

CHALMERS



An Approach to Scheduling in a Hardware-Software Co-Design Toolchain

*Master of Science Thesis
in the Programme Integrated Electronic System Design*

NIKITA FROLOV

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science & Engineering
Göteborg, Sweden, October 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

An Approach to Scheduling in a Hardware-Software Co-Design Toolchain

NIKITA FROLOV

© NIKITA FROLOV, October 2011.

Examiner: PER LARSSON-EDEFORS

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, October 2011

Abstract

Much like VLIW, statically scheduled architectures that expose all control signals to the compiler offer much potential for highly parallel, energy-efficient performance. A cornerstone to effective compilation for such architectures is an effective solution to the phase ordering problem, i.e., planning the cooperation between instruction scheduling and register allocation.

Existing heuristic algorithms that approach this problem are hard to analyze and to break down to reusable concepts that might lead to better algorithms, which is one of the major obstacles for adoption of VLIW architectures. An approach based on a combination of a domain-specific language (DSL) embedded in a higher-order language and a constraint satisfiability engine makes it possible to structure the problem and abstract away from generic search space exploration methods.

Bau is a novel compilation infrastructure that leverages the LLVM compilation tools and the MiniSAT solver to generate efficient code for one such exposed architecture, FlexCore. A compiler construction library is built that allows the compiler writer to express scheduling and resource constraints declaratively, as a set of constraints in a DSL, each describing one property of a valid schedule. It provides a framework to rapidly modify aspects of a backend and explore tradeoffs between compilation time and quality of compiled code.

A compiler implemented using this library can generate programs that are 1.2–1.5 times more compact than ones generated either by a baseline MIPS R2K compiler or a basic-block-based, sequentially phased scheduler. However, further optimization of the instruction lowering pass is needed to improve performance.

Acknowledgments

Foremost, I'd like to thank Per Larsson-Edefors, Magnus Sjölander and Sally McKee for supervision, guidance, ideas and insights they provided. Additionally, I'd like to thank Tung Hoang, Kasyab Subramaniyan and Alen Bardizbanyan for sharing their expertise of FlexCore internals, and John Hughes for a discussion that helped me define the scope of this thesis.

Nikita Frolov, Göteborg, October 2011.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Functional languages	2
1.3	Hardware-software co-design	4
1.4	Challenges	6
2	Toolchain architecture	8
2.1	Compilation flow	10
2.2	Target description	13
2.2.1	Instruction lowering	13
2.2.2	Interconnect configuration	15
2.3	FlexCore	15
2.3.1	Binary operations	15
2.3.2	Memory operations	16
2.3.3	Control flow	16
2.3.4	Printing the assembly	18
3	Scheduling as CSP	19
3.1	Problem encoding	20
3.2	Instruction placement	21
3.3	Value placement	21
3.4	Code insertion	23

3.5	Encoding with triples	23
3.5.1	Execution unit properties	24
3.5.2	Dataflow properties	25
3.5.3	Non-orthogonal datapath interconnect	26
3.5.4	Location properties	26
3.5.5	Value path properties	26
3.5.6	Value paths of globally live values	30
3.6	Encoding with pairs	30
3.6.1	Execution unit properties	31
3.6.2	Instruction properties	32
3.6.3	Value properties	33
3.6.4	Location properties	34
3.7	Solution interpretation	35
4	Defining constraints	37
4.1	Translation to CNF	38
4.2	Constraint templates	38
4.3	Quantification	39
4.3.1	Universal quantifiers	39
4.3.2	Existential quantifiers	40
4.3.3	Composition strategy	41
5	Results	42
6	Conclusion and future work	44
7	Related work	47
A	Bau source code	54
B	Integration with LLVM	73

List of Figures

1.1	Datapath interconnect	6
2.1	Compiler source code layout	10
2.2	Compilation flow	11
2.3	Top-level organization of RTN assembly	11
2.4	Unscheduled and scheduled forms of assembly	12
2.5	Description of dataports before and after scheduling	12
2.6	Target type class	13
3.1	A value should be stored after it is defined and at before it used. . .	27
3.2	Allocating memory in a SAT solver instance	31
3.3	Liveness in a data flow graph	35
4.1	From problems to constraints	39
4.2	Restricting constraint scope	39

List of Tables

5.1	Benchmark compilation times (in seconds)	42
5.2	Number of instruction words in innermost functions	43
5.3	Benchmark execution times (in FlexCore simulator cycles)	43

1

Introduction

1.1 Overview

Engineering is an art of compromise. Beginning with requirements definition, the design process of a technological artifact revolves around finding the best possible trade-off between properties of tools and materials that engineers have at their disposal. Implications to be considered belong to all stages of the artifact lifecycle: design, production, use, maintenance and recycling. Computer systems are built from many tightly coupled hardware and software components, and it is not only the choice of components themselves that matters. It is not enough to design for performance, power consumption, production, operation and service costs — the time-to-market bears great importance in dynamic economies.

Domain-specific languages and hardware-software co-design, while perceived as unrelated areas of research and belonging accordingly to fields of computer science and computer engineering, have much in common in their incentives, methods and goals. Both were conceived to provide the system designer with a simple tool that will combine abstract expressiveness of a high-level language with fine-grained resource management of a low-level language and elaborate the idea of *interface design* in order to exploit the trade-off between abstraction and control over implementation details, in the opposite ends of the hardware-software stack.

Functional languages are being increasingly adopted as the compiler technology

brings the performance on par with widespread imperative languages and emerging DSLs reduce the learning curve. However, there is still room for improvement because members of the functional programming community are mostly focused on front-end optimizations, which are close to their area of expertise, and rely on the infrastructure created for imperative languages in the first place to generate machine code. Moreover, the targeted machines were designed to run imperative languages, as well. From the compiler middle- and back-ends' point of view and from point of view of the hardware, functional languages still are second-class citizens.

At the same time, the progress in microtechnology has made vast resources available to computer engineers to work with. They are expected to build machines that will meet the ever-growing needs of software but continuous increase of frequency has proven to be ineffective due to power and heat constraints. Exploitation of parallelism is one design approach that is widely studied. Another approach is concerned with a shift from fixed, "one-size-fits-all" hardware to more flexible architectures, involving decisions on portions of software to be implemented in hardware. The area of hardware-software co-design studies what costs are associated with different variants of partitioning of the system into hardware and software components, how coarse should the components be, and how to automate these decisions.

Two problems stated above are, in fact, solutions for each other. Functional languages lack extensive toolchain and hardware support, but flexible hardware allows the system designer to choose what computational primitives to support. This thesis has started as an attempt to connect the demand of imposed high-level programming with supply of raw processing power. After conducting a pre-study on what improvements should be made (Section 1.2) and what can be practically implemented (Section 1.3), a particular problem was chosen that unites both areas (Section 1.4).

1.2 Functional languages

Attempts to increase performance of functional programs by running them on a machine designed for *graph reduction* as opposed to a von Neumann machine

were undertaken in early 1970s when the first LISP machines were built [1]. By the end of 1980s the performance of commercially available computers designed with imperative programs in mind was growing at a faster pace than university research groups could design and fabricate innovative language-specific architectures. The language technology community has moved on to studies of effective implementation of functional languages on top of commodity hardware. Today, with availability of rapid prototyping platforms, such as FPGAs, the interest has renewed, and new experimental architectures appeared, for example, Reduceron [2].

A number of abstract machines were designed to address the gap between models of computation based on term rewriting (utilized by functional languages) and changing the memory state (utilized by mainstream, von Neumann-esque computer architectures) and to serve as an intermediate representation in a compiler. In the *pure graph reduction* approach, programs are transformed into sequences of combinators, either belonging to a predefined set [3] or derived from the program (*supercombinators*) [4]. The resulting code still retains much of the interpretative behavior of a graph reduction machine. Influenced by the concept of supercombinators, the approach of *programmed graph reduction* relies on static analysis more than pure graph reduction to decrease need for dynamic dispatch. The control flow (that is, the order in which expressions should be evaluated) is extracted from the program graph, and the program is expressed with a fixed set of instructions that are easy to translate to the native instruction set of a von Neumann architecture. Implementations of this approach include the G-machine [5] and its variants and successors [6], among them being the STG-machine used by the popular Glasgow Haskell Compiler [7].

After a functional program is transformed from an expression graph to sequential code, it is compiled either for a virtual machine, such as JVM (Scala, Clojure), .NET (F#) or BEAM (Erlang), or for hardware (Haskell, ML family). This stage of compilation is commonly broken down into a middle-end, performing optimizations independent of the source language and the targeted hardware architecture, and a back-end generating native code. From the middle-end point of view, functional programs have the following differences from imperative ones:

- aggressive pointer chasing

Representation of functions and values as linked-list structures requires several pointer dereferences for a single reduction step, which causes many branch mispredictions and cache misses, especially with tagless abstract machines. The STG-machine design does not match superscalar processors well, and ILP is not exploited. [8]

- data is immutable and is referenced almost immediately after allocation;

Due to short lifetimes of data, large cache associativity has less impact, and different data layouts are required to exploit locality [9]. Prefetching of frequently accessed data degrades performance, although it does decrease L2 cache misses in some cases. Improved cache performance primarily comes from an increase in L2 cache size [10], but caches that support runtime tuning of associativity, line size and block replacement policies might be useful for some applications [11].

1.3 Hardware-software co-design

An influential paper by Steele and Sussman [12] is an early example of putting the problem of hardware-software partitioning into the context of language technology. A LISP system is described as a hierarchy of virtual machines where a boundary between hardware and software can be placed arbitrarily. In the 30 years that have passed since the publication of this paper, the costs of hardware prototyping have radically dropped, thanks to customer-reprogrammable devices such as FPGAs. It is possible now to design and evaluate a new processor architecture completely in the lab. However, a straightforward hardware implementation of a language interpreter cannot compete with results of 30 years of improvements in computer architecture and, especially, compiler technology.

A trade-off between versatility of “sea of gates”-like devices and maturity of general purpose processors is found among template-based and reconfigurable processors. Configurable architectures can trade generality for increased performance and lower power for domain-specific applications [13]. For instance, the type and number of functional units and application-specific accelerators can be selected at

design time (TCE[14], MOLEN[15]), and the datapath between these units can then be specified to support the specific communication behaviors of applications in the target domain. Both approaches are implemented by the FlexCore architecture [16].

Just as programmed graph reduction simplifies the execution-time planning of computation, static instruction scheduling reduces the processor complexity and can potentially provide better results than dynamic scheduling, because static compilers have more resources at their disposal. Architectures that expose the control signals for all execution units and the interconnect addresses to the compiler require wide control words (as in VLIW [17]). Customizing both computational units and their communication medium can improve computational efficiency in terms of performance and power, but allowing such flexibility complicates the compiler, which must satisfy many more concurrent resource constraints and instruction dependencies than typical, phase-based code generators. Nonetheless, transforming dynamic instruction scheduling done with fixed-function logic into static instruction scheduling done by a compiler enables dramatically more compact schedules.

The FlexSoC scheme provides an architectural template, FlexCore, of a design-time configurable (application-specific), exposed architecture whose datapath units communicate through an interconnect that, in its most complex instantiation, forms a crossbar switch. Figure 1.1 depicts the output of a particular datapath unit (that is, an output port) connected to a register. The output of the register is routed to several input multiplexers, each driving the input of a datapath unit (that is, an input port).

Assuming there are M output ports and N input ports, the interconnect template can, at most, support $N \cdot M$ communication paths. To avoid area, delay, and power dissipation overheads from additional wiring and multiplexers (muxes), paths that turn out to be superfluous (not used in any of the intended applications) should never be instantiated [18].

