Optimal Scheduling for Exposed Datapath Architectures with Buffered Processing Units by ASP

Marc Dahlem

Insiders Technologies GmbH, Kaiserslautern, Germany
https://insiders-technologies.de

Anoop Bhagyanath, Klaus Schneider

Department of Computer Science, University of Kaiserslautern, Germany
https://es.cs.uni-kl.de

Abstract

Conventional processor architectures are restricted in exploiting instruction level parallelism (ILP) due to the relatively low number of programmer-visible registers. Therefore, more recent processor architectures expose their datapaths so that the compiler (1) can schedule parallel instructions to different processing units and (2) can make effective use of local storage of the processing units. Among these architectures, the Synchronous Control Asynchronous Dataflow (SCAD) architecture is a new exposed datapath architecture whose processing units are equipped with first-in first-out (FIFO) buffers at their input and output ports.

In contrast to register-based machines, the optimal code generation for SCAD is still a matter of research. In particular, SAT and SMT solvers were used to generate optimal resource constrained and optimal time constrained schedules for SCAD, respectively. As Answer Set Programming (ASP) offers better flexibility in handling such scheduling problems, we focus in this paper on using an answer set solver for both resource and time constrained optimal SCAD code generation. As a major benefit of using ASP, we are able to generate *all* optimal schedules for a given program which allows one to study their properties. Furthermore, the experimental results of this paper demonstrate that the answer set solver can compete with SAT solvers and outperforms SMT solvers. *This paper is under consideration for acceptance in TPLP.*

KEYWORDS: Answer Set Programming, Exposed Datapath Architectures, Optimal Scheduling, Code Generation

---

1 Introduction

1.1 Motivation

Memory access has always been a critical issue for improving the execution time of programs for at least the following reasons: First, the relative improvement of memory access time has not been as good as the relative improvement of execution time of other instructions of processors. Second, the sequential access to the main memory limits the availability of data for concurrent executions by functional units in processors. The former is tackled to some extent by the development of expensive memory hierarchies starting with the registers of the processor itself. Storing intermediate values of a program in registers allows processors to access these values much faster compared to the next components in the memory hierarchy. However, the number of registers of a register file and number of its ports, i.e., the number of parallel reads/writes of
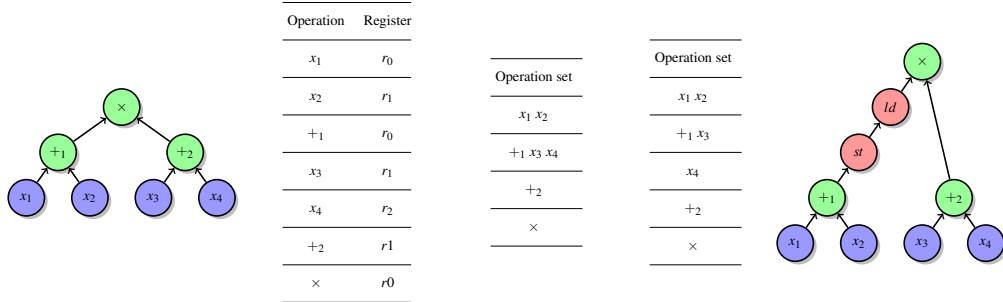
Operation	Register
x_1	r_0
x_2	r_1
$+_1$	r_0
x_3	r_1
x_4	r_2
$+_2$	$r1$
$\times$	$r0$

Operation set
$x_1\ x_2$
$+_1\ x_3\ x_4$
$+_2$
$\times$

Operation set
$x_1\ x_2$
$+_1\ x_3$
x_4
$+_2$
$\times$

Figure 1: (a) An example expression tree, (b) a depth-first ordering of operations and register assignment, (c) operations bundled together for concurrent execution, (d) operations bundled together for concurrent execution on a machine with only 2 write ports in register file, and (e) an expression tree with spill code.

registers, are usually quite low. Increasing these numbers is difficult since the number of registers is directly encoded in the instruction sets. Changing it requires corresponding changes in processors and compilers. Also, increasing the number of ports of register files and processing units (PUs) quickly leads to a bottleneck in wiring these on the chips (Aletà et al. 2007). However, a small number of ports of register files quickly becomes a communication bottleneck in processors since all values are written to and read from registers. This is particularly true for recent processors that are employing many functional units. Therefore, while introducing registers was a good idea for sequential processors, it now limits the use of instruction level parallelism (ILP) for recent architectures.

As an example, consider the expression tree shown in Figure 1(a): With a register file containing 4 registers and 4 read/write ports, this simple expression can be evaluated in only 3 steps if all levels are executed in parallel. With only 3 registers, one can avoid using load/store instructions by ordering the tree nodes and assigning registers by a depth-first traversal in an optimal way according to (Sethi and Ullman 1970). Figure 1(b) shows the operation order and the corresponding register assignment. The Figure 1(c) divides the sequence of operations into sets, where all operations in a set can be executed concurrently. Clearly, it takes 4 steps to evaluate the expression tree in parallel this way. Note that even without the overhead of load and store instructions (i.e., given a sufficient number of registers to avoid memory accesses), it is apparent that the register-based code limits the use of ILP. Moreover, if only 2 write ports are available in the register file, at least 5 steps are then required to evaluate the expression as shown by the independent sets of operations in Figure 1(d). Note that at most 2 operations can be bundled together since each operation must write its result to a register and only 2 write ports are available in the register file. Finally, if only 2 registers would be available, one has to insert spill code in that the obtained result of $x_1 + x_2$ is temporarily stored in memory and loaded in a register after having evaluated $x_3 + x_4$ as shown in Figure 1(e). The expression evaluation is then further delayed by the latency of the memory access. Hence, the number of programmer-accessible registers influences the use of ILP. Therefore, in a processor with many processing units for parallel execution, it is desirable to communicate values directly between processing units without using global registers in between.

