at the head of some output buffer. Furthermore, $n < m$ holds since more than one next operands in $\{x_1, \ldots, x_m\}$ may be behind any y_i in some output buffer. This results in following the input edges: $\{x_1 \rightsquigarrow y_1, \ldots, x_n \rightsquigarrow y_n\}$.

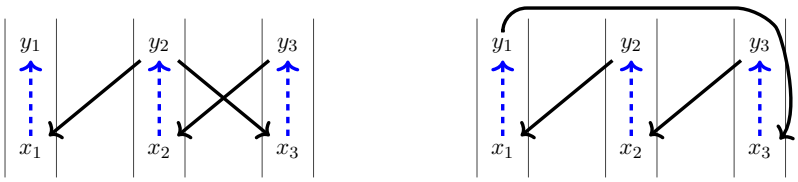
Since $\forall_i x_i$ is not at the head of any output buffer, each x_i must succeed one of $\{y_1, \ldots, y_n\}$ in some output buffer. Consider any x_i. If x_i succeeds y_i, then $y_i \prec x_i$ holds yielding an output edge as shown in Figure 4.6a. The input and output edges now form a cycle in input buffer i, thus not satisfying constraint $\mathcal{C}_s$ contradicting our assumption that the ordering $\prec$ satisfies the constraint $\mathcal{C}_s$. Therefore, any x_i must succeed $y_j \mid j \neq i$.

(a) *Cycle in buffer i* **(b)** *Cycle spanning buffers 1 and 2*

Figure 4.6.: *Ordering of variables in input buffers*

Consider x_1. Without loss of generality, assume that x_1 succeeds y_2 in some output buffer. Now consider x_2. If x_2 succeeds y_1, then $y_1 \prec x_2$ holds yielding a cycle in input buffers 1 and 2 as shown in Figure 4.6b. Thus, it does not satisfy the constraint $\mathcal{C}_s$, which contradicts our assumption. Therefore, x_2 must succeed $y_j \mid j > 2$. Again, without loss of generality, assume that x_2 succeeds y_3 in some output buffer. Now, consider x_3. If x_3 succeeds y_2 or y_1, this results in cycles shown in Figures 4.7a and 4.7b, respectively. Again,

not satisfying constraint $\mathcal{C}_s$ thus contradicting our assumption. Therefore, x_3 must succeed $y_j \mid j > 3$. Continuing the argument, consider x_n. Since x_n is not at the head of any output buffer, it must succeed one of $\{y_1, \ldots, y_n\}$ in some output buffer. However, if x_n succeeds any $y_j \mid j \leq n$, this results in a cycle spanning input buffers $j, \ldots, n$ as shown in Figure 4.8. Thus, inevitably the buffer constraint $\mathcal{C}_s$ is not satisfied, contradicting our initial assumption that the variable ordering $\prec$ satisfies constraint $\mathcal{C}_s$. It is therefore the case that any non-optimal schedule must not satisfy the buffer constraint $\mathcal{C}_s$.



(a) *Cycle spanning buffers* 2 *and* 3 **(b)** *Cycle spanning buffers* 1, 2 *and* 3

Figure 4.7.: *Ordering of variables in input buffers* 1, 2 *and* 3 *(from left to right)*

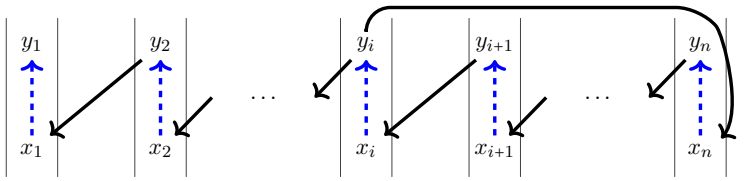


Figure 4.8.: *Ordering of variables in input buffers* $1, \ldots, n$ *(from left to right) forming a cycle spanning buffers* $i, \ldots, n$

■

Theorem 4.2 (Necessary and Sufficient) *Constraint* $\mathcal{C}_s$ *is both necessary and sufficient for optimal basic block scheduling on SCAD.*

Proof This follows directly from Lemmas 4.1 and 4.2. ■

4.2.2. Move code generation

After proving the sufficiency of constraint $\mathcal{C}_s$, it is easy to *generate a move program* for the given basic block for execution on the given SCAD machine with a variable ordering $\prec$ and a PU assignment α that satisfies the constraint $\mathcal{C}_s$. First load variables (or leaf nodes of the DAG representing the given basic block) are produced in the output buffer of PU 0, i.e., the LSU. For this, simply extract the variables assigned to LSU, sort them according to $\prec$ and generate

moves to load these variables in the sorted order to the output buffer of the LSU. Next, extract variables assigned to PUs $\{1, \ldots, p\}$. This determines the entire sequence of operand values that must pass through each input buffer i and the entire sequence of result values that must pass through each output buffer j. Let tail$[i]$ denote the next operand value to move to input buffer i and head$[j]$ denote the current result value at the head of the output buffer j. Now, the move code for the computation of the basic block can be generated as follows:

- determine input and output buffer pairs (i, j) such that head$[j]$ and tail$[i]$ are the same. The Lemma 4.2 guarantees that such a pair of input and output buffers will exist as long as there are still values to move from output to input buffers.
- generate moves to transport the value from the output buffer j to the input buffer i for each determined pair (i, j) and update the respective tail$[i]$ and head$[j]$ values.
- generate moves to store variables that appear as root nodes of the DAG, if found at any head$[j]$.
- repeat the above steps until all result values are either moved to respective input buffers or stored in the main memory.

4.2.3. Optimizing execution time

So far, we have considered resource-constrained optimal scheduling since our effort was to derive an optimal schedule of programs on SCAD machines with a given number of PUs so that any reduction in performance due to overhead operations is avoided. Consider the simple program $x_2 = x_0 + x_1; x_3 = x_0 * x1$. Clearly, for any order of loading of x_0 and x_1 by the LSU, these operand values can be moved so that x_2 and x_3 can be produced by the same PU. Even when a SCAD machine with 2 PUs is provided, a satisfying assignment of the SAT constraints developed so far may only use 1 PU to execute the above program. However, an optimal schedule that minimizes the execution time will assign x_2 and x_3 to different PUs for concurrent execution. Therefore, the next step is to optimize the execution time for the maximal use of ILP contained in programs or equivalently to minimize the execution time of programs. For this purpose, we introduce the desired execution time t as an additional input parameter that modifies our problem statement as follows: *The problem is to determine if the basic block can be executed on the SCAD machine in time t without any dup and/or swap overhead. If so, determine the schedule of the basic block for the SCAD machine.*

To include further constraints regarding the execution time, we consider the assignment of variables to time slots, which is defined as follows:

- ***time slot assignment relation*** $\theta_{i,j}$ for $x_i \in \mathcal{V}$ and $j \in \{0, \ldots, t-1\}$
  - $\theta_{i,j}$ means that $x_i \in \mathcal{V}$ is produced in time slot j

Every variable has to be produced in one and only one time slot, as expressed in the ***unique time slot assignment constraint*** $\mathcal{C}_6$ in 4.14. Similar to the unique PU assignment constraint $\mathcal{C}_1$, the first part of $\mathcal{C}_6$ asserts that any variable x_i is assigned to at least one time slot, and the second part enforces that if x_i is assigned to any time slot k, then it is not assigned to any other time slots $j \neq k$.

$$\mathcal{C}_6 :\Leftrightarrow \left(\bigwedge_{x_i \epsilon \mathcal{V}} \bigvee_{k=0}^{t-1} \theta_{i,k} \right) \wedge \left(\bigwedge_{x_i \epsilon \mathcal{V}} \bigwedge_{k=0}^{t-1} \theta_{i,k} \Rightarrow \bigwedge_{j=0, j \neq k}^{t-1} \neg \theta_{i,j} \right) \tag{4.14}$$

The assignment of variables to time slots must respect both the data dependency of the variables and the production order of the variables on the same PU. To formulate these, the following notation is used to express that a variable x_i is assigned to an earlier time slot than variable x_j (i.e., x_i is produced before x_j):

$$\tau_i < \tau_j :\Leftrightarrow \bigvee_{k=0}^{t-1} \theta_{i,k} \wedge \left(\bigvee_{m=k+1}^{t-1} \theta_{j,m} \right) \tag{4.15}$$

$\tau_i < \tau_j$ holds if x_i is assigned to any time slot k and x_j is assigned to a time slot $m > k$.

The ***'time slot respect data dependency' constraint*** $\mathcal{C}_7$ in 4.16 restricts time slot assignments in that the operand variables must be assigned to earlier time slots than respective target variables.

$$\mathcal{C}_7 :\Leftrightarrow \bigwedge_{i=0}^{\ell-1} \tau_{\mathsf{srcL}(i)} < \tau_{\mathsf{tgt}(i)} \wedge \tau_{\mathsf{srcR}(i)} < \tau_{\mathsf{tgt}(i)} \tag{4.16}$$

The ***'time slot respect ordering' constraint*** $\mathcal{C}_8$ 4.17 restricts time slot assignments in that the time slots assigned to variables produced by the same PU must agree with the ordering of these variables.

$$\mathcal{C}_8 :\Leftrightarrow \bigwedge_{i=0}^{\ell-1} \bigwedge_{j=0}^{\ell-1} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \Rightarrow \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \Rightarrow \tau_{\mathsf{tgt}(i)} \prec \tau_{\mathsf{tgt}(j)} \\ \wedge \\ x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \Rightarrow \tau_{\mathsf{tgt}(j)} \prec \tau_{\mathsf{tgt}(i)} \end{array} \right) \tag{4.17}$$

The above constraints are added to $\mathcal{C}_s$ (in 4.13) to obtain a new constraint formula for *optimal scheduling of the given basic block on the given SCAD machine where optimal refers to both, absence of any overhead and guarantee an upper bound 't' of execution time.*

$$\mathcal{C}_s :\Leftrightarrow \bigwedge_{i=1}^{5} \mathcal{C}_i \wedge \bigwedge_{i=6}^{8} \mathcal{C}_i \tag{4.18}$$

Finally, it is worth mentioning that instead of introducing a boolean time slot assignment variable $\theta_{i,j}$ for any x_i and then deriving τ_i to build constraints, it is as well possible to encode τ_i directly as an integer variable. However, the resulting boolean and linear integer constraints will necessitate the use of satisfiability modulo theories (SMT) solvers [BhSc17].

4.2.4. Adapting constraints for DO-SCAD

As we have seen, the assignment of instructions to PUs is restricted by the variable ordering. On the one hand, the variable ordering constraints avoid the use of overhead, which will otherwise adversely affect the performance. On the other hand, these constraints reduce the flexibility of the SCAD compiler to perform the instruction assignment for a maximal use of ILP. For example, consider the following program:

$$\begin{aligned} x_3 &= x_0 \odot_0 x_1 \\ x_4 &= x_2 \odot_1 x_3 \\ x_5 &= x_1 \odot_2 x_4 \end{aligned} \tag{4.19}$$

The program can be scheduled without overhead on a SCAD machine with only one PU and one LSU. Clearly, variables $\{x_0, x_1, x_2\}$ are load variables assigned to LSU, while variables $\{x_3, x_4, x_5\}$ are assigned to the PU. The resulting optimal assignment of variables to time slots (to minimize execution time) in given in Table 4.1b. Note that the SCAD machine takes 6 cycles to execute the program. However, an ideal schedule that maximizes the use of ILP, shown in Table 4.1a, will only need 5 cycles to execute the same program.

clk	instructions
1	x_0
2	x_1
3	x_2 x_3
4	x_4
5	x_5

(a) *by ideal (DO-SCAD) scheduler*

clk	instructions
1	x_0
2	x_2
3	x_1
4	x_3
5	x_4
6	x_5

(b) *by SCAD scheduler*

Table 4.1.: *Minimal time slot assignments for Program 4.19*

The variables x_2 and x_3 are assigned to the same time slot 3 in the ideal schedule. Note that x_1 is the right operand of target variable x_3 ($x_3 = x_0 \odot_0 x_1$). Therefore, x_1 should be produced before x_3 and thus also before variable x_2 in the ideal schedule where both x_2 and x_3 are assigned to the same time slot (see Table 4.1a). However, this is not possible due to the following reason: x_4 and x_5 are mapped to the single available PU. Since $x_5 = x_1 \odot_2 x_4$ (x_4 is the right operand of the target variable x_5), variable x_4 must be produced before producing variable x_5, i.e., $x_4 \prec x_5$ (see Table 4.1b). The left operands of x_4 and x_5 are load variables x_2 and x_1, respectively, which are produced in the output buffer of the LSU. Since $x_4 \prec x_5$ holds and both are mapped to the single PU, x_2 must be moved to the left input buffer of the PU before moving x_1 to the same buffer. This requires that x_2 is loaded from the main memory by the LSU before loading x_1, i.e., x_2 should be produced in the output buffer of the LSU before producing x_1. Otherwise, overhead operations will be required to rotate copies of x_1 to access the copy of x_2. Therefore, the ordering constraint

$x_2 \prec x_1$ in the LSU output buffer forbids the assignment of x_2 and x_3 in the same time slot in the SCAD schedule.

Unlike buffers in SCAD, there is a pool of entries at inputs and outputs of PUs in the dynamically ordered SCAD (DO-SCAD) equipped with tag-matching hardware. This avoids any need for a compiler determined ordering in DO-SCAD. Thus providing full flexibility to the DO-SCAD compiler to assign instructions to PUs to utilize maximal ILP. Due to the absence of any ordering, no overhead operations are required. Therefore, the *optimal schedule in DO-SCAD simply refers to an assignment of instructions to PUs that maximizes the use of ILP or minimizes the execution time.* Consequently, the conjunction of the unique PU assignment constraint $\mathcal{C}_1$, the unique time slot assignment constraint $\mathcal{C}_6$ and the 'time slot respect data dependency' constraint $\mathcal{C}_7$ is enough to derive the final constraint formula $\mathcal{C}_{do}$ for optimal scheduling for DO-SCAD:

$$\mathcal{C}_{do} :\Leftrightarrow \mathcal{C}_1 \wedge \mathcal{C}_6 \wedge \mathcal{C}_7 \tag{4.20}$$

The ideal schedule in Table 4.1a for program 4.19 is therefore possible in DO-SCAD since there are no ordering constraints. In the experimental results provided in Section 4.3, we show that with only a few PUs, a SCAD compiler can determine an appropriate variable ordering that avoids the use of overhead. Moreover, we also show that compromises in the exploited ILP due to the buffer constraint (or the ordering constraint) in SCAD are an exception and not the expectation.

4.2.5. Adapting constraints for SO-SCAD

There is provision in input buffers in SCAD to enforce a statically determined order of operand values at runtime locally in each PU, irrespective of the order of arrival of these values at respective input buffers. However, in a statically ordered SCAD (SO-SCAD), the PUs rely only on the order of arrival of operand values to determine the next operation to be executed. The order of arrival of values in turn depends on when these values are produced in some output buffer(s). Recall that two ways of enforcing a compiler determined order of instructions in SO-SCAD PUs was discussed in Section 2.3: (1) The control unit must administer the firing of PUs at statically determined instruction issue times to ensure that values are produced and transported in the same order that they must be consumed. (2) The instructions that produce operand values targeted to the same input buffer are assigned by the compiler to the same PU so that these values originate from and are transported from the same output buffer in the expected order. We refer to the former as SO-SCAD$_\mathrm{I}$ and the latter as SO-SCAD$_\mathrm{P}$ machines where subscripts 'I' and 'P' indicates that the ordering is enforced by static instruction issue and static instruction placement, respectively. Consequently, an *optimal schedule in SO-SCAD$_I$ refers to an assignment and ordering of instructions on PUs and statically determined instruction issue (firing of PUs) times, such that the use of ILP is maximized. An optimal schedule in SO-SCAD$_P$ refers to an assignment and ordering of*

instructions on PUs such that the use of ILP is maximized.

To establish how instruction issue times and instruction placement are determined for these machines, let us consider two instructions $x_{\mathsf{tgt}(i)} = x_{\mathsf{srcL}(i)} \odot_i x_{\mathsf{srcR}(i)}$ and $x_{\mathsf{tgt}(j)} = x_{\mathsf{srcL}(j)} \odot_j x_{\mathsf{srcR}(j)}$ of a basic block. In both SCAD and SO-SCAD, if the instructions are executed on different PUs, it is possible to move their operands to the corresponding input buffers irrespective of the ordering of operands in some output buffers. Assume that the instructions are executed on the same PU k. If $x_{\mathsf{tgt}(i)}$ (respectively $x_{\mathsf{tgt}(j)}$) is produced before $x_{\mathsf{tgt}(j)}$ (respectively $x_{\mathsf{tgt}(i)}$), then we should be able to move the operand $x_{\mathsf{srcL}(i)}$ (respectively $x_{\mathsf{srcL}(j)}$) before the operand $x_{\mathsf{srcL}(j)}$ (respectively $x_{\mathsf{srcL}(i)}$), to the left input buffer of PU k. Therefore, if both left operands $x_{\mathsf{srcL}(i)}$ and $x_{\mathsf{srcL}(j)}$ are produced by the same PU, we must enforce the ordering $x_{\mathsf{srcL}(i)} \preceq x_{\mathsf{srcL}(j)}$ (respectively $x_{\mathsf{srcL}(j)} \preceq x_{\mathsf{srcL}(i)}$), in both SCAD and SO-SCAD. Assume now that both left operands are produced by different PUs. As we have seen, the buffer constraint $\mathcal{C}_5$ (4.12) ensures in SCAD that the appropriate transportation of operands $x_{\mathsf{srcL}(i)}$ and $x_{\mathsf{srcL}(j)}$ is not cyclically blocked by any other variables in the basic block. However, the buffer constraint works under the premise that input buffers in PUs in SCAD tolerates any random arrival order of operand values by reordering operand values at runtime if required, and this way respects the compiler determined order of instructions on each PU. On the other hand, input buffers in SO-SCAD are 'pure' FIFO buffers. In SO-SCAD$_{\mathrm{I}}$, the firing times of PUs are statically determined and dynamically enforced by the control unit to ensure that the compiler determined order of instructions is respected. Therefore, in SO-SCAD$_{\mathrm{I}}$, if both left operands $x_{\mathsf{srcL}(i)}$ and $x_{\mathsf{srcL}(j)}$ are produced by different PUs, the control unit must trigger the firing of the PU that will produce $x_{\mathsf{srcL}(i)}$ (respectively $x_{\mathsf{srcL}(j)}$) before triggering the firing of the PU that will produce $x_{\mathsf{srcL}(j)}$ (respectively $x_{\mathsf{srcL}(i)}$). In other words, the production times of variables or variable schedule time slots τ are statically determined such that $\tau_{\mathsf{srcL}(i)} \prec \tau_{\mathsf{srcL}(j)}$ (respectively $\tau_{\mathsf{srcL}(j)} \prec \tau_{\mathsf{srcL}(i)}$). A similar argument applies for the right operands. Accordingly the buffer constraint for SO-SCAD$_{\mathrm{I}}$ is expressed in constraint 4.21.

$$\mathcal{C}_9 :\Leftrightarrow \bigwedge_{i,j=0}^{\ell-1} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \\ \wedge \\ \tau_{\mathsf{srcL}(i)} \prec \tau_{\mathsf{srcL}(j)} \wedge \tau_{\mathsf{srcR}(i)} \prec \tau_{\mathsf{srcR}(j)} \end{array} \right) \\ \vee \\ \left(\begin{array}{c} x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \\ \wedge \\ \tau_{\mathsf{srcL}(j)} \prec \tau_{\mathsf{srcL}(i)} \wedge \tau_{\mathsf{srcR}(j)} \prec \tau_{\mathsf{srcR}(i)} \end{array} \right) \end{array} \right) \tag{4.21}$$

In SO-SCAD$_{\mathrm{P}}$, the compiler simply allocates to the same PU those instructions that produce operand values targeted to the same input buffer. Therefore, if target variables $x_{\mathsf{tgt}(i)}$ and $x_{\mathsf{tgt}(j)}$ are produced by the same PU k, both left operands $x_{\mathsf{srcL}(i)}$ and $x_{\mathsf{srcL}(j)}$ are assigned to the same PU enforcing the ordering $x_{\mathsf{srcL}(i)} \preceq x_{\mathsf{srcL}(j)}$ if $x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)}$ or the ordering $x_{\mathsf{srcL}(j)} \preceq x_{\mathsf{srcL}(i)}$ if

$x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)}$. This is expressed in buffer constraint 4.22 for SO-SCAD$_\mathrm{P}$.

$$\mathcal{C}_{10} :\Leftrightarrow \bigwedge_{i,j=0}^{\ell-1} \beta_{\mathsf{tgt}(i),\mathsf{tgt}(j)} \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} x_{\mathsf{tgt}(i)} \prec x_{\mathsf{tgt}(j)} \\ \wedge \\ \beta_{\mathsf{srcL}(i),\mathsf{srcL}(j)} \wedge x_{\mathsf{srcL}(i)} \prec x_{\mathsf{srcL}(j)} \\ \wedge \\ \beta_{\mathsf{srcR}(i),\mathsf{srcR}(j)} \wedge x_{\mathsf{srcR}(i)} \prec x_{\mathsf{srcR}(j)} \end{array} \right) \\ \vee \\ \left(\begin{array}{c} x_{\mathsf{tgt}(j)} \prec x_{\mathsf{tgt}(i)} \\ \wedge \\ \beta_{\mathsf{srcL}(i),\mathsf{srcL}(j)} \wedge x_{\mathsf{srcL}(j)} \prec x_{\mathsf{srcL}(i)} \\ \wedge \\ \beta_{\mathsf{srcR}(i),\mathsf{srcR}(j)} \wedge x_{\mathsf{srcR}(j)} \prec x_{\mathsf{srcR}(i)} \end{array} \right) \end{array} \right) \tag{4.22}$$

To obtain the final constraint formula $\mathcal{C}_{so_i}$ (respectively $\mathcal{C}_{so_p}$) for SO-SCAD$_\mathrm{I}$ (respectively SO-SCAD$_\mathrm{P}$), simply replace buffer constraint $\mathcal{C}_5$ (constructed for SCAD) in 4.18 by the buffer constraint $\mathcal{C}_9$ (respectively $\mathcal{C}_{10}$) constructed above:

$$\mathcal{C}_{so_i} :\Leftrightarrow \bigwedge_{i=1}^{4} \mathcal{C}_i \wedge \mathcal{C}_9 \wedge \bigwedge_{i=6}^{8} \mathcal{C}_i \tag{4.23}$$

$$\mathcal{C}_{so_p} :\Leftrightarrow \bigwedge_{i=1}^{4} \mathcal{C}_i \wedge \mathcal{C}_{10} \wedge \bigwedge_{i=6}^{8} \mathcal{C}_i \tag{4.24}$$

In SCAD, the values consumed by the same input buffer may be produced by different PUs without coordinating their production times. In SO-SCAD$_\mathrm{I}$, the compiler must ensure a total order of production times of values consumed by each input buffer, but they may still be produced by different PUs since statically determined instruction issue times are enforced at runtime by controlling the firing times of PUs. In SO-SCAD$_\mathrm{P}$, these values must be produced by the same PU in the order in which they must be consumed by each input buffer. Therefore, clearly the buffer constraint for SO-SCAD$_\mathrm{P}$ is stricter compared to SO-SCAD$_\mathrm{I}$, which again is stricter compared to SCAD. This means that as we consider SCAD < SO-SCAD$_\mathrm{I}$ < SO-SCAD$_\mathrm{P}$ machines, it becomes increasingly difficult to avoid overhead operations. For example, consider the following program:

$$\begin{aligned} x_2 &= x_1 \odot_0 x_0 \\ x_3 &= x_2 \odot_1 x_2 \\ x_4 &= x_0 \odot_2 x_3 \\ x_5 &= x_3 \odot_3 x_1 \end{aligned} \tag{4.25}$$

Load variables x_0 and x_1 are assigned to the LSU, while variables $\{x_2, x_3, x_4, x_5\}$ can be assigned to any available PUs. Assignments of variables (equivalently instructions) to a minimal number of PUs in SCAD, SO-SCAD$_\mathrm{I}$ and SO-SCAD$_\mathrm{P}$ are shown in Table 4.2. Note that 2 PUs (excluding LSU or PU 0) are required in SCAD to execute the program without any overhead. This is because of the following reason: Computing x_4 is data dependent on x_2. Therefore, if x_2 and x_4 are assigned to the same PU 1, $x_2 \prec x_4$ must hold.