FlexSoC's exposed architecture is not a conventional instruction set architecture (ISA) and has no fixed set of assembly instructions. It is up to the designer to decide what execution units will be connected to the datapath and what their functionality would be. Operations at the machine level can instead be expressed in *register transfer notation* (RTN) specifying operations to be performed on output

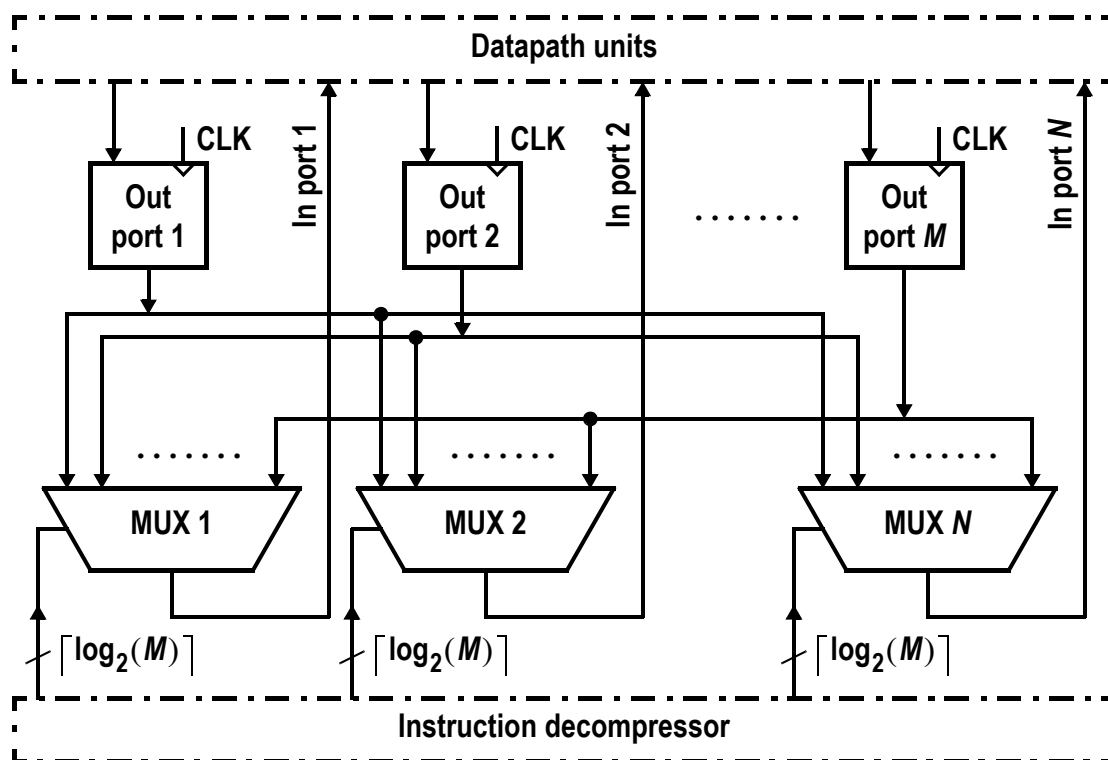


Figure 1.1: In its most complex configuration the datapath interconnect acts as a crossbar switch, supporting communication from each of M output ports to each of N input ports.

port registers of the various datapath units. The output port from which a value is read represents the address of the interconnect multiplexer, and the operation represents the control signals (that is, the op-code) to a specific datapath unit. A special RTN code reads a value from the register file without affecting any output ports from the execution units. Decoded control words are simply concatenations of the RTN operations of all datapath units for a given clock cycle. (Compact representation of these control words along with the design of efficient decoders represent an orthogonal path of research [19].)

1.4 Challenges

Mapping of high-level language primitives to dedicated hardware blocks demands extensive support in the compiler back-end. Compiling for an architecture that

is design-time configurable with an exposed datapath requires a flexible compiler back-end. Even the simplest changes in microarchitecture, such as adding more units of a known type (for example, more ALUs) and/or modification of the interconnect, require changes in the instruction scheduler and the register allocator, and adding support for application-specific accelerators with new semantics is not trivial. Existing reconfigurable computing toolchains (including FlexSoC's) do not address these problems.

Traditional compilers rely on fixed algorithms to schedule for microarchitectures with known interconnect and known numbers and types of functional units. For instance, out-of-order architectures implement routines for assignment instructions to units in microcode, so the ISA can remain unchanged when the number of units changes. In contrast, in an exposed architecture the forwarding of data between datapath units is explicitly compiler-controlled, and the compiler has to be provided with information about available units.

In von Neumann architectures, instructions use the register file to read operands and write results, but hardware forwarding is often introduced to reduce latency. In an architecture with a flexible datapath template architectural registers belong to output ports of datapath units, not to the register file, automatically storing the “last computed value”. The name of a register where an operand can be found is not known until the instruction computing the operand will be assigned to a unit. Vice versa, an instruction cannot be assigned to a unit before the locations of its operands are known. Thus, the compiler cannot perform instruction scheduling and register allocation sequentially, as a traditional compiler would do.

The compilation issues arising from performing instruction scheduling and register allocation are commonly known as the *phase ordering problem*. While partial solutions for this problem in von Neumann architectures are used to find a better trade-off between ILP and memory pressure, in exposed architectures a complete solution is required just to get the programs run. This thesis researches a method to perform instruction scheduling and register allocation concurrently, globally, in a single phase. Section 2 discusses the general design of a flexible compiler. Section 3 formulates the phase ordering problem as a *constraint satisfaction problem*. Section 4 introduces a domain-specific language for describing scheduling problems.

2

Toolchain architecture

Two open source compiler infrastructures with wide community support — GCC [20] and LLVM [21] — can be used to implement a language- and target-independent parts of a language toolchain. Both implement complete compiler functionality (from language front-ends, to multiple optimization passes, instruction schedulers, and target-specific code generators) and are open to customization for non-orthodox applications.

Modifying GCC can require daunting effort. On the other hand, the highly modular LLVM uses a single intermediate representation for every compilation step, making front-ends, back-ends, and optimization passes independent from each other — they can be developed in separate source trees and even written in different languages. Many languages can already be compiled to LLVM bytecode, and many optimization techniques are implemented as LLVM passes. Although there is much LLVM support for developing RISC or CISC back-ends (for example, instruction selectors and schedulers, register allocators, and peephole optimizers), these cannot be readily reused for an exposed architecture.

LLVM provides a DSL, `TableGen`, that is used for abstract target and machine descriptions. Parts of a back-end can be automatically generated from `TableGen` descriptions, but even for mainstream architectures, such as x86 or ARM, it is not comprehensive. Some of machine instruction semantics still have to be given as C++ code, and the glue code (also in C++) is still required to develop a

complete LLVM back-end[22]. Automatic compiler reconfiguration is not possible with `TableGen`, and manual reconfiguration requires knowledge of both C++ and `TableGen`, which is not common among hardware developers who would most likely have the knowledge of architecture and be involved in back-end writing.

Code generator components included with LLVM are not of great use with exposed architectures, either, because they were designed for a sequential compilation flow. LLVM instruction schedulers and register allocators are implemented as separate *passes*, and a parallel approach is necessary, as explained in Section 1.4. Code analyses implementations, such as liveness analysis, cannot be reused, as well, because LLVM does not implement them in a way to work on the common LLVM internal representation, but on machine-specific ones. Machine-specific representations that are predefined by LLVM are not general enough to support some essential features of exposed architectures, such as compiler-controlled value forwarding. They are integrated with available register allocators too tightly. For example, they are used to maintain a portion of the register allocator state[23]. Thus, Bau has to reimplement some of the functionality provided by `Machine*` classes of LLVM.

The toolchain described in this thesis includes its own back-end infrastructure independent from LLVM. LLVM still provides middle-end functionality and makes it possible to reuse language front-ends targeting LLVM. Back-end infrastructure is implemented as a Haskell library, `Bau` (Appendix A). `Bau` allows the compiler writer to capture both target-independent functionality of a back-end and target-specific hardware details. The compiler based on `Bau` is a standalone program that that compiles LLVM bytecode files into RTN code. RTN code has to be assembled then to the binary code of the target architecture. An example of compilation flow integration is given in Appendix B.

Source tree layout of a complete compiler is shown in Fig. 2.1. The compiler is targeting variants of the FlexCore processor implementing the FlexSoC scheme. Variant-independent code is organized into the *Bau* library. Section 2.1 describes proposed compilation flow. Section 2.2 introduces a DSL for target platform description (modules `RTN`, `Target` and `AsmWriter`). Problems arising when scheduling for an exposed architecture are elaborated in Sections 3 (modules `Schedule` and `Constraints`) and 4 (module `Primitives`).

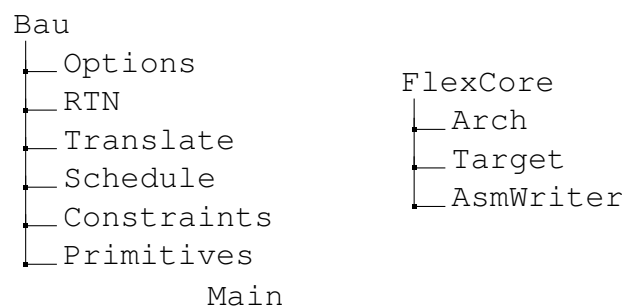


Figure 2.1: Compiler source code layout

2.1 Compilation flow

A back-end targeting an exposed architecture has to implement instruction decoding and reordering, replacing the complicated control logic of a conventional CISC/RISC-like pipeline. The output of such backend has the form of microinstructions, or RTN. Compiling LLVM bytecode to RTN assembly requires two steps: lowering the LLVM instructions to the RTN micro-operations (μops) supported by a given architectural instance, and allocating resources to assign μops to particular execution units and to assign variables to registers. Figure 2.2 illustrates the flow chart of activities implementing these steps.

The part of the RTN assembly definition that is specific to an architecture variant has provided by the compiler writer. Variant-specific data types, such as unit names (or *unit types*, not to be confused with Haskell types), unit modes and *dataports*, should be declared as extensions of data type families, and a type variable (`arch`) is used to enumerate members of these type families. In order to separate instruction lowering from resource allocation, RTN assembly has two forms — the unscheduled form and the scheduled form. Both forms share top-level organization of the code into functions annotated with parameter lists `[Name]` and information about global (`Lives`) and local (`Edges`) value dependencies and basic blocks, as shown on Fig. 2.3.

The difference between unscheduled and scheduled forms is made at the `BasicBlock` level. The unscheduled form uses unscheduled μops , not assigned to cycles (corresponds to the `BBU` constructor). In the scheduled form (`BBS` constructor), μops are organized into instruction words, as shown on Fig. 2.4.

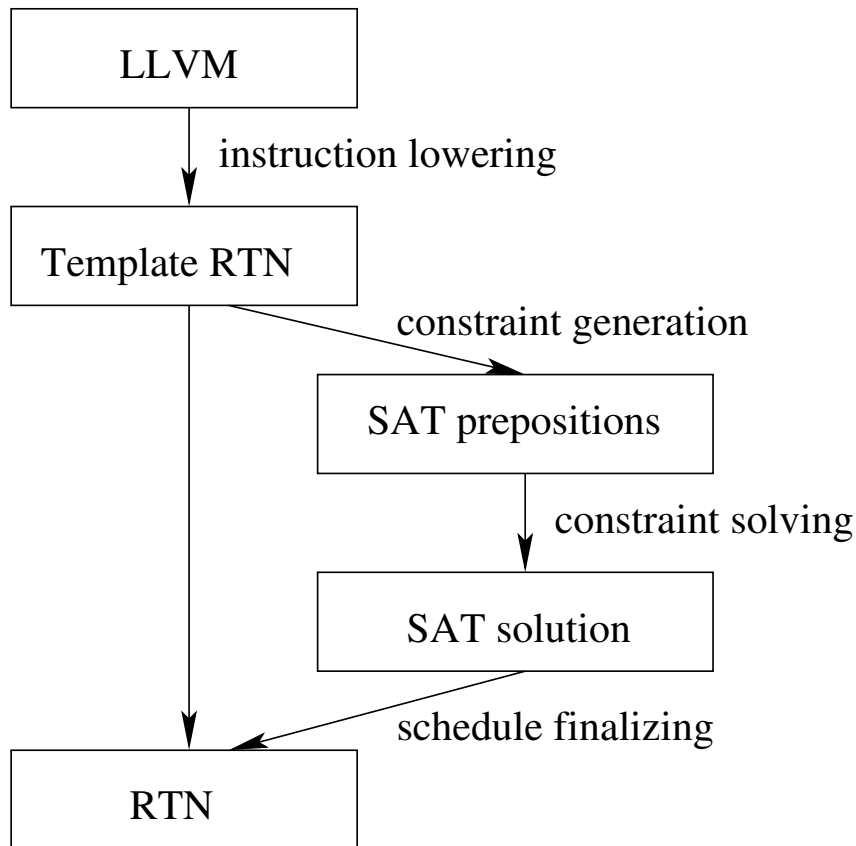


Figure 2.2: Compilation flow

```

data Target arch => Assembly arch = A Name [Function arch]
data Function arch = F Name [Name] [BasicBlock arch] (Lives, Edges)

```

Figure 2.3: Top-level organization of RTN assembly

A `MicroOp` defines a value with a `Name` that is computed on a unit of a given `UnitType` in a particular `Mode`. Data dependencies of a value are expressed with a list of operands `[DataPort]`. Every value name can be defined only once, by one μop . This implies that RTN assembly is in static single assignment form (SSA), like LLVM bytecode. An optionally non-empty list of control dependencies `[Name]` can be provided.