Exposed datapath architectures offer this feature by allowing the compiler to move values directly from one PU to another PU. These architectures usually contain a large number of PUs

with local storage. However, their current compiler technology (Lee et al. 1998; Smith et al. 2006; Äijö et al. 2016; Guo et al. 2006; Schilling et al. 2009), although utilizing register bypassing, still relies on classic code generators where the use of a minimal number of registers is in the focus rather than maximizing the degree of ILP. In (Bhagyanath et al. 2016), we therefore suggested a code generation technique for exposed datapath architectures that is based on a breadth-first traversal rather than the classic depth-first traversal (Sethi and Ullman 1970) over the syntax trees. To that end, we considered the Synchronous Control Asynchronous Dataflow (SCAD) architecture that is a special exposed datapath architecture whose PUs have buffers for each input and output port. The classic depth-first traversal was motivated by the reuse of registers, while the breadth-first version was motivated by executing independent instructions in parallel.

However, with a limited number of PUs, computational overhead in the form of swap and duplication operations becomes necessary for SCAD machines (Bhagyanath et al. 2016) (see Section 2). Whether an overhead-free schedule exists for a given program on a SCAD machine with a given number of PUs was formalized in (Bhagyanath and Schneider 2016) as a satisfiability (SAT) problem. Repeated invocation of the SAT solver by incrementing the number of PUs allowed one to determine the minimal number of PUs required to execute programs without any computational overhead. Since execution time is a more important metric in instruction scheduling, we refined the SAT encoding in (Bhagyanath and Schneider 2017) to a satisfiability modulo theories (SMT) problem, introducing execution time as an additional parameter for optimization. The non-linearity of buffer constraints (a property specific to SCAD as discussed in Section 3) did not allow the use of integer linear programming solvers that is more commonly used in instruction scheduling. While the SAT solver could process up to 15-instruction programs with a timeout of 5 seconds, the SMT solver could only process up to 12-instruction programs even with a timeout of 60 seconds.

1.2 Contributions

In this paper, we refine the SAT encoding of (Bhagyanath and Schneider 2016)) to an Answer Set Programming (ASP) problem and use Clingo (Gebser et al. 2007) to obtain answer sets (schedules) of varying execution times. Execution time constraints can be intuitively included in the ASP encoding using a graph theoretic formulation. Paths of maximal length in the directed acyclic graph helps to find the schedule(s) with optimal execution time. Furthermore, the ASP encoding can be easily adapted to different optimization criteria like to minimize execution time on a given number of PUs, or to minimize the number of PUs to achieve a given execution time. The specific contributions in this paper are the following:

- We encode time constrained SCAD scheduling as an ASP problem by formulating straight-line programs (without branches) as directed acyclic graphs and determine optimal schedules using the answer set solver Clingo.
- We show that the ASP encoding can be easily modified to solve optimization problems with different optimization criteria.
- We provide experimental results that clearly demonstrate the improved feasibility of ASP-based SCAD code generation in processing larger programs compared to code generation using SAT and SMT solvers.

1.3 Outline

The rest of the paper has the following outline: Section 2 presents sufficient details of the SCAD architecture and its code generation problem. Section 3 considers then a first code optimization problem in terms of an ASP problem in that we determine the minimal number of PUs to generate overhead-free code (this was the SAT problem considered in (Bhagyanath and Schneider 2016)). Section 4 introduces then a new ASP formulation for determining the minimal execution time of a given straightline program and SCAD machine. Section 5 generalizes both ASP problems in that we will then determine the the minimal number of PUs required to run a given program in a given bound of time or conversely the minimal execution time for a given number of PUs. Section 6 presents experimental results that we obtained using the ASP solver Clingo. As we will show there, our new ASP formulation leads to much better results than the previous SAT and SMT encodings and is even more flexible in that we can consider different optimization problems in ASP alone. Finally, Section 7 considers related work, and Section 8 lists our conclusions.

2 Background

In this section, we briefly describe the basics of SCAD architectures and the related code generation problem. For a more detailed description of SCAD and the optimal code generation problem discussed here see (Bhagyanath and Schneider 2016).

2.1 SCAD Architectures

The general organization of PUs in a SCAD architecture is shown in Figure 2(a). Although only two-input one-output PUs are shown, a PU in a SCAD architecture may also implement functions with an arbitrary number of inputs and outputs. Each PU shown in Figure 2(b) has queues (or first-in-first-out (FIFO) buffers) at its input and output ports. Input and output buffers have unique addresses which are used in move instructions which are the only kind of instructions of the SCAD machine: A move instruction $src \rightarrow tgt$ means that a value from output buffer src shall be sent to input buffer tgt.

Input and output buffers are connected to two interconnection networks: The *move-instruction bus (MIB)* (given in red color) is used to synchronously send move instructions from the control unit to the PUs owning output buffer src and input buffer tgt. Second, the *data transport network (DTN)* (given in green color) is used later by the PUs to asynchronously send values whenever these are available. All buffers store pairs (adr, val) of entries. For an input buffer, adr is the address of the output buffer of the PU that has already 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 is waiting for 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 $\rightarrow$ tgt* whose semantics is to move a value from the head of output buffer src to the tail of input buffer tgt. The MIB is used to synchronously register move instructions $src \rightarrow tgt$ issued by the control unit on the PUs owning buffers src and tgt, respectively. The PUs continuously snoop the MIB for move instructions $src \rightarrow tgt$. If an output buffer has address src, it will add the entry $(tgt, \perp)$ to its tail. At the same
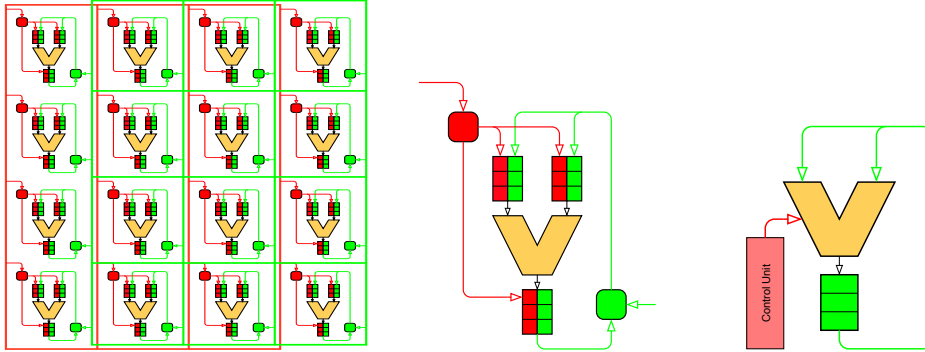
Figure 2: (a) Architecture of a SCAD Processor (b) A processing unit in SCAD and (c) Architecture of queue machine