Consequently, $x_1 \prec x_0$ must hold since x_1 and x_0 are left operands of x_2 and x_4, respectively. Similarly, computing x_5 is data dependent on x_2. If x_2 and x_5 are assigned to the same PU 1, $x_2 \prec x_5$ and consequently $x_0 \prec x_1$ must hold since x_0 and x_1 are right operands of x_2 and x_5, respectively. Due to the contradicting constraints $x_1 \prec x_0$ and $x_0 \prec x_1$, an extra PU 2 is used in SCAD to compute x_4.

PU	instr order
0	$x_0 \prec x_1$
1	$x_2 \prec x_3 \prec x_5$
2	x_4

(a) *in SCAD*

PU	instr order
0	$x_0 \prec x_1$
1	$x_2 \prec x_3$
2	x_4
3	x_5

(b) *in SO-SCAD$_I$*

PU	instr order
0	$x_0 \prec x_1$
1	x_2
2	x_4
3	x_5
4	x_3

(c) *in SO-SCAD$_P$*

Table 4.2.: *Minimal PU assignments for Program 4.25*

At least 3 PUs (excluding LSU or PU 0) are needed in a SO-SCAD$_\mathrm{I}$ architecture to execute the same program without any overhead. This is because in a SO-SCAD$_\mathrm{I}$ machine, we can no longer schedule x_3 and x_5 in the same PU 1. Since computing x_5 is data-dependent on x_3, $x_3 \prec x_5$ must hold if they are computed by the same PU 1. Note that x_2 and x_1 are right operands of x_3 and x_5, respectively. Therefore, x_2 must be moved to the right input buffer of PU 1 before moving x_1 to the same buffer. This is however not possible in SO-SCAD$_\mathrm{I}$ since x_2 can be produced only after producing x_1 (x_2 is data-dependent on x_1). This was not a concern in SCAD since x_2 and x_1 are produced by different PUs, and thus have no need to be ordered.

Finally, 4 PUs (excluding LSU or PU 0) are needed in a SO-SCAD$_\mathrm{P}$ architecture to execute the same program without any overhead. This is because, in a SO-SCAD$_\mathrm{P}$ machine, x_2 and x_3 can no longer be scheduled in the same PU 1. Note that x_1 and x_2, which are left operands of x_2 and x_3, respectively, will have to be assigned to the same PU in SO-SCAD$_\mathrm{P}$ if x_2 and x_3 are produced by the same PU. This is not possible since x_1 is a load variable assigned to LSU (or PU 0), and x_2 must be assigned to another PU. A similar argument applies for right operands x_0 and x_2. Experimental results in the following section clearly show that while the use of overhead is easily avoided in SCAD, it is considerably more difficult to avoid the use of overhead in SO-SCAD$_\mathrm{I}$ and SO-SCAD$_\mathrm{P}$.

4.3. Experiments

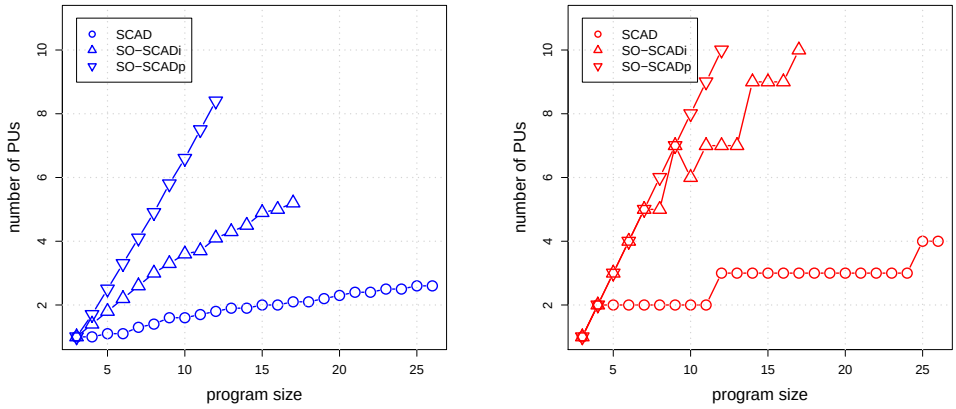
To generate input programs for experiments, a random basic block generator was implemented that accepts number of nodes n (program size) and number of levels l (program level) as inputs. The basic block is generated by randomly choosing two predecessors of every node, ensuring that the DAG has l levels. Clearly, for a n node basic block, $l := \{2, \ldots, n-1\}$ levels are possible.

A hundred basic blocks were generated for every pair (n, l) *in each described experiment* so that as many different patterns of DAGs are covered. We use Microsoft's Z3 solver [MoBj08a] to find satisfying assignments for boolean constraints for optimal basic block scheduling in SCAD and its variant architectures. First, we compare in Section 4.3.1 different SCAD variants in terms of difficulty in avoiding the use of overhead instructions and subsequent use of ILP. Section 4.3.2 compares the execution of queue oriented move code with register oriented move code by a cycle-accurate SCAD simulator. The feasibility of optimal SCAD code generation is discussed in Section 4.3.3.

4.3.1. Execution paradigms

For each randomly generated program, we determine the minimal number of PUs required in SO-SCAD$_\mathrm{P}$, SO-SCAD$_\mathrm{I}$ and SCAD architectures to execute the program optimally without any overhead. Recall that the decision version of the optimal code generation was mapped to SAT where a given program was, if possible, compiled to optimal move code for execution in a SCAD (or its variant) machine with a given number of PUs. To derive the minimal number of PUs, we perform a binary search between the theoretical minimal number (set as 1) and the maximal number (set to program size). The average-case and worst-case minimum number of PUs required in SCAD and its variant architectures to optimally execute programs of different sizes are shown in Figures 4.9a and 4.9b, respectively. With a timeout of 60 seconds, programs of sizes (nodes in DAG) 26, 17 and 12 were successfully processed for SCAD, SO-SCAD$_\mathrm{I}$ and SO-SCAD$_\mathrm{P}$ architectures, respectively. The reason for the lesser feasibility of optimal compilation as we consider SCAD $\prec$ SO-SCAD$_\mathrm{I}$ $\prec$ SO-SCAD$_\mathrm{P}$ is that the minimal number of PUs required in these machines for overhead-free execution of programs increases considerably in that order. The SAT solver has to handle more boolean constraints with more PUs.

The DO-SCAD architectures are not bound by any ordering constraint (and thus do not require any overhead). As expected, for other architectures, more PUs are required to avoid overhead with the increase in program size, as seen in Figure 4.9. However, the rate of increase in numbers of PUs varies considerably among these architectures. It is most difficult to avoid overhead in SO-SCAD$_\mathrm{P}$, often requiring a number of PUs proportional to the program size. Note that size 12 programs require on an average 8 PUs (and 10 PUs in the worst case). This means that for a SO-SCAD$_\mathrm{P}$ machine with a given number of PUs, the compiler will often have to introduce a lot of overhead to execute real programs degrading the overall performance. Therefore, while SO-SCAD$_\mathrm{P}$ offers simple hardware, the compiler will often find it prohibitively difficult (requiring a lot of PUs) or even impossible (with a limited number of PUs) to utilize ILP contained in programs. The situation is more optimistic for SO-SCAD$_\mathrm{I}$ architectures with a lower rate of increase in the number of PUs. The size 17 programs require on an average 5 PUs (but 10 PUs in the worst case) for overhead-free (optimal) execution. Although SO-SCAD$_\mathrm{I}$ offers simple hardware (same as SO-SCAD$_\mathrm{P}$), the control unit in SO-SCAD$_\mathrm{I}$ triggers the firing of PUs at statically determined instruction issue times, in contrast to



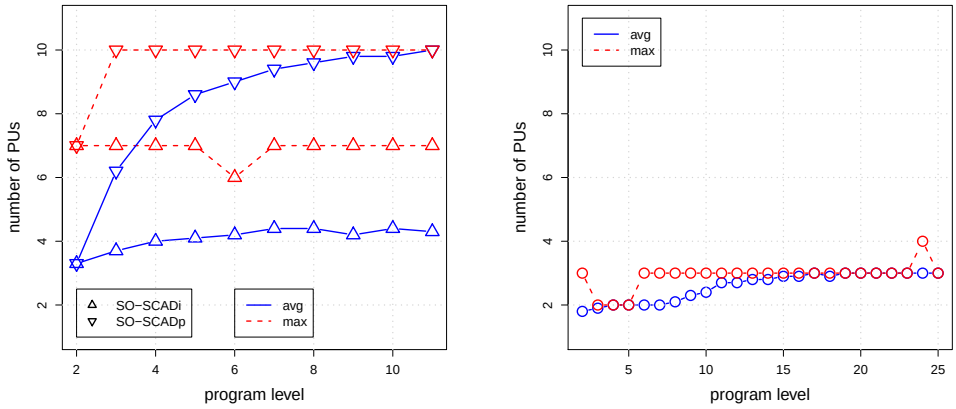
(a) *Average case* **(b)** *Worst case*

Figure 4.9.: *Minimal numbers of PUs required to execute programs of different sizes in SCAD and its variant architectures*

SO-SCAD$_P$ where the arrival of input values drives the firing of PUs. It is this flexibility to determine the instruction issue times that allows the SO-SCAD$_I$ compiler to more effectively avoid the overhead compared to the SO-SCAD$_P$ compiler. However, as already mentioned, the static instruction issue cannot adapt to a variable latency of PUs, memory accesses, and DTN, which is often restricting the use of ILP.

Compared to SO-SCAD architectures, only a few PUs are required in a SCAD architecture for the compiler to avoid the use of any overhead. Even for size 26 programs, less than 3 PUs on average (only 4 PUs in worst case) are enough (see Figure 4.9). Therefore, the organization of input buffers in SCAD (more complex compared to pure FIFO buffers in SO-SCAD and simpler compared to tag matching hardware in DO-SCAD) imparts more flexibility to the compiler in determining an instruction schedule so that the use of overhead is comfortably avoided compared to compilers for SO-SCAD variants. Real programs are easily executed without any overhead in SCAD machines with a limited number of PUs as shown by heuristic experimental results in Section 5.6. Moreover, the instruction issue in the dataflow order tolerates a variable component latency and further ensures the effective use of ILP in programs.

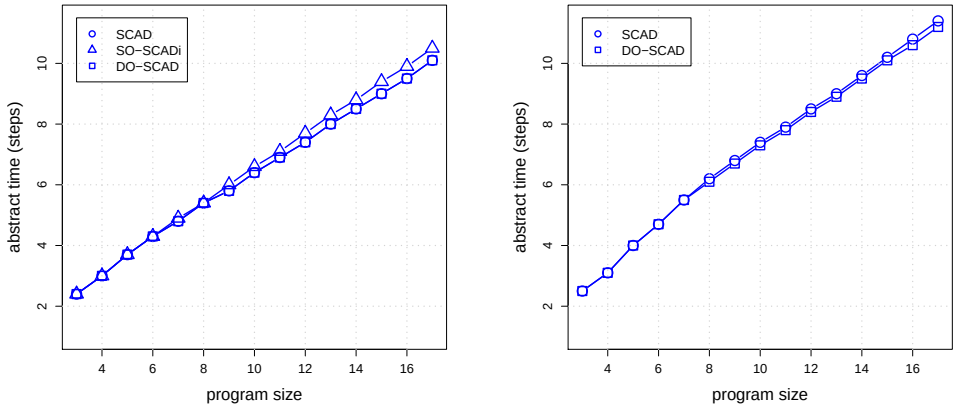
The minimal numbers of PUs required in different architectures with varying levels of DAGs (representing programs or basic blocks) are shown in Figure 4.10. The average number of PUs (in all architectures) increases with the number of levels. With more levels, there are more data dependencies that constrain the SCAD compiler in ordering dependent instructions in the same PU. In SO-SCAD$_I$, in addition to the ordering of dependent instructions in the same PU, data dependencies also constrain the compiler in determining



(a) *size 12 programs on SO-SCAD* **(b)** *size 26 programs on SCAD*

Figure 4.10.: *Minimal PUs required to execute programs of different levels*

the issue times of all dependent instructions. In SO-SCAD$_P$, data dependencies additionally restrict the compiler in allocating dependent instructions to PUs. Therefore, the impact of increase in program levels is more apparent in SO-SCAD$_P$ compared to SO-SCAD$_I$ (see Figure 4.10a) and least apparent in SCAD (see Figure 4.10b).



(a) *on SCAD variants with minimal PUs required in SO-SCAD$_I$* **(b)** *on SCAD and DO-SCAD with minimal PUs required in SCAD*

Figure 4.11.: *Average minimal time to execute programs of different sizes*

Finally, we want to determine for each architecture the degree of exploited ILP when enough PUs are available for an overhead-free execution. To derive

the minimal execution time (for maximizing the use of ILP), we again perform a binary search between a theoretical minimum (set to 1) and the maximum (set to program size). Note that the derived minimal execution time is abstract (in number of steps) since only a unit instruction latency is taken into account by the SAT encoding and other architectural timings such as fetch time, decode (registration of move instruction) time, time taken for values to ripple through buffers, etc. are abstracted out. The abstract steps should serve well to measure the relative use of ILP by different SCAD variants. In a first experiment, we determine the minimal number of PUs required in the SO-SCAD$_{\mathrm{P}}$ architecture for an overhead-free execution. With this number of PUs, we then observed that the minimal execution time derived for the SO-SCAD$_{\mathrm{P}}$, SO-SCAD$_{\mathrm{I}}$, SCAD, and DO-SCAD machines are equal. This is expected since the minimal number of PUs required in SO-SCAD$_{\mathrm{P}}$ for an optimal execution is nearly the same as the input program size. In the next experiment, the minimal number of PUs required in the SO-SCAD$_{\mathrm{I}}$ architecture was determined, and the minimum execution times with this many PUs in SO-SCAD$_{\mathrm{I}}$, SCAD, and DO-SCAD machines are derived (see Figure 4.11a). Similarly, minimum execution time in SCAD and DO-SCAD with the minimal number of PUs required for SCAD is shown in Figure 4.11b. It is clear from both graphs that once enough PUs are available in the SO-SCAD or the SCAD architecture to avoid the use of any overhead, they can exploit nearly maximal ILP (exploited by DO-SCAD) contained in programs. This means that the overhead is the main limiting factor in exploiting ILP.

4.3.2. Queue and register based SCAD code

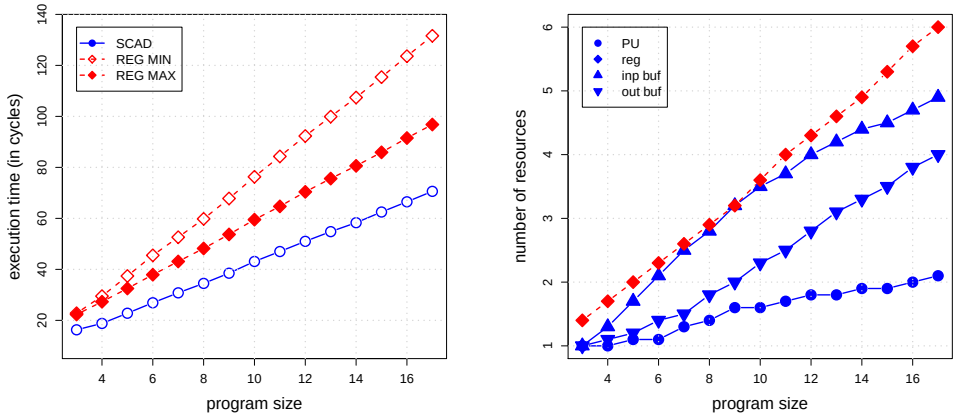
In this section, we generate optimal queue-based and optimal register-based move code, and compare the execution of both using a cycle-accurate SCAD simulator[1]. The optimal queue-based move code is generated as follows: Derive the minimal number of PUs required for an overhead-free compilation of the input program. With this number of PUs, derive a schedule (PU assignment and variable ordering) for a minimum execution time (maximum use of ILP) of the input program. Generate move code from this PU assignment and variable ordering. The optimal register-based move code is generated as follows: Allocate variables in the program to a minimal number of registers using the well known Chaitin-Briggs heuristic [Chai04] (that yields nearly optimal results). The resulting dataflow graph is used to build relevant SAT constraints (similar to constraints for DO-SCAD, i.e., without buffer or ordering constraints) so that instructions are assigned to PUs to maximize the use of ILP, taking care of any false dependencies. The same number of PUs are used in both queue-based and register-based compilation. With the PU assignment and allocation of variables to registers, it is straightforward to generate a move program: For each instruction, move operand values from registers to the corresponding input buffers and move result values from output buffers to the respective mapped registers. To summarize, both queue-based and

[1] `https://es.cs.uni-kl.de/tools/teaching/ScadSim.html`

register-based move programs use storage (buffer and registers) to accommodate all intermediate program values. To also study the effect of the number of register file ports, we execute register-based move code on, (1) a SCAD machine with a single-ported register file referred to as REG-MIN and (2) a SCAD machine with a multi-ported register file with as many ports as registers, referred to as REG-MAX. Recall that the organization of components in a SCAD architecture allows the use of register files just like any PU.

The buffer size can be critical for executing a program on a SCAD machine. With larger buffers, there will be fewer control unit stalls yielding better performance. The program will deadlock if a minimal buffer size (specific to a program) is not guaranteed (the control unit stalls forever due to lack of space in a buffer). However, since randomly generated basic blocks are small programs, we observe that the impact of buffer size can be neglected. Clearly, one entry in each input and output buffer is sufficient to successfully execute register-based move programs since all intermediate values are stored in registers. We measure the minimal number of entries in input and output buffers required for executing queue-based move programs. Clearly, free space in input buffers allows values to be transported from output buffers, thus freeing up space in the output buffers. Also, free space in output buffers allows PUs to consume values from input buffers, thus freeing up space in the input buffers. Therefore, the minimal input buffer size and the output buffer size required to execute a move program often depend on each other. To derive the minimal buffer sizes, we first fix the output buffer size to a large value (equal to the program size) and then determine the minimal input buffer size required for the successful execution of the move program. In a second step, with the input buffer size set to this derived minimum value, we determine the minimal output buffer size required for the successful execution of the move program. Input and output buffers of the same sizes are assumed in all PUs. Finally, we configure a unit latency for all components (PU, LSU, and DTN) in the SCAD machine.

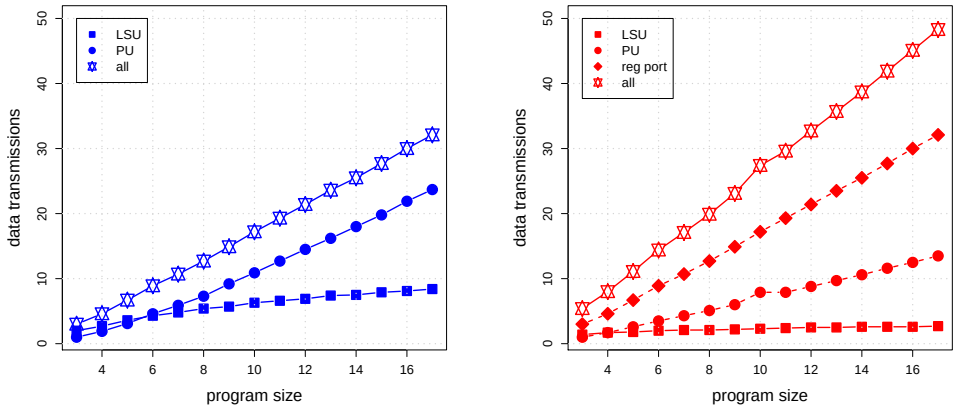
For randomly generated input programs of different sizes, the average execution time (in cycles) of the corresponding queue compiled move programs in SCAD, register compiled move programs in REG-MIN and REG-MAX, are shown in Figure 4.12a. As expected, the register-based move programs take more cycles to execute compared to the queue-based move programs due to additional register write moves to transport the computed result from the output of PUs to the registers. In queue-based move programs, values are directly communicated from output buffers of producer PUs to input buffers of consumer PUs. Besides additional register writes, the contention of simultaneous register accesses on the single read/write port leads to further execution cycles in REG-MIN. The gap in execution time between REG-MIN, REG-MAX, and SCAD understandably widens with the increase in program size due to more intermediate values in larger programs. The number of resources (PUs, registers, input and output buffer sizes) used is shown in Figure 4.12b. Together, the size of input and output buffers per PU is more than the overall number of registers. However, note that only up to 2 PUs are used on an average.



(a) *Execution time* **(b)** *Resource usage*

Figure 4.12.: *Average measure of parameters in queue-based and register-based SCAD program executions*

With more PUs, more buffers will be available for storing intermediate values, thus requiring less size per buffer. Moreover, buffers scale better compared to a register file. For an exact comparison, we will need to measure hardware resource usage (in terms of area, power, and access times), which is not covered in this thesis.



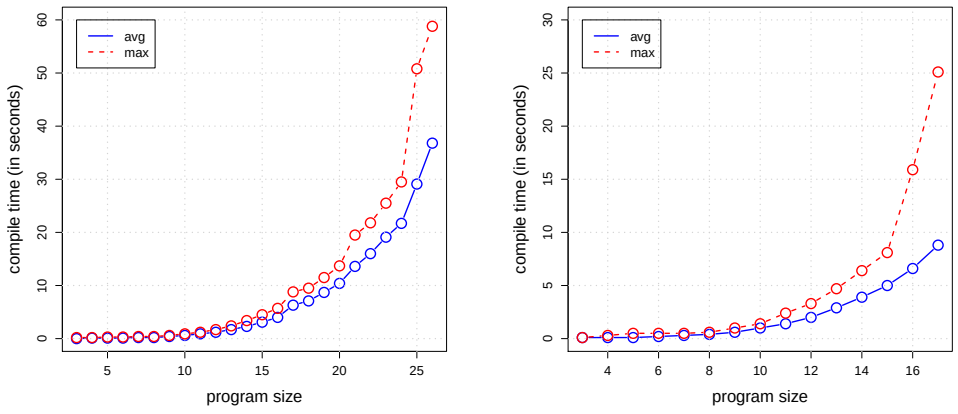
(a) *Queue-based SCAD program* **(b)** *Register-based SCAD program*

Figure 4.13.: *Average data transmissions in program executions*

The next experiment compares data transmission pattern in the execution of queue-based and register-based move programs. Results are shown in Fig-

ures 4.13a and 4.13b, respectively. Note that both graphs use the same scale for the x and y axes. The overall data transmission (or transportation of values) for programs of the same size is higher in execution using registers due to the additional register writes, compared to the execution by direct communication of intermediate results. Importantly, note that the bulk of this data traffic is concentrated on a limited number of ports in the register file. This clearly reveals the register file as the bottleneck in utilizing ILP and the hotspot in power consumption. Meanwhile, the data traffic in the execution of queue-based move program is distributed among processing units where each processing unit has dedicated input and output ports. Thus, we have a distributed communication pattern, which is important for the continued scaling of performance with the increase in ILP contained in programs. Note that the number of data transmissions by the LSU is more in the execution of queue-based move programs. This is because we directly instantiate the relevant number of copies of a loaded variable in the output buffer of the LSU. With more load variables, we may instead choose to load only one copy, move this copy to the input buffer of a PU and instantiate the relevant number of copies in the output buffer of that PU.