In a scheduled `BasicBlock` μops are also mapped to unit instances. This is done by setting the `Int` field of every `MicroOp` to a positive value. Lists of operands (`[DataPort]`) are also rewritten after scheduling to refer to *dataport*

```

data BasicBlock arch = BBU Label [MicroOp arch]
                    | BBS Label [InstrWord arch]
data MicroOp arch = MO Name (UnitType arch) (Mode arch)
                  [(DataPort arch)] Int [Name]
newtype InstrWord arch = IW [MicroOp arch]

```

Figure 2.4: Unscheduled and scheduled forms of assembly

```

data DataPort arch = DP PortType (UnitType arch) Int Name | DPU ArgDesc
data PortType = In | Out deriving Show

```

Figure 2.5: Description of dataports before and after scheduling

names. A dataport name is defined as the name of a unit instance that computes a given operand. When a μop becomes scheduled, LLVM SSA value names of its operands (`ArgDesc`) are replaced with dataport names, as shown on Fig. 2.5.

Static single assignment form of LLVM bytecode is preserved in this transformation, and lowering can be accomplished by instantiating instruction templates defined in terms of variant-specific μops and provided by the compiler writer. Besides, SSA form has well-known benefits as the foundation for an intermediate representation:

- resource allocation is delayed, therefore fewer false dependencies are introduced, and better utilization of processor resources can be achieved;
- value names are never redefined, so every register transfer has a unique name, and greater freedom is achieved in scheduling transfers through the interconnect.

The second step is resource allocation — choosing units to map μops to, and the dataports from which to read operands. Because every unit has a register for every output dataport to store its result, the choice of dataports for a μop depends on the unit used to compute a given operand, and allocation of registers has to consider that values can be forwarded between units directly through the interconnect or saved in pipeline buffers and not in the register file. Resource allocation can be expressed as a constraint satisfaction problem, and recent progress in SAT solver implementation makes such tools attractive building blocks for powerful

```
class Target arch where
  data UnitType arch
  data Mode arch
  constraints :: [Constraint arch]
  lower :: (Name, InstrDesc) -> [MicroOp arch]
```

Figure 2.6: Target type class

schedulers, due to both the simplicity of formulating and refining problems and to the impressive performance [24]. The scheduling problem can then be generalized by expressing it as a set of constraint templates and then using a template engine to produce constraints; a stand-alone SAT solver finds the solution for a given set of resources.

2.2 Target description

Both LLVM and RTN code are represented with a hierarchy of Haskell abstract data types, and the translator can be implemented by straightforward, recursive pattern matching. While traversing the code tree, the translator calls a target-specific, instruction lowering function for each leaf. The parts of the compiler that are specific to a given architectural instance must then: 1) define abstract datatypes that represent new execution units, and 2) define lowering functions that transform LLVM instructions to RTN code by implementing the Target type class (Fig. 2.6).

2.2.1 Instruction lowering

Most of LLVM instructions can be put into one of four groups: binary arithmetic and logic operators, basic block terminators, memory operations and operations with data structures (type conversion and vector modification). Two important instructions that do not fall into these groups are `phi` (merges SSA values defined in predecessor basic blocks) and `call` (performs a function call). Arithmetic, logic and memory operators usually can be translated to a dataflow graph of μ ops, because access to special-purpose registers is not required. However, terminator

instructions often depend on precise placement of pipeline control μ ops (such as register file operations or jumps), and control flow aspects have to be expressed in lowered code. The [Name] field of `MicroOp` type should be used to list values that have to be available for reading through the interconnect immediately before a given instruction starts executing.

The matching is done for a specific LLVM instruction and operand type, and generated code has to preserve the SSA form. The following example matches the LLVM `add` instruction applied to 32 bit integer operands. The generated RTN code consists of one ALU instruction with mode bits set for the addition operation, and the SSA value name `v` defined in LLVM bytecode is used as the RTN SSA value name, as well. The instruction does not have any control dependencies, hence the list of additional dependencies is empty. Its data dependencies are operands `a` and `b` for which dataports have to be defined by the scheduler (this is why the DPU dataport constructor is used):

```
lower (v, IDBinOp BOAdd (TDInt U 32) a b) =
    [ MO v ALUOp AO_ADDU (DPU a) (DPU b) 0 [] ]
```

By matching on custom LLVM intrinsic functions, it is possible to generate code for accelerators with arbitrary functionality. Because calls to intrinsics are represented with function calls that can have an arbitrary number of arguments, it is necessary to generate the list of operands for an accelerator instruction by mapping the unscheduled dataport constructor to all intrinsic arguments:

```
lower (v, IDCall t "llvm.viterbi" args) =
    [ MO v VITERBI None (map (\a -> (DPU $ AV a)) args 0 []) ]
```

This matching rule means that an LLVM instruction that calls (`IDCall`), which is an intrinsic function named `llvm.viterbi()`¹, is translated into a new RTN μ op defined as `VITERBI`. No mode bits (`None`) are set, and intrinsic arguments are simply passed as μ op operands. Complex translation rules can generate multiple μ ops.

While being associated with the LLVM `phi` instruction, the function prologue is generated by the scheduler that guarantees that ϕ -values defined in predecessor

¹All intrinsics are defined in `llvm.*` namespace by convention.

basic blocks are being placed in the same memory location that instructions using the defined values will refer to. References to function arguments are also inserted by the scheduler, so no explicit prologue code has to be generated by the instruction lowering function.

As symbolic value names are preserved at this stage, value forwarding and spilling between output port registers, the register file, and memory is not defined until later. Forwarding and spilling code is produced by constraints described in Section 3.7.

2.2.2 Interconnect configuration

An execution unit is characterized not only by its intended functionality, but also by types of inputs and outputs and by latency. A configuration file contains a list of triples, where every triple denotes a type of execution unit, the number of units of this type available, and the unit type's delay:

```
type Resources = forall arch . Map (UnitType arch) (Int, Int)
```

For example, a configuration file entry for a Viterbi accelerator might be written like this:

```
(VITERBI, (1, 3))
```

Such entry would mean “this architecture variant has one (1) Viterbi accelerator, and its delay is three (3) cycles”. The scheduler relies on this information to avoid hazards when allocating resources.

2.3 FlexCore

The target description for the baseline variant of FlexCore is shipped as a part of the Bau toolchain. It can be modified to create a new, accelerator-extended variants or serve as a primer for writing descriptions.

2.3.1 Binary operations

Since the baseline FlexCore implementation includes a multiplier in addition to an ALU, lowering of arithmetic and logic instructions requires two rules:

```

lower (v, IDBinOp BOMul (TDInt False _) a b) =
  [ MO v Mult None [DPU a, DPU b] 0 [] ]
lower (v, IDBinOp m (TDInt False _) a b) =
  [ MO v ALUOp (mode m) [DPU a, DPU b] 0 [] ]
where mode m =
  case m of {BOAdd -> ADDU ; BOSub -> SUBU ;
             BOAnd -> AND ; BOOr -> OR ; BOXor -> XOR ;
             BOShL -> SLL ; BOLShR -> SRL ; BOAShR -> SHR }

```

2.3.2 Memory operations

Lowering of memory access instructions has to consider LLVM pointer arithmetic.

Lowering of loads and stores is straightforward:

```

lower (v, IDLoad t a) = [ MO v LSRead (LSW 4) [DPU a] 0 [] ]
lower (v, IDStore t a b) = [ MO v LSWrite (LSW 4) [DPU b, DPU a] 0 [] ]

```

But LLVM assembly supports compound types too. Besides global variables defined at compile time, it is possible to dynamically allocate data structures, as well, with the `alloca` instruction:

```

lower (v, IDAlloca t tsize n) =
  [ MO (v) RegRead sp [] 0 []
  , MO (v++"_0") PC Imm [(DPU $ AI $ tsize*n)] 0 []
  , MO (v++"_1") ALUOp SUBU [(DPU $ AV v), (DPU $ AV (v++"_0"))] 0 []
  , MO (v++"_2") RegWrite sp [(DPU $ AV (v++"_1"))] 0 [] ]

```

The `GetElementPtr` instruction, which is used for accessing members of data structures, does not have to be lowered explicitly because pointer offsets are calculated statically, and addresses are calculated in the value-reading code inserted during scheduling.

2.3.3 Control flow

Every basic block in LLVM ends with an instruction that modifies the control flow (the *terminator* instruction). Lowering of basic block terminator instructions is straightforward:


```

lower (v, IDRet t r) = [ MO (v++"_0") RegWrite v1 [DPU r] 0 [] ]
                        ++ lower (v, IDRetVoid)
lower (v, IDRetVoid) = [ MO (v++"_10") RegRead ra [] 0 []
                        , MO (v++"_11") PC JumpSA [a v 10] 0 [] ]

lower (v, IDBrCond c l1 l2) =
  [ MO (v++"_t") PC BNEZA [DPU l1, DPU c] 0 []
  , MO (v++"_f") PC BEQZA [DPU l1, DPU c] 0 [v++"_t"] ]
lower (v, IDBrUncond l) = [ MO v PC JumpSA [DPU l] 0 [] ]

```

Notably, there is no rule for the `switch` instruction. It does not need to be implemented because LLVM optionally includes a pass that lowers it to ordinary branches, which saves the backend developer's effort.

A calling convention has to be implemented both by lowering rules and scheduling constraints because passing arguments between functions belongs to register allocation, which must be managed by the scheduler:

```

lower (v, IDCall t f args) =
  [ MO (v++"_0") RegRead sp [] 0 []
  , MO (v++"_1") PC Imm [DPU $ AI 32] 0 []
  -- push $fp
  , MO (v++"_2") ALUOp SUB [a v 0, a v 1] 0 []
  , MO (v++"_4") RegRead fp [] 0 []
  , MO (v++"_6") LSWrite (LSW 4) [a v 2, a v 4] 0 []
  -- push $ra
  , MO (v++"_3") ALUOp SUB [a v 0, a v 2] 0 []
  , MO (v++"_5") RegRead ra [] 0 []
  , MO (v++"_7") LSWrite (LSW 4) [a v 3, a v 5] 0 []
  -- save $sp in $fp
  , MO (v++"_8") RegWrite fp [a v 0] 0 []
  -- update $ra
  , MO (v++"_9") PC GetPC [] 0 []
  , MO (v++"_10") PC Imm [DPU $ AI 96] 0 []
  , MO (v++"_11") ALUOp ADD [a v 9, a v 1] 0 []
  , MO (v++"_12") RegWrite fp [a v 0] 0 []

```

```
-- jump to entry point  
, MO (v++"_13") PC JumpSA [DPU f] 0 [v++"_12"]  
, MO (v) RegRead v1 [] 0 [v++"_13"]]
```

2.3.4 Printing the assembly

Assembler development is beyond the scope this thesis. We leverage the existing FlexCore assembler to transform RTN assembly to FlexCore machine code.