time, the input buffer with address tgt will add the pair $(src, \bot)$ to its tail. If any of these buffers is full, a feedback signal $fullBuffer$ is given. Then, no buffer is written and the control unit (CU) is stalled until there is space in both buffers to register the move successfully. Therefore, the move instructions are registered synchronously. After this, the execution in the PUs and the actual data transports are carried out asynchronously once the values will be available.

Assume a PU with n inputs and m outputs has valid values $x_1, \ldots, x_n$ at the heads of its input buffers. If each of its m output buffers has enough free space, the corresponding operation will fire. Thus, PU execution is only driven by the availability of input data. This will produce output values $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$ for every output buffer. The pair $(adr, \bot)$ closest to the head of the jth output buffer is then replaced by $(adr, f_j(x_1, \ldots, x_n))$.

A message in the DTN is a triple (src, tgt, val) where src is the output buffer where val was produced and tgt is the input buffer the message is being sent to. When an output buffer src finds the pair (tgt, val) with $val \neq \bot$ at its head, it will produce the message (src, tgt, val) for transmission by DTN and will then remove the pair from the buffer. When receiving the message, the input buffer tgt will replace the entry $(src, \bot)$ closest to its head with (src, val).

There is at least one *load/store unit* (LSU) that handles memory accesses that are of course also required for SCAD machines. As expected, the input buffers contain the address and value to be stored for a store operation, and the address and a target buffer's address for a load value.

Branch instructions are handled as follows by the CU: if the target of a move instruction is the CU itself, it is meant to be the program counter. In this case, the CU stops fetching move instructions and has to wait until this value arrives at the head of its input buffer associated with the program counter. Otherwise, it will simply increment the program counter and place it on its input buffer's head associated with the program counter so that the next move instruction in the program order is fetched in the subsequent clock cycle.

2.2 Code Generation for SCAD

A queue machine (see Figure 2(c)) (Feller and Ercegovac 1981) reads operands for executing an operation from the head of a queue and adds the results to the tail of that queue. Generating a queue program to evaluate an expression tree is done by a breadth-first traversal of the tree (Feller and Ercegovac 1981). A consistent left to right or right to left traversal ensures that operands required to execute operations at one level are available in the queue in the correct order.

Basic blocks of programs are often represented as 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 (Schmit et al. 2002). 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 (Schmit et al. 2002) as shown in Figure 3. The given expression DAG is first levelized which means that operations must only refer to operands at the same level. This can be easily achieved by introducing *dup* operations which take a value from the head of the queue and add some number of copies of it to the tail of the queue. Then, the graph is planarized which means that crossing edges are removed by inserting *swap* operations which take two values from the head of the queue, and add them in exchanged order to the tail of the queue. One can sometimes avoid the introduction of *swap* operations by suitable ordering of input or output nodes, but not in general. Finally, another levelization is usually required since *swap* operations may be placed at new levels. A consistent left to right or right to left traversal is now performed on the level-planar graph to generate the queue program.
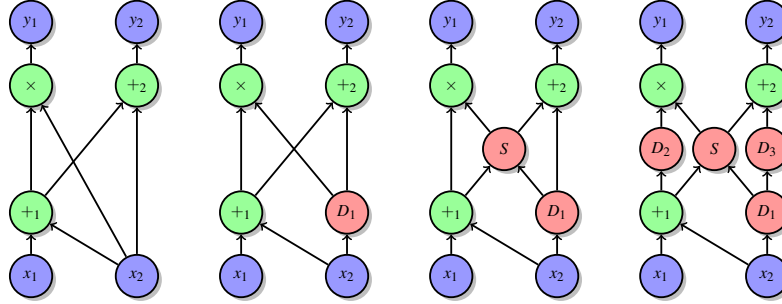

Figure 3: (a) An expression DAG with its (b) levelized and (c) planarized version, and (d) the final level-planar expression DAG

The first approach to generate code for a given SCAD machine was to translate queue code to move code whose execution on the SCAD machine simulates the execution of the queue code on the queue machine (Bhagyanath et al. 2016). It was observed that SCAD requires less overhead compared to queue machines since SCAD machines have multiple buffers (associated with multiple PUs) while queue machines only have one central queue. In other words, any basic block can be executed on a SCAD machine without overhead provided there are enough PUs. The SAT encoding in (Bhagyanath and Schneider 2016) was then used to determine the minimal number of PUs required in a SCAD machine to execute a given basic block without any overhead. This was refined to an SMT encoding in (Bhagyanath and Schneider 2017) to determine the minimal execution time of a given basic block on a SCAD machine with a given number of PUs (greater than or equal to the minimal number required for overhead free execution).

3 Optimal Scheduling Regarding Processing Units by ASP

The first question we want to answer by ASP is the one considered in (Bhagyanath and Schneider 2016) using SAT solvers: Given a basic block *B*, determine the minimal number of PUs required to schedule *B* on a SCAD machine without duplication and swap operations. Using ASP, we can additionally count the number of those schedules, and can further analyze their structure since we can compute *all* optimal schedules.