4.3.3. Feasibility

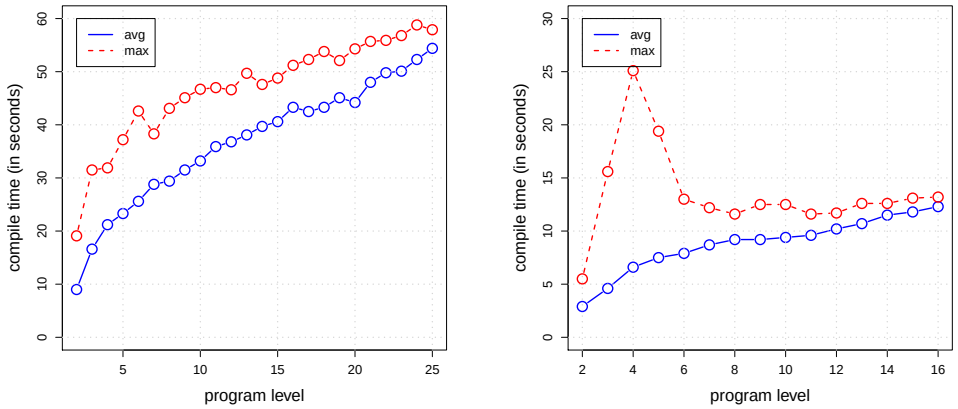


(a) *Resource constrained* **(b)** *Time constrained*

Figure 4.14.: *Compile time for optimal SCAD code generation for programs of different sizes*

We also study the feasibility of our optimal code generation by SAT solver. The average and maximal times taken by the SAT solver to derive a queue-based schedule (both resource-constrained and time-constrained) for programs of different sizes are shown in Figure 4.14. As expected, the time taken by SAT solver to derive a feasible schedule increases with the program size. A larger

number of program variables means a larger number of PU assignment relations ($\alpha_{i,j}$) to consider in the resource-constrained SAT problem. In addition, the time-constrained SAT problem must consider a larger number of time slot assignment relations ($\theta_{i,j}$) with increasing program size. With a timeout of 60 seconds, programs containing up to 26 variables were successfully processed for resource-constrained SCAD code generation (see Figure 4.14a). But for the time-constrained move code generation, programs containing only up to 17 variables were successfully processed (see Figure 4.14b).



(a) *Resource constrained code generation for programs of size* 26

(b) *Time constrained code generation for programs of size* 17

Figure 4.15.: *Compile time for optimal SCAD code generation for programs of different levels*

Figure 4.15 shows the SAT solver's runtime to derive a queue-based schedule for programs of different levels. The time taken increases with the number of levels since the SAT solver has to handle more data dependency constraints with increasing program levels. On the other hand, more data dependencies restrict the number of choices for variable orderings leaving the SAT solver with fewer variable orders to check to derive a feasible schedule. Therefore, in general, for two-level or three-level DAGs, the SAT solver can find a feasible schedule in less time due to many choices of variable orderings. However, for certain exceptions (difficult DAG structures), the SAT solver has to explore all possible variable orderings to determine a feasible one. This is why we often see spikes in the maximal compile time in Figures 4.15a and 4.15b.

Chapter 5

Heuristics for Code Generation

Contents

Since optimal SCAD code generation by SAT solvers is only feasible for small programs, heuristics are required to compile larger programs to move code for the execution on SCAD machines. In Section 5.2, we present a heuristic for assigning variables in a program to a minimal number of output buffers (or PUs) in a SCAD machine so that *resource-constrained* move code can be generated in *program order* without any *dup* or *swap* overhead. Recall that the number of copies of values to produce must also be determined at compile time. Depending on the control flow path, the number of uses of a value or the number of copies of a value to produce, may differ. Therefore, we present in Section 5.3, program transformations to equalize the number of uses of a value along each control flow path so that the number of copies to produce is uniquely determined. Finally, Section 5.4 describes move code generation.

5.1. Preliminaries

This section introduces some preliminary information necessary to understand the rest of this chapter. The MiniC[1] language and its compiler intermediate representation, used to implement the heuristic, are introduced in Section 5.1.1. Some basic definitions from compiler literature and notations used in the rest of this chapter are given in Section 5.1.2. Static single information (SSI) form [Anan99; Sing06], a compiler intermediate representation, is briefly described and its properties are outlined in Section 5.1.3.

5.1.1. The MiniC Language and Compiler

MiniC is a simple imperative programming language developed by the Embedded Systems Group[2] at the University of Kaiserslautern. It has a minimal set of data types, namely boolean (**bool**), unsigned/signed integers of machine width (**nat** and **int**, respectively) and arrays. For these data types, there are the usual boolean and arithmetic operations needed to implement reasonable benchmarks. The language is strictly typed and allows programmers to convert types using cast expressions like (**nat**) τ to enforce the type checker to give the expression τ the type **nat**. Functions are not allowed to be recursive and are inlined by the MiniC compiler. MiniC provides the standard statements listed in Table 5.1, where S, S_i are statements, φ is a boolean expression, λ is a left-hand side expression and τ, τ_i are right-hand side expressions. A simple

Statement type	Syntax
assignment	$\lambda = \tau;$
sequence	$S_1; S_2$
conditional	**if**(φ) S_1 [**else** S_2]
while loop	**while**(φ) S_l
for loop	**for** $(i = \tau_l \ldots \tau_u)$ S_l
do-while loop	**do** S_l **while**(φ)
function call	$f(\tau_1, \ldots, \tau_n);$
return	**return** τ

Table 5.1.: *Statements in MiniC language*

compiler framework is available for MiniC that offers syntax analysis, type-checking, and a translation to an intermediate format referred to as command language (similar to three-address code). A command program of length ℓ is a sequence of command instructions $(I_0, I_1, \ldots, I_{\ell-1})$. Available instruction types are listed in Table 5.2, where x, x_i and y are variables, $\odot$ is a primitive binary arithmetic or logic operator and i is an immediate value used to represent instruction I_i as branch target.

[1] `https://es.cs.uni-kl.de/tools/teaching/MiniC.html`
[2] `https://es.cs.uni-kl.de`

Command instruction	Syntax
copy variable	$y = x$
assignment	$y = x_1 \odot x_2$
array access	$y = x_1[x_2]$
array assignment	$y[x_1] = x_2$
unconditional branch	**goto** i
conditional branch	**if** x **goto** i

Table 5.2.: *Instructions of the Command language*

5.1.2. Basic Definitions and Notations

The MiniC compiler also offers further functions that split the command programs into control-flow graphs which is the basis for many code optimization techniques.

Definition 5.1 ⟨ Control-Flow Graph ⟩
A control-flow graph (CFG) is a program representation as a directed graph $G_C = (V_C, E_C, I_s, I_e)$ *with command instructions as set of nodes* V_C*, set of edges* E_C *and two special nodes* I_s *and* I_e *where* I_s *is the start node with no incoming edge and* I_e *is the end node with no outgoing edge.*

In CFG representation of a command program $(I_0, I_1, \ldots, I_{\ell-1})$, $(I_{i-1}, I_i) \in E_C$ for all $i \in \{1 \ldots \ell - 1\}$ and $(I_i, I_j) \in E_C$ if I_i is a branch instruction whose target is I_j. I_s denotes start node I_0 and I_e denotes the end node $I_{\ell-1}$. A CFG is also often defined at the granularity of basic blocks, which is a piece of straight line code, i.e., there are no jumps into or out of the middle of a block.

Definition 5.2 ⟨ Path ⟩
A path of length $\ell \geq 0$ *from a node* I_u *to a node* I_v *is a sequence of nodes* $(I_{k_0}, I_{k_1}, \ldots, I_{k_\ell})$ *such that* $I_{k_0} = I_u$, $I_{k_\ell} = I_v$ *and* $(I_{k_{i-1}}, I_{k_i}) \in E_C$ *for* $i \in \{1 \ldots \ell\}$.

A node I_v is *reachable* from I_u if and only if there is a path from I_u to I_v in the CFG. Every node is reachable from the start node I_s. The end node I_e is reachable from every other node.
In a CFG representation of a command program $(I_0, I_1, \ldots, I_{\ell-1})$, an edge (I_i, I_j) is called a forward-edge if $i < j$ and a back-edge otherwise. Notice that MiniC is a structured programmed language in that only structured control flow constructs [BoJa66] are available for constructing MiniC programs. This

structured control flow is further retained by the MiniC compiler in that the CFG of command programs are reducible flow graphs where possible loop structures are shown in Figure 5.1 and are defined as follows:

Definition 5.3 ⟨ Loop ⟩
A sequence of command instructions $(I_i, \ldots, I_{i+m})$ *where* $m \geq 0$*, in a command program* $(I_0, I_1, \ldots, I_{\ell-1})$*, is a loop if* I_i *is the single entry point,* I_i *or* I_{i+m} *is the single exit point, and there exists a back-edge* (I_{i+m}, I_i)*.*

(a) *Entry and exit at* I_i **(b)** *Entry at* I_i *and exit at* I_{i+m}

Figure 5.1.: *Possible control-flow graphs of loops in MiniC*

Finally, we define the concepts of dominance and post-dominance that are necessary to understand the construction and properties of SSI programs explained in the next section.

Definition 5.4 ⟨ Dominance ⟩
A node I_u *dominates a node* I_v*, denoted by* I_u `dom` I_v*, if every path from the start node* I_s *to* I_v *contains* I_u*. Every node dominates itself. If* I_u `dom` I_v *and* $I_u \neq I_v$*, then* I_u *strictly dominates* I_v *denoted by* I_u `sdom` I_v*. The node* I_u *is the unique immediate dominator of node* I_v*, denoted by* I_u `idom` I_v*, if* I_u `sdom` I_v*, but* I_u *does not strictly dominate any other strict dominators of* I_v*.*

Definition 5.5 ⟨ Dominance Frontier ⟩
A node I_v is in the dominance frontier set of node I_u, denoted by $\mathcal{DF}(I_u)$, if I_u dominates an immediate predecessor of I_v, but does not strictly dominate I_v, i.e., $\mathcal{DF}(I_u) = \{I_v \mid (I_w \rightsquigarrow I_v) \ \wedge \ (I_u \ \texttt{dom} \ I_w) \ \wedge \ \neg(I_u \ \texttt{sdom} \ I_v)\}$

Intuitively, the dominance of I_u reaches until, but not including I_v so that I_v is a control flow join point as shown in Figure 5.2a. Post-dominance and the post-dominance frontier are defined analogously.

(a) $I_v \in \mathcal{DF}(I_u)$ **(b)** $I_u \in \mathcal{PDF}(I_v)$

Figure 5.2.: *Examples for intuitive understanding of dominance and post-dominance frontiers*

Definition 5.6 ⟨ Post-Dominance ⟩
A node I_v post-dominates a node I_u, denoted by I_v $\texttt{pdom}$ I_u, if every path from I_u to the end node I_e contains I_v. Every node post-dominates itself. If I_v $\texttt{pdom}$ I_u and $I_v \neq I_u$, then I_v strictly post-dominates I_u denoted by I_v $\texttt{spdom}$ I_u. The node I_v is the unique immediate post-dominator of node I_u, denoted by I_v $\texttt{ipdom}$ I_u, if I_v $\texttt{spdom}$ I_u, but I_v does not strictly post-dominate any other strict post-dominators of I_u.

Definition 5.7 ⟨ Post-Dominance Frontier ⟩
A node I_u is in the post-dominance frontier set of node I_v, denoted by $\mathcal{PDF}(I_v)$, if I_v post-dominates an immediate successor of I_u, but does not strictly post-dominate I_u, i.e., $\mathcal{PDF}(I_v) = \{I_u \mid (I_u \rightsquigarrow I_w) \ \wedge \ (I_v \ \texttt{pdom} \ I_w) \ \wedge \ \neg(I_v \ \texttt{spdom} \ I_u)\}$

Intuitively, the post-dominance of I_v reaches until, but not including I_u so that I_u is a control flow split point as shown in Figure 5.2b.

5.1.3. Static Single Information (SSI) Representation

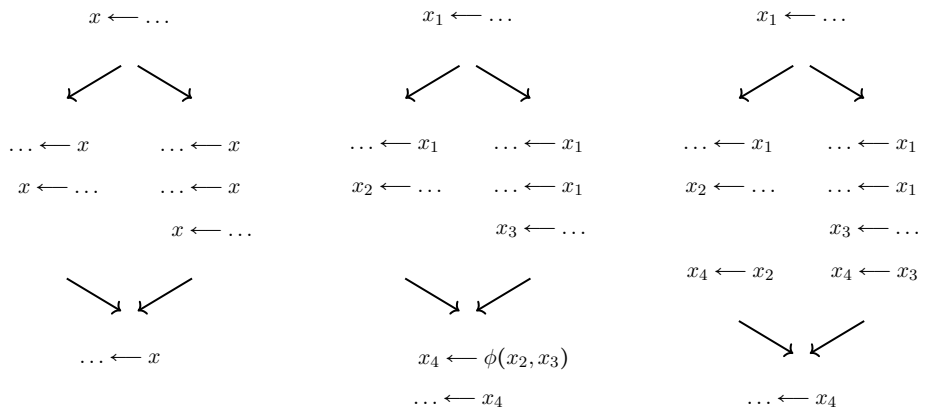
Static single information (SSI) [Anan99; Sing06] is an extension of the well established static single assignment (SSA) [AlWZ88; CFRW91; RoWZ88] compiler intermediate representation. In the following, the SSA representation is first discussed, followed by its extension to SSI. A program is in SSA form if each variable is defined (statically or textually) only once. A non-SSA program is transformed into an SSA program by replacing each definition of a variable with a unique definition and renaming each use of the original variable using the new variable whose unique definition reaches it. Clearly, multiple definitions may merge from different paths to a control flow join and reach a given use. In this case, a ϕ-function is placed at the join point that selects the appropriate variable depending on which path was executed. Consequently, the following properties hold for SSA (and SSI) programs.

Property 5.1 ⟨ Unique Definition ⟩
Each variable in a program in SSA (and SSI) form has a unique definition.

Property 5.2 ⟨ Dominance ⟩
The unique definition of any variable x in a program in SSA (and SSI) form, dominates all uses of x.

For example, consider the CFGs shown in Figure 5.3a. The variable definitions are made unique, and variable uses are appropriately renamed to construct the SSA program in Figure 5.3b. Note that a ϕ-function is used to assign x_4, with x_2 if the left control flow path was taken or with x_3 if the right control flow path was taken. This is a pseudo assignment in that the execution of a ϕ-function requires knowledge of the past control flow. Therefore, to execute SSA programs, ϕ-function assignments will have to be replaced by actual assignments that respect the semantics of the ϕ-function. A naive elimination of ϕ-functions places copy assignments at the end of each predecessor basic block. For example, $x_4 \longleftarrow x_2$ and $x_4 \longleftarrow x_3$ are placed at the end of the basic blocks in the left path and the right path, respectively (see Figure 5.3c).

The SSA form is constructed in two phases: First, ϕ-functions are inserted for each variable so that only a single definition of the variable reaches its uses. Second, variables are uniquely named when defined, and their uses are appropriately renamed. If a variable is defined in instruction I, then ϕ-functions are inserted for that variable at dominance frontiers (more precisely at iterated dominance frontiers) of I ($\mathcal{DF}(I)$) to construct a minimal SSA



(a) *Non-SSA program* **(b)** *SSA program* **(c)** *After elimination*

Figure 5.3.: *An example for SSA transformation and elimination*

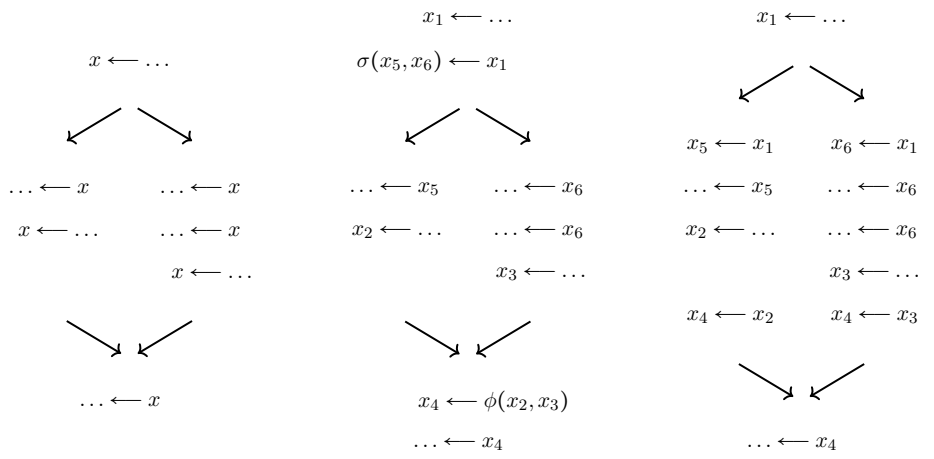
form [CFRW91; ApPa02]. The pruned SSA form reduces the number of ϕ-functions by not inserting the ϕ-function for a variable at program points where the variable is not live anymore.

SSI form, introduced in [Anan99] and more concisely revisited in [Sing06], extends the SSA form by additionally treating uses of a variable similar to how SSA form treats variable definitions. More precisely, while the SSA form establishes that each use of a variable is dominated by its unique definition (Property 5.2), the SSI form further establishes that the unique definition of a variable is post-dominated by each of its uses.

Property 5.3 ⟨ Post-Dominance ⟩
Each use of any variable x in a program in SSI form post-dominates the unique definition of x.

Clearly, the definition of a variable may separate to different paths at a control flow split and reach different uses of the variable along these different paths. Therefore, in the same way that ϕ-functions are inserted at join points to construct SSA programs, σ-functions are further inserted at split points to construct SSI programs. The σ-function assigns a variable whose definition reaches a split point to an appropriate variable depending on which path will be executed in the future. For example, consider CFG in Figure 5.4a (same as Figure 5.3a). Note that in addition to the ϕ-function at the join point, a σ-function is inserted at the split point to derive the SSI program in Figure 5.4b. The σ-function will assign x_1, to variable x_5 if the left control flow path will be taken in future or to variable x_6 if the right control flow path will be taken. The assignments to σ-functions are also pseudo assignments since the

execution of a σ-function requires knowledge of the future control flow. Both ϕ and σ-function assignments will have to be replaced by appropriate actual assignments to execute SSI programs. Analogous to eliminating ϕ-functions, a naive elimination of σ-functions places copy assignments at the beginning of each successor basic block. For example, $x_5 \longleftarrow x_1$ and $x_6 \longleftarrow x_1$ are placed at the beginning of basic blocks in the left path and the right path, respectively (see Figure 5.4c).



(a) *Non-SSI program* **(b)** *SSI program* **(c)** *After elimination*

Figure 5.4.: *An example for SSI transformation and elimination*

While SSI is introduced in [Anan99], an alternative definition and algorithm are provided in [Sing06]. A more recent work [BBDR12] discovered that both definitions are not equivalent. To distinguish between these definitions, the authors introduce the notion of upward-exposed use, where the use of a variable x is upward-exposed at a program point p if there is a path from p to the use that does not go through any other use of x. In the SSI definition in [Anan99], at most one use of each variable is upward-exposed at each program point p. A pseudo-use of each variable is assumed at the end node I_e of the CFG so that this property holds even at program points where the variable is not live. Clearly, these pseudo-uses are only considered for the SSI construction and not for liveness analysis. However, in the SSI definition in [Sing06], at most one use of each variable is upward-exposed only at the definition points and *not* at all program points. Hence, the definition of SSI in [Anan99] is more constrained compared to that in [Sing06]. For more details, see [BBDR12]. Interestingly, the construction algorithm in [Sing06] produces an SSI form that corresponds to the definition in [Anan99]. We use this algorithm for the SSI construction so that the following property can be stated:

Property 5.4 ⟨ Unique Upwards-Exposed Use ⟩
For each point in a program in SSI form, there is a unique upward-exposed use of each variable.

To construct an SSI form, ϕ and σ-functions are inserted in the first phase. The ϕ-functions are inserted for each variable so that the dominance property 5.2 is satisfied. The σ-functions are inserted for each variable so that both the post-dominance property 5.3 and the unique upwards-exposed use property 5.4 are satisfied. Notice that the insertion of ϕ-functions introduces new definitions that may trigger the insertion of corresponding σ-functions and vice versa. Therefore, it is necessary to perform a fixpoint iteration that alternately introduces ϕ and σ-functions. In the second phase, variables are given unique names when defined, and their uses are appropriately renamed. Analogous to the insertion of ϕ-functions in a minimal SSA form, a minimal SSI form is constructed by inserting σ-functions for a variable at post-dominance (more precisely at iterated post-dominance) frontiers of I ($\mathcal{PDF}(I)$) where the variable is used in instruction I. Clearly, the elimination of ϕ and σ-functions results in extra overhead in the form of copy assignments. To minimize these, we use the pruned SSI form where the number of ϕ and σ-functions are reduced by not inserting these for a variable at program points where the variable is not live.

5.2. Buffer Assignment

We first explore in Section 5.2.1 execution scenarios for any pair of variables, such that if assigned to the same output buffer (or same PU), *dup* and/or *swap* overhead operations will be required to read (or dequeue) their values. Restrictions are then introduced for the move code generation so that the dataflow analysis framework [Kild73] can be used to compute whether it is safe to map a given pair of variables to the same output buffer as explained in Section 5.2.2. Eventually, a buffer interference graph (similar to the traditional register interference graph) is computed whose coloring yields a valid assignment of variables in the program to output buffers in the SCAD machine so that overhead-free move code can be generated.

5.2.1. Variable Interference

Let x^i denote the i^{th} value (i.e., the value from the i^{th} write) of variable x. Clearly, x^i has a single write instance and may have any number of read instances. Figure 5.5 shows the write instance and read instances of x^i, where $\mathsf{w}(x^i)$ denotes the time at which x^i is produced, $\mathsf{r}_m(x^i)$ denotes the time at which the m^{th} read of x^i occur and c is the total number of reads of x^i before $\mathsf{w}(x^{i+1})$) if any. Assume that the variables x and y are assigned to the same output buffer (i.e., scheduled on the same PU) in a SCAD machine. If

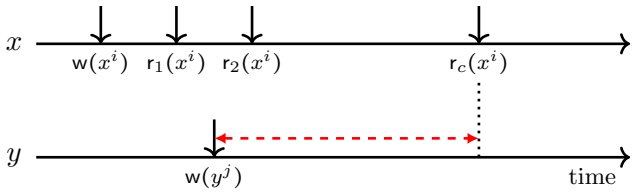


Figure 5.5.: *Write and read instances of variables x and y during program execution*

there exists an instance j of variable y such that $\mathsf{w}(y^j) > \mathsf{w}(x^i)$ as shown in Figure 5.5, then any read of y^j can only occur after all remaining reads of x^i. At time $\mathsf{w}(y^j)$, all copies of y^j are 'blocked' by the remaining unread copies of x^i in the output buffer as shown in Figure 5.6. The dashed line (in red color) shows the timeline during which it is not possible to read any copy of y^j unless all unread copies of x^i are rotated (to the same or a different output buffer) to bring y^j to the head of the output buffer. Therefore, if it is required to read (dequeue) y^j during this timeline, the variable y is said to be blocked by the variable x, denoted by $y \sqsubset x$. If either y is blocked by x or vice versa, then x and y are said to 'interfere' with each other denoted by $x \sqcap y$. In this case, x and y must be assigned to different output buffers in the SCAD machine to avoid overhead (rotation of values). Thus, the following definitions are in order:

$\rightarrow$	y^j	$\ldots$	y^j	x^i	$\ldots$	x^i	$\rightarrow$

Figure 5.6.: *Content of the output buffer (the tail is on the left and the head is on the right hand side) at time $\mathsf{w}(y^j)$ (from Figure 5.5)*

Definition 5.8 ⟨ Blocking: $y \sqsubset x$ ⟩
A variable y is 'blocked' by another variable x, denoted by $y \sqsubset x$, if and only if the following holds:

$$\exists i, j \in \mathbb{N} \mid \left(\begin{array}{c} \big(\mathsf{w}(y^j) > \mathsf{w}(x^i)\big) \\ \wedge \\ \big(\exists m, n \in \mathbb{N} \mid \mathsf{r}_n(y^j) < \mathsf{r}_m(x^i)\big) \end{array} \right) \tag{5.1}$$

Definition 5.9 ⟨ Interfering: $x \sqcap y$ ⟩
A pair of variables, x and y 'interfere' with each other, denoted by $x \sqcap y$, if and only if the following holds:

$$(y \sqsubset x) \vee (x \sqsubset y) \tag{5.2}$$

Clearly, the 'interfere' operator $\square$ is commutative while the 'blocked' operator $\sqsubset$ is not commutative.

5.2.2. Computing the Interference Graph

The order in which variables are written and read determines if they interfere. Therefore, to compute the interference of variables, we enforce that variables are written and read in the order that they are defined and used, respectively, in the command program. This order is enforced by restricting the move code generation in that command instructions are considered in program order for move code generation. That is, operand moves of any command instruction I_i are scheduled before scheduling any operand moves of command instruction I_{i+1}. Thus, variable writes and reads occur in program order at runtime, facilitating the use of the well-known dataflow analysis framework [Kild73] to compute whether $y \sqsubset x$ holds for any pair of variables x and y.