3

Scheduling as CSP

Scheduling is a classical example of a constraint satisfaction problem (CSP). The unscheduled RTN representation can be transformed into valid RTN by substituting full names of execution units and data ports. When the choice of an execution unit instance is a matter of an instance number, the choice of data ports to read operands from is not straightforward. Because of spilling, an operand might come not only directly from another execution unit, but also from a buffer unit, from the register file, or from the load/store unit (essentially, from memory). Thus, two subsets of constraints can be identified — one to define the instruction placement in space (units) and time (cycles) and another to define forwarding of values between units and cycles.

Entries of the schedule can be represented with a tuple of parameters and constraints imposed on these parameters would define a valid schedule. Indexed variables assigned with true by the SAT solver will denote valid entries. Statements, as shown in Section 3.2, can then be rewritten as propositional logic expressions [25]. The problem for a SAT solver to satisfy would be a conjunction of constraints for every instruction, pair of instructions, or execution unit. We can only ban invalid schedules, not mandate valid ones.

Besides two major sets of constraints, more can be thought of, for example, to guide the scheduler towards a more power- or performance-optimized solution, by casting away schedules that are suboptimal by given criteria. Additional con-

straints will not require changes to the solution interpreter, since they will only refine existing solutions, and not bring in a new class of solutions. The same holds, for example, for trace scheduling — the interpreter will already have a generic procedure for code rearrangement.

After the solver returns relations, they can be used by a solution interpreter that builds the actual RTN code. Interpretation of instruction schedules is one-to-one template substitution — every μop is annotated with the ID number of a unit instance it is scheduled on. Spilling code generation is performed when the value schedule is interpreted — not only dataport names are supplied into a μop instead of value names, but additional μops are inserted to ensure that values are stored in memory after they are computed and read back when they are required by other μops . Different spilling strategies can be implemented to either optimize for access latency or for total number of transfers. Current version of Bau uses a simple strategy that allocates faster memory for values that are to be used sooner.

3.1 Problem encoding

In order to encode a problem as a SAT constraint, it is necessary to identify what is the meaning of a single SAT variable. The scheduler is supposed to find a relationship between μops (or instructions) and values defined by them, *unit instances*, *memory locations* and *cycles*. The simplest encoding would be using four indices for each variable. But a four-dimensional array of variable would not only grow fast in size together with growth of program sizes but also will require many additional constraints to forbid obviously wrong combination of indices, such as instances and locations with non-matching value types.

Memik and Fallah’s paper [25] uses three indices for denoting operations, units and cycles, correspondingly, and the initial approach used in this thesis was to reuse constraints defined by them for instructions and try to encode the problem of value placement in a similar fashion. Section 3.5 describes this approach. However, the unit instance index is only used for checking if an instance is not assigned with more than one instruction at a time and if it is assigned with an instruction of matching type. In all other constraints, it only increases complexity of a problem by an order of magnitude. Three indices are also redundant for the value assignment problem.

It became possible to encode the scheduling problem with tens of thousands of variables and millions clauses instead of hundreds of thousands of variables and tens to hundreds millions of clauses. The approach described in Section 3.6 does not only reduce the solution time, but also the memory required.

3.2 Instruction placement

After the program has been expressed in terms of μ ops implemented by available execution units, assignment of operations to units and values to registers (both output port registers and register file) can be performed. The scheduling problem can be defined as the following [25]:

- every instruction should be assigned to exactly one unit and exactly one cycle;
- every execution unit performs just one instruction at a time;
- types of operands and result should match types of execution unit and registers;
- no instruction appears before its arguments have been defined; and
- no instruction appears after its result is used.

3.3 Value placement

FlexCore architecture establishes two major classes of registers — unit output port registers and registers provided by units attached to the interconnect. Every execution unit has a register that holds the output value¹, and an implementation of the FlexSoC scheme (for example, FlexCore) can have dedicated buffering units that do not perform any computation but are used as scratch registers and can be accessed faster than the regular register file. Different classes of registers do not share a uniform interface — an RTN instruction can only reference output ports, but in order to read or write a value from or to a buffer or register file additional spilling code is required.

¹A unit may compute more than one output value and have multiple output registers

A *value path* is a set of slots in the value placement schedule that belong to consecutive cycles between slots where the defining (*def*) and using (*use*) instructions are placed. A value can be passed between execution units in several ways:

- directly through the interconnect (only supported in the version of Bau current to this thesis if *def* and *use* are placed on consecutive cycles)
- through the buffering unit
- through the register file
- through the stack (which is known as spilling in conventional compilers)

Template RTN still retains SSA form, so the number of users of every value is finite and known. There should be a path between a value definition and a value use, and for values with many users those paths may overlap. If a value has many users across the program, the value path can become sophisticated. Value paths define where a value will reside at a given point in time, and in the first iteration the properties of a valid value placement schedule can be defined as following:

- a memory location can hold just one value at a time;
- types of a memory location and of a value should match; and
- a value should be stored continually at the same location after it is defined and before it is used.

It should be noted that characteristics of different memory levels do not have to be considered by the scheduler, so, for example, no preference between placing a value in a register or on the stack is made. The task of the scheduler is to ensure that resource conflicts are absent, but the differentiation between memories of low (buffers and registers, available only in limited numbers) and high latency (the stack) is coarse at this compilation step. The solution interpreter has to perform further analysis and determine what placement of values suits the optimization criteria better (Section 3.7).

Every basic block has its independent instruction schedule, because μ ops cannot be moved between basic blocks without an analysis that will prove that code movements will not change the semantics of a program. Currently, Bau does not implement this kind of analysis. Nevertheless, values may be live across basic

blocks, and value placement constraints cannot consider independent basic blocks. A limited form of liveness analysis has to be performed to determine what values are live in a given basic block and, vice versa, in what basic blocks is a given value live. More fine-grained analysis (cycle-relative, as performed in conventional compilers) is not possible because instruction and value placements are interdependent.

3.4 Code insertion

Some tasks in the compilation flow that are not related to resource allocation still cannot be solved by lowering rules alone. Parts of template code have to be inserted conditionally by the scheduler. They include:

- merging of ϕ -values by storing them in one location;
- choice between buffers, registers, or spilling;
- generation of code to read/write values from buffers, registers, or stack;
- passing arguments to a function on a call;
- reading function arguments;
- reading function return value.

3.5 Encoding with triples

The triple encoding described in this section results in a model by one order of magnitude larger than one generated in the pair encoding described in Section 3.6 and was not ultimately used for implementation of Bau.

An instruction schedule consists of triples of microinstruction name (that is, name of SSA value defined by it), name of an execution unit it is assigned to (that is, unit type + unit number) and cycle number. Every possible schedule entry can be encoded by a Boolean variable with three indexes x_{ouc} , where o corresponds to value name, u to unit name, and c to cycle number. The maximum possible number of cycles C in the schedule is equal to the sum of latencies of all instructions in a basic block to be scheduled. Every μop and execution unit instance are also given a numerical identifier with the maxima of O being equal to the number of

microinstructions in a basic block and U being equal to the number of all unit instances regardless of their type.

A value schedule consists of triples of SSA value name, name of a memory location (would that be a buffering unit, a register in the register file, or a memory address) and cycle number in a basic block. Every possible schedule entry can be encoded by a Boolean variable with three indexes y_{olc} , where o corresponds to value name, l to location name, and c to cycle number. Not surprisingly, many constraints resemble the ones for instruction scheduling. Considerations for numbering of locations and cycles are also the same (Section 3.2). The maximum location index is calculated as the sum of the maximum number of values that never are used outside of a basic block that defines them and the number of all values that are transferred between basic blocks.

3.5.1 Execution unit properties

Every unit runs no more than one instruction at a time

For every unit instance u there should not be any pair of variables assigned with true at the same cycle:

$$\forall u \in U \forall c \in C \forall o_1, o_2 \in O \quad \bar{A}((o_1, u, c), (o_2, u, c)) : o_1 = o_2 \quad (3.1)$$

In basic Boolean operators:

$$o_1 \neq o_2, \bigwedge_{\substack{u \in U \\ c \in C}} \overline{(o_1, u, c) \wedge (o_2, u, c)} \quad (3.2)$$

Unit and instruction types match

If types of an instruction and an execution unit do not match (for example, an ALU operation cannot be assigned to a load-store unit), corresponding variables should never be assigned with true:

$$\forall o \in O \forall u \notin U(o) \forall c \in C \quad \bar{A}(o, u, c) \quad (3.3)$$

In basic Boolean operators:

$$\bigwedge_{\substack{o \in O \\ u \notin U(o) \\ c \in C}} \overline{(o, u, c)} \quad (3.4)$$

3.5.2 Dataflow properties

Every instruction has exactly one entry in the schedule

As the instruction types are known, the search space can be reduced by limiting the number of units where an instruction o can be placed to those of a corresponding type. First, an instruction o should have no more than one entry in the schedule. This is achieved by demanding impossibility of every pair of variables to be assigned with true:

$$\begin{aligned} \forall o \in O \forall u_1, u_2 \in U(o) \forall c_1, c_2 \in C \\ \exists((o, u_1, c_1), (o, u_2, c_2)) : (u_1, c_1) = (u_2, c_2) \end{aligned} \quad (3.5)$$

In basic Boolean operators:

$$(u_1, c_1) \neq (u_2, c_2), \quad \bigwedge_{\substack{o \in O \\ u_1, u_2 \in U(o) \\ c_1, c_2 \in C}} \overline{(o, u_1, c_1) \wedge (o, u_2, c_2)} \quad (3.6)$$

Conjunction is commutative, and the order of variables in a pair is not important, so redundant pairs have to be removed by imposing an additional restriction on unit and cycle indices: $u_2 \cdot c_n + c_2 > u_1 \cdot c_n + c_1$.

Second, an instruction o should have no less than one entry, which is assured by a disjunction of all possible places of i in the schedule:

$$\forall o \in O \forall u \in U(o) \forall c \in C \exists(o, u, c) \quad (3.7)$$

In basic Boolean operators:

$$\bigwedge_{\substack{o \in O \\ u \in U(o) \\ c \in C}} (o, u, c) \quad (3.8)$$

Data dependencies are not broken

Given a set LE of pairs of instructions (that is, local value dependency edges) where instruction use depends on the result of instruction def and has a delay d , it could be assumed that use should never be scheduled before cycle d after def :

$$\begin{aligned} \forall def, use \in LE \forall u_1 \in U(def) \forall u_2 \in U(use) \forall c_1, c_2 \in C \\ \exists ((use, u_1, c_1), (def, u_2, c_2)) : c_1 + d \geq c_2 \end{aligned} \quad (3.9)$$

In basic Boolean operators:

$$c_1 + d \geq c_2, \quad \bigwedge_{\substack{def, use \in LE \\ u_1 \in U(def) \\ u_2 \in U(use) \\ c_1, c_2 \in C}} \overline{(def, u_1, c_1) \vee (use, u_2, c_2)} \quad (3.10)$$

3.5.3 Non-orthogonal datapath interconnect

Instructions in a def - use cannot be assigned to unit instances that lack a route in a customized interconnect as defined in configuration file:

3.5.4 Location properties

Similarly to execution units, memory locations can process (that is, store) only one value at a time and only values of corresponding types can be processed. Constraints 3.1 and 3.3 can be applied to value schedules unchanged to enforce these two properties.

3.5.5 Value path properties

A graphical illustration of the value path concept for a locally live value is given on Fig. 3.1.