The input to our ASP encoding is a basic block that is represented as a list of variable assignments. The DAG representation and list of variable assignments for a basic block is shown in Figure 4.
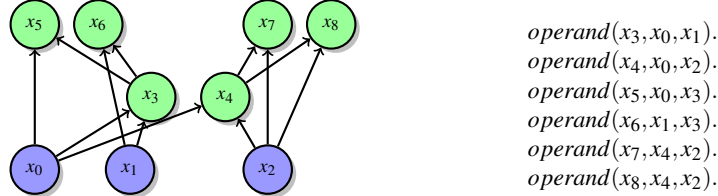

$$operand(x_3, x_0, x_1).$$
$$operand(x_4, x_0, x_2).$$
$$operand(x_5, x_0, x_3).$$
$$operand(x_6, x_1, x_3).$$
$$operand(x_7, x_4, x_2).$$
$$operand(x_8, x_4, x_2).$$

Figure 4: (a) An example basic block as DAG, (b) the same DAG as ASP-code

The expected output from the ASP encoding is/are schedule(s) of the given basic block on a SCAD machine. The schedule consists of two parts:

- *Assignment:* which variable is produced by which PU?
- *Ordering:* in which order are the variables assigned to the same PU produced?

Assignment or mapping of variables to PUs and the ordering of variables on PUs must allow generation of move code to execute the basic block on the SCAD machine without any duplication and swap overhead. In other words, with the given variable assignment and ordering, one must be able to move values from output buffers to input buffers of PUs such that firing of computations on PUs produces values in the given order. Recall that all buffers are FIFO buffers and a PU fires if values are available at the heads of its input buffers. For example, a few valid schedules of the basic block in Figure 4 are shown in Figure 5. Note that although only one copy of each variable is shown, during actual execution as many copies of a variable are produced in the output buffer as needed in the basic block. For Solution 1, the following is a valid sequence of move instructions that executes the basic block on the SCAD machine respecting the given order of variables (every PU_n has two input buffers, the left one named $PU_{n,L}$ and the right one $PU_{n,R}$): $x_0 \rightarrow PU_{0,L}, x_0 \rightarrow PU_{0,L}, x_0 \rightarrow PU_{1,L}, x_1 \rightarrow PU_{0,L}, x_1 \rightarrow PU_{1,R}, x_2 \rightarrow PU_{1,R}, x_2 \rightarrow PU_{1,R},$ $x_2 \rightarrow PU_{0,R}, x_3 \rightarrow PU_{0,R}, x_3 \rightarrow PU_{0,R}, x_4 \rightarrow PU_{1,L}, x_4 \rightarrow PU_{1,L}$. Similarly, one can generate valid move code for other solutions. It should also be noted that values of the variables x_0, x_1 and x_2 are not computed, but loaded from the main memory. To load a variable value, we simply move the memory address of that variable to the left input buffer of the PU to which the variable is assigned. Since the memory addresses are known beforehand, we may place this move anywhere in the above sequence of moves and are not shown for simplicity.

In our ASP encoding, we start by extracting all variables from our basic block/program with the following code.

$$var(X) \quad :- \quad operand(X, Y, Z).$$
$$var(Y) \quad :- \quad operand(X, Y, Z).$$
$$var(Z) \quad :- \quad operand(X, Y, Z).$$

Our generate part demands every variable to be assigned to exactly one PU, in the sense that the variable should be produced by this PU and put to the output buffer. Furthermore, if two values V_1 and V_2 are assigned to the same PU, they have to be ordered (either $V_1 < V_2$ or $V_2 < V_1$).

$$1 \{asgn(VAR, PU) : pu(PU)\} 1 \quad :- \quad var(VAR).$$
$$order(V_1, V_2), order(V_2, V_1) \quad :- \quad asgn(V_1, PU), asgn(V_2, PU), V_1 \neq V_2.$$
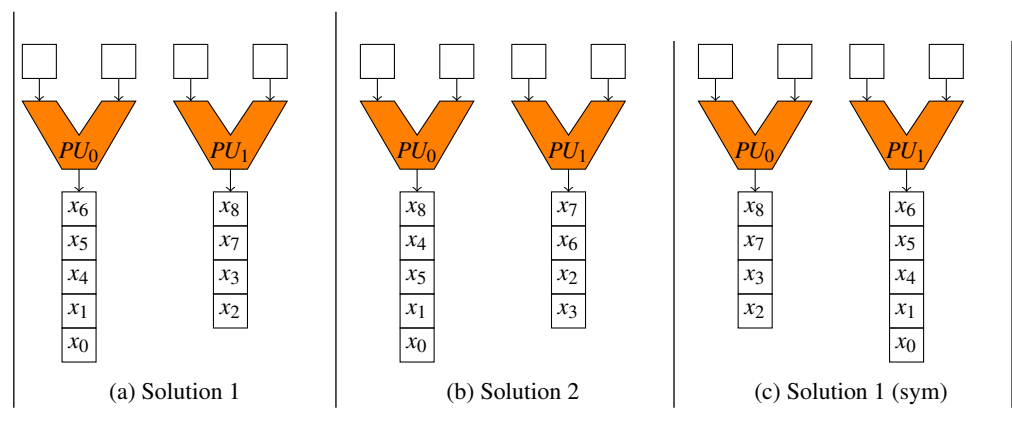
Figure 5: Three valid variable assignments and orderings that allow computation of the basic block in Figure 4 without any overhead

Our test part demands the order to fulfill additional constraints to allow the variable productions in a way that the computations of the basic block still can be performed. As shown in (Bhagyanath and Schneider 2016) and (Bhagyanath and Schneider 2017), it is sufficient that the order of two variables V_1, V_2 produced by a PU PU is preserved in their operands $V_{1,L/R}$ and $V_{2,L/R}$, if both operands are produced by the same PU PU_2.