First, we define the domain of values on which the dataflow analysis framework is employed.

Definition 5.10 ⟨ var-def (v, I) tuple ⟩
A variable definition tuple (v, I) *contains a variable* v *and an instruction* I *that define (write) variable* v*.*

Definition 5.11 ⟨ var-use (v, I) tuple ⟩
A variable use tuple (v, I) *contains a variable* v *and an instruction* I *that use (read) variable* v*.*

Let W and R denote the set of all possible var-def and var-use tuples respectively in a given command program. Furthermore, let $\mathsf{W}(x)$ and $\mathsf{R}(x)$ denote the subsets of W and R, respectively, consisting of tuples that correspond to variable x, i.e.,

$$\begin{aligned} \mathsf{W}(x) &= \{(x, I) \mid (x, I) \in \mathsf{W}\} \\ \mathsf{R}(x) &= \{(x, I) \mid (x, I) \in \mathsf{R}\} \end{aligned} \tag{5.3}$$

Next 'may' approximations of reaching variable definition tuples and live variable use tuples at each program point are computed and finally results from these analyses are used to compute the interference of variables on buffers in a SCAD machine. The analyses are described followed by a computation of the buffer interference graph.

Reaching Definition Tuples

A var-def tuple (v, I) reaches a program point p if there exists at least one path from the point immediately following instruction I to p such that the variable v is not redefined (overwritten) along that path. The goal of reaching definitions analysis is to compute for every program point the set of var-def tuples that reach this program point. Clearly, reaching definitions analysis is a forward analysis performed on the set of var-def tuples. The following transfer function is used for each instruction I.

$$\begin{aligned} \mathsf{in}(I) &= \bigcup_{I' \rightsquigarrow I} \mathsf{out}(I') \\ \mathsf{out}(I) = \mathsf{f}_s(\mathsf{in}(I)) &= (\mathsf{in}(I) \setminus \mathsf{kill}(I)) \cup \mathsf{gen}(I) \end{aligned} \tag{5.4}$$

where $\mathsf{in}(I)$ and $\mathsf{out}(I)$ are the sets of var-def tuples that reach the entry and the exit of instruction I, respectively. Instruction I' in $I' \rightsquigarrow I$ is a predecessor of I. The set of var-def tuples generated by instruction I is denoted by $\mathsf{gen}(I)$ and $\mathsf{kill}(I)$ denotes the set of var-def tuples killed by instruction I. See Table 5.3 for $\mathsf{gen}(I)$ and Table 5.4 for $\mathsf{kill}(I)$ for each command instruction type. Note that the union operator $\cup$ is used as the *meet* or *join* operator to compute the 'may' approximation.

Command Instr I	$\mathsf{gen}(I)$: var-def tuples
$y = x$	$\{(y, I)\}$
$y = x_1 \odot x_2$	$\{(y, I)\}$
$y = x_1[x_2]$	$\{(y, I)\}$
$y[x_1] = x_2$	$\{\}$
goto i	$\{\}$
if x **goto** i	$\{\}$

Table 5.3.: *Variable definition tuples generated by each command instruction type*

Command Instr I	$\mathsf{kill}(I)$: var-def tuples
$y = x$	$\mathsf{W}(y)$
$y = x_1 \odot x_2$	$\mathsf{W}(y)$
$y = x_1[x_2]$	$\mathsf{W}(y)$
$y[x_1] = x_2$	$\{\}$
goto i	$\{\}$
if x **goto** i	$\{\}$

Table 5.4.: *Variable definition tuples killed by each command instruction type*

For any given program, starting with empty initial values, the analysis evaluates the equation system 5.4 until a fixpoint is reached or as long as there are changes (see Algorithm 1). This derives for every program point p, a set of var-def tuples that reaches p denoted by D_p.

Algorithm 1: Reaching definitions analysis

```
1  Function ReachingDefs(CFG):
       // initialize
2      forall node I in CFG do
3          out[I] ⟵ ∅
4          in[I]  ⟵ ∅
5      end
       // iterate
6      repeat
7          foreach node I in CFG do
8              in[I]  ⟵ ⋃_{I'⇝I} out[I']
9              out[I] ⟵ (in[I] ∖ kill(I)) ∪ gen(I)
10         end
11     until out[I] unchanged for all nodes I
       // return
12     return in[I] for each node I
```

Live Use Tuples

A var-use tuple (v, I) may be reached from a program point p if there exists at least one path from p to the instruction I such that the variable v is not defined (written) along that path. The goal of live uses analysis is to compute for every program point, the set of var-use tuples that may be reached from this program point. Unlike reaching definitions analysis, live uses is a backward analysis performed on the set of var-use tuples. The following transfer function is used for each instruction I.

$$\begin{aligned} \mathsf{out}(I) &= \bigcup_{I \rightsquigarrow I'} \mathsf{in}(I') \\ \mathsf{in}(I) &= \mathsf{f}_I(\mathsf{out}(I)) = (\mathsf{out}(I) \setminus \mathsf{kill}(I)) \cup \mathsf{gen}(I) \end{aligned} \tag{5.5}$$

where $\mathsf{in}(I)$ and $\mathsf{out}(I)$ are sets of var-use tuples that may be reached from the entry and the exit of instruction I, respectively. Instruction I' in $I \rightsquigarrow I'$ is a successor of I. Furthermore, $\mathsf{gen}(I)$ is the set of var-use tuples generated by instruction I and $\mathsf{kill}(I)$ denotes the set of var-use tuples killed by instruction I. See Table 5.5 for $\mathsf{gen}(I)$ and Table 5.6 for $\mathsf{kill}(I)$ for each command instruction type. Again, note that the union operator $\cup$ is used as the *meet* or *join* operator to compute the 'may' approximation.

For any given program, starting with empty initial values, the analysis evaluates the equation system 5.5 until a fixpoint is reached or as long as there are changes (see Algorithm 2). This derives for every program point p, a set of var-use tuples U_p that may be reached from p.

Command Instr I	$\mathsf{gen}(I)$: var-use tuples
$y = x$	$\{(x, I)\}$
$y = x_1 \odot x_2$	$\{(x_1, I), (x_2, I)\}$
$y = x_1[x_2]$	$\{(x_2, I)\}$
$y[x_1] = x_2$	$\{(x_1, I), (x_2, I)\}$
goto i	$\{\}$
if x **goto** i	$\{(x, I)\}$

Table 5.5.: *Variable use tuples generated by each command instruction type*

Command Instr I	$\mathsf{kill}(I)$: var-use tuples
$y = x$	$\mathsf{R}(y)$
$y = x_1 \odot x_2$	$\mathsf{R}(y)$
$y = x_1[x_2]$	$\mathsf{R}(y)$
$y[x_1] = x_2$	$\{\}$
goto i	$\{\}$
if x **goto** i	$\{\}$

Table 5.6.: *Variable use tuples killed by each command instruction type*

Algorithm 2: Live use tuples analysis

```
1  Function LiveUses(CFG):
       // initialize
2      forall node I in CFG do
3          in[I]  ⟵ ∅
4          out[I] ⟵ ∅
5      end
       // iterate
6      repeat
7          foreach node I in CFG do
8              out[I] ⟵ ⋃_{I⇝I'} in[I']
9              in[I]  ⟵ (out[I] ∖ kill(I)) ∪ gen(I)
10         end
11     until in[I] unchanged for all nodes I
       // return
12     return in[I] for each node I
```

Computation

Using results from the aforementioned analyses, we now determine for any given pair of variables x and y, if y is blocked by x (i.e., $y \sqsubset x$). In other words, we determine if there exists any program point p whose execution lies

on the dashed line (in red color) in Figure 5.5. In the following, we use the notations:

$$\mathsf{D}_p(x) = \{I \mid (x, I) \in \mathsf{D}_p\}$$
$$\mathsf{U}_p(x) = \{I \mid (x, I) \in \mathsf{U}_p\}$$

First, we identify any program point p so that there exists a path from a definition of x (say instruction I_d^x) to p via a definition of y (say instruction I_d^y) such that x is not overwritten along that path and y is not overwritten along the path from I_d^y to p. Such a path exists only if the following holds:

$$\exists I_d^y \in \mathsf{D}_p(y) \mid \left(\exists I_d^x \in \mathsf{D}_{I_d^y}(x) \mid I_d^x \in \mathsf{D}_p(x)\right) \tag{5.6}$$

Note that this is equivalent to identifying

$$\exists i, j \mid \mathsf{t}(p) > \mathsf{w}(y^j) > \mathsf{w}(x^i)$$

where $\mathsf{t}(p)$ is the time at which the program point p is executed.

Next we identify for the same program point p, if there exists a path from p to a use of x (say instruction I_u^x) via a use of y (say instruction I_u^y) such that x is not defined (written) along that path and y is not defined (written) along the path from p to I_u^y. Such a path exists only if the following holds:

$$\exists I_u^y \in \mathsf{U}_p(y) \mid \left(\exists I_u^x \in \mathsf{U}_{I_u^y}(x) \mid I_u^x \in \mathsf{U}_p(x)\right) \tag{5.7}$$

Note that this is equivalent to identifying

$$\exists m, n \in \mathbb{N} \mid \mathsf{t}(p) < \mathsf{r}_n(y^j) < \mathsf{r}_m(x^i)$$

for the same i, j and $\mathsf{t}(p)$.

If both such paths exist, we confirm from Definition 5.8 that variable y is blocked by x or $y \sqsubset x$ holds. Then, clearly the execution time of program point p, $\mathsf{t}(p)$ lies on the dashed line (in red color) in Figure 5.5. Therefore, for any given pair of variables x and y, if there exists a point p in the program such that the following holds:

$$\begin{gathered}\left(\exists I_d^y \in \mathsf{D}_p(y) \mid \left(\exists I_d^x \in \mathsf{D}_{I_d^y}(x) \mid I_d^x \in \mathsf{D}_p(x)\right)\right) \\ \wedge \\ \left(\exists I_u^y \in \mathsf{U}_p(y) \mid \left(\exists I_u^x \in \mathsf{U}_{I_u^y}(x) \mid I_u^x \in \mathsf{U}_p(x)\right)\right)\end{gathered} \tag{5.8}$$

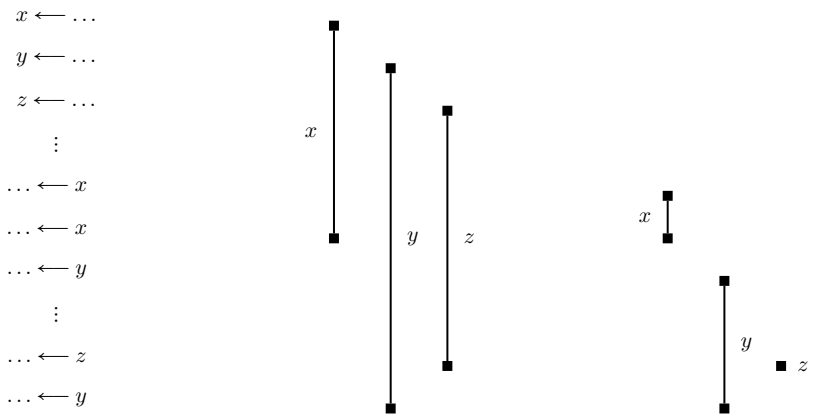
then y is blocked by x or $y \sqsubset x$ holds. Clearly, the above condition for identifying relevant program points is necessary but not sufficient due to probable infeasible paths. That is, if y is blocked by x at runtime then the above condition must be satisfied. However, if the above condition is satisfied, it may also be the case that y is not blocked by x.

From Definition 5.9, variables x and y interfere (i.e., $x \sqsupset y$ holds) if either $y \sqsubset x$ or $x \sqsubset y$ is true. By computing the same for all pairs of variables in the command program, a buffer interference graph $G = (V, E)$ is constructed with variables as vertices V, and there exists an edge between any two vertices if

Algorithm 3: Constructing the buffer interference graph

```
1  Function ConstructBufferInterferenceGraph(CFG):
       // initialize
2      V ⟵ set of all nodes in CFG except start and end nodes
3      E ⟵ ∅ // empty set of edges
4      P ⟵ set of all variable pairs
       // dataflow analyses
5      D ⟵ ReachingDefs(CFG)
6      U ⟵ LiveUses(CFG)
       // return true if x is blocking y, otherwise false
7      Function Blocking(x, y):
           /* return true if any definition of x reaches any definition of
              y that in turn reaches node I                                  */
8          Function DefOrder(I):
9              yDefs ⟵ {I_y | (y, I_y) ∈ D[I]}
10             foreach node I_y in yDefs do
11                 xDefs ⟵ {I_x | (x, I_x) ∈ D[I_y]}
12                 if xDefs ≠ ∅ then return true
13             end
14             return false
           /* return true if any use of x is live at any use of y that in
              turn is live at node I                                         */
15         Function UseOrder(I):
16             yUses ⟵ {I_y | (y, I_y) ∈ U[I]}
17             foreach node I_y in yUses do
18                 xUses ⟵ {I_x | (x, I_x) ∈ U[I_y]}
19                 if xUses ≠ ∅ then return true
20             end
21             return false
22         foreach node I in CFG do
23             if DefOrder(I) ∧ UseOrder(I) then return true
24         end
25         return false
       // construct buffer interference graph
26     foreach variable pair (x, y) in P do
           // add edge (x, y) to E if x and y interfere
27         if Blocking(x, y) ∨ Blocking(y, x) then
28             E ⟵ E ∪ {(x, y)}
29         end
30     end
       // return buffer interference graph
31     return (V, E)
```

the corresponding variables interfere (see Algorithm 3). Any coloring heuristic may now be used to assign colors (output buffers or PUs) to vertices (variables) such that interfering variables are always assigned to different output buffers or PUs. This guarantees that when command instructions are considered in program order for move code generation, relevant operand values will always be found at the head of respective output buffers for transportation to destination input buffers. Furthermore, a coloring of the buffer interference graph will guarantee that multiple operands of the same command instruction are mapped to different output buffers. Note that live use tuples analysis (Algorithm 2) returns a set of live use tuples at the entry of each command instruction I (in[I]). Now consider for example, the assignment instruction I := $y = x_1 \odot x_2$. Clearly, both $I \in \mathsf{U}_I(x_1)$ and $I \in \mathsf{U}_I(x_2)$. Therefore, from 5.7, the entry of instruction I will serve as a program point that reaches a use of x_1 (I) via a use of x_2 (again I) and vice versa. Thus, it will be determined that x_1 and x_2 interfere irrespective of the order of the definitions of x_1 and x_2.



(a) *A program* **(b)** *Live ranges* **(c)** *Use intervals*

Figure 5.7.: *Live ranges and use intervals in a program*

5.2.3. Remarks

The buffer assignment is based on more liberalized rules compared to the traditional assignment of variables to registers: The variables live at a program point are assigned to different registers. However, we can map two live variables to the same buffer. In addition to liveness, if the use intervals of variables also overlap, then they must be assigned to different buffers to avoid overhead in the generated move code (see Figure 5.7 for example). For the given program, since live ranges of variables x, y and z overlap, they must be assigned to different registers to execute classic register architectures. However, the use intervals of only y and z overlap so that only y and z are needed to be assigned to different buffers for an overhead-free execution in SCAD archi-

tectures. This is not surprising since buffers can accommodate different live values, albeit restricting access to only the one at the head of the buffer.

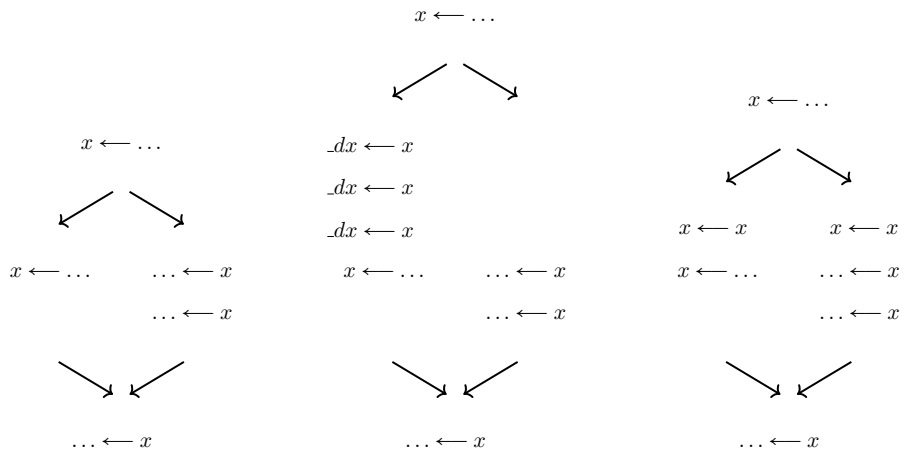
5.3. Balancing Variables

Any PU in a SCAD machine must produce as many copies of the computed result in its output buffer as there are uses of that value in the program. However, the number of uses of a value may vary depending on which control flow path is taken. Since the number of copies of values to produce must be determined at compile-time and the program control flow is only known at runtime, the SCAD compiler must transform the program in some way so that the number of uses of a value along each control flow path is equalized. In other words, all variables assigned to buffers must be balanced in the program where balancing is defined as follows:

Definition 5.12 ⟨ Balanced Variable ⟩
A variable x is balanced in a program if for every program point p, the number of uses of x in each path from p to the end node I_e are equal.

Equivalently, a variable is said to be 'balanced' in a program if the number of uses of that variable is uniquely determined at all program points. For SCAD compilation, it is necessary to balance variables so that the number of copies to produce is uniquely determined for each variable at compile time. Consider for example, the conditional execution shown in the control-flow graph in Figure 5.8a. If the left control flow path is taken, the value of variable x from the definition above the control flow split point is not used since x is redefined in the left path. If the right control flow path is taken instead, the value of x is used 3 times (2 uses in the right path and 1 use after the control flow join point). However, a predetermined number of copies must be produced in the output buffer of a PU when x is defined above the split point, and all copies must be consumed irrespective of the future control flow so that there are no stray values (values not accounted for).

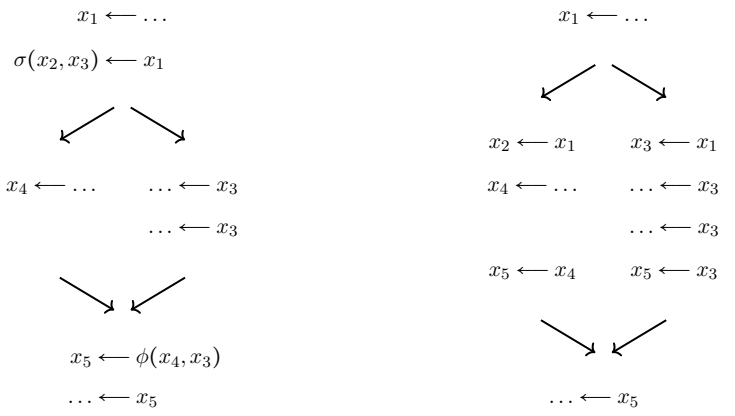
We propose two approaches to balance any variable x in a program: (1) Dummy uses of variable x in the form of assignments $\_dx \longleftarrow x$ to dummy variable $\_dx$, are introduced as shown in Figure 5.8b. Note that x is now used 3 times along both left and right paths so that the first definition of x can now safely produce 3 copies. The dummy assignments are translated to move instructions to special address *null* so that these values are simply *discarded* by the SCAD machine at runtime (given in Section 5.4). The algorithm to balance variables in programs by introducing dummy assignments is presented and discussed in Section 5.3.1. (2) Alternatively, we introduce copy assignments of variable x to itself ($x \longleftarrow x$) in the left and the right control flow paths as shown in Figure 5.8c. Note that x is now used 1 time along each path so



(a) *not balanced* **(b)** *dummy assignments* **(c)** *copy assignments*

Figure 5.8.: *Balancing variables by dummy assignments and copy assignments*

that the first definition of x needs to produce only 1 copy. It is easy to see that the elimination of σ-functions in SSI programs will provide exactly this transformation. The program in Figure 5.8a after SSI transformation is shown in Figure 5.9a. A naive elimination of $\sigma(x_2, x_3) \longleftarrow x_1$ places copy assignments $x_2 \longleftarrow x_1$ and $x_3 \longleftarrow x_1$ in the left and the right paths, respectively, as shown in Figure 5.9b, so that x_1 is used once along each path. We prove in Section 5.3.2 that all variables are inherently balanced in SSI programs and therefore, directly use the *SSI transformation* for balancing programs [Schn18].



(a) *SSI program* **(b)** *After elimination*

Figure 5.9.: *Balancing variables by SSI transformation*

The dependence of the number of variable uses on the number of loop iterations poses an additional challenge in balancing variables. If there exists

a path in a loop body where a variable is only used, the number of copies of this variable to produce will depend on the number of loop iterations. This is not desirable since the number of loop iterations is often unknown at compile time. Moreover, even if statically known, these are often large numbers, so that instantiating these many copies of a variable will require huge buffer sizes in SCAD. Due to this reason, all variable uses in loops must be bounded where bounding is defined as follows:

Definition 5.13 ⟨ Bounded Variable ⟩
A variable x is bounded in a loop body $(I_{ls}, \ldots, I_{le})$ if there does not exist any path from loop start I_{ls} to loop end I_{le} where x is only used.

We prove in Section 5.3.2 that the SSI transformation bounds all unbounded variables in a given program so that there is no need to explicitly bound variables when balancing a program by SSI transformation. However, when a program is balanced by discarding copies (i.e., by introducing dummy assignments), all unbounded uses of any variable x must be explicitly bounded. To this end, we introduce an additional variable x' to bound the use of x in a loop. Consider for example, the loop in Figure 5.10a with an unbounded use of x. The new variable x' is introduced as shown in Figure 5.10b. Note that both x and x' are now bounded since they are defined in the loop body, and thus the number of copies to produce for each definition of x and x' is independent of the number of loop iterations. Each definition in this example may produce one copy. The algorithm to bound variables in programs is given in Section 5.3.1.

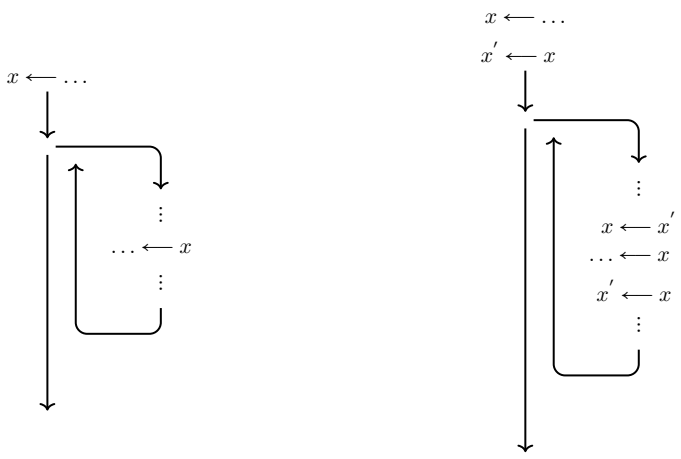
5.3.1. Balancing by Discarding Copies

Recall that MiniC is a structured programming language that allows programmers to only use structured control flow constructs. Therefore, modifications of the abstract syntax tree (AST) representation of a program are guaranteed to retain the block structure of the program, while modifications of the simpler control-flow graph (CFG) representation (i.e., at the intermediate command language level) may lose the block structure of the program if not carefully done. Due to this reason, we decide to implement bounding and balancing of variables at the MiniC language level and not at the intermediate command language level.