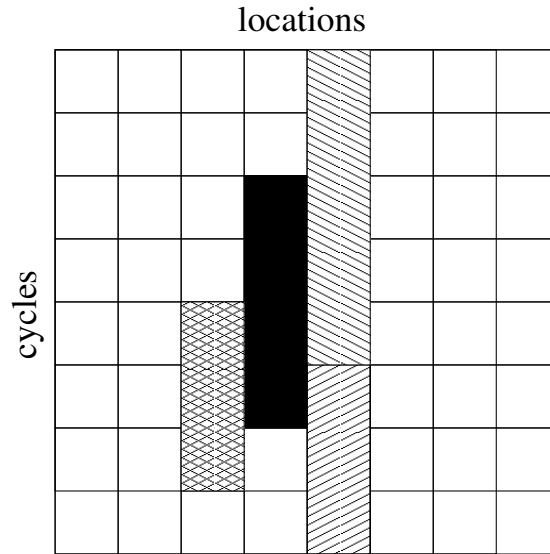


Figure 3.1: A value should be stored after it is defined and at before it used.

Value path that spans inside of one basic block:

$$C_{VPL} = \{c : c \in C, c_{def} < c < c_{use}\} \quad (3.11)$$

A value path can't include more than one location

For every assignment of *def* and *use* instructions to task slots on non-consecutive cycles, a value path should exist such that *def* is always assigned to the same memory location. For every possible cycle range between *def* and *use* all paths that include more than one location should be forbidden:

$$\begin{aligned} &\forall[(def, u_{def}, c_{def}), (use, u_{use}, c_{use})] \forall c \in C \exists(l_i, c) : \\ &\{l_1, l_2 \dots l_n\} \subseteq L, l_1 \neq l_2 \neq \dots \neq l_n, c \in C, c_{use} < c < c_{def}, \end{aligned} \quad (3.12)$$

In basic Boolean operators:

$$\begin{aligned}
n = c_{use} - c_{def} - 1, l_1 \dots l_n \in \binom{L}{n}, l_1 \neq l_2 \neq \dots \neq l_n, \\
\bigwedge_{(def, use) \in LE} \overline{(def, u_{def}, c_{def}) \wedge (use, u_{use}, c_{use})} \wedge \bigwedge_{\substack{l \in \binom{L}{n} \\ c \in C_{VPL}}} (def, l, c)
\end{aligned} \tag{3.13}$$

Innermost NAND clauses in (3.13) are only false when all their member variables are true. For every value path that includes more than one location, the constraint has one NAND clause forbidding it, so paths with just one location are allowed.

A value path can't have gaps

A value should never be overwritten in the middle of its value path. For every combination of *def-use* cycle ranges and location

$$\begin{aligned}
\forall [(def, u_{def}, c_{def}), (use, u_{use}, c_{use})] \forall l \in L \forall c \in C \exists (def, l, c) : \\
c > c_{def}, c < c_{use}
\end{aligned} \tag{3.14}$$

In basic Boolean operators:

$$\begin{aligned}
\bigwedge_{(def, use) \in LE} \left(\overline{[(def, u_{def}, c_{def})] \vee \bigvee_{\substack{l \in \binom{L}{n} \\ c \in C_{VPL}}} (def, l, c)} \right) \wedge \\
\overline{[(use, u_{use}, c_{use})] \vee \bigvee_{\substack{l \in \binom{L}{n} \\ c \in C_{VPL}}} (def, l, c)}
\end{aligned} \tag{3.15}$$

(3.15) is only true when all subexpressions (implications) are true, that is, the value is placed at least one location in a cycle between *def* and *use*.

A value cannot be placed outside its value path

All placements of a value outside of its value path should be explicitly forbidden, because to store an undefined value is impossible and to store a used one is unnecessary.

$$\begin{aligned} & \forall [(def, u_{def}, c_{def}), (use, u_{use}, c_{use})], \bar{A}(def, l, c) : \\ & l \in L, c \in C, 1 \leq c \leq c_{def}, c_{use} < c \leq |C| \end{aligned} \quad (3.16)$$

In basic Boolean operators:

$$\begin{aligned} & (use, def) \in LE, l \in L, c \in C, 1 \leq c \leq c_{def} \\ & \overline{(def, u_{def}, c_{def})} \vee \overline{(def, l, c)} \end{aligned} \quad (3.17)$$

$$\begin{aligned} & (use, def) \in LE, l \in L, c \in C, c_{use} < c \leq |C| \\ & \overline{(use, u_{def}, c_{def})} \vee \overline{(def, l, c)} \end{aligned} \quad (3.18)$$

Implication in (3.17) is only false when a placement appears before *def*, and implication in (3.18) is only false when a placement appears after *use*.

Locally live values can be eligible for forwarding

If and only if *def* and *use* are placed in consequent cycles, then *def* can be forwarded directly through the interconnect and does not have to be stored, in which case there is no value path to be defined by (3.12), (3.14) and (3.16).

$$\forall l \in L, c \in C, [(def, u_{def}, c_{def}), (use, u_{use}, c_{def} + 1)] \leftrightarrow \bar{A}(def, l, c) \quad (3.19)$$

In basic Boolean operators:

$$\bigwedge_{\substack{(use, def) \in LE \\ u_{def}, u_{use} \in U \\ c_{def} \in C}} \left(\bigwedge_{l \in L, c \in C} [(\overline{def, u_{def}, c_{def}}) \vee (\overline{use, u_{use}, c_{def} + 1}) \vee (\overline{def, l, c})] \right) \quad (3.20)$$

The implication in (3.20) is only true when all member variables, that is, a placement of a value in a location is forbidden when *def* and *use* are placed in consecutive cycles, and, vice versa, placement of *def* and *use* in consecutive cycles is forbidden, when a value is placed in a location.

3.5.6 Value paths of globally live values

A value path of a globally live value (*global value path*) spans across several basic blocks and can include branches and loops. It can be modelled as a directed graph with vertices representing basic blocks where the value is live and edges connecting pairs of consecutive basic blocks if the value is live at the end of the predecessor, and the successor is using the value or if its respective successors are.

Constraints defined in (3.12), (3.14) and (3.16) do not have to be modified to ensure existence of global value paths. However, the exact method of quantifying over individual cycles is implementation-specific (Section 4.3.1).

3.6 Encoding with pairs

An encoding that uses three indexes for every variable is excessive, as, for example, in most cases every instruction has just one unit instance to be placed on, and the choice of execution unit is obvious. Similarly with value placement, the beginning of a value path always takes place on the cycle following the one where a value is defined, which makes the value cycle index redundant.

The approach to encoding ultimately used in Bau uses two two-dimensional tables, one to relate μ ops to cycles and another to relate values (which have one-to-one relationship with μ ops) to memory locations. The upper estimate of cycle lengths of a schedule is a sum of all μ ops' latencies, and a better estimate is a topic for future work. The upper estimate of location number is a sum of amounts of

buffers and registers, and the number spill slots for the current scheduler iteration. The code on Fig. 3.2 illustrates use of the `relation` function that allocates an array of Boolean variables in the SAT solver.

```
(emptyISched, emptyVSched) :: Solver -> Resources arch
                                -> BasicBlock arch -> Relation Task Slot
emptyISched s res (BBU label ops) =
    relation s ((0, 0), (length ops - 1, maxCycleIndex res ops - 1))
emptyVSched s res (BBU label ops) =
    relation s ((0, 0), (length ops - 1, maxLocationIndex res - 1))
```

Figure 3.2: Allocating memory in a SAT solver instance

3.6.1 Execution unit properties

No more than N instructions of a given type can be executed in one cycle

The choice of a unit instance to assign an instruction to can be made by the solution interpreter provided no more instructions are scheduled in one cycle than there are unit instances available that can run this type of instructions. This constraint replaces both constraints that mandate assignment one instruction to a unit instance and correspondence of instruction and instance types. It is sufficient to check that at least N+1 instructions are not scheduled at the same time:

$$\forall ut \in UT \forall o \in O : \forall c \in C, o_1 \dots o_n \in \binom{n}{O} \quad (3.21)$$

$$type(o) = ut, n = units(ut), \exists ((o_1, c), \dots, (o_n, c))$$

In basic Boolean operators:

$$type(o) = ut, \bigwedge_{\substack{ut \in UT \\ o \in O \\ c \in C}} \overline{\bigwedge_{i \in \binom{n}{O}} (o_i, c)} \quad (3.22)$$

3.6.2 Instruction properties

The properties of the dataflow graph remain the same with this approach except that variables lack the unit instance index.

Every instruction has exactly one entry in the schedule

First, an instruction o should have no more than one entry in the schedule. This is achieved by demanding impossibility of every pair of variables to be assigned with true:

$$\forall o \in O \forall c_1, c_2 \in C \neg((o, c_1), (o, c_2)) : c_1 = c_2 \quad (3.23)$$

In basic Boolean operators:

$$c_1 \neq c_2, \bigwedge_{\substack{o \in O \\ c_1, c_2 \in C}} \overline{(o, c_1) \wedge (o, c_2)} \quad (3.24)$$

Second, an instruction o should have no less than one entry, which is assured by a disjunction of all possible places of i in the schedule:

$$\forall o \in O \forall c \in C \exists (o, c) \quad (3.25)$$

In basic Boolean operators:

$$\bigwedge_{\substack{o \in O \\ c \in C}} (o, c) \quad (3.26)$$

Data dependencies are not broken

Given an instruction use dependent on the result of instruction def and a delay d , it could be assumed that use should never be scheduled before cycle d after def :

$$\forall def, use \in LE \forall c_1, c_2 \in C \neg((use, c_1), (def, c_2)) : c_1 + d \geq c_2 \quad (3.27)$$

In basic Boolean operators:

$$c_1 + d \geq c_2, \quad \bigwedge_{\substack{def, use \in LE \\ c_1, c_2 \in C}} \overline{(def, c_1) \vee (use, c_2)} \quad (3.28)$$

3.6.3 Value properties

Every instruction defining a value has exactly one entry in the schedule

First, every instruction def should have no more than one entry in the schedule. This is achieved by demanding impossibility of every pair of variables to be assigned with true:

$$\forall (def, use) \in E \forall l_1, l_2 \in L \bar{\exists}((def, l_1), (def, l_2)) : l_1 = l_2 \quad (3.29)$$

In basic Boolean operators:

$$l_1 \neq l_2, \quad \bigwedge_{\substack{def, use \in E \\ l_1, l_2 \in L}} \overline{(def, l_1) \wedge (def, l_2)} \quad (3.30)$$

Second, an instruction def should have no less than one entry, which is assured by a disjunction of all possible places of i in the schedule:

$$\forall def, use \in E \forall l \in L \exists (def, l) \quad (3.31)$$

In basic Boolean operators:

$$\bigwedge_{\substack{def, use \in E \\ l \in L}} (def, l) \quad (3.32)$$

No instruction not defining a value has entries in the schedule

In order not to confuse the solution interpreter, all instructions that don't define values and can't result in writing a value to a location should never be assigned with a location:

$$\forall o \notin E \forall l \in L \neg \Delta(o,l) \quad (3.33)$$

In basic Boolean operators:

$$\bigwedge_{\substack{o \notin E \\ l \in L}} \overline{\Delta(o,l)} \quad (3.34)$$

3.6.4 Location properties

Don't store values that can be forwarded

A location with index 0 is a virtual location used for values that can be forwarded directly and thus don't have to be assigned with a real location. If defining and using instruction can be scheduled on consecutive cycles, then this virtual location should be used:

$$\forall (def, use) \in LE \forall c_1, c_2 \in C, c_1 + d_{def} = c_2 \exists (def, 0) \quad (3.35)$$

In basic Boolean operators:

$$c_1 + d_{def} = c_2 \bigwedge_{\substack{def, use \in LE \\ c_1, c_2 \in C}} [\overline{\Delta(def, c_1)} \vee \overline{\Delta(use, c_2)} \vee \Delta(def, 0)] \quad (3.36)$$