$$
\begin{aligned}
order(V_{1,L}, V_{2,L}) \quad :- \quad & asgn(V_1, PU), asgn(V_2, PU), V_1 \neq V_2, order(V_1, V_2), \\
& operand(V_1, V_{1,L}, V_{1,R}), operand(V_2, V_{2,L}, V_{2,R}), \\
& asgn(V_{1,L}, PU_2), asgn(V_{2,L}, PU_2), V_{1,L} \neq V_{2,L}. \\
order(V_{1,R}, V_{2,R}) \quad :- \quad & asgn(V_1, PU), asgn(V_2, PU), V_1 \neq V_2, order(V_1, V_2), \\
& operand(V_1, V_{1,L}, V_{1,R}), operand(V_2, V_{2,L}, V_{2,R}), \\
& asgn(V_{1,R}, PU_2), asgn(V_{2,R}, PU_2), V_{1,R} \neq V_{2,R}.
\end{aligned}
$$

Additionally, the order must also preserve the data dependencies of the basic block/DAG. For example, if an operand $V_{1,L}$ is produced by the same PU as its consumer V_1, it is required that first the operand is produced and afterwards its consumer. This must be generalized not only to the operands, but to all predecessors of a variable V_1 in the initial DAG. Therefore, the initial DAG is constructed, and a predicate that holds if a variable is a predecessor in this DAG. This predicate can then be used in the constraint to remove all orders which do not correspond to the predecessors order.

$$
\begin{aligned}
node(X) \quad :- \quad & var(X). \\
edge\_initial(X,Y) \quad :- \quad & operand(Y,X,\_). \\
edge\_initial(X,Y) \quad :- \quad & operand(Y,\_,X). \\
rootNode(X) \quad :- \quad & not\ edge\_initial(\_,X), node(X).
\end{aligned}
$$

$$
\begin{aligned}
predecessor(X,Y) \quad :- \quad & edge\_initial(X,Y). \\
predecessor(X,Z) \quad :- \quad & predecessor(X,Y), predecessor(Y,Z).
\end{aligned}
$$

$$
\quad :- \quad predecessor(X,Y), asgn(X, PU), asgn(Y, PU), order(Y,X).
$$

To conclude, the transitive closure of the ordering relation is used in order to be able to compare every position of every variable on the same PU.

$$order(V_1, V_3) \quad :- \quad order(V_1, V_2), order(V_2, V_3).$$

Using this basic encoding, we can now use ASP to find the minimal amount of PUs needed to allow the execution of a given basic block:

$$pu(0..PUS - 1) \ :- \ amountPUs(PUS).$$
$$possiblePUAmount(0..max\_pus).$$
$$1\ \{amountPUs(N): \ possiblePUAmount(N)\}\ 1.$$
$$\#minimize\{N: amountPUs(N)\}.$$

The above encoding led to 8800 different solutions, 3 of them have been picked out and are shown in Figure 5. At a closer look, one can realize that the example (c) is a symmetric solution of (a). To avoid this overhead, symmetry breaking constraints have been added, which basically sort the PUs by the minimally assigned variable.

$$minimum(PU, S) \quad :- \quad S = \#min\{VAR: asgn(VAR, PU)\}, pu(PU), asgn(\_, PU).$$
$$\quad :- \quad minimum(PU, S), minimum(PU_2, S_2), PU_2 > PU, S > S_2.$$

This symmetry breaking constraint removes all rotations and combinations of PUs with the corresponding assignments. In numbers, $\frac{N}{amountPU!}$ many solutions are left if there are N schedules given for *amountPU* many PUs. For example, if there are $N = 2016$ schedules of basic block on 4 PUs, $\frac{2016}{24} = 84$ unique solutions are left after symmetry breaking. In our example (see Figure 4), the amount of results given by ASP is reduced by $\frac{8800}{2!}$ many solutions from 8800 to 4400.

4 Optimal Scheduling Regarding Execution Time

The main contribution of this paper is a new method to determine the execution time for scheduling a basic block. Similar to the SMT encoding (Bhagyanath and Schneider 2017), an assignment of variables to timeslots is used in this encoding. However, this approach did not scale well as the number of constraints grows enormously with the number of variables. Using ASP, we found a more direct way for determining the minimal execution time.

As stated in the previous section, the initial program/basic block can be represented by a DAG. The basic information added by a schedule is the introduction of additional edges into this DAG (all orders from the same PU which have not been ordered in the initial DAG). All answer sets are therefore the same as introducing all possibilities of additional edges into the graph with the only constraint that there are only minimal possible paths through the graph (minimal amount of PUs) without reaching a cycle.

From an abstract perspective, the longest path though this non-cyclic combined graph is the execution time/execution steps needed to compute the solutions of the basic block. By adding different costs for every OP-code, PU or adding costs for moving values, this can be easily refined to a particular real hardware implementation.

We illustrate our new approach by an example: Consider solution a) from Figure 5. It has the additional edges $x_0 \rightarrow x_1$, $x_1 \rightarrow x_4$, $x_4 \rightarrow x_5$, and $x_5 \rightarrow x_6$ introduced by the order of PU_0 and $x_2 \rightarrow x_3$, $x_3 \rightarrow x_7$, and $x_7 \rightarrow x_8$ introduced by the order of PU_1. If we add these edges to the initial DAG, we can find a longest path through this combined DAG representing the amount of steps needed to execute the basic block with this order on a SCAD machine. In case of solution 1, there are at most 5 steps.
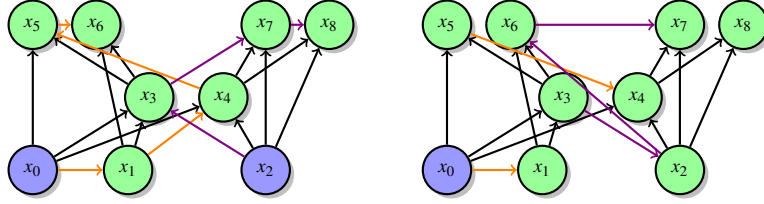
Figure 6: Combined DAG for Solutions 1 and 2 from Figure 5(a) and (b)