Statements in MiniC are defined recursively (see Table 5.1). Algorithm 4 implements the bounding of any variable x in any MiniC statement S recursively from innermost to outermost loops. Note that for each statement type S, the use of x is first bounded in the contained statements if any, followed by the bounding of x in statement S if S is a loop statement. For example, for the while statement $S :=$ while(φ,S_l), the use of x is first bounded in the loop body statement S_l to obtain S_l^b. The variable x is then bounded in this while

Algorithm 4: Bound variable x in loops in statement S

```
1  Function BoundLoops(S, x):
2      switch S do // statement type
3          case seq(S1,S2) do                                  // sequence:  S1; S2
4              S1^b ⟵ BoundLoops(S1, x)
5              S2^b ⟵ BoundLoops(S2, x)
6              return seq(S1^b,S2^b)
7          end
8          case cond(φ,S1,S2) do            // conditional:  if(φ) S1 [else S2]
9              S1^b ⟵ BoundLoops(S1, x)
10             S2^b ⟵ BoundLoops(S2, x)
11             return cond(φ,S1^b,S2^b)
12         end
13         case while(φ,Sl) do                          // while loop:  while(φ) Sl
14             Sl^b ⟵ BoundLoops(Sl, x) // bound inner loops
15             if x is unbounded in Sl^b then
16                 Sl^b ⟵ seq(asg(x, x'), seq(Sl^b, asg(x', x)))
17                 return seq(asg(x', x), while(φ,Sl^b))
18             end
19             return while(φ,Sl^b)
20         end
21         otherwise do                                        // other statements
22             return S
23         end
24     end
```



(a) *Unbounded variable use* **(b)** *After bounding*

Figure 5.10.: *Bounding variable use in loop*

loop by introducing new assignments $\text{asg}(x, x')$ and $\text{asg}(x', x)$ using variable x' as explained in Figure 5.10. The do-while and for-loop statements are handled similarly and are not shown in Algorithm 4. Moreover, the for loops and do-while loops are easily expressed in terms of while loop as follows:

Loop	Corresponding while loop
for $(i = \tau_l \ldots \tau_u)$ S_l	$i = \tau_l$; **while**$(i < \tau_u)$ $\{S_l;\ i = i + 1\}$
do S_l **while**(φ)	S_l; **while**(φ) S_l

To ensure that all variables are bounded in all loops in the program, variable uses are bounded in function statements in the MiniC program followed by the main program statement.

After bounding variable uses in loops, balancing of variables is performed by a bottom-up traversal of the AST representation of the program. The bottom-up traversal is necessary since balancing any variable x in any statement S requires knowledge of the number of uses of x at the exit of S. This is clear from the example in Figure 5.8b, where three dummy assignments are placed in the left path since x is used twice in the right path and once after the exit of the conditional statement (i.e., after the control flow join point). The recursive function that implements the bottom-up traversal of AST to balance variable x in statement S given the number of uses of x at the exit of S (u_{exit}) is given in Algorithm 5. Note that for each statement type S, x is first balanced in the contained statements if any, before balancing x in this statement S. For a sequence statement $\text{seq}(S_1, S_2)$, statement S_2 is first balanced followed by S_1. This implements a bottom-up traversal. Clearly, dummy assignments or discarding of copies are needed only for those statements that contain control flow split points. Finally, the balanced statement and the use count of x at the entry of the balanced statement (u_{entry}) are re-

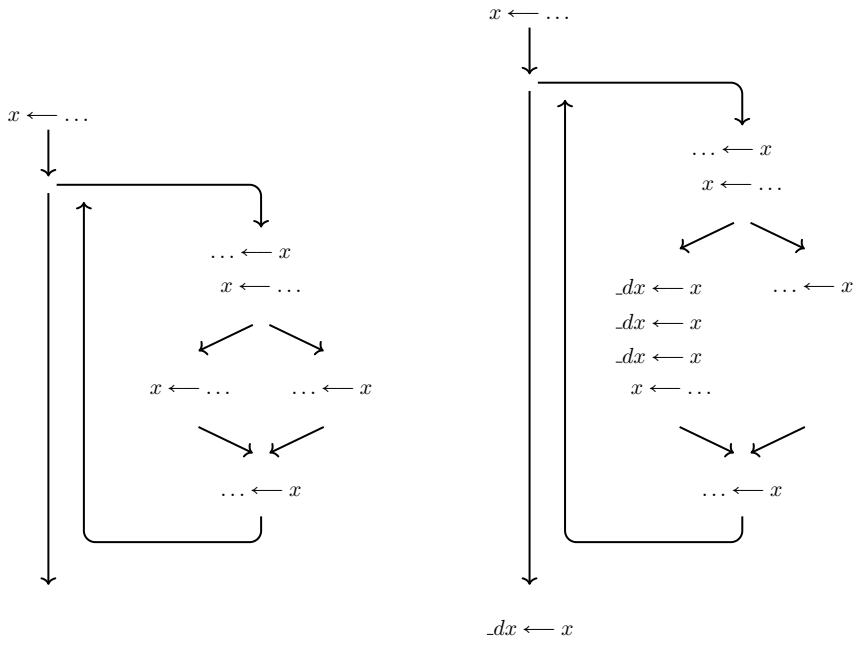
turned, so that u_{entry} serves as the number of uses of x at the exit of the next statement considered in the bottom-up traversal. Note that u_{entry} must be evaluated even for atomic statements that do not require balancing. For the assignment statement $\text{asg}(\lambda, \tau)$, u_{entry} is either set to 0 if x is defined by the assignment or evaluated by summing the use counts of x in the left-hand side expression λ and the right-hand side expression τ. This is done similarly for the function call statement $\text{fun}(\tau_1, \ldots, \tau_n)$ and the return statement $\text{return}(\tau)$. For the sequence statement $S := \text{seq}(S_1, S_2)$, the number of uses evaluated at the entry of S_2 is provided as the use count at the exit of S_1 when balancing the statement S_1. The number of uses of x subsequently evaluated at the entry of S_1 is returned.

Consider the balancing of variable x in a conditional statement $S := \text{cond}(\varphi, S_1, S_2)$ in Algorithm 5 in the context of the example in Figure 5.8b. The number of uses at the exit of S is $u_{exit} = 1$. Assume that the left path in Figure 5.8b corresponds to the if statement S_1 and the right path corresponds to the else statement S_2. The statements S_1 and S_2 are balanced and respective entry use counts $u_{S_1} = 0$ and $u_{S_2} = 3$ are evaluated. Next, the use count at the control flow split point is fixed as the maximum of the 'if' path and the 'else' path, and the appropriate number of copies are discarded in the other path. In this case, $u_{S_2} - u_{S_1} = 3$ copies of the variable x are discarded in the left (if) path. Finally, the number of uses of x in the condition expression φ and the use count of x at the split point are added to determine the use count at the entry of the conditional statement S. The main idea behind balancing a variable use this way is to have as many copies of the variable as its maximal use count at the control flow split point and then discard unnecessary copies depending on the path taken.

Observe that the use count of a variable at a control flow split point in a conditional statement $S := \text{cond}(\varphi, S_1, S_2)$ does not affect the balancing of that variable in its contained statements S_1 and S_2. However, due to back edges in loops, the use count of a variable at the split point will affect the balancing of that variable in its contained loop body statement. Due to this reason, for the loop statements, we first determine the use count at the split point and then balance variables in the loop body statement in a second phase. In more detail, consider the balancing of variable x in the while statement $S := \text{while}(\varphi, S_l)$ in the context of example in Figure 5.11a. The variable x is not used after the while loop, thus $u_{exit} = 0$. To balance x in the contained loop body statement S_l, we have to derive the number of uses of x at the exit of S_l, which is dependent on the use count at the split point. At the control flow split, the program control flows either to the exit of the loop statement S or to the entry to the loop body statement S_l. Therefore, to determine the maximal use count of x at the split point, besides u_{exit}, we have to evaluate the use count at the entry to the loop body statement denoted by u_{S_l} in Algorithm 5. Note that the number of uses of x at the exit of S_l, will not impact the use count at the entry of S_l because variable uses in loops are already bounded, which guarantees that x is defined in all paths in S_l where it is used. Now invoking the balance statement function with S_l and x as

Algorithm 5: Balance variable x in loop bounded statement S

```
1  Function BalanceStmt(S, x, u_exit):
2      Function Uses(e):            // count number uses of x in expression e
3          return number uses of x in e
4      Function Discard(n):         // assignment statements to discard x
5          return sequence of n discard assignments asg(_dx, x)
       // balance statement S
6      switch S do // statement type
7          case asg(λ,τ) do                                  // assignment:  λ = τ;
8              if x is defined by S then return (S, 0)
9              else return (S, Uses (λ) + Uses (τ))
10         end
11         case seq(S1,S2) do                                // sequence:  S1; S2
12             S2^b, u_S2  ⟵ BalanceStmt(S2,x,u_exit)
13             S1^b, u_entry ⟵ BalanceStmt(S1,x,u_S2)
14             return (seq(S1^b,S2^b), u_entry)
15         end
16         case cond(φ,S1,S2) do          // conditional:  if(φ) S1 [else S2]
17             S1^b, u_S1 ⟵ BalanceStmt(S1,x,u_exit)
18             S2^b, u_S2 ⟵ BalanceStmt(S2,x,u_exit)
19             if u_S1 > u_S2 then S2^b ⟵ seq(Discard(u_S1 − u_S2), S2^b)
20             else S1^b ⟵ seq(Discard(u_S2 − u_S1), S1^b)
21             u_entry ⟵ ((u_S1 > u_S2)?u_S1 : u_S2) + Uses(φ)
22             return (cond(φ,S1^b,S2^b), u_entry)
23         end
24         case while(φ,Sl) do                       // while loop:  while(φ) Sl
25             _ ,u_Sl ⟵ BalanceStmt(Sl,x,0)
26             u       ⟵ ((u_exit > u_Sl)?u_exit : u_Sl) // control flow split
27             Sl^b, _ ⟵ BalanceStmt(Sl,x, u + Uses(φ))
28             u_entry ⟵ u + Uses(φ)
29             if u_exit > u_Sl then
30                 return (while(φ,seq(Discard(u_exit − u_Sl), Sl^b)), u_entry)
31             else
32                 return (seq(while(φ,Sl^b), Discard(u_Sl − u_exit)), u_entry)
33             end
34         end
35         case fun(τ1,...,τn) do                 // function call:  f(τ1,...,τn);
36             return (fun(τ1,...,τn), u_exit+ Uses(τ1) +...+ Uses(τn))
37         end
38         case return(τ) do                                 // return :  return τ
39             return (return(τ), u_exit + Uses(τ))
40         end
41         otherwise do report invalid statement
42     end
```



(a) *Not balanced* **(b)** *Balanced*

Figure 5.11.: *Balancing variables in loops by discarding copies*

arguments will return the use count of x at the entry of S_l, and a balanced S_l which is ignored. In this example, $u_{S_l} = 1$ since x is used once in the loop body before it is defined. The use count u at the control flow split point is finally fixed to the maximum of u_{S_l} and u_{exit}. The number of uses of x at the exit of S_l is now easily obtained by adding the use count at the control flow split and the use count of x in the condition expression φ. This value can now be used to balance the loop body statement S_l. In this example, balancing x in the loop body will yield three dummy assignments in the left path of the conditional statement inside the loop body. Finally, variable x is balanced in the loop statement $S := \text{while}(\varphi, S_l)$ by discarding the appropriate number of copies after the loop exit or at the entry to the loop body. One copy of x is discarded after the loop exit in the given example.

5.3.2. Balancing by SSI Transformation

In this section, we present the SSI transformation as an alternative means to balance variables in programs by proving that all variables in an SSI program are balanced by construction. To this end, we first prove that variables are bounded in loops in an SSI program and then prove that they are balanced.

Lemma 5.1 (Bounded SSI) *All variables in a SSI program are bounded in all loops in the program.*

Proof Assume on the contrary that a variable x is not bounded in a loop $(I_{ls}, \ldots, I_{le})$. Therefore, from Definition 5.13, there must exist a path P where x is not defined and a sequence of $n \geq 1$ uses of variable x, denoted by $(I_{u_1}, \ldots, I_{u_n})$, appear. Let p denote a program point at the exit of the last use I_{u_n}. At program point p, the use I_{u_1} is upward-exposed via the back edge (I_{le}, I_{ls}) of the loop. Also, any use of x after the loop (pseudo-use at program end node I_e in case x is not used after the loop) is also upward-exposed at the program point p. This violates the unique upward-exposed use property 5.4 of programs in SSI form. Thus, contradicting our assumption that the program is in SSI form. ■

Theorem 5.1 (Balanced SSI) *All variables in a SSI program are balanced.*

Proof Lemma 5.1 states that all variables in a program in SSI form are bounded. Therefore, it is enough to consider a single iteration to count the number of uses of variables in any loop. In other words, back edges can be ignored. Assume now that a variable x is not balanced at any point p in the given program. Let I_d be the unique definition of x (see the unique definition property 5.1 of SSI programs). Clearly, program point p must appear after I_d in the command program since the use count of x at all program points before I_d is zero. Since x is not balanced at p, there exists at least two paths P_i and P_j from the program point p to the end node I_e with a different number of uses of x. Without loss of generality, assume that any use I_u is a part of path P_i, but not a part of path P_j. Since there exists a path from p to I_u and I_d *dom* I_u (see the dominance property 5.2 of SSI programs), there must exist a path from I_d to p. Therefore, there exists a path from the unique definition I_d to p to the end node I_e (along the path P_j), where the use I_u does not appear. This violates the post-dominance property 5.3 of SSI programs, which states that each use of x post-dominates its unique definition, thus contradicting our assumption that the program is in SSI form. ■

5.4. Move Code Generation

Once the buffer assignment (assignment of instructions to PUs) and the number of copies to produce for each variable definition are determined, it is easy to generate the final move code by considering the command instructions in

program order. Table 5.7 lists command instructions and the corresponding move instructions, where $u(x)$ denotes the PU that produce x, n denotes the number of copies to produce and the memory address of any array variable x is denoted by $adr(x)$. The operand moves corresponding to a command instruction may be ordered randomly since the buffer assignment guarantees that these operands are assigned to different output buffers or PUs. For example, x_1 and x_2 in the assignment command instruction $y = x_1 \odot x_2$ may be moved to left and right input buffers, respectively, of $u(y)$ in any order. Since our SCAD simulator currently only supports primitive binary arithmetic and logic operations supported by the command language (denoted by $\odot$), *dup* and *swap* opcodes are not yet supported. Therefore, a copy command $y = x$ is currently implemented by adding x to the immediate value 0 in $u(y)$, incurring an additional transportation overhead for the moving immediate value 0 to the right input buffer of $u(y)$.

Command Instr	Corresponding SCAD Move Instructions			
$y = x$	$u(x)@o \rightarrow u(y)@l$;	$0 \rightarrow u(y)@r$;	$+ \rightarrow u(y)@op$;	$n \rightarrow u(y)@cp$;
$y = x_1 \odot x_2$	$u(x_1)@o \rightarrow u(y)@l$;	$u(x_2)@o \rightarrow u(y)@r$;	`op`$\rightarrow u(y)@op$;	$n \rightarrow u(y)@cp$;
$y = x_1[x_2]$	$adr(x_1) \rightarrow u0@l$;	$u(x_2)@o \rightarrow u0@r$;	$+ \rightarrow u0@op$;	$1 \rightarrow u0@cp$;
	$u0@o \rightarrow lsu@l$;		`ld`$\rightarrow lsu@op$;	$1 \rightarrow lsu@cp$;
	$lsu@o \rightarrow u(y)@l$;	$0 \rightarrow u(y)@r$;	$+ \rightarrow u(y)@op$;	$n \rightarrow u(y)@cp$;
$y[x_1] = x_2$	$adr(y) \rightarrow u0@l$;	$u(x_1)@o \rightarrow u0@r$;	$+ \rightarrow u0@op$;	$1 \rightarrow u0@cp$;
	$u0@o \rightarrow lsu@l$;	$u(x_2)@o \rightarrow lsu@r$;	`st`$\rightarrow lsu@op$;	
goto i	`i`$\rightarrow pc$;			
if x **goto** i	$u(x)@o \rightarrow cu@c$;	`i`$\rightarrow cu@then$;	$pc \rightarrow cu@else$;	

Table 5.7.: *Command instructions and corresponding SCAD move instructions*

Currently, the SCAD compiler does not perform any array index analysis so that all array accesses and assignments are directed to the main memory. Furthermore, we reserve PU 0 ($u(0)$) for evaluating the memory address of array elements, which is then transported to the load-store unit (lsu) for loading or storing array elements. See the array access ($y = x_1[x_2]$) and the array assign ($y[x_1] = x_2$) command instructions in Table 5.7. Unconditional branching (**goto** i) is implemented by moving the branch target i to the special address pc that denotes the program counter. For conditional branching (**if** x **goto** i), the branch target i is moved to 'then' input lane of the control unit followed by moving the branch condition produced in $u(x)@o$ to the condition input lane of the control unit. Though we list the move instruction $pc \rightarrow cu@else$ for the conditional branching, this is automatically done by the control unit. For more details on how the control flow is implemented in SCAD, see Section 2.1.3. Finally, dummy assignments of the form $\_dx = x$ are translated to move instruction $u(x)@o \rightarrow null$ that simply discard a copy of x from the head of the output buffer $u(x)@o$. Appendix A lists all intermediate results from the compilation (input MiniC program, program after bounding variables in loops, program after balancing variable uses, command program, buffer interference graph and final move program) of a simple benchmark.

5.5. Remarks on Buffer Size

Given enough space in the output buffers, a single slot is sufficient in input buffers in a SCAD processor to guarantee that the move program generated by the heuristic discussed in the previous sections will successfully execute without deadlocking (i.e., without control unit stalling forever due to full buffers). This is attributed to the following characteristics of the heuristic: (1) The operand moves of each command instruction are scheduled consecutively. This means, to execute an operation on a PU, the corresponding data transports to different input buffers of the PU are registered consecutively before registering the data transports for the next operation. (2) Command instructions are considered in program order for move code generation. This means, when data transports are registered to execute an operation on a PU, all data transports necessary to produce the relevant operand values for this operation would have already been registered. Therefore, even if the control unit should stall at this point of time, it is guaranteed that the operand values will be eventually transported to the input buffers and are consumed by the PU, freeing up space in its input buffers.

5.6. Experiments

We experimentally compare the execution of benchmarks by direct instruction communication with the execution by using registers to hold intermediate results. For the former, *queue-based move programs* are generated using the heuristic discussed in the previous sections. The Chaitin-Briggs heuristic [Chai04] is used to color the buffer interference graph so that variables are assigned to a minimal number of PUs. Recall that the queue-based code generation heuristic produces *resource constrained* move programs that use a minimal number of PUs to execute benchmarks. The *register-based move program* is generated as follows: Allocate variables to a minimal number of registers by coloring the register interference graph, again using the Chaitin-Briggs heuristic [Chai04]. Assign instructions to PUs by list-based scheduling that attempts to minimize the execution time using the given resources. The same number of PUs are then used in both queue-based and register-based move code generation. With the PU and register assignments, the move program is easily obtained by generating the following moves for each command instruction: move the operands from the registers to the corresponding input buffers of the PU that executes this instruction, and move the result from the output buffer to the respective mapped register. In both register-based and queue-based code generation, the array variables are mapped to the main memory and PU 0 is reserved for computing array element addresses.

We use a cycle-accurate SCAD simulator[3] to execute both queue-based and register-based move programs. To study the impact of the number of ports in the register file, register-based move programs are executed on both (1) a SCAD machine with a single-ported register file, denoted by REG-MIN, and

[3] `https://es.cs.uni-kl.de/tools/teaching/ScadSim.html`

(2) a SCAD machine with q multi-ported register file with as many ports as registers, denoted by REG-MAX. Once all intermediate results are accommodated in the respective local storage, the sizes of input and output buffers have a similar impact on the execution time of both register-based and queue-based move programs, in that the control unit will stall if not enough space is available in buffers to register the move instruction. Therefore, we use the same buffer sizes to execute both move programs. To this end, execution times are measured for two configurations of buffer sizes: (1) First, we use a *minimal* number of entries in the input and output buffers required for the queue-based move program (i.e., the move program generated by the heuristic where variables are balanced by discarding copies). Currently, the queue-based code generation heuristic does not consider buffer sizes. Therefore, we check if a given buffer size is sufficient or not for the successful execution of the program in the SCAD simulator. It is already known from the heuristic that the required minimal input buffer size is 1, given large enough output buffers. The sizes of input buffers are fixed to 1 to determine the required minimal output buffer size by simulation. (2) Next, we provide *enough* space (size 20) in input and output buffers so that the execution times of both queue-based and register-based move programs are compared, avoiding a differing impact of minimal buffer sizes, if any.

Program	Label	Input
factorial	fact	$\{1,\ldots,12\}$
fibonacci	fib	$\{1,\ldots,20\}$
sumup	sumup	add numbers $\{1,\ldots,500\}$ in a loop
euclid	euclid	compute gcd of pairs $\{101,1001\},\ldots,\{150,1050\}$
heron	heron	compute square root of first 20 perfect squares
daxpy	daxpy	vector length 100
eratosthenes sieve	sieve	determine prime numbers in $\{1,\ldots,100\}$
insertion sort	insort	reverse sorted array $[15,\ldots,1]$
bubble sort	bbsort	reverse sorted array $[15,\ldots,1]$
matrix multiply	matmul	4×6 and 6×8 matrices
image convolution	imgconv	6×6 image and 3×3 kernel

Table 5.8.: *List of benchmarks*

Table 5.8 lists benchmarks (with corresponding inputs used for experiments), that comprises of both regular parallelism (daxpy, matrix multiply, image convolution) and irregular (other benchmarks) parallelism. The simulated SCAD machine has one load-store unit (LSU) for memory accesses, and the other PUs are capable of executing any standard binary operation. Arithmetic, comparison, and bitwise operations are configured to have a unit delay, except for multiplication and division operations that take 8 and 20 cycles, respectively, to execute. Memory accesses (read and write) take 50 cycles. The DTN is configured to have a latency of $\log_2 p$ where p is the number of processing units. This is usually the number of stages required in a multi-stage interconnection network to communicate values from p senders to p receivers.

Furthermore, all components (PUs, LSU, and DTN) are pipelined.

Recall that two program transformations were discussed to balance variables in a program for move code generation: balancing by discarding copies and by SSI transformation. In the rest of this section, we refer to the former as NORMAL balanced move program and the latter as SSI balanced move program. We also abbreviate the execution of the NORMAL balanced move program on the SCAD machine by NORMAL execution in the following explanations. Similarly, SSI, REG-MIN, and REG-MAX executions. The number of cycles taken for different execution types using *minimal* and *enough* buffer sizes are shown in Figures 5.12 and 5.13, respectively. Note that the cycles for different execution types relative to one another appear similar in both figures, with only a few minor differences discussed later. This affirms our postulate that sizes of input and output buffers have a similar impact on the execution of both queue-based and register-based move programs. The execution of a NORMAL balanced move program on SCAD is the most efficient in terms of the number of cycles since all register accesses are bypassed in comparison to the execution of the corresponding register-based move programs. The REG-MIN execution is considerably slower due to the contention of simultaneous register accesses on the single port of the register file. Clearly, register accesses and the number of register file ports have a huge impact on the execution time, particularly for programs exhibiting regular parallelism (daxpy, matrix multiply, image convolution) since there are more parallel accesses (writes and reads) of intermediate values.

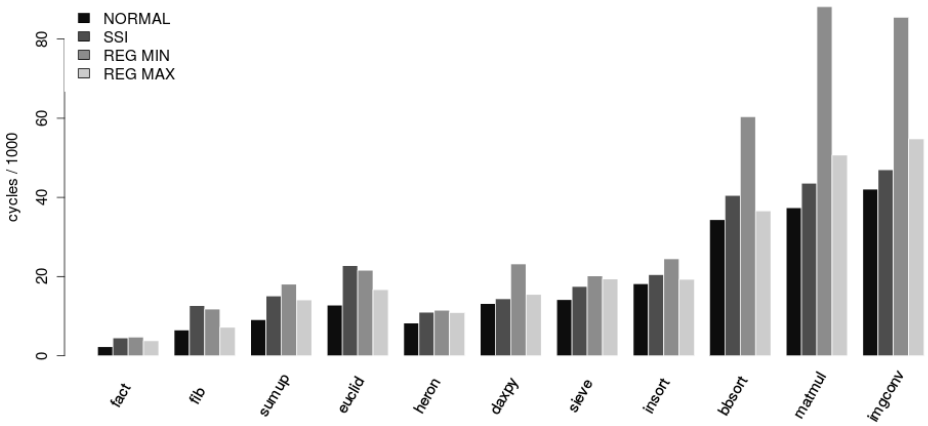


Figure 5.12.: *Execution time using minimal buffer sizes*

Notice that unlike the optimal code generation experiments, the REG-MAX (that uses an ideal register file) execution cycles of many benchmarks are comparable to that of the corresponding NORMAL balanced move programs in SCAD. This is because the experiments using optimal code only considered basic blocks as programs. The control flow in the above benchmarks introduces additional overhead in the execution by direct instruction communication.