No writes to a location before stored value is read

A location cannot be assigned to two values that can potentially be live at the same time. Due to interdependency of instruction scheduling and register allocation problems in architectures with a flexible datapath it is not possible to perform a full-fledged liveness analysis, as it is impossible to calculate cycle precise liveness ranges based on instruction placement alone, so this constraint forbids many valid schedules. This is an area for future improvement, because register reuse decreases and thus unnecessary spilling may be performed. A limited analysis is performed to show that, for example, values dependent on each other, values never used outside of their defining basic blocks and values live in non-overlapping loops and

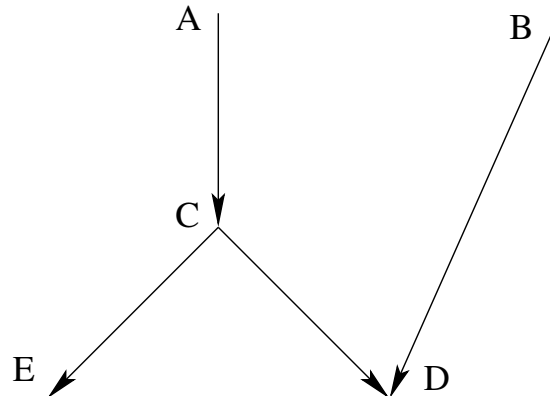


Figure 3.3: Values A and C cannot compete for the same memory location at the same time, but A and B or B and C could be competing. Despite belonging to the same path, E can compete with C, because D depends on C.

branches are never live together and don't have to be included in the constraint (Fig. 3.3). S is a subset of O produced by this analysis.

$$\forall l \in L \forall (o_1, o_2) \in S \quad \bar{A}((o_1, l), (o_2, l)) \quad (3.37)$$

In basic Boolean operators:

$$\bigwedge_{\substack{o_1, o_2 \in S \\ l \in L}} \overline{(o_1, l) \wedge (o_2, l)} \quad (3.38)$$

3.7 Solution interpretation

The schedule returned by the SAT solver contains the information about on what cycles to place instructions generated by the lowering pass and memory access instructions inserted by the scheduler, and names of memory locations to place SSA values in. Indices of unit instances are missing, but it is guaranteed that every cycle contains no more instructions than there are units of a given type available. The solution interpreter has to choose numbers of instances and fill in the names of dataports to read the values from.

It is possible that the solver is unable to find a schedule for a given number of memory locations. In this case the solver is restarted with a different number of allowed spills.

4

Defining constraints

A SAT solver interface DSL — `satchmo` library ¹, which is based on an extension to the `State` monad, was initially used as a foundation for this DSL for description of scheduling constraints. It implements the SAT monad that performs marshalling of SAT problems expressed as Haskell values into DIMACS format accepted by SAT solvers and unmarshalls solutions back to Haskell values. The basic building block for SAT problems is `assert`, a monadic function that stores individual clauses (disjunctions of boolean variables, possibly negated) in the SAT monad. A series of `asserts` makes a conjunction of clauses, allowing for representation of a CNF expression. After the problem is constructed from clauses, the SAT solver should be invoked with `solve`. Assignments to boolean variables can be extracted from the SAT monad with `decode`.

`satchmo` implements a primitive for logical relations — assignments of truth values to tuples. Elements of relations would correspond to indexed boolean variables that constitute boolean propositions representing the constraints, the data structure representing the relation (a Haskell `Array`) can be then used to translate unscheduled RTN code to its scheduled form (Section 3.7).

However, on large problems `satchmo` has shown to be a performance bottleneck because problem representations have to be constructed twice — first, in the SAT monad, and second, as an internal data structure of MiniSAT. It became

¹<http://dfa.imn.htwk-leipzig.de/satchmo/>

impractical on problems with more than several millions of clauses. A minimalist drop-in interface to MiniSAT Haskell wrapper was developed in form of `assert` and `solve` functions behaving similarly to ones provided by `satchmo`. Also, this interface can be more easily extended to support MiniSAT incremental solving than `satchmo`, which will prove useful with one of the planned Bau extensions.

4.1 Translation to CNF

All-purpose algorithms for translation of logical operators to CNF often give an exponential growth of the expression size, either in terms of clauses, or auxiliary variables. The set of supported logical operators should be carefully designed, but even a restricted operator set can be used to construct powerful constraints, for example, translation of following operators to CNF is straightforward:

- AND: $a_1 \wedge a_2 \wedge \dots \wedge a_n$ (already in CNF)
- OR = $a_1 \vee a_2 \vee \dots \vee a_n$ (already in CNF)
- NAND = $\overline{a_1 \wedge a_2 \wedge \dots \wedge a_n} = \bar{a}_1 \vee \bar{a}_2 \vee \dots \vee a_n$
- NOR: $\overline{a_1 \vee a_2 \vee \dots \vee a_n} = \bar{a}_1 \wedge \bar{a}_2 \wedge \dots \wedge \bar{a}_n$
- Exclusive OR: $a \oplus b = \overline{a \leftrightarrow b} = (\bar{a} \vee b) \wedge (\bar{b} \vee a)$

Constraints described in Sections 3.2 and 3.3 are built using only these simple operators. However, resulting boolean expressions still cannot be trivially represented in Haskell code, which gives the use case for this DSL.

4.2 Constraint templates

Fig. 4.1 shows how a higher-level constraint is defined as a function of architecture variant description, unscheduled RTN assembly and memory allocated in a SAT solver instance.

Many constraints are intended to work on the level of a basic block, and the environment binding code is the same in every such constraint. Such constraints can make use of the combinator shown on Fig. 4.2 to setup their environment.

```

type Relation a b = Array (a,b) Lit
type Schedule = (Relation Task Slot, Relation Task Slot)
type Problem = (Resources, [Schedule], [BasicBlock], Edges)
type Constraint = Problem -> IO ()

```

Figure 4.1: From problems to constraints

```

type BBProblem = [MicroOp arch], Resources arch, Schedule, [Edge]
bbConstr :: (BBProblem -> IO ()) -> Constraint

```

Figure 4.2: Restricting constraint scope

There is no need to have separate templates for instruction-only constraints and constraints imposed on both instruction and value schedules. While not every basic block has a value schedule (ones that do not define values do not have value schedules), the template does not have to check if a basic block defines values. Value constraints do not consider non-defining basic blocks because their top universal quantifier quantifies over *def-use* edges. Non-defining blocks have none, so they are not affected.

4.3 Quantification

All constraints presented in Section 3 have similar structure. A basic operator (as selected in Section 4.1) is applied to variables that belong to a defined domain. Scheduling constraints are thus defined by nesting an operator that will enforce the truth value for one variable, for a pair of variables, or more (existential quantifier) inside of an operator that identifies variables, pairs of variables or larger collections from the schedule (universal quantifier).

4.3.1 Universal quantifiers

Most commonly, quantification is performed over all elements of a set of tasks, resources or cycles, and the ranges can be obtained with `allTasks`, `allResources` and `allCycles` functions. Most constraints will need quantification over slots, to enforce a task-related property, or tasks, to enforce a slot-related property.

```
allSlots :: [Slot]
```

```
taskSlots :: -> Range -> Index -> [Entry]
```

```
taskSlotPairs :: Range -> Index -> [(Entry, Entry)]
```

A special case is when a subset of slots is selected to enforce properties of dependency graphs:

```
depSlotPairs :: Range -> LEdge -> [(Entry, Entry)]
```

```
brokenDepSlotPairs :: Range -> LEdge -> [(Entry, Entry)]
```

Or vice versa, properties can be enforced for execution units or locations:

```
slotTaskPairs :: Range -> Index -> [(Entry, Entry)]
```

```
defUsePairs :: [LEdge] -> [(LEdge, LEdge)]
```

```
consDepSlots :: Range -> LEdge -> [(Entry, Entry)]
```

4.3.2 Existential quantifiers

Most of the inner expressions include only a single variable or a pair of variables. For single variables, implementations of OR and NOR are provided:

```
atLeastOne :: Schedule -> [Entry] -> IO ()
```

```
noOne :: Schedule -> [Entry] -> IO ()
```

Variable pairs are commonly used in NAND expressions:

```
neverBoth :: [(Entry, Entry)] -> IO ()
```

```
neverAll :: [Entry] -> IO ()
```


4.3.3 Composition strategy

The following combinator implements nesting of a universal and an existential combinator:

```
nest2 :: [a] -> (b -> IO ()) -> (a -> b) -> IO ()
```

A complete constraint could be written like following (3.27 is given as an example):

```
defBeforeUse =  
  bbConstr $ \ (s, res, (isch, _), ops, l_deps) ->  
    nest2  
      l_deps  
      (brokenDepSlots (allSlots isch))  
      (neverBoth s isch)
```

5

Results

For comparison of two toolchains, FlexTools [26] and newly developed Bau, three benchmarks were selected from EEMBC benchmarks suite — autocorrelation (Autocor), fast Fourier transform (FFT) and Viterbi decoder (Viterbi). Toolchains are compared in compilation speed, execution speed of compiled programs and resulting assembly code size.

Table 5.1 gives running times for scheduling passes. The reason not to measure complete compilation times is that front- and middle-ends used (GCC and LLVM) are third-party software not specific to FlexCore. A radical decrease in compilation speed is attributed to the use of a SAT solver for most of the scheduling instead of a heuristic algorithm. Compilation speed may be improved by better estimates of schedule lengths that will reduce SAT problem sizes.

	Autcor	FFT	Viterbi
FlexTools	3	4	4
Bau	71	87	94

Table 5.1: Benchmark compilation times (in seconds)

Table 5.2 gives sizes of resulting assembly code. The sizes of scheduling pass inputs are given to show the difference in density of code the schedulers are working with. A more verbose intermediate representation of LLVM keeps more informa-

tion about the program that can be readily used by the Bau scheduler, which gives an improvement from 20% to 50% in final assembly size.

	Autcor	FFT	Viterbi
MIPS	38	337	324
FlexTools	42	392	307
LLVM	70	569	617
Bau	33	224	261

Table 5.2: Number of instruction words in innermost functions

Table 5.3 gives benchmark execution times. The current iteration of Bau does not give much of improvement over FlexTools. The reason for this is absence of certain microcode optimizations implemented by FlexTools, suboptimal estimation of schedule length that may introduce many empty cycles and suboptimal generated spilling code that may introduce unnecessary latencies.

	Autcor	FFT	Viterbi
FlexTools	0.9k	33k	232k
Bau	1.0k	31k	214k

Table 5.3: Benchmark execution times (in FlexCore simulator cycles)

6

Conclusion and future work

Memik and Fallah’s work on expressing scheduling problems as satisfiability problems [25] has demonstrated the viability of the approach and inspired our own work. Besides reducing efforts of developing compilers for new variants of configurable architectures by separating architecture-specific resource constraints from the generic constraints, it is possible to separate the scheduling algorithm from the scheduling engine (the SAT solver) and reduce the code base that has to be maintained by compiler developers. We have demonstrated that modularized design of the scheduler establishes clear boundaries between loosely-related properties of the schedule and allows for better utilization of hardware resources. However, the set of constraint combinators is by no means exhaustive, and scheduler writers should not limit their designs with the features available.

- *RTN optimization.* LLVM implements common subexpression elimination and loop invariant code motion passes, but after LLVM instructions are translated to RTN microinstructions, unnecessary duplicated code occurs again. [27]
- *Support for vector instructions.* LLVM assembly can express vector operations natively, so back-ends can translate them to instructions for SIMD units. FlexSoC’s compile-time reconfigurability makes it a good platform for building SIMD units of arbitrary size, provided that the compiler can do necessary scheduling, and instruction compression overhead is not too big.