However, if we look at the combined DAG for Solution 2, we can see that there exists at least one path with cost 6. Although Solution 1 and 2 can both be scheduled on 2 PUs, Solution 1 would be a better choice for an optimal compiler as it requires less steps. This minimization can also be modeled in the ASP encoding: First, we constructed the combined DAG by adding all order relations introduced by our result to the initial DAG.

$$
\begin{aligned}
edge(X,Y) &:- edge\_initial(X,Y). \\
edge(X,Y) &:- order(X,Y). \\
initialNode(X) &:- not\ edge(\_,X),\ node(X).
\end{aligned}
$$

With the combined DAG, which represents all variable dependencies, we can calculate the longest path and take this as the execution time. Here, it is assumed that the combined DAG has no cycles, and therefore the length of the path can be used as cost measure.

$$
\begin{aligned}
pathCosts(X,1) &:- initialNode(X). \\
pathCosts(Y,N+1) &:- edge(X,Y),\ pathCosts(X,N),\ N<(M+1),\ amountVars(M).
\end{aligned}
$$

$$
maximalCost(N) \quad :- \quad N = \#max\{C : pathCosts(\_,C)\}.
$$

Finally, it is possible to answer the minimization question with a final minimize statement.

$$
\begin{aligned}
\#minimize\{N@1 : &\ maximalCost(N), N > \#inf\}. \\
:- &\ maximalCost(\#inf).
\end{aligned}
$$

With this minimize statement, we can determine all solutions that have minimal execution time for a given minimal number of PUs (the infimum #inf had to be used to ensure that the minimize statement does not allow to have no *maximalCost* defined at all). As our total results for Figure 4 to be scheduled on 2 PUs have been 8800 possible schedules, it was figured out that only 520 of them are optimal (260 with symmetry breaking). The solutions (a) and (c) shown in Figure 5 are minimal and have cost 5. Solution (b) is not optimal as it has cost 6.

However, are there even better schedules possible on a SCAD machine with more PUs? And how many PUs are needed to execute a program/basic block as fast as possible (with minimum execution time at all)? The answer to the first question is simple: The depth of the initial DAG represents the overall lower bound to the execution time. Therefore, the more interesting question is the second one.

The answer can be easily computed by our ASP encoding if we switch the priorities of the two minimize statements for PUs and execution time. One out of the 6912 possible minimal schedules (288 with symmetry breaking) for our example DAG from Figure 4 can be found in Figure 7. As it can be seen, our initial basic block can be scheduled on 4 PUs with the minimal cost of 3 (as this was the maximal depth in the initial DAG).
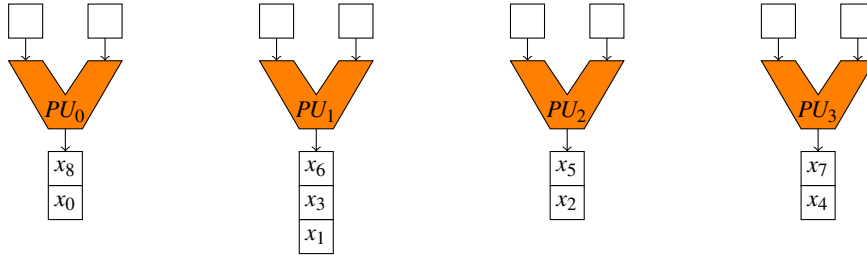
Figure 7: One example to minimize the amount of execution time and then find the minimal number of PUs needed to schedule the example basic block shown in Figure 4

5 Enhanced Optimal Schedulings

So far, we have been able to answer the questions about the minimal execution time, and the minimal execution time for the least possible amount of PUs. More realistic is to consider both resource and time constraints together instead of after each other. For example, we may wish to determine the minimal number of PUs to execute a given program/basic block within N steps. As another example, we may wish to determine the minimal execution time of a given program with a given number of PUs. To solve these problems, only minor changes are required in our ASP encoding which shows the benefits of using ASP. As the *maximalCost* is already available in our model, we can answer the first question by adding just one constraint that states that the cost does not exceed a given constant for maximal executions.

$$:- \quad maximalCost(N), \, N > max\_execution. $$

If we execute this ASP encoding on our example basic block from Figure 4 together with an upper bound of execution time 4, results as shown in Figure 8 can be found. We can see that we need at least 3 PUs if it is needed to have a real-time behaviour of at most 4 steps.
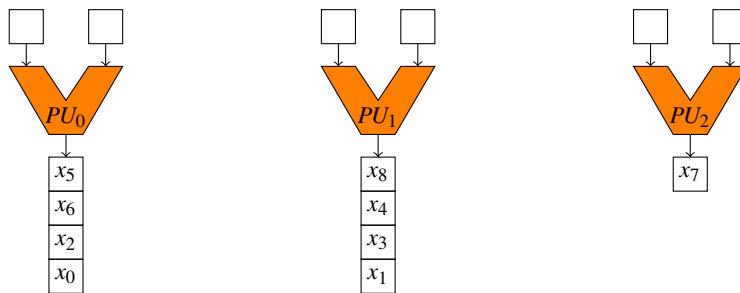

Figure 8: One example out of 8088 (without symmetries: 1348) to minimize the amount of PUs with an upper bound to 4 execution steps from the example basic block shown in Figure 4

Second, we can introduce a resource limitation of PUs by restricting the answers to not exceed our PU limitation:

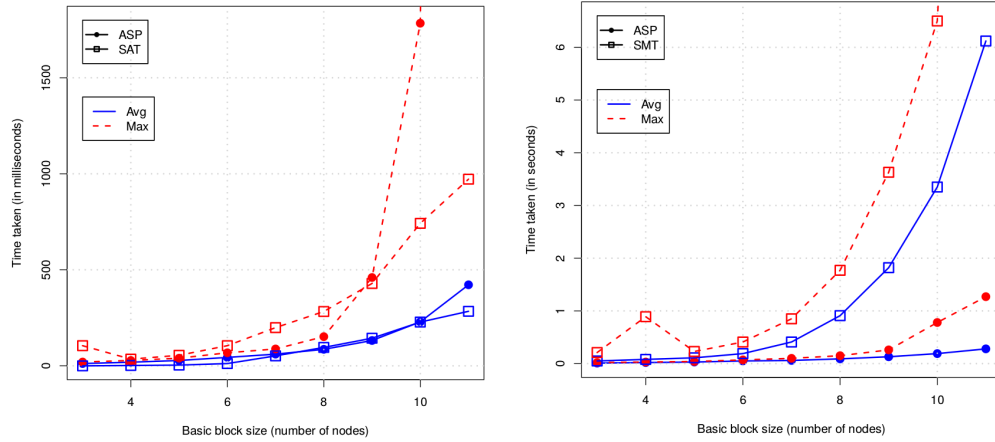$$:- \quad amountPUs(N), \, N > pus\_available. $$

Figure 9: (a) Average and maximum time required by SAT and ASP solvers to derive resource constrained schedules for programs of different sizes. (b) Average and maximum time taken by SMT and ASP solvers to derive time constrained schedules for programs of different sizes.

6 Comparison with SAT and SMT/Benchmarks

We used the same random basic block generator as in (Bhagyanath and Schneider 2016) to generate basic blocks whose DAG representations have a given size *n* and a given number of levels *l* [1]. For every pair (node,level), 1000 basic blocks were generated. Schedules are derived for each basic block using the ASP solver Clingo with the following optimization criteria: (1) Minimize number of PUs required to execute the basic block without overhead. This is used for a comparison with the SAT encoding in (Bhagyanath and Schneider 2016). (2) Minimize the number of PUs and then minimize the time needed to execute the basic block on a SCAD machine without overhead. This is used for a comparison with the SMT encoding in (Bhagyanath and Schneider 2017). All the runs are performed on an Intel Core-i5 (4 x 2.67 GHz) desktop computer with 8 GB RAM running Ubuntu 14.04, the same machine that has been used for the SMT and the SAT encoding. We used the newest Clingo version 5.2.2 for the experiments, our own SAT solver, and Microsoft's Z3 SMT solver.

The resulting execution times of Clingo are given in Figure 9, in comparison to SAT (a) and SMT (b). For the comparison with the SAT encoding, only PU optimization has been included in the ASP template to perform a fair comparison. We could not detect any significant differences between the average execution times. This is a quite good result because Clingo produced *all* possible answer sets, whereas the SAT encoding only produced one satisfying model. However, there seems to be the trend that ASP will lead to higher maximal CPU times compared to SAT.

Because the SMT encoding was searching the minimal execution time on minimal numbers of PUs, minimization of the execution time has been included to the ASP encoding with second preference. It was quite interesting to see that the CPU times needed by Clingo did even decrease compared to the version without execution time minimization. As minimizing the execution time of a schedule was straightforwardly added to ASP, this is directly visible in the result, as the ASP encoding outperforms the SMT encoding by far. We could easily compute schedules with minimal execution times up to 20 nodes without having memory, time or other resource problems.

---

[1]The tools are available at https://es.cs.uni-kl.de/tools/teaching/

7 Related Work

Answer set programming is known to be well-suited for solving scheduling problems (see e.g., (Dodaro and Maratea 2017) and (Balduccini 2011)) since optimization statements have found their way into ASP (Brewka et al. 2003). Optimal scheduling for multicore processors has been investigated in (Kumar and Delgrande 2009) using path cost minimization algorithms for weighted graphs. It allows mapping tasks to CPUs, if they are split before into independent tasks. The algorithms presented are directly usable for quadcore and dualcore CPUs and need manual changes for others.

Symbolic system synthesis has been performed in (Ishebabi et al. 2009) and (Andres et al. 2013). These synthesized systems are build to match best with a given problem instance, and find the most suitable system design. In our paper, we already have a system design, and want to find the best scheduling for this design.

There is a very interesting approach generating the best operations which lead to execution time minimization in (Brain et al. 2006) and (Crick et al. 2009). Although the workflow proposed there is focusing on optimal code synthesis, and not finding an optimal schedule, it is still related to this paper, especially because they introduced a levelized architecture for reducing the search space. The basic idea is kept in mind for future optimizations of our work. The work shown in (De Angelis et al. 2012) focused in a similar way on the generation of concurrent programs described by structural and behavioral properties.

8 Conclusions

This paper introduced and evaluated the optimal code generation of basic blocks for SCAD machines using ASP. The flexibility of ASP can be used to adapt existing encodings to solve new resource and time constrained optimization problems. The use of ASP solvers is competitive to SAT solvers, and outperforms the previously used SMT solvers. Additionally, ASP not only computes one optimal schedule, but all optimal solutions so that it is now possible to study their properties.

References

ALETÀ, A., CODINA, J., GONZÁLEZ, A., AND KAELI, D. 2007. Heterogeneous clustered VLIW microarchitectures. In *Code Generation and Optimization (CGO)*, San Jose, California, USA, pp. 354–366. IEEE Computer Society.

ANDRES, B., GEBSER, M., SCHAUB, T., HAUBELT, C., REIMANN, F., AND GLASS, M. 2013. Symbolic system synthesis using answer set programming. In P. CABALAR AND T. SON (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 8148 of *LNCS*, Corunna, Spain, pp. 79–91. Springer.