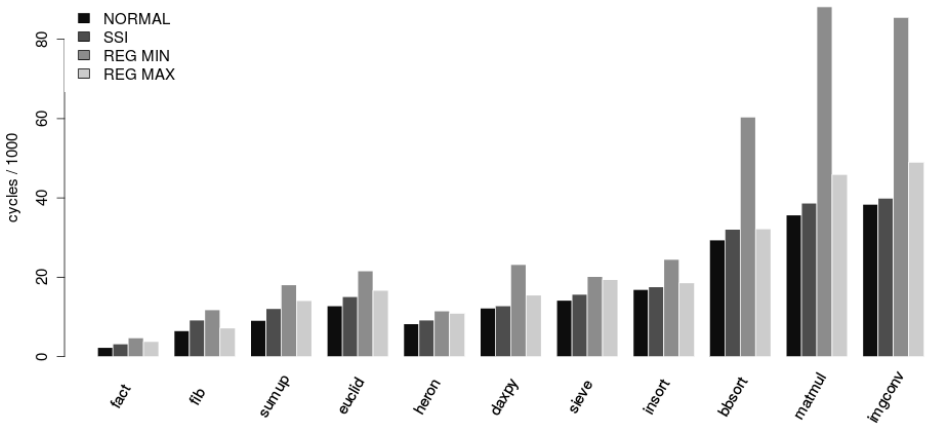


Figure 5.13.: *Execution time using enough buffer sizes*

First, in the form of new copy assignments for bounding variables. Second, as additional cycles to discard unnecessary copies introduced by the balancing transformation. Furthermore, the copy assignments of the form $y \leftarrow x$ incur extra overhead for the direct instruction communication since x need to be rotated via the input buffer to the output buffer of the PU to which y is assigned. When using registers to hold values, this simply means copying one register's content (or an immediate value) to another register. The impact of these overheads is apparent from Figure 5.14 that shows the total number of firings of PUs. The numbers of PU firings for most benchmarks are considerably more in the execution of NORMAL balanced move programs compared to the corresponding register-based move programs. It is not difficult to see that with programs offering more ILP (possibly by various techniques such as loop unrolling, superblock and hyperblock formations) and with larger numbers of PUs in SCAD, the negative effect of the overhead will quickly amortize since the overall overhead will then be distributed between more PUs. This is seen to a slight extent for the matrix multiplication and the image convolution benchmarks in Figures 5.12 and 5.13. Unfortunately, we do not yet have a heuristic that maximizes the use of ILP in SCAD with any given number of PUs. This will be addressed in future work.

As expected, SSI balanced move programs of all benchmarks take longer to execute compared to NORMAL balanced move programs (Figures 5.12 and 5.13). Balancing by the SSI transformation introduces extra overhead in the form of copy assignments used at the control flow split points. Again, this is clearly seen in Figure 5.14 in that PUs in the SCAD machine fire more often when executing SSI balanced move programs. However, NORMAL balancing will require larger buffers to successfully execute generated move programs since maximal copies are produced when values are computed. This space-time trade-off in computation is reflected in Figure 5.15 that shows the minimal number of resources required for different execution types. Clearly, SSI

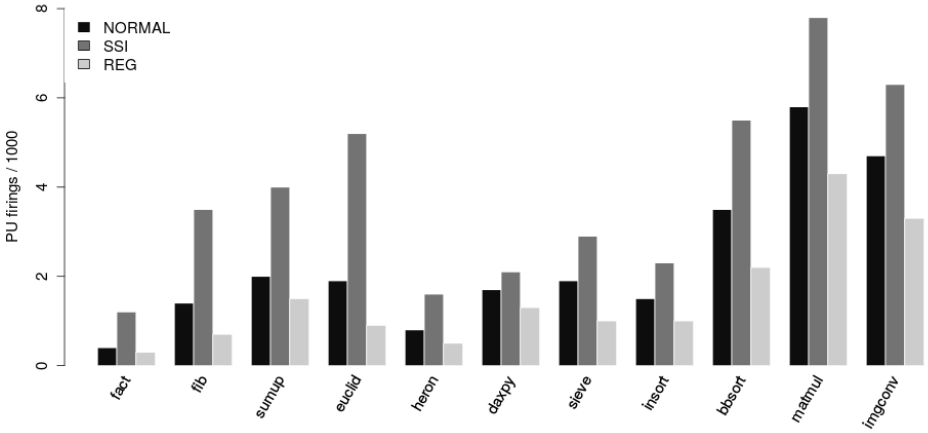


Figure 5.14.: *Number of PU firings*

balanced move programs require less buffer size for all benchmarks. Minimal output buffer sizes are shown with input buffer sizes fixed to 1. Although we configured the same buffer sizes for all PUs in SCAD, each PU will not use all slots in its buffers. In fact, only a few (one or two) PUs will utilize all buffer slots because our heuristic tries to assign as many instructions as possible to the same buffer to use a minimal number of PUs (i.e., resource-constrained code generation). In addition to minimal PUs (required by the NORMAL balanced benchmark programs), minimal registers required by the register-based compilation are also given in Figure 5.15. However, we will need to measure the hardware resource usage for an exact comparison of the register file size and the buffer size, which is not covered in this thesis.

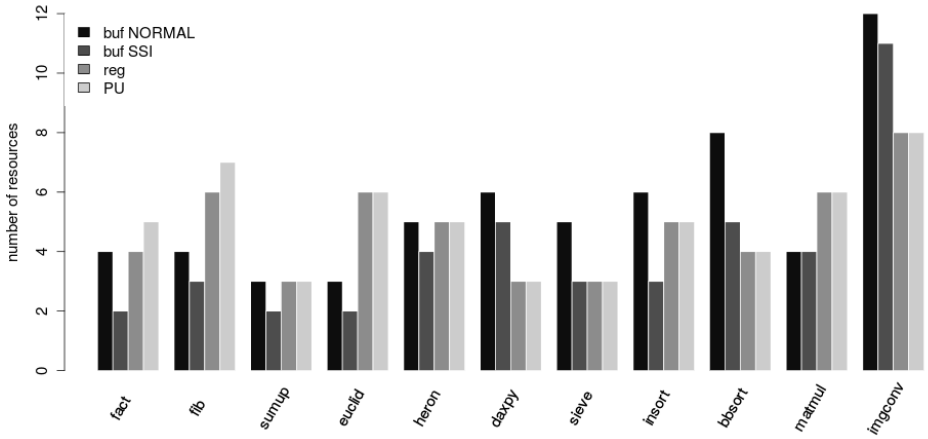


Figure 5.15.: *Minimal resource usage*

In Figure 5.12, the execution of the SSI balanced programs in SCAD has a lower runtime compared to REG-MIN executions for most benchmarks except fibonacci and euclid. This is again more apparent for parallel benchmarks like daxpy, matrix multiply, and image convolution. For these benchmarks, it can be observed that the SSI execution even outperforms the execution in REG-MAX (using ideal register files). Now compare the SSI execution with the execution in REG-MIN and REG-MAX in Figure 5.13. It is interesting to see that provided enough buffer size, the execution by direct instruction communication using the SSI transformation is faster than the execution by using registers for all benchmarks (except for the fibonacci benchmark where the SSI execution lags behind REG-MAX execution). This is because SSI balanced move programs have more move instructions due to the additional copy assignments introduced by the SSI elimination. With more move instructions and less buffer size, the control unit in SCAD would frequently stall. Some of these control unit stalls are avoided in experiments with enough buffer size.

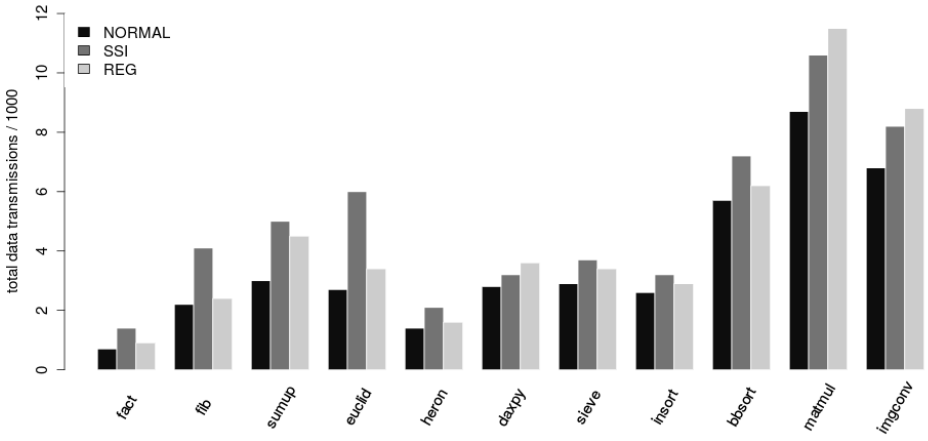


Figure 5.16.: *Total number of data transmissions*

The total number of data transmissions by all hardware units for each execution type is shown in Figure 5.16. The NORMAL execution has the least number of data transmissions. Execution in REG (REG-MIN or REG-MAX) contain comparatively more communication of values due to additional writes to registers. Even more values are communicated by the interconnect in the SSI execution due to additional copying (or duplicating) of values by the SSI elimination. However, for more parallel benchmarks (daxpy, matrix multiply, and image convolution), the overall data communication in the SSI execution is less than that in the REG execution. This is reflected as a better performance of these benchmarks by the SSI execution compared to the REG-MIN and the REG-MAX executions (see again Figures 5.12 and 5.13). It is again interesting to observe that if programs offer more ILP, the negative impact of register accesses on the performance aggravates while that of the overhead amortizes quickly. We expect to confirm this observation in the future by

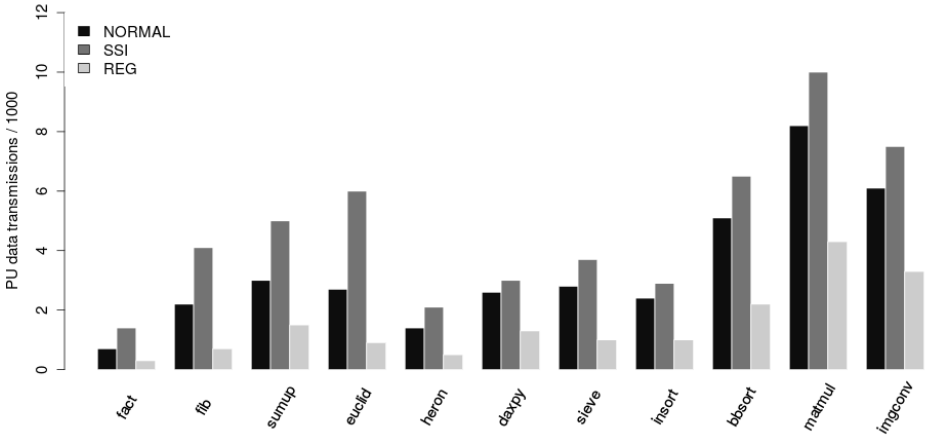


Figure 5.17.: *Number of data transmissions by PUs*

developing heuristics to generate SCAD code that will maximize the use of ILP. The total number of data transmissions by PUs for each execution type is shown in Figure 5.17. As expected, the overall data communication in the execution by direct instruction communication is distributed among PUs. In contrast, all data traffic is routed through a limited number of register file ports in the execution where registers are used to hold intermediate values.

Notice that the execution of register-based move code resembles the execution in VLIW processors, where each instruction reads operand values from registers and writes its result to a register. Similar to the bundling of instructions in VLIW, independent move instructions can be bundled in SCAD. Also, the execution in SCAD can benefit from various VLIW compiler techniques to find enough independent instructions for concurrent execution. Therefore, it is not unfair to consider the above experiments as a comparison of the execution by direct instruction communication in SCAD and the execution in a VLIW processor corresponding to REG-MAX{MIN}.

Chapter 6

Related Work

In this chapter, we review processor architectures concerning the use of ILP with an emphasis on their capability to bypass register usage or implement direct instruction communication. We also mention aspects of SCAD that distinguish it from the reviewed architectures.

Superscalar processors [John91; SmSo95] execute typical reduced instruction set computer (RISC) instructions where operands and results of instructions are encoded by register addresses. Dynamically scheduled superscalar processors perform out-of-order execution of RISC programs by Tomasulo's algorithm [Toma67]. They fetch and decode multiple instructions at once filling a reservation station and a reorder buffer. Instructions in the reservation station whose operand values are available are dispatched to processing units (PUs) for execution. So, allocation of instructions to PUs (*instruction placement*) and firing times of PUs (*instruction issue*) are determined at runtime (see Figure 6.1). During the execution of an instruction, other instructions that consume the result of this instruction might still be decoded and put into the reservation station. Since any entry in the reservation station is a probable consumer instruction, each PU must broadcast its result to all reservation station entries. To that end, n PUs arbitrate on a single common data bus (CDB) that is snooped by all m reservation station entries. If any consumer instruction is not yet decoded when the result is broadcast, the result is communicated to this consumer instruction via the register. Therefore, the instruction window for direct instruction communication is determined by the size of the reservation station. However, scaling the reservation station size and the number of PUs in a superscalar processor is limited by the $n : 1$ arbitration of PUs on the CDB and the $1 : m$ broadcast from the CDB to the reservation station entries. In contrast, the input and output buffers of SCAD are scalable, and the data transport network (DTN) in SCAD establishes a unicast $n : n$ network where n is the number of PUs.

In superscalar processors, instructions still have to read their operands from registers in the program order in the decode stage. Each instruction has to write its result to a register in the write-back stage again in the program order (from the reorder buffer). The number of registers that can be read or

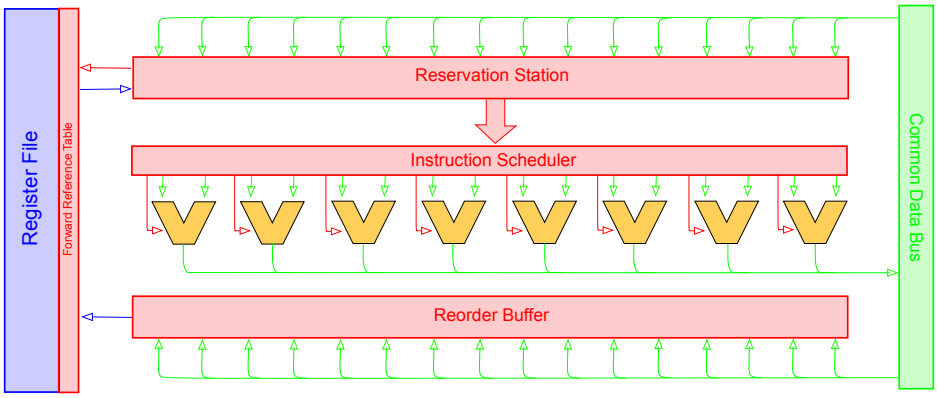


Figure 6.1.: *Execution framework of a superscalar processor*

written at any point of time is limited by the number of ports of the register file. Furthermore, compilers have to generate spill code to temporarily store intermediate results in memory if the number of programmer-accessible registers is not sufficient, thus reducing the use of ILP (see Figure 1.1). Increasing the number of registers is difficult since this number is directly encoded in the RISC instruction format. Other major drawbacks are the enormous power consumption and the limited scalability: In addition to executing programs, the processor also takes care of instruction scheduling in that the data dependencies of instructions are tracked, and instructions are allocated to PUs (instruction placement) at runtime. Superscalar processors rely on branch prediction to keep the PUs busy for control-flow intensive programs.

The *very long instruction word (VLIW) processors* [FERN84], contrary to superscalar processors, are quite simple machines. It is the responsibility of the VLIW compiler to track data dependencies of instructions, allocate instructions to PUs, and also determine PU firing times. So, both instruction placement and instruction issue are determined statically by the compiler. The number and latency of PUs are exposed to the compiler so that it can bundle independent instructions in a very long instruction word. The hardware simply executes each instruction in a bundle simultaneously. The general structure of VLIW architectures is shown in Figure 6.2. Many sophisticated techniques like trace scheduling [Fish81] and hyperblock scheduling [MLCH92] for acyclic regions, modulo scheduling (or software pipelining) [Lam88; Rau94] and loop unrolling [LaHw95] for cyclic regions (or loops), etc. are used by the VLIW compilers to find sufficient independent instructions from across basic blocks to utilize the available PUs. Statically scheduled VLIW processors have significantly less power consumption than dynamically scheduled superscalar processors. However, while the same binary programs can run on very different superscalar processors that share the same instruction set, the programs have to be compiled for each particular VLIW processor, which restricted these processor architectures to the field of embedded computing [FiFY05].

Since most VLIW architectures use typical RISC instructions, each instruc-

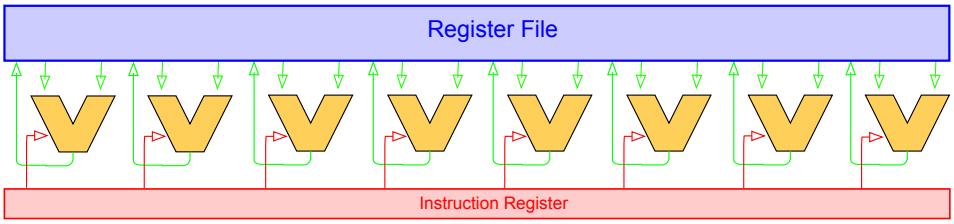


Figure 6.2.: *Execution framework of a VLIW processor*

tion in a very long instruction must read its operands from registers and write its result to a register. If the number of registers (again, this number is encoded in the RISC instruction format) is not sufficient, the VLIW compilers have to generate spill code to temporarily store intermediate results in the main memory, which reduces the use of ILP (see again Figure 1.1). Furthermore, a processor with n issue slots usually requires a register file with $2 * n$ read ports and n write ports, so that each of the n instructions bundled together can simultaneously read two operands and write one result. Clearly, with the increase of number of PUs, the number of ports of the register file must also be increased to improve the use of ILP. However, the required wiring to allow all PUs to access all read and write ports in the register file in a cycle quickly reaches limits in its scalability. It is observed in [ZyKo98] that with the increase in the number of ports, the power dissipation in a register file increases at the rate of n^2 to n^3 while the area and the access times have been shown to increase at the rate of n^3 and $n^{3/2}$ in [RDKM00]. This lead to the development of clustered VLIW architectures where PUs can only access predefined subsets of registers, which is a further hurdle for the compilers to generate suitable code.

The effect of *exposing datapaths* of the processor to the compiler to possibly reduce the register file pressure in VLIW processors was studied in [HoCo94] and yielded quite impressive results. It was found that it is possible to bundle and execute 2 instructions per cycle with only a two-ported register file, which requires four read ports and two write ports in the original VLIW. Also, 3.6 instructions per cycle is supported with a six-ported register file, which requires eight read ports and four write ports in the original VLIW architecture. *Transport-triggered architectures (TTA)* [Corp94; HoCo94a; Corp99], whose execution framework is shown in Figure 6.3, was used to conduct the aforementioned study. The PUs and the register file are connected via a move instruction bus in TTA so that the PUs can communicate values with each other without having to write and read a central register file. The PUs have uniquely addressed single registers at their input and output ports. One of the input ports of each PU is designated as the 'trigger' port (shown in dark green color in Figure 6.3) so that the data transport to this port will trigger the execution of the respective PU, thus the name 'transport-triggered'. Similar to SCAD, TTAs are also programmed by move instructions. When the control unit issues a move instruction via the move instruction bus, the transport of

a value from register *src* to register *tgt* is performed immediately again via the move instruction bus. Clearly, several move instructions may be grouped in a bundle and issued via a bus with many lanes. Again like in SCAD, the PUs in TTA may implement any arbitrary application-specific function without affecting the instruction set. Extensive research is done on compilers and design space exploration for TTAs [1].

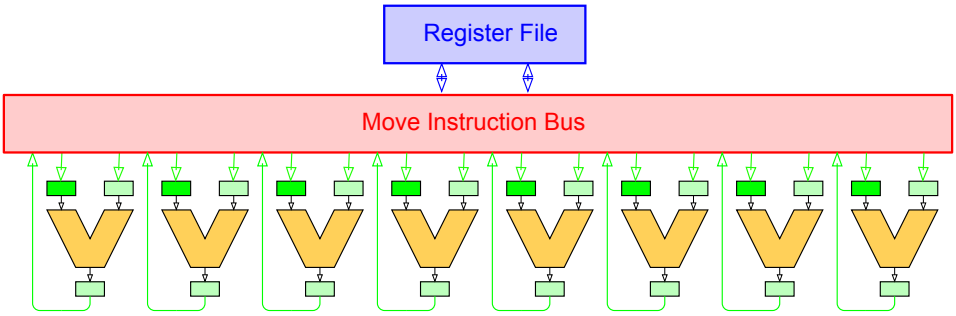


Figure 6.3.: *Execution framework of a TTA processor*

TTAs are, similar to VLIW, statically scheduled architectures where the compiler determines instruction placement and instruction issue. In fact, they are an extreme case of static scheduling since not just the firing times of operations in PUs but also the times of data transports corresponding to operations are statically determined. In SCAD, data transports are only registered on the issue of the respective move instructions, while the actual transport of the data is only carried out when the respective values are available. Therefore, SCAD processors adapt to arbitrary latencies of PUs and memory accesses similar to superscalar processors. In contrast, compilers for TTA and VLIW processors must often consider a worst-case latency of hardware units to derive a correct static schedule. Static instruction issue inhibits the execution in dataflow order since instructions with uncertain latency such as memory loads (in the context of cache hit/miss) cannot be optimally accommodated in the static schedule and will stall the entire execution engine. Thus, static instruction issue is a limiting factor for the use of ILP. Moreover, TTAs will still need to use a central register file since, unlike the FIFO buffers in SCAD, they use single registers at PU inputs and outputs. Accordingly, the compilers will have to assign some intermediate results (that are not bypassed) to general purpose registers. Although register allocation is less critical than in VLIW, the use of general purpose registers still limits the amount of ILP the architecture can make use of.

Execution in some architectures bears a strong resemblance to the execution paradigm of TTA in the sense that they are statically scheduled architectures with exposed datapaths. The *MOVE-Pro* architecture [HSMC11] has a buffer with multiple entries at each PU output, instead of a single register in TTA. This increases the chances of bypassing more register file accesses. Since the

[1] `http://openasip.org/`

data transport to the trigger input port of a PU in TTA triggers the firing of that PU, this data transport must always be scheduled after scheduling the move instructions to the other input ports. The PUs in MOVE-PRO does not have a designated trigger port. Instead, the data transport to any input port can trigger the PU's firing. To this end, the triggering information is encoded by setting or clearing an opcode field in any move instruction. This increases the flexibility to schedule move instructions in MOVE-Pro compared to TTA. Furthermore, MOVE-Pro offers improved code density by proposing new instruction formats. The *synchronous transfer architecture (STA)* [CRSM04] is more similar to TTA. However, instead of a limited bus, it uses a multiplexer array to allow any PU to receive operands for execution from other PUs directly. Both the operation to be executed on a PU and the interconnect configuration for its operands are encoded in an instruction, several of which may be bundled together for concurrent execution. There are other similar architectures that not only allow the compiler to move values between PUs, but also expose more finer microarchitectural details. In *FlexCore* [TSBS08], PUs are connected by a full crossbar. Statically scheduled [ScSL09] instructions of a native ISA encode both the configuration of the crossbar network and the operations to be executed on individual PUs. Also, the use of an instruction decompression unit that unfolds instructions in an application-specific ISA to the more detailed native ISA, is proposed to improve the code density. Notice that the execution in the statically ordered variant of SCAD where the compiler determines the instruction issue resembles the execution paradigm of TTA-like architectures.