- *Better estimates of basic block length.* A straightforward approach to calculating schedule length, as described in Section 3.2, results in a larger search space than necessary and potentially longer schedules, even if empty instruction words are eliminated. Weak constraints do not necessarily mean that the search time would be big, since a suboptimal solution could still be found fast [28]. But a more comprehensive approach to analyzing the scheduling outcome will also help to coin constraints that would impact scheduling speed, as well, which would become an important factor with bigger benchmarks or applications.
- *Iterative optimization of basic block length.* The problem of finding a lower bound of a schedule length is as hard as finding the schedule itself, so it is unfeasible to try to estimate it. A viable approach to optimization is finding a valid schedule of a length with a statically estimated upper bound and iteratively adding new constraints to the SAT solver instance that will forbid placement of instructions of increasing amount of cycles in the end of basic blocks, which is quicker than solving the problem from scratch with new scheduling bounds. After a number of iterations the problem will become unsatisfiable, which would mean that a lower bound is found.
- *Choice of optimization criteria.* There are many schedules for one program that have a valid behavior, but they have different outcomes in terms of consumed power and execution time. After the solver returns a schedule, it might be run through an analyzer to determine if it satisfies power and performance requirements. To avoid blind searches across many possible schedules, further constraints can be defined to explicitly focus on power or performance, which would require a more clear mechanism for constraint description.
- *Better liveness analysis.* A pipeline controlled by a compiler makes it impossible to separate scheduling into two sequential phases, instruction scheduling and register allocation. It means that existing liveness analysis techniques are inapplicable, and a very basic approach to defining liveness has to be taken to generate value placement constraints. Although, a constraint exists that defines value schedule precisely, but it is too hard computationally to be practically used. A trade-off has to be found.

- *Tighter integration between scheduler and spill code generator* When memory unit throughput is in deficit, and additional cycles are inserted to transfer values, the performance suffers. In many cases, scheduling can be performed to hide the latency of memory further, and additional constraints can be written to capture such behavior. Assignment of value groups to registers should not be random.
- *Peephole optimization of jumps.* LLVM does not rely on the lexical order of basic blocks in the assembly code and demands explicit listing of basic block successors. A straightforward lowering generates excessive jump code that can be removed because FlexCore assembly does not require explicit jumps to lexically following basic blocks.
- *Direct forwarding of values across basic blocks.* The simplest implementation of the scheduler does not try to prove that a value can be directly forwarded (and used) to all successors of basic blocks and always assigns a real location to a value that is used globally. This might lead to a higher memory pressure than necessary.

7

Related work

The presented work relies both on previous approaches to generalize the scheduling problem and off-the-shelf tools for implementing a prototype scheduler. This section lists sources that provide better insight into the domain.

Instruction reordering and register allocation are two interdependent scheduling phases with opposite goals. During instruction reordering parallelism is exploited at the cost of increasing register pressure and spilling. Optimization criteria of a register allocator are exactly opposite — decreasing the number of spills at the cost of parallelism. Many approaches to choosing an optimal scheduling phase sequence were summarized by Norris and Pollock [29]. They have also proposed several strategies for phase communication and making the instruction scheduler and the register allocator mutually sensitive. Recent works on the topic [30, 31] have not departed from the scheme of separate but iteratively communicating phases but barely proposed alternative communication strategies. A notable exception is [32] where serialization of a CFG is performed incrementally. However, neither approach is directly applicable to an exposed architecture where the compiler has control over instances of execution units and the forwarding paths that a value can take.

A major reason not to unite instruction reordering and register allocation is NP-hardness of a combined problem [33]. Building phase communication strategies upon heuristics improves performance but makes it hard to guarantee a specific

scheduling outcome in a larger number of cases than it was thought of during design of heuristics. While SAT solvers might still employ heuristics to find solution quickly [24, 34], we gain in clarity of the problem definition directly by imposing constraints on the schedule that do not change the optimization criteria and indirectly with constraints that increase scheduling speed [28]. Improving performance of SAT solvers is a topic of ongoing research, with proposed support for parallelization of solvers to run on multicore and multinode computers [35, 36], GPUs [37] and FPGAs [38]. As the scheduling problem definition is made separate from the scheduling engine, it will benefit from future faster solvers without any changes.

Bibliography

- [1] Richard D. Greenblatt, Thomas F. Knight, John T. Holloway, and David A. Moon. A LISP machine. In *Proc. of 5th Workshop on Computer Architecture for Non-Numeric Processing*, pages 137–138, 1980.
- [2] M. Naylor and C. Runciman. The reduceron reconfigured. In *Proc. of the 15th ACM SIGPLAN international conference on Functional programming*, pages 75–86, 2010.
- [3] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.
- [4] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proc. of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, 1982.
- [5] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 58–69, 1984.
- [6] L. Augustsson and T. Johnsson. Parallel graph reduction with the (v , g)-machine. In *Proc. of the 4th international conference on Functional programming languages and computer architecture*, pages 202–213, 1989.
- [7] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992.

-
- [8] N. Nethercote and A. Mycroft. The cache behaviour of large lazy functional programs on stock hardware. *SIGPLAN Not.*, 38:44–55, June 2002.
- [9] Langendoen K. and Agterkamp D.-J. Cache behaviour of lazy functional programs. In *Proc. 4th Int. Workshop on Parallel Implementations of Functional Languages*, 1992.
- [10] Ahmad A. and DeYoung H. Cache performance of lazy functional programs on current hardware. Technical Report 15-740, Carnegie Mellon University, November 2009.
- [11] Moritz A., Frank M. I., and Amarasinghe S. FlexCache: A framework for flexible compiler generated data caching. In *Intelligent Memory Systems*, volume 2107 of *Lecture Notes in Computer Science*, pages 135–146. Springer Berlin / Heidelberg, 2001.
- [12] G. L. Steele, Jr. and G. J. Sussman. Design of a lisp-based microprocessor. *Commun. ACM*, 23:628–645, November 1980.
- [13] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March 2000.
- [14] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala. Codesign toolset for application-specific instruction-set processors. In *Proc. SPIE*, volume 6507, 2007.
- [15] E. Moscu Panainte, K.L.M. Bertels, and S. Vassiliadis. The molen compiler backend for reconfigurable architectures. In *Compiler and Architecture Seminar 2005*, December 2005.
- [16] J. Hughes, K. Jeppson, P. Larsson-Edefors, M. Sheeran, P. Stenstrom, and L. Svensson. FlexSoC: Combining flexibility and efficiency in SoC designs. In *Proc. IEEE NorChip Conference*, pages 52–55, November 2003.
- [17] J. Fisher, J. Ellis, J.C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proc. ACM SIGPLAN Symposium on Compiler Construction*, pages 37–47, June 1984.

- [18] T.T. Hoang, U. Jälmlund, E. der Hagopian, K.P. Subramaniyan, M. Sjölander, and P. Larsson-Edefors. Design space exploration for an embedded processor with flexible datapath interconnect. In *Proc. IEEE International Conference Application-Specific Systems, Architectures, and Processors*, pages 55–62, July 2010.
- [19] M. Thuresson, M. Sjölander, and P. Stenstrom. A flexible code compression scheme using partitioned look-up tables. In *Proc. High Performance Embedded Architectures and Compilers*, pages 95–109, January 2009.
- [20] GNU Compiler Collection. <http://gcc.gnu.org>.
- [21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, pages 75–86, March 2004.
- [22] The LLVM Target-Independent Code Generator. <http://llvm.org/docs/CodeGenerator.html>.
- [23] Writing an LLVM Pass. <http://llvm.org/docs/WritingAnLLVMPass.html>.
- [24] N. Een and N. Sörensson. An extensible SAT-solver. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, pages 333–336. Springer, May 2003.
- [25] S.O. Memik and F. Fallah. Accelerated SAT-based scheduling of control/data flow graphs. In *Proc. IEEE International Conference on Computer Design*, pages 395–400, September 2002.
- [26] T. Schilling, M. Sjölander, and P. Larsson-Edefors. Scheduling for an embedded architecture with a flexible datapath. In *Proc. IEEE Symposium on VLSI*, pages 151–156, May 2009.
- [27] Ian Finlayson, Gang-Ryung Uh, David Whalley, and Gary Tyson. Improving low power processor efficiency with static pipelining. *Interaction between Compilers and Computer Architecture, Annual Workshop on*, 0:17–24, 2011.

- [28] J.M. Crawford and A.B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. Conference on Artificial Intelligence (AAAI)*, pages 1092–1097, July 1994.
- [29] C. Norris and L.L. Pollock. Experiences with cooperating register allocation and instruction scheduling. *International Journal of Parallel Programming*, 26(3):241–284, 1998.
- [30] I. Cutcutache and W.-F. Wong. Fast, frequency-based, integrated register allocation and instruction scheduling. *Software: Practice and Experience*, 38(11):1105–1126, September 2008.
- [31] D.R. Koes. Register allocation aware instruction selection. Technical Report CMU-CS-09-169, Carnegie Mellon University School of Computer Science, October 2009.
- [32] N. Johnson and A. Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Proc. of the 12th International Conference on Compiler Construction*, pages 1–16, April 2003.
- [33] R. Motwani, K.V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report TR 698, Courant Institute, July 1995.
- [34] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th ACM/IEEE Design Automation Conference*, pages 530–535, June 2001.
- [35] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *Proc. of the 12th Asia and South Pacific Design Automation Conference*, pages 926–931, January 2007.
- [36] Y. Hamadi and L. Sais. ManySAT: a parallel SAT solver. *Satisfiability, Boolean Modeling and Computation*, 6(12):245–262, June 2009.
- [37] C.J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proc.*

IEEE/ACM 35th International Symposium on Microarchitecture, pages 306–317, November 2002.

- [38] A. Dandalis and V.K. Prasanna. Run-time performance optimization of an FPGA-based deduction engine for SAT solvers. *ACM Transactions on Automation of Electronic Systems*, 7(4):547–562, October 2002.

A

Bau source code

The directory structure is shown on Fig. 2.1.

Listing A.1: Bau/RTN.hs

```
{-# LANGUAGE TypeFamilies, FlexibleContexts, ExistentialQuantification,
      ImpredicativeTypes, StandaloneDeriving, UndecidableInstances #-}

module Bau.RTN where

import Data.Array
import qualified Data.Map as M
import qualified LLVM.Core as L
import MiniSat

-- RTN is an assembly language on its own, and it's structure is not different
-- from other assemblies.

type Name =String -- for values and functions
type Label =String -- for BBs

data Target arch =>Assembly arch =A Name [Function arch] [L.GlobalDesc]
data Function arch =F Name [Name] [BasicBlock arch] LEdges
data BasicBlock arch =BBU Label [MicroOp arch] |BBS Label [InstrWord arch]

-- Every microinstruction defines one SSA value as output of an execution unit
-- of a particular type with given SSA values as inputs. There might be several
-- identical units attached to the interconnect. A unit may or may be not used
-- in a schedule. If the unit is referred before scheduling is performed, the
-- unit id field is 0 (blank unit). An optional list of *control* dependencies
-- can be provided.
```