BALDUCCINI, M. 2011. Industrial-size scheduling with ASP+CP. In J. DELGRANDE AND W. FABER (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 6645 of *LNCS*, Vancouver, Canada, pp. 284–296. Springer.

BHAGYANATH, A., JAIN, T., AND SCHNEIDER, K. 2016. Towards code generation for the synchronous control asynchronous dataflow (SCAD) architectures. In R. WIMMER (Ed.), *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, Freiburg, Germany, pp. 77–88. University of Freiburg.

BHAGYANATH, A. AND SCHNEIDER, K. 2016. Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers. In E. LEONARD AND K. SCHNEIDER (Eds.), *Formal Methods and Models for Codesign (MEMOCODE)*, Kanpur, India, pp. 143–152. IEEE Computer Society.

BHAGYANATH, A. AND SCHNEIDER, K. 2017. Exploring the potential of instruction-level parallelism of exposed datapath architectures with buffered processing units. In A. LEGAY AND K. SCHNEIDER (Eds.), *Application of Concurrency to System Design (ACSD)*, Zaragoza, Spain, pp. 106–115. IEEE Computer Society.

BRAIN, M., CRICK, T., DE VOS, M., AND FITCH, J. 2006. TOAST: Applying answer set programming to superoptimisation. In S. ETALLE AND M. TRUSZCZYŃSKI (Eds.), *International Conference on Logic Programming (ICLP)*, Volume 4079 of *LNCS*, Seattle, WA, USA, pp. 270–284. Springer.

BREWKA, G., NIEMELA, I., AND TRUSZCZYNSKI, M. 2003. Answer set optimization. In G. GOTT-LOB AND T. WALSH (Eds.), *International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, pp. 867–872. Morgan Kaufmann.

CRICK, T., BRAIN, M., DE VOS, M., AND FITCH, J. 2009. Generating optimal code using answer set programming. In E. ERDEM, F. LIN, AND T. SCHAUB (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 5753 of *LNCS*, Potsdam, Germany, pp. 554–559. Springer.

DE ANGELIS, E., PETTOROSSI, A., AND PROIETTI, M. 2012. Synthesizing concurrent programs using answer set programming. *Fundamenta Informaticae 120*, 3-4 (July), 205–229.

DODARO, C. AND MARATEA, M. 2017. Nurse scheduling via answer set programming. In M. BALDUC-CINI AND T. JANHUNEN (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 10377 of *LNCS*, Espoo, Finland, pp. 301–307. Springer.

FELLER, M. AND ERCEGOVAC, M. 1981. Queue machines: An organization for parallel computation. In W. BRAUER, P. BRINCH HANSEN, D. GRIES, C. MOLER, G. SEEGMÜLLER, J. STOER, N. WIRTH, AND W. HÄNDLER (Eds.), *Conpar 81*, Volume 111 of *LNCS*, Nürnberg, Germany, pp. 37–47. Springer.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. clasp: A conflict-driven answer set solver. In C. BARAL, G. BREWKA, AND J. SCHLIPF (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 4483 of *LNCS*, Tempe, AZ, USA, pp. 260–265. Springer.

GUO, J., LIMBERG, T., MATUS, E., MENNENGA, B., KLEMM, R., AND FETTWEIS, G. 2006. Code generation for STA architecture. In W. NAGEL, W. WALTER, AND W. LEHNER (Eds.), *Euro-Par 2006 Parallel Processing*, Volume 4128 of *LNCS*, Dresden, Germany, pp. 299–310. Springer.

ISHEBABI, H., MAHR, P., BOBDA, C., GEBSER, M., AND SCHAUB, T. 2009. Application of ASP for automatic synthesis of flexible multiprocessor systems from parallel programs. In E. ERDEM, F. LIN, AND T. SCHAUB (Eds.), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Volume 5753 of *LNCS*, Potsdam, Germany, pp. 598–603. Springer.

KUMAR, V. AND DELGRANDE, J. 2009. Optimal multicore scheduling: An application of ASP techniques. In E. ERDEM, F. LIN, AND T. SCHAUB (Eds.), *Logic Programming and Nonmonotonic Reasoning (LP-NMR)*, Volume 5753 of *LNCS*, Potsdam, Germany, pp. 604–609. Springer.

LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. 1998. Space-time scheduling of instruction-level parallelism on a raw machine. In D. BHANDARKAR AND A. AGARWAL (Eds.), *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, USA, pp. 46–57. ACM.

SCHILLING, T., SJÄLANDER, M., AND LARSSON-EDEFORS, P. 2009. Scheduling for an embedded architecture with a flexible datapath. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Tampa, FL, USA, pp. 151–156. IEEE Computer Society.

SCHMIT, H., LEVINE, B., AND YLVISAKER, B. 2002. Queue machines: hardware compilation in hardware. In J. ARNOLD AND K. POCEK (Eds.), *Field-Programmable Custom Computing Machines (FCCM)*, Napa, California, USA, pp. 152–160. IEEE Computer Society.

SETHI, R. AND ULLMAN, J. 1970. The generation of optimal code for arithmetic expressions. *Journal of the ACM (JACM) 17*, 4 (October), 715–728.

SMITH, A., BURRILL, J., GIBSON, J., MAHER, B., NETHERCOTE, N., YODER, B., BURGER, D., AND MCKINLEY, K. 2006. Compiling for EDGE architectures. In *Code Generation and Optimization (CGO)*, New York, New York, USA, pp. 185–195. IEEE Computer Society.

ÄIJÖ, T., JÄÄSKELÄINEN, P., ELOMAA, T., KULTALA, H., AND TAKALA, J. 2016. Integer linear programming-based scheduling for transport triggered architectures. *ACM Transactions on Architecture and Code Optimization (TACO) 12*, 4 (January), 59:1–59:22.