Numerous other architectures expose communication details in hardware to the compiler. To mention a few: The RAW/Tilera architecture [WTSS97; LBFS98] consists of simple pipelined RISC processor cores arranged in a grid and connected by mesh networks. Each core is equipped with a local instruction and data memory. A few designated registers of each core are connected to the network. A software-controlled network switch in each core is configured by the compiler to read from registers of another core connected to the network, thus supporting direct communication from a producer instruction in one core to the consumer instruction in another core. In the explicit datapath wide single instruction multiple data (SIMD) architecture [WSCH15], a set of PUs are arranged in a circular layout where each unit is connected to its left and right neighbors. There is a control unit that can talk to all the PUs. It is programmed using very long instructions that encode for each PU, the source and destination of its operands, and the role of the control unit. The statically scheduled Mill[2] architecture uses a fixed-length FIFO buffer (called Belt) to store operands and results of the execution, and this way eliminates the general purpose register file. Unlike buffers in SCAD, the PUs in the Mill processor may directly read operands from any location in the Belt. AMIDAR (Adaptive Microinstruction Driven Architecture) [GaHo05a] is a general model for building processors that features a set of functional units (meaning any hardware component) connected to each other via a communication structure. A

[2] `https://millcomputing.com/technology/docs/belt/`

token generator distributes the so-called tokens to these functional units over a dedicated token generator network. Each token encodes, in addition to an operation code, the destination functional unit to where the operation's result is to be sent. The token also contains a tag value that is either attached after incrementing or attached as is to the sent result, depending on yet another information encoded by the token. Programs consist of instructions where each instruction may contain an arbitrary number of tokens, allowing any function to be computed by direct communication of values between the functional units. The AMIDAR model has been often used to execute Java bytecode. For a recent review of exposed datapath architectures, see [JKVT15].

With the growing number of PUs in processors, the execution of programs is becoming increasingly communication dominated due to longer wires and the subsequent decrease in wire transmission speeds [AHKB00; HoYS98]. Therefore, the instruction schedulers must consider on-chip wire delays when allocating PUs to producing and consuming instructions. Clearly, dynamic instruction placement (as in superscalar processors) is limited in handling growing wire delays, whereas compiler-driven static instruction placement can better optimize communication delays. On the other hand, the dynamic instruction issue is preferred over the static instruction issue to achieve a higher use of ILP. This evoked interest in *static placement dynamic issue (SPDI)* [NKBM04] architectures, which combined a compiler-determined allocation of PUs to instructions and a hardware-determined issue of instructions. According to this categorization, superscalar processors are classified as DPDI (dynamic placement dynamic issue) architectures, whereas VLIW, TTA, and other statically scheduled exposed datapath processors are classified as SPSI (static placement static issue) architectures. The SCAD architecture follows the SPDI execution model, thus retaining the benefits of static instruction placement and dynamic instruction issue.

The *TRIPS* [BKMD04] (Tera-op, Reliable, Intelligently adaptive Processing System) architecture, an Explicit Dataflow Graph Execution (EDGE) architecture, presents the SPDI scheduling model. Recall that in superscalar processors, all PUs have to arbitrate on a single common data bus, which is snooped by each reservation station entry to possibly fetch its operands. This restricted direct instruction communication is necessary since when the hardware allocates an instruction to PU, all consumer instructions might not yet be fetched and decoded. This limits the scaling of PUs and the reservation station in dynamically scheduled superscalar processors. TRIPS avoids this bottleneck by employing an SPDI block-atomic execution mode: Figure 6.4 shows the execution framework of TRIPS where each PU has its own set of reservation station entries (instruction buffers), each of which is connected to output ports of PUs via a 2D mesh network. The TRIPS compiler [SBGM06] transforms segments of the program's control-flow graph to large basic blocks called hyperblocks by various techniques such as loop unrolling, if-conversion, predication, etc., and map these hyperblocks to the computational engine. Each instruction in a hyperblock is accommodated in a statically determined reservation station slot (static placement) and encodes physical locations (again

reservation station slots) of its consumer instructions. The availability of operand values in any reservation station slot triggers the firing of the corresponding PU executing the respective instruction. The computed result is then directly communicated to its consumer instructions via the network. This way, all instructions within a hyperblock communicate directly with one another. Registers and memory are used for the communication between hyperblocks. In fact, a hyperblock execution is considered complete when it performs a specific number of register writes and memory stores. Clearly, there is a program counter in TRIPS that steers the control flow in programs at the level of hyperblocks. Notice that the hyperblock execution in TRIPS is analogous to the RISC instruction execution in superscalar processors so that various techniques to improve the use of ILP like superscalarity (multiple hyperblocks in flight simultaneously), out-of-order execution, and branch prediction (for speculative execution), are also used in TRIPS at the level of hyperblocks [SBGM06].

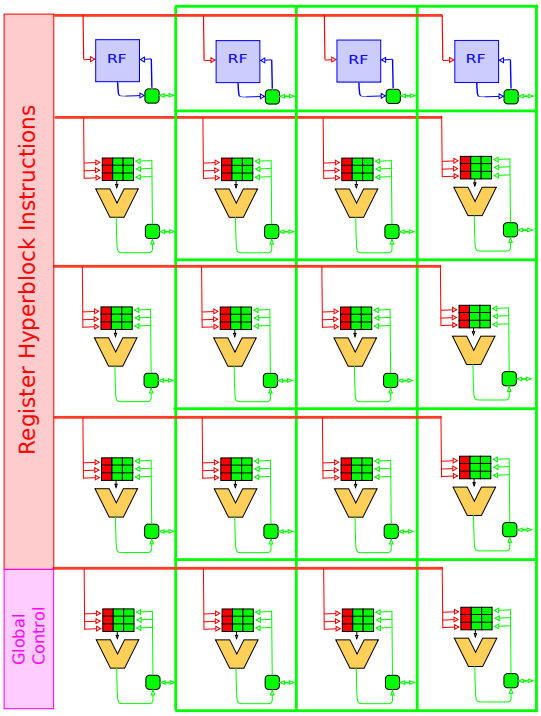


Figure 6.4.: *Execution framework of a TRIPS processor*

Although the use of hyperblocks considerably reduces register file accesses in TRIPS, the use of registers for communicating between hyperblocks will still adversely affect the potential use of ILP by the architecture. It is not easy to scale the instruction buffers (reservation station) local to each PU in TRIPS, compared to FIFO buffers in SCAD that scale better so that more values can be accommodated for direct communication at any point of time. Moreover, since each slot in a reservation station is addressed by the interconnection network, the interconnection network in TRIPS must scale proportional to

the scaling of reservation stations and PUs. Meanwhile, in SCAD, the interconnection network needs to scale only with the number of PUs because only the head of output buffers and the tail of input buffers are connected to the DTN in SCAD. On the downside, the operand values in SCAD must ripple their way to the right slot in the input buffers. Each PU in TRIPS can execute instructions in its reservation station and transport the results in the order that respective operands become available. Out-of-order execution of instructions and transport of results is also possible in SCAD PUs with an appropriate hardware support [JaSW17], with the constraint that multiple values send from an output buffer to the same input buffer are transported in order. However, when allocating instructions to PUs, dependent instructions are usually allocated to the same PU and independent instructions to different PUs to effectively use the ILP in programs. This would speak logically against spending additional resources for out-of-order execution and transport locally within each PU.

There exist other block-oriented architectures whose instructions consist of large blocks of operations where operations in a block communicate their results with one another directly without using a separate local storage (i.e., operations within a block are executed in dataflow order). *DySER* (Dynamically Specializing Execution Resources) [GoHS11] is similar to TRIPS in that the compiler determined dataflow graphs are atomically mapped and executed. To this end, an array of PUs is used in the execution stage of the usual processor pipeline, which can communicate intermediate results to each other via a 2D mesh network. The PU array is configured at runtime to implement specialized functions. Once configured, the specialized function may be used many times to accelerate appropriate program segments. The compiler uses profiling to determine parts of the code to accelerate this way. By allowing runtime reconfiguration, multiple different program segments may be accelerated in multiple phases of the execution. Note that unlike TRIPS, where the entire program is transformed to hyperblocks, blocks are generated only for parts of the program in DySER. *Tartan* [MCCV06] and *Conservation cores* [VSGG10] also uses dataflow execution to accelerate parts of programs, but they do not support hardware reconfiguration at runtime. Intra-block communication in these block-oriented architectures avoids the use of registers, while registers or memory are still required for inter-block communication. In SCAD, the use of registers is completely avoided using code generation inspired from classical queue machines.

So far, we have considered processors whose execution paradigm is fundamentally based on the control-flow computing model. The model does not express any parallelism due to its following defining characteristics: First, a program counter is used to steer the control flow of programs. Second, the intermediate results are stored in an updateable memory (or shared namespace). The adverse impact of the former is alleviated in the above processors by techniques such as branch prediction, speculative execution, the formation of superblocks and hyperblocks. The SCAD processor can also benefit from these techniques, but we do not experimentally study the impact of these in

SCAD in this thesis. The adverse impact of the latter is mitigated by direct communication of values between instructions. As has been seen, the architectures discussed above vary in the degree to which direct instruction communication is supported or equivalently in the extent to which dataflow execution is used to improve the exploited ILP. In SCAD, *all* intermediate results are communicated directly between the processing units, though sometimes values have to be rotated (overhead) before transporting them to the consumer processing units. With more processing units, there is less overhead. Therefore, the execution paradigm in SCAD finds an equilibrium between control-flow and dataflow execution styles. See [YAJE14] for a recent survey and classification of architectures based on the blend of control-flow and dataflow execution styles.

Dataflow processors [DeMi75; Davi78; VeFi78; KiYK83; GuKW85; ArCu84; PaCu90a; SSMP07] directly instantiate the dataflow computing model [KaMi66; Denn74; Kahn74], that has the following contrasting characteristics compared to the control-flow model of computation: The next instruction to fetch and execute is determined by the availability of operands and *not* addressed by a program counter. The intermediate results of computations are directly communicated from producer instructions to consumer instructions and are *not* overwritten using a shared namespace. For the execution in dataflow processors, the entire program is represented as a directed graph called a dataflow graph that is directly executed in hardware. The nodes and edges in a dataflow graph represent the instructions and data dependencies between instructions, respectively. That is, the operands for an instruction arrive via incoming edges of the corresponding node. The node can fire as soon as operands arrive, and the computed result is forwarded to the consumer instructions via outgoing edges. Therefore, dataflow programs directly expose all available ILP that can be used by dataflow processors. Though special languages were developed for programming dataflow computers, most dataflow based systems convert usual imperative programs to dataflow graphs. It is straightforward to translate basic blocks by the conversion to static single assignment (SSA) [AlWZ88; CFRW91; RoWZ88] form where each intermediate result is given a unique name (no shared names). The control flow in usual programs is often translated by introducing additional data-steering instructions like switch and select nodes (see Section 2.1.3).

Clearly, direct instruction communication is inherent in dataflow programs where each instruction encodes the addresses of the consumer instructions that wait for its computed result. The first dataflow processors were based on so-called static dataflow architectures [DeMi75; Davi78; VeFi78] where at most only one value (often referred to as token) is allowed to propagate along each edge in the dataflow graph. This way, race conditions by different dynamic token instances (from different loop iterations) at an edge are avoided that would otherwise create an ambiguity in identifying matching tokens for executing instructions. However, the main drawback of static dataflow processors is that the restriction of only one token per edge prevents multiple iterations of a loop from executing simultaneously. The dynamic dataflow architectures

[KiYK83; GuKW85; ArCu84; PaCu90a; SSMP07] overcome this limitation by assigning tags to data tokens and allowing multiple tokens at edges, similar to the dynamically ordered variant of SCAD. The tags associated with data tokens are used to disambiguate dynamic instances of respective target instructions. To this end, the dynamic dataflow processors must employ tag management instructions to manage and assign tags to data tokens.

The main disadvantage of dynamic dataflow processors is the additional overhead of matching tags to identify the correct operands for executing instructions, often requiring expensive associative memory implementations [GuKW85]. Even though notable attempts were made to solve the *token matching problem* (see Explicit Token Store (ETS) architecture [PaCu90a]), it remained as a difficulty in realizing an efficient implementation of dataflow processors. Another significant problem of dataflow processors is the difficulty in using conventional memory. Since some tokens will inevitably have to share the same memory slot due to the limited size of the data memory, the use of conventional memory hardware demands an ordering of memory write and read operations. It is impossible to ensure an ordering of load and store instructions in the dataflow execution model since the instructions fire when operands are available (and not in any predefined order). This necessitates additional support in dataflow processors for ensuring the desired ordering of load and store operations of a centralized data memory. This is also the reason why dataflow systems could not efficiently enforce sequential memory semantics that the imperative languages require.

WaveScalar [SSMP07] is a recent dataflow architecture that executes programs in waves, which are dataflow graphs corresponding to the maximal acyclic code regions of the program's control-flow graph. They are constructed similar to hyperblocks by loop unrolling and if-conversions. Waves cannot contain loops. To allow the simultaneous execution of different instances of a loop body, the data tokens travel with a wave-number that serves as tag to identify correct operands to execute instructions at runtime. To this end, a wave-advance instruction is used for tag management in that it increments the wave-number of data tokens as they travel from one wave to another one. Operand matching in SCAD does not require complicated token matching hardware. Clearly, the instructions are issued (PUs are fired) dynamically in WaveScalar, like in other dataflow processors. The instruction placement has both a static and dynamic component [MSPP06a]. The waves are clustered into smaller segments by the compiler, and when any instruction in a segment needs to execute, that segment is mapped at runtime into a single PU. Therefore, instructions are fetched and replaced at runtime in the granularity of segments. The compiler groups instructions into segments by performing a depth-first traversal of the dataflow graphs, thus possibly grouping dependent instructions in a segment. The latency for communicating operands is optimized by loading instruction segments hierarchically to the tiled computational engine that employs a hierarchical on-chip interconnect. The compilers for SPDI architectures (including SCAD) can better optimize communication delays by statically assigning instructions to the processing units. It is also

worth mentioning that the synchronous registration of move instructions in SCAD will ensure that when a token arrives at a PU, there will be space reserved for that token in the input buffer of the PU. This obviates any back pressure in the data transport network. The PUs in WaveScalar may reject tokens if these cannot be accommodated due to lack of space, so that the senders must then retry later. Conventional programming languages are supported in WaveScalar by a wave-ordered memory mode. To this end, the compiler annotates load and store instructions in each wave that encodes an ordering constraint (using sequence and ripple numbers) of these memory accesses, which is then respected by the memory system. The memory accesses in a wave are executed only after executing memory accesses in all previous waves.

Chapter 7

Conclusions

The use of instruction-level parallelism (ILP) by commercial processor architectures face unavoidable limits on its further scalability. These architectures are fundamentally based on a sequential control-flow computing model. The more concurrent dataflow computing model has not yet been fully exploited for commercial systems due to their inefficient implementations with conventional hardware. Since the control-flow model itself does not express any parallelism, these architectures employ various techniques to effectively use ILP contained in programs and this way adopt a hybrid control-flow dataflow model of computation. The use of registers is an innate reason for the limited use of ILP in these architectures. To this end, more recent architectures expose their datapath to the compiler, so that use of registers is bypassed by directly communicating intermediate results from producer processing units to consumer processing units. It was observed that although these architectures are studied in great detail, they still use registers to execute programs. We proposed a novel exposed datapath architecture named Synchronous Control Asynchronous Dataflow (SCAD) that uses a hybrid control-flow dataflow model of computation, where the use of registers is completely avoided by the code generation for SCAD based on classical queue machines. SCAD uses FIFO buffers at output and input ports of processing units, and output buffers are connected to the input buffers using any general interconnection network. The datapaths between output and input buffers are exposed to the compiler. SCAD is programmed by a sequence of move instructions of the form $src \rightarrow tgt$ that instructs a data transport from the output buffer *src* to the input buffer *tgt*. Execution in SCAD adheres to the control-flow computing model in that it uses a program counter to steer the control-flow of programs. Similar to other control-flow processors, well-known compiler techniques (generation of superblocks and hyperblocks) and hardware techniques (branch prediction and speculative execution) can be used to exploit ILP across the control flow boundaries.

This thesis focused on the dataflow aspect of execution in SCAD, where intermediate results are directly communicated from producer processing units to consumer processing units and are never overwritten using a shared names-

pace (registers). In other words, the use of registers, which is an inherent bottleneck in utilizing ILP contained in programs, is completely avoided in SCAD. This is achieved by a synergy of the execution paradigm and the code generator. Move code generation for SCAD inspired from classical queue machines utilizes the exposed datapaths in SCAD to the fullest, eliminating any need for other local storage (registers) and also tends to maximize the degree of exploited ILP. Queue oriented code generation suggests a breadth-first traversal over expression trees in contrast to the traditional depth-first traversal in classic register-based architectures. The depth-first traversal was motivated by the reuse of registers, while the breadth-first version is motivated to exploit maximal ILP and to eliminate the use of registers. On the one hand, queue oriented SCAD code can execute programs by direct instruction communication (without registers). However, on the other hand, SCAD code obtained by direct translation from queue code contains overhead in terms of rotation steps to access relevant values, not at the head of output FIFO buffers.

We observed that it is not a good idea to directly translate queue code to SCAD code since SCAD machines often require less overhead than the queue machine. With more processing units (thus more input and output buffers) in SCAD, there are fewer restrictions in accessing values. It is important to avoid the overhead since these additional operations degrade the use of ILP, enabled in the first place by direct instruction communication in SCAD. We established that it is, in general, an NP-hard problem to compile a program for overhead-free execution on a given SCAD machine. Consequently, boolean constraints are formulated for optimal (overhead-free) compilation so that SAT solvers may be used to determine a minimal number of processing units required in SCAD to execute a given program without the use of any overhead operations. The minimal number of PUs is an indirect measure of the SCAD compiler's effectiveness in avoiding overhead or equivalently the flexibility of the SCAD compiler in maximizing the use of ILP by direct instruction communication. To this end, we also studied variants of the SCAD machine, namely the statically ordered SCAD (SO-SCAD) and the dynamically ordered SCAD (DO-SCAD), to compare it with classic control-flow and dataflow architectures, respectively.

In processing units in SCAD, the operands for executing the next operation (or ordering of operations in processing units) are determined by the compiler and enforced by the hardware. In simpler processing units in SO-SCAD, the operands for executing the next operation are both determined and enforced by the compiler. This resembles the classic control-flow (or register) architectures where the compiler encodes operands and the result of an instruction by register addresses. In processing units in DO-SCAD, the operands for the next operation are both determined and enforced at runtime by complex token matching hardware resembling classic dataflow architectures. We found by experiments that the SO-SCAD architecture often requires many processing units to avoid the overhead when executing programs by direct instruction communication. In other words, the naive hardware support for direct instruction communication in SO-SCAD takes away a lot of the flexibility from

the compiler in utilizing ILP contained in the programs. At the same time, programs can be compiled with only a few processing units for overhead-free execution in SCAD. Given these minimal numbers of processing units, SCAD can even exploit nearly the maximal ILP similar to DO-SCAD but without the expensive token matching hardware. To summarize, the experimental study revealed that the execution paradigm in SCAD strikes an essential balance between hardware complexity and compiler flexibility.

Since optimal compilation for SCAD by SAT solvers could only process small programs, we developed heuristics to compile real benchmarks. A novel buffer interference analysis assigned instructions to the processing units so that overhead-free move code is generated by considering instructions in program order. We executed programs both by direct instruction communication and by using registers using a cycle-accurate SCAD simulator. To this end, a representative set of benchmarks were used that featured both regular and irregular parallelism. It was observed that the execution by direct instruction communication was faster even when we used ideal multi-ported register files (with as many ports as registers) to execute register-based programs. More importantly, the data transmission pattern showed that the communication (transmission and reception) of values was distributed among processing units in execution by direct instruction communication. On the contrary, as expected, the bulk of data traffic in the execution of register-based programs was routed through a limited number of ports of the register file, revealing the register file as a bottleneck in the use of ILP and as a hotspot in power consumption.

Future work

Experiments

We already suggested that SCAD architectures can benefit from well established hardware techniques (branch prediction) and compiler techniques (superblock and hyperblock formations) to exploit ILP across control flow boundaries. However, no experiments are carried out in this regard since we focused mainly on direct instruction communication in SCAD in this thesis. It is important to experimentally compare the impact of these techniques in SCAD with that in conventional *superscalar* and *VLIW processors*. For this purpose, it is necessary to develop a heuristic for SCAD code generation that will maximize the use of ILP rather than the resource-constrained code generation heuristic discussed in this thesis. On the hardware front, the immediate next step is to prototype a SCAD processor in hardware and to compare it with traditional superscalar and VLIW implementations in terms of area, cycle time, and power consumption.

Execution paradigm

Asynchronous control unit: In SCAD, the source and target addresses of a move instruction are registered synchronously in the corresponding input

and output buffer, respectively. If one of these buffers is full, the control unit has to stall. This necessitates using a bus (MIB) that must be snooped by all input and output buffers. Though multiple lanes may be used to register independent moves simultaneously, the performance of applications with a huge amount of ILP is expected to saturate due to the use of the MIB. The MIB is also a hurdle in scaling SCAD to larger numbers of processing units. Therefore, it is desirable to allow an asynchronous registration of addresses in buffers. To this end, the SCAD program will consist of a sequence of addresses per buffer. These independent sequences may be registered on the respective buffers using any parallel interconnection network (*address transport network*) that scales better than buses. We envision an architecture where both program (sequences of addresses) registration and data flow proceed asynchronously. However, with an asynchronous control unit, different aspects of the execution paradigm and code generation must be looked at carefully for correctness. Furthermore, since a program counter currently steers control flow in SCAD, this asynchronous operation of the control unit (or the parallel registration of addresses) must inevitably synchronize at control flow boundaries in the program.

Control flow: The control flow is currently implemented in SCAD by branch move instructions whose target is the control unit. Predication may be used with loop unrolling to construct hyperblocks that cover conditional statements. However, branch prediction is still required to execute multiple loop iterations simultaneously. Branch prediction and ensuing speculative execution require a lot of broadcast (one to many) communication from the control unit to all processing units, which will eventually limit the scalability of SCAD to larger numbers of processing units. Moreover, while branch prediction is complemented by a synchronous control unit, it does not augur well with the nature of an asynchronous control unit discussed above. It is therefore important to consider other approaches in handling loops. For example, we may explore possibilities of implementing the *switch* node in SCAD (see Section 2.1.3) to execute loops similar to dataflow processors.

Timing predictability: Though not explored in this thesis, predictable execution time was one of the factors considered when the execution paradigm of SCAD was designed [BhJS15]. Memory accesses (especially caches) and out-of-order execution in processing units adversely affect timing predictability of the underlying hardware [HLTW03]. In SCAD, the number of memory accesses is reduced by the use of FIFO buffers coupled with direct instruction communication. Moreover, execution local to each processing unit proceeds in program order. These factors speak in favour of SCAD as a time-predictable processor architecture. We will investigate in greater detail the timing predictability of SCAD architectures and worst case execution time analysis of move programs.

Model of computation: Move programs are inherently difficult to read and understand. Consequently, directly verifying the properties of a move program is a daunting task. Therefore, it is desirable to derive a *denotational semantics of the SCAD machine* by formally studying the SCAD computational model,

which shall make it easier to reason about move programs and subsequently aid in verifying them.

Code generation

Heuristics: The optimal code generation for SCAD by SAT solvers can generate move code given both resource and time constraints. However, only small programs can be processed by the SAT solver. The heuristics discussed for SCAD code generation can only generate overhead-free move code that uses a minimal number of processing units (resource-constrained) for execution. It is necessary to determine heuristics with constraints on the execution time to maximize the use of ILP. It is further important to consider input and output buffer sizes in SCAD processing units for generating move code that will work with given buffer sizes.

Compilation phases: In classic compilers, register assignment and instruction scheduling are conflicting phases in that an early register assignment may constrain instruction scheduling and vice versa. Analogously, the conflicting compilation phases in SCAD are *overhead optimization* and *instruction scheduling*. In the discussed heuristic, we obtain a constrained instruction schedule to generate overhead-free SCAD code. It is also necessary to consider other phase orderings, i.e., to optimize overhead for an instruction schedule that maximizes the use of ILP. Finally, combined approaches should also be explored.

Dataflow and functional programs: In imperative programs, the same variable is often used to hold different values during different stages of the program, and sequential execution is required to guarantee program correctness. This is in harmony with the execution paradigm followed by register machines where registers are used to hold different values during different stages of the program execution. However, in the SCAD execution paradigm, intermediate program values are preferably communicated directly from the producer PUs to the consumer PUs instead of storing these in some central register file. In other words, SCAD deals with values rather than variables. This execution paradigm is closer to dataflow and functional programs, where variables obey the single-assignment rule: each variable is assigned at most once, and it is never over-written. Therefore, it is worthwhile to study the characteristics of dataflow and functional programs to possibly learn further efficient approaches to generate SCAD code.