APPENDIX A. BAU SOURCE CODE

```
data InstrWord arch =IW [MicroOp arch]
data MicroOp arch =MO Name (UnitType arch) (Mode arch) [(DataPort arch)] Int [Name]
    -- instance

-- Operands are defined implicitly as a given execution unit can only read its
-- input registers, and an execution unit has to route the result into one of
-- pipeline registers through interconnect. Every pipeline register is
-- identified by its input and output dataports.

-- In order to keep assembly syntax similar to a more conventional one,
-- switching is expressed by listing output ports as arguments instead of
-- listing input ports as results. Ports are defined by variant-specific unit
-- type instances.

data DataPort arch =DP PortType (UnitType arch) Int Name |DPU L.ArgDesc
data PortType =In |Out deriving (Show, Eq)

deriving instance (Eq (UnitType arch), Eq (Mode arch)) =>Eq (MicroOp arch)
deriving instance (Eq (UnitType arch), Eq (Mode arch)) =>Eq (InstrWord arch)
deriving instance (Eq (UnitType arch)) =>Eq (DataPort arch)

-- Defines an interface for accessing architecture-specific translation rules
-- and constraints. Translator implementation chooses a set of rules by
-- specifying full type signature.

-- define value @ cycle or store value @ location
type Index =Int
type Entry =(Index, Index)

-- (instructions, values)
type Relation a b =Array (a,b) Lit
type Schedule =(Relation Index Index, Relation Index Index)

-- analysis-related
type Lives =M.Map Label [Name]
type LEdge =(Index, Index)
type LEEdges =M.Map Label [LEdge]

type Resources arch =M.Map (UnitType arch) (Int, Int) -- unit type, amount, execution
    latency (in cycles)
type Problem arch =(Solver, Resources arch, [Schedule], [BasicBlock arch], LEEdges)
type BBProblem arch =(Solver, Resources arch, Schedule, [MicroOp arch], [LEdge])
type Constraint arch =Problem arch ->IO ()
type BBConstraint arch =BBProblem arch ->IO ()

class (Eq (Mode arch), Eq (UnitType arch), Ord (UnitType arch), Enum (UnitType arch),
    Bounded (UnitType arch)) =>Target arch where
    data UnitType arch
    data Mode arch
    isMemory :: UnitType arch ->Bool
```

```

constraints :: [Constraint arch]
lower :: (Name, L.InstrDesc) → [MicroOp arch]

-- Hjelpfunksjoner

bblabel (BBU label _) =label
bbops (BBU _ ops) =ops
opname (MO name _ _ _ _) =name
opunittype (MO _ ut _ _ _) =ut
opargs (MO _ _ _ args _ _) =args

isDPValue (DPU (L.AV _)) =True
isDPValue _ =False

dpValue (DPU (L.AV x)) =x
dpValue _ =""

maxUnitIndex :: Resources arch →Int
maxUnitIndex =foldr ((+)ofst) 0oM.elems

-- 1 "special" location, 4 buffers, 32 registers sans 7 special registers
-- FIXME: should be defined in the variant configuration
maxLocationIndex res =1+4+32-7

```

Listing A.2: Bau/Translate.hs

```

{-# LANGUAGE FlexibleContexts #-}

module Bau.Translate (translateModule) where

import System.FilePath
import Data.List
import qualified Data.Map as M
import Data.Maybe
import Data.Tuple
import Control.Monad
import qualified LLVM.Core as L
import qualified LLVM.FFI.Core as FFI
import System.IO (stderr, hPutStrLn)

import Bau.RTN
import Bau.Schedule

translateModule :: (Target arch) ⇒Resources arch →String →IO (Assembly arch)
translateModule res file =do
  m ←L.readBitcodeFromFile file
  fs ←L.getFunctions m
  renames ←mapM findRenamesInFunction fs
  fRtn ←mapM translateFunction $ zip renames fs
  sched ←mapM (scheduleFunction res 0) fRtn

```



```

globals ←L.getGlobalVariables m
globals2 ←mapM translateGlobalValue globals
return $ A (takeFileName file) sched globals2

translateFunction (renames, (fname, f)) =do
  bbs ←L.getBasicBlocks f
  params ←L.getParams f
  (bbsRtn, dfg) ←mapAndUnzipM (translateBasicBlock renames) bbs
  return $ F fname (map fst params) bbsRtn (M.fromList dfg)

translateBasicBlock renames (bblabel, bb) =do
  is ←L.getInstructions bb
  isRtn ←concatMapM (translateInstruction renames) is
  l_deps ←returnConcatMap (localuses isRtn) $ isRtn
  return ((BBU bblabel isRtn), (bblabel, l_deps))

translateInstruction renames (v, i) =do
  i2 ←L.getInstrDesc i
  let v' =case M.lookup v renames of
        Just v'' →v''
        Nothing →v
  return $ lower (v', i2)

translateGlobalValue (gname, v) =do
  init ←FFI.getInitializer v
  L.getGlobalDesc gname init

findRenamesInFunction :: (String, FFI.ValueRef) →IO (M.Map Name Name)
findRenamesInFunction (_, f) =do
  bbs ←L.getBasicBlocks f
  bbsRenames ←mapM findRenamesInBasicBlock bbs
  return $ M.fromList $ concat bbsRenames

findRenamesInBasicBlock :: (String, FFI.ValueRef) →IO [(Name, Name)]
findRenamesInBasicBlock (_, bb) =do
  is ←L.getInstructions bb
  is2 ←filterM (λ x →liftM L.isValConvOp $ L.getInstrDescsnd $ x) is
  renames ←mapM (λ x →do
    old ←liftM L.getValConvArg $ L.getInstrDescsnd $ x
    new ←L.getValueNameUosnd $ x
    return (old, new)) is2
  return renames

-- Hjelpfunksjoner

localuses :: (Eq (UnitType arch), Eq (Mode arch)) ⇒[MicroOp arch] →MicroOp arch →
  [LEdge]
localuses ops op =(map mkEdge)ofilter (isDefUse op) $ ops
  where mkEdge use =fromMaybe (error "Translate:_localuses:_index_not_found")
    (do defIndex ←elemIndex op ops

```

```

        useIndex ←elemIndex use ops
        return (defIndex, useIndex)
    isDefUse (MO def _ _ _ _ _) (MO _ _ _ _ args _ _) =
        (DPU $ L.AV def) `elem` args

concatMapM f xs =liftM concat (mapM f xs)

```

Listing A.3: Bau/Schedule.hs

```

{-# LANGUAGE ScopedTypeVariables, FlexibleContexts #-}

module Bau.Schedule(scheduleFunction) where

import Data.Array
import qualified Data.Map as M
import Data.Maybe
import Control.Monad ( forM, forM_, filterM, zipWithM, liftM )
import MiniSat
import System.IO (stderr, hPutStrLn)

import Bau.RTN
import Bau.Primitives
import Bau.Constraints

--type Solution =A.Array Entry Bool -- +present in the schedule or not

-- a function is the optimal unit of program text to be scheduled because
-- schedules for basic blocks have to consider inter-BB data transfers at very
-- least, and module-wide and cross-module code moving was already done by LLVM
-- (inlining/linking)
scheduleFunction :: (Target arch) =>Resources arch →Int →Function arch →IO (Function
    arch)
scheduleFunction res spills (F name params bbs l_deps) =do
    hPutStrLn stderr $ name ++"()..."
    -- construct and solve the problem
    s ←newSolver
    ischs ←mapM (emptyISched s res) bbs
    vschs ←mapM (emptyVSched s res) bbs
    hPutStrLn stderr $ "no_more_than_" ++show spills ++"_spill_slots..."
    num_vars ←minisat_num_vars s
    hPutStrLn stderr $ "added_variables:_" ++show num_vars
    forM_ constraints ($ (s, res, zip ischs vschs, bbs, l_deps))
    num_clauses ←minisat_num_clauses s
    hPutStrLn stderr $ "added_clauses:_" ++show num_clauses
    b ←solve s []
    -- interpret the solution
    case b of
        True →do
            bbsScheduled ←zipWithM (buildBBS s res vschs) bbs ischs
            deleteSolver s

```

```

    return $ F name params bbsScheduled l_deps
  _ → if spills == 20
    then fail $ "F_" ++ name ++ ":_scheduling_failed"
    else scheduleFunction res (spills+2) (F name params bbs l_deps)

-- Fill in the blanks in the schedule

buildBBS :: (Target arch) ⇒ Solver → Resources arch → [Relation Int Int] → BasicBlock
arch → Relation Int Int → IO (BasicBlock arch)
buildBBS s res vschs (BBU label ops) isch = do
  let entry2Op (o, _) es = op2Unit (ops !! o) (total (opunittype $ ops !! o) - left
    (opunittype $ ops !! o) es) -- :: MicroOp arch
      total = fst ◦ (res M.!)
      left t = length ◦ filter ((== t) ◦ opunittype ◦ (ops !!)) ◦ fst
      op2Unit (MO v ut m dp _ _) id = MO v ut m dp id []
      cycle2Word (e:es) = (entry2Op e es) : (cycle2Word es)
      cycle2Word [] = []
  sol2Cycles ← mapM (ops2Cycle s isch) [0..length ops-1] -- :: IO [[Entry]]
  return $ (BBS label) ◦ (filter (/= (IW []))) ◦ (map (IW ◦ cycle2Word)) $ sol2Cycles

ops2Cycle :: Solver → Relation Int Int → Int → IO [Entry]
ops2Cycle s sch n = do
  let cycleVars = filter (λ ((_, c), _) → c == n) (assocs sch)
      trueVars ← filterM (isVarTrue s) cycleVars
  return $ map fst trueVars

isVarTrue :: Solver → (Entry, Lit) → IO Bool
isVarTrue s (_, t) = do
  tv ← modelValue s t
  case tv of
    Just True → return True
    _ → return False

-- Determine scheduling bounds

maxCycleIndex :: (Ord (UnitType arch)) ⇒ Resources arch → [MicroOp arch] → Int
maxCycleIndex res = foldr ((+) ◦ latency ◦ opunittype) 0
  where latency = snd ◦ (res M.!)

-- Instruction placement bounds

--emptyISched :: Solver → Resources arch → BasicBlock arch → Relation Task Slot
emptyISched s res (BBU label ops) =
  relation s ((0, 0), (length ops - 1, maxCycleIndex res ops - 1))

--emptyVSched :: Solver → Resources arch → BasicBlock arch → Relation Task Slot
emptyVSched s res (BBU label ops) =
  relation s ((0, 0), (length ops - 1, maxLocationIndex res - 1))

-- global unit index → unit instance number

```

```

unitId :: (Bounded (UnitType arch), Enum (UnitType arch), Ord (UnitType arch)) =>
  Resources arch ->Int ->Int
unitId res n =seek (minBound :: Bounded (UnitType arch) =>UnitType arch) n 0
  where seek ut n i |i +utCard res ut <n =seek (succ ut) n (i +utCard res ut)
              |i +utCard res ut ≥n =i +utCard res ut - n +1
  utCard res =fsto(res M.!)

```

Listing A.4: Bau/Constraints.hs

```

{-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}

module Bau.Constraints(
  defaultInstrConstrs, defaultValueConstrs
) where

import Control.Monad ( forM_ )
import Data.Array
import qualified Data.Map as M
import Data.List
import MiniSat

import Bau.RTN
import Bau.Primitives

-- Scheduling constraints that assign instructions to units

defaultInstrConstrs :: (Target arch) =>[Constraint arch]
defaultInstrConstrs =[nOpsAtATime, placeAtMostOnce, placeAtLeastOnce, defBeforeUse]

-- Execution unit properties

-- no more than N intructions of given type at a time
-- NAND N+1 vars in a cycle - no need to check larger subsets
-- will replace both oneTask and resourceType for instructions
-- placeAtMostOnce, placeAtLeastOnce, defBeforeUse stay
-- baseline FlexCore has ~11 unit instances, so there will be 11 times less
  instruction vars

nOpsAtATime =
  bbConstr $ λ(s, res, (isch, _), ops, _) ->
    -- forAll unit types
    forM_ (M.keys res) $ λut ->do
      -- forAll ops in ut
      let opsannotated =zip [0..length ops-1] $ ops
          utops =map fstofilter ((== ut)oopunittypeosnd) $ opsannotated
          overbook =1 +(fsto(res M.!) $ ut)
          --forAll 'ut' op combinations of length 'available instances +1'
          forM_ (subseqN overbook utops) $ λnn ->
            -- forAll cycles
            forM_ (allSlots isch) $ λc ->

```

APPENDIX A. BAU SOURCE CODE

```
neverAll s isch $ zip nn $ replicate (length nn) c

-- Dataflow properties

-- every value is computed only once ⇒ every instruction is placed no more than
-- once, and only on instances of corresponding unit types
placeAtMostOnce =
  bbConstr $ λ(s, res, (isch, _), ops, _) →
    nest2
      (allTasks isch)
      (taskSlotPairs (allSlots isch))
      (neverBoth s isch)

-- every value is computed only once ⇒ every instruction is placed no less than
-- once
placeAtLeastOnce =
  bbConstr $ λ(s, res, (isch, _), ops, _) →
    nest2
      (allTasks isch)
      (taskSlots (allSlots isch))
      (atLeastOne s isch)

-- no value is computed before its dependencies are available (one clause for
-- every def-misplaced use