Bibliography

[ArCu84] Arvind and D.E. Culler. *The Tagged Token Dataflow Architecture*. Technical Report FLA Memo 229. Cambridge, Massachusetts, USA: MIT Lab for Computer Science, 1984.

[AdGh96] S.V. Adve and K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial". In: *IEEE Computer* 29.12 (Dec. 1996), pp. 66–76.

[AHKB00] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. "Clock rate versus IPC: the end of the road for conventional microarchitectures". In: *International Symposium on Computer Architecture*. Vancouver, British Columbia, Canada: ACM, 2000, pp. 248–259.

[Anan99] C.S. Ananian. *The Static Single Information Form*. Technical Report MIT-LCS-TR-801. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, 1999.

[Ande17] M. Anders. "Complexity Analysis of Code Generation for the SCAD Machine". Bachelor. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, Oct. 2017.

[ApPa02] A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. 2nd ed. Cambridge University Press, 2002.

[AlWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. "Detecting Equality of Variables in Programs". In: *Principles of Programming Languages (POPL)*. San Diego, California, USA: ACM, 1988, pp. 1–11.

[BoJa66] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Communications of the ACM* 9.5 (May 1966), pp. 366–371.

[BhJS15] A. Bhagyanath, T. Jain, and K. Schneider. "Poster Abstract: A Time-Predictable Model of Computation". In: *Real-Time Systems Symposium*. San Antonio, Texas, USA: IEEE Computer Society, 2015, p. 376.

[BhJS16] A. Bhagyanath, T. Jain, and K. Schneider. “Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. Freiburg, Germany: University of Freiburg, 2016, pp. 77–88.

[BBDR12] B. Boissinot, P. Brisk, A. Darte, and F. Rastello. “SSI Properties Revisited”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 11S.1 (June 2012), 21:1–21:23.

[BhSc16] 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*. Kanpur, India: IEEE Computer Society, 2016.

[BhSc17] A. Bhagyanath and K. Schneider. “Exploring the Instruction-Level Parallelism Potential of Exposed Datapath Architectures with Buffered Processing Units”. In: *Application of Concurrency to System Design*. Ed. by A. Legay and K. Schneider. Zaragoza, Spain: IEEE Computer Society, 2017.

[BhSc17a] 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)*. Ed. by Y. Patt and S.K. Nandy. Samos, Greece: IEEE Computer Society, 2017, pp. 1–10.

[BKMD04] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. “Scaling to the End of Silicon with EDGE Architectures”. In: *IEEE Computer* 37.7 (July 2004), pp. 44–55.

[Chai04] G. Chaitin. “Register allocation and spilling via graph coloring”. In: *ACM SIGPLAN Notices* 39.4 (Apr. 2004), pp. 66–74.

[CRSM04] G. Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis. “Synchronous Transfer Architecture (STA)”. In: *International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece: Springer Berlin Heidelberg, 2004, pp. 343–352.

[Corp94] H. Corporaal. “Design of Transport Triggered Architectures”. In: *Great Lakes Symposium on VLSI (GLSVLSI)*. Notre Dame, IN, USA: IEEE Computer Society, 1994, pp. 130–135.

[Corp99] H. Corporaal. “TTAs: Missing the ILP complexity wall”. In: *Journal of Systems Architecture* 45.12-13 (June 1999), pp. 949–973.

[CFRW91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (Oct. 1991), pp. 451–490.

[Davi78] A.L. Davis. "The architecture and system method of DDM1: A recursively structured Data Driven Machine". In: *International Symposium on Computer Architecture (ISCA)*. Palo Alto, CA, USA: ACM, 1978, pp. 210–215.

[Denn74] J.B. Dennis. "First Version of a Data-Flow Procedure Language". In: *Programming Symposium*. Ed. by B. Robinet. Vol. 19. LNCS. Paris, France: Springer, 1974, pp. 362–376.

[DeMi75] J.B. Dennis and D.P. Misunas. "A Preliminary Architecture for a Basic Dataflow Processor". In: *International Symposium on Computer Architecture (ISCA)*. Ed. by W.K. King and O. Garcia. Houston, TX, USA: ACM, 1975, pp. 126–132.

[FeEr81] M. Feller and M.D. Ercegovac. "Queue machines: An organization for parallel computation". In: *Conpar 81*. Vol. 111. LNCS. Nürnberg, Germany: Springer, 1981, pp. 37–47.

[FGQF12] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla. "Assessing the suitability of the NGMP multicore processor in the space domain". In: *International Conference on Embedded Software*. New York, NY, USA: ACM, 2012, pp. 175–184.

[FiFY05] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[FERN84] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, and A. Nicolau. "Parallel processing: a smart compiler and a dumb machine". In: *ACM SIGPLAN Notices* 19.6 (1984), pp. 37–47.

[Fish81] J.A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". In: *IEEE Transactions on Computers* C-30.7 (July 1981), pp. 478–490.

[GaHo05a] S. Gatzka and C. Hochberger. "The AMIDAR Class of Reconfigurable Processors". In: *The Journal of Supercomputing* 32.2 (2005), pp. 163–181.

[GoHS11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. "Dynamically Specialized Datapaths for Energy Efficient Computing". In: *International Symposium on High Performance Computer Architecture*. San Antonio, TX, USA: IEEE Computer Society, 2011, pp. 503–514.

[GuKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. "The Manchester prototype dataflow computer". In: *Communications of the ACM* 28.1 (Jan. 1985), pp. 34–52.

[Gros00] J.P. Grossman. *Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation*. unpublished manuscript. 2000.

[HoCo94] J. Hoogerbrugge and H. Corporaal. "Register file port requirements of transport triggered architectures". In: *Microarchitecture (MICRO)*. San Jose, California, USA: IEEE Computer Society, 1994, pp. 191–195.

[HoCo94a] J. Hoogerbrugge and H. Corporaal. "Transport-Triggering vs. Operation-Triggering". In: *Compiler Construction (CC)*. Ed. by P. Fritzson. Vol. 786. LNCS. Edinburgh, UK: Springer, 1994, pp. 435–449.

[HSMC11] Y. He, D. She, B. Mesman, and H. Corporaal. "MOVE-Pro: A low power and high code density TTA architecture". In: *International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece: IEEE, 2011, pp. 294–301.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. "The influence of processor architecture on the design and the results of WCET tools". In: *Proceedings of the IEEE* 91.7 (July 2003), pp. 1038–1054.

[HoYS98] M. Horowitz, Chih-Kong Ken Yang, and S. Sidiropoulos. "High-speed electrical signaling: overview and limitations". In: *IEEE Micro* 18.1 (Jan. 1998), pp. 12–24.

[JKVT15] P. Jääskeläinen, H. Kultala, T. Viitanen, and J. Takala. "Code Density and Energy Efficiency of Exposed Datapath Architectures". In: *Journal of Signal Processing Systems* 80.1 (July 2015), pp. 49–64.

[John91] W. Johnson. *Superscalar Microprocessor Design*. Englewood Cliffs, New Jersey, USA: Prentice Hall, 1991.

[JaSW17] T. Jain, K. Schneider, and F. Walk. "Out-of-Order Execution of Buffered Function Units in Exposed Data Path Architectures". In: *Reconfigurable Architectures Workshop (RAW)*. Ed. by D. Göhringer and D. Sciuto. Orlando, Florida, USA: IEEE Computer Society, 2017, pp. 229–234.

[JoWa89] N.P. Jouppi and D.W. Wall. "Available instruction-level parallelism for superscalar and superpipelined machines". In: *Architectural Support for Programming Languages and Operating Systems*. Boston, Massachusetts, USA: ACM, 1989, pp. 272–282.

[Kahn74] G. Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Information Processing*. Ed. by J.L. Rosenfeld. Stockholm, Sweden: North-Holland, 1974, pp. 471–475.

[Karp72] R.M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by R.E. Miller and J.W. Thatcher. Yorktown Heights, New York, USA: Plenum Press, New York, 1972, pp. 85–103.

[Kild73] G.A. Kildall. “A unified approach to global program optimization”. In: *Principles of Programming Languages (POPL)*. Boston, Massachusetts, USA: ACM, 1973, pp. 194–206.

[KaMi66] R.M. Karp and R.E. Miller. “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing”. In: *SIAM Journal on Applied Mathematics (SIAP)* 14.6 (Nov. 1966), pp. 1390–1411.

[KiYK83] M. Kishi, H. Yasuhara, and Y. Kawamura. “DDDP-a Distributed Data Driven Processor”. In: *International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 1983, pp. 236–242.

[Lam88] M. Lam. “Software pipelining: an effective scheduling technique for VLIW machines”. In: *Programming Language Design and Implementation*. Atlanta, Georgia, USA: ACM, 1988, pp. 318–328.

[LBFS98] 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*. San Jose, California, USA: ACM, 1998, pp. 46–57.

[Lee06] E.A. Lee. “The problem with threads”. In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[LaHw95] D.M. Lavery and W.W. Hwu. “Unrolling-based optimizations for modulo scheduling”. In: *Microarchitecture (MICRO)*. Ann Arbor, Michigan, USA: IEEE Computer Society, 1995, pp. 327–337.

[MLCH92] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. “Effective compiler support for predicated execution using the hyperblock”. In: *Microarchitecture (MICRO)*. Portland, Oregon, USA: IEEE Computer Society, 1992, pp. 45–54.

[MoBj08a] L. Mendonça de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. LNCS. Budapest, Hungary: Springer, 2008, pp. 337–340.

[McKe04] S.A. McKee. “Reflections on the memory wall”. In: *Computing Frontiers*. Ischia, Italy: ACM, 2004, pp. 162–167.

[MSPP06a] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S.J. Eggers. "Instruction scheduling for a tiled dataflow architecture". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Ed. by J.P. Shen and M.R. Martonosi. San Jose, California, USA: ACM, 2006, pp. 141–150.

[MCCV06] M. Mishra, T.J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S.C. Goldstein. "Tartan: evaluating spatial computation for whole program execution". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Ed. by J.P. Shen and M.R. Martonosi. San Jose, California, USA: ACM, 2006, pp. 163–174.

[Mosb93] D. Mosberger. "Memory consistency models". In: *ACM SIGOPS: Operating Systems Review* 27.1 (Jan. 1993), pp. 18–26.

[NKBM04] R. Nagarajan, S.K. Kushwaha, D. Burger, K.S. McKinley, C. Lin, and S.W. Keckler. "Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures". In: *Parallel Architectures and Compilation Techniques*. Antibes Juan-les-Pins, France: IEEE Computer Society, 2004, pp. 74–84.

[VonN45] John von Neumann. *First draft of a report on the EDVAC*. Tech. rep. Moore School of Electrical Engineering, University of Pennsylvania, June 1945.

[NoPa12] J. Nowotsch and M. Paulitsch. "Leveraging multi-core computing architectures in Avionics". In: *European Dependable Computing Conference*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 132–143.

[PaCu90a] G. Papadopoulos and D. Culler. "Monsoon: An Explicit Token-Store Architecture". In: *International Symposium on Computer Architecture*. Ed. by J.-L. Baer and L. Snyder. Seattle, Washington, USA: IEEE Computer Society, 1990, pp. 82–91.

[RGGQ12] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla. "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments". In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), 34:1–34:25.

[Rau94] B. Ramakrishna Rau. "Iterative modulo scheduling: an algorithm for software pipelining loops". In: *Microarchitecture*. San Jose, California, USA: IEEE Computer Society, 1994, pp. 63–74.

[RaFi93] B.R. Rau and J.A. Fisher. "Instruction-level Parallel Processing: History, Overview, and Perspective". In: *Journal of Supercomputing* 7.1-2 (1993), pp. 9–50.

[RDKM00] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. "Register organization for media processing". In: *High-Performance Computer Architecture (HPCA)*. 2000, pp. 375–386.

[RoWZ88] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Principles of Programming Languages (POPL)*. San Diego, California, USA: ACM, 1988, pp. 12–27.

[Schn18] A. Schneiders. "Using Static-Single-Information-Form for SCAD Code Generation". Bachelor. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, June 2018.

[Sing06] J. Singer. *Static program analysis based on virtual register renaming*. Technical Report UCAM-CL-TR-660. Computer Laboratory, University of Cambridge, Feb. 2006.

[ScLY02] H. Schmit, B. Levine, and B. Ylvisaker. "Queue machines: hardware compilation in hardware". In: *Field-Programmable Custom Computing Machines*. Napa, California, USA: IEEE Computer Society, 2002, pp. 152–160.

[SBGM06] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. "Compiling for EDGE architectures". In: *International Symposium on Code Generation and Optimization*. New York, NY, USA: IEEE, 2006.

[StNu04] R.C. Steinke and G.J. Nutt. "A unified theory of shared memory consistency". In: *Journal of the ACM* 51.5 (Sept. 2004), pp. 800–849.

[SmSo95] J. Smith and G. Sohi. "The Microarchitecture of Superscalar Processors". In: *Proceedings of the IEEE* 83.12 (1995), pp. 1609–1624.

[ScSL09] T. Schilling, M. Själander, and P. Larsson-Edefors. "Scheduling for an Embedded Architecture with a Flexible Datapath". In: *Annual Symposium on VLSI*. Tampa, Florida, USA: IEEE Computer Society, 2009, pp. 151–156.

[SeUl70] R. Sethi and J.D. Ullman. "The Generation of Optimal Code for Arithmetic Expressions". In: *Journal of the ACM* 17.4 (Oct. 1970), pp. 715–728.

[SSMP07] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S.J. Eggers. "The WaveScalar Architecture". In: *ACM Transactions on Computer Systems* 25.2 (May 2007), pp. 1–54.

[TSBS08] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing". In: *Journal of Signal Processing Systems* 57.1 (Apr. 2008), pp. 5–19.

[Toma67] R.M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33.

[VSGG10] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M.B. Taylor. “Conservation cores: reducing the energy of mature computations”. In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Ed. by J.C. Hoe and V.S. Adve. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 205–218.

[VeFi78] R. Vedder and D. Finn. “The Hughes Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing”. In: *International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society, 1985, pp. 324–332.

[Voll70] R. Vollmar. “Über einen Automaten mit Pufferspeicherung”. In: *Computing* 5.1 (1970), pp. 57–70.

[WSCH15] L. Waeijen, D. She, H. Corporaal, and Y. He. “A Low-Energy Wide SIMD Architecture with Explicit Datapath”. In: *Journal of Signal Processing Systems* 80.1 (2015), pp. 65–86.

[WTSS97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, et al. “Baring it all to Software: RAW Machines”. In: *IEEE Computer* 30.9 (Sept. 1997), pp. 86–93.

[Wall91a] D.W. Wall. “Limits of instruction-level parallelism”. In: *Architectural Support for Programming Languages and Operating Systems*. Santa Clara, California, USA: ACM, 1991, pp. 176–188.

[WGRS09] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. “Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (Jan. 2009), pp. 966–978.

[WuMc95] W. A. Wulf and S. A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24.

[YAJE14] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. “Hybrid Dataflow/von-Neumann Architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (June 2014), pp. 1489–1509.

[ZyKo98] V. Zyuban and P. Kogge. “The energy complexity of register files”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. New York, NY, USA: ACM, 1998, pp. 305–310.

Appendix A

Compilation Example

```
1  // ------------------------------------------------------
2  // The function below computes integer approximation to the
3  // square root of a given natural number a. It is known as
4  // Heron's algorithm, but is also derived by Newton-Raphson
5  // iteration of f(x) = x^2-a to compute roots.
6  // ------------------------------------------------------
7
8  nat res;
9
10 thread heron {
11     // variables
12     nat a,xold,xnew;
13
14     // given natural number
15     a = 100;
16
17     // compute square root
18     xnew = a;
19     xold = a;
20     do {
21        xold = xnew;
22        xnew = (xold + a/xold)/2;
23     } while(xnew < xold);
24
25     // store result
26     res = xold;
27 }
```

Listing A.1: *Program*

Register oriented

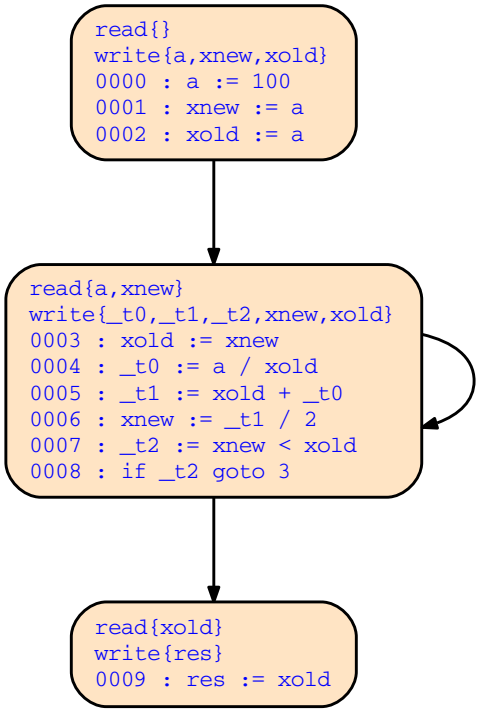


Figure A.1.: *Command program*

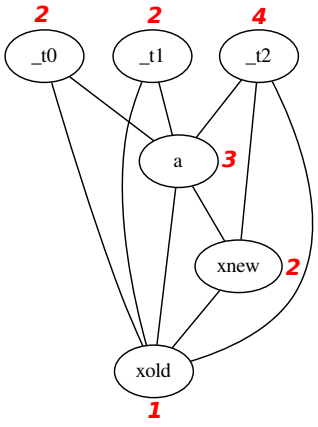


Figure A.2.: *Colored register interference graph*

```
1  $100 -> reg3 //a := 100
2  reg3 -> reg2 //xnew := a
3  reg3 -> reg1 //xold := a
4  reg2 -> reg1 //xold := xnew
5  reg3 -> pu0@l //_t0 := a / xold
6  reg1 -> pu0@r
7  (/,1) -> pu0@opcp
8  pu0@o -> reg2
9  reg1 -> pu0@l //_t1 := xold + _t0
10 reg2 -> pu0@r
11 (+,1) -> pu0@opcp
12 pu0@o -> reg2
13 reg2 -> pu0@l //xnew := _t1 / 2
14 $2 -> pu0@r
15 (/,1) -> pu0@opcp
16 pu0@o -> reg2
17 reg2 -> pu0@l //_t2 := xnew < xold
18 reg1 -> pu0@r
19 (<,1) -> pu0@opcp
20 pu0@o -> reg4
21 $3 -> cu@r //if _t2 goto 3
22 reg4 -> cu@l
23 $0 -> lsu@l //res := xold
24 reg1 -> lsu@r
25 st -> lsu@opcp
```

Listing A.2: *Register based move program*

Queue oriented

```
1  nat res;
2
3  thread heron {
4      nat a1;
5      nat a,xold,xnew;
6
7      a = 100;
8
9      xnew = a;
10     xold = a;
11     a1 = a;
12     do {
13        a = a1; // bound use of a
14        xold = xnew;
15        xnew = (xold + a/xold)/2;
16        a1 = a; // bound use of a1
17     } while(xnew < xold);
18
19     res = xold;
20 }
```

Listing A.3: *Bounded program*

```
1  nat res;
2
3  thread heron{
4  {
5      nat _dN;
6      nat a1;
7      nat a,xold,xnew;
8
9      a = 100;
10
11     xnew = a;
12     xold = a;
13     a1 = a;
14     do {
15         _dN = xold; // discard xold copy
16         a = a1;
17         xold = xnew;
18         xnew = ((xold + (a / xold)) / 2);
19         a1 = a;
20     } while((xnew < xold))
21     _dN = xnew; // discard xnew copy
22     _dN = a1;   // discard a1 copy
23
24     res = xold;
25 }
```

Listing A.4: *Balanced program*

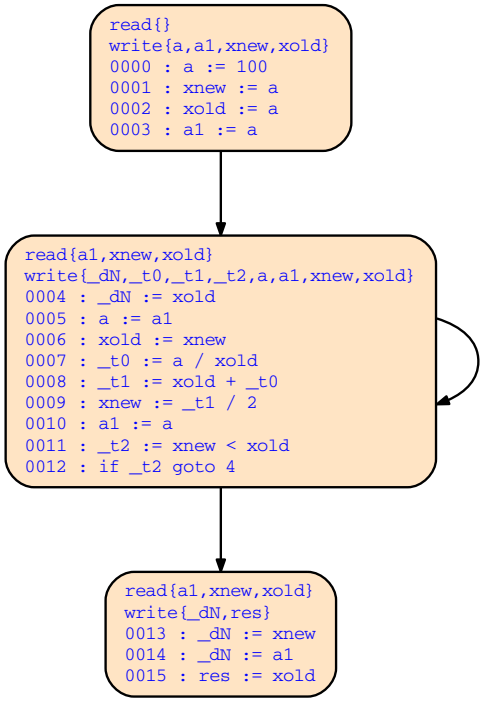


Figure A.3.: *Balanced command program*

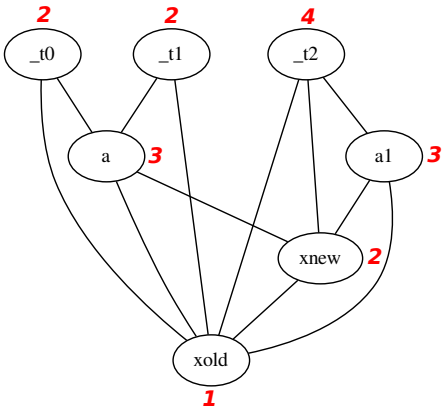


Figure A.4.: *Colored buffer interference graph*

```
1  $100 -> pu3@l //a := 100
2  $0 -> pu3@r
3  (+,3) -> pu3@opcp
4  pu3@o -> pu2@l //xnew := a
5  $0 -> pu2@r
6  (+,1) -> pu2@opcp
7  pu3@o -> pu1@l //xold := a
8  $0 -> pu1@r
9  (+,1) -> pu1@opcp
10 pu3@o -> pu3@l //a1 := a
11 $0 -> pu3@r
12 (+,1) -> pu3@opcp
13 pu1@o -> null //_dN := xold
14 pu3@o -> pu3@l //a := a1
15 $0 -> pu3@r
16 (+,2) -> pu3@opcp
17 pu2@o -> pu1@l //xold := xnew
18 $0 -> pu1@r
19 (+,4) -> pu1@opcp
20 pu3@o -> pu2@l //_t0 := a / xold
21 pu1@o -> pu2@r
22 (/,1) -> pu2@opcp
23 pu1@o -> pu2@l //_t1 := xold + _t0
24 pu2@o -> pu2@r
25 (+,1) -> pu2@opcp
26 pu2@o -> pu2@l //xnew := _t1 / 2
27 $2 -> pu2@r
28 (/,2) -> pu2@opcp
29 pu3@o -> pu3@l //a1 := a
30 $0 -> pu3@r
31 (+,1) -> pu3@opcp
32 pu2@o -> pu4@l //_t2 := xnew < xold
33 pu1@o -> pu4@r
34 (<,1) -> pu4@opcp
35 $12 -> cu@r //if _t2 goto 4
36 pu4@o -> cu@l
37 pu2@o -> null //_dN := xnew
38 pu3@o -> null //_dN := a1
39 $0 -> lsu@l //res := xold
40 pu1@o -> lsu@r
41 st -> lsu@opcp
```

Listing A.5: *Queue based move program*

Appendix B

Curriculum Vitae

Berufserfahrung

2008–2009 **Senior Software Engineer**
Infineon Technologies
XC2000 Microcontroller Family Device Driver team

2007–2008 **Software Engineer**
Cypress Semiconductors
West Bride Peripheral Controller Device Driver team

2005–2007 **Software Engineer**
Delphi Automotive Systems
Engine Management Systems team

Akademische Ausbildung

2010–2012 **MSc in Electrical and Computer Engineering** *Note 1,3*
Specialized in Embedded Systems
Technische Universität Kaiserslautern (TUK)
Masterarbeit: *Real-time scheduling of variable duration task graphs on multiple resources*

2001–2005 **BTech in Electrical and Electronics Engineering** *83%*
National Institute of Technology Calicut (NITC), India
DSP based active noise control

Schulausbildung

2001 **Vordiplom** *92%*
Kendriya Vidyalaya Kottayam, India

1999 **Abitur** *91%*
Kendriya Vidyalaya Kottayam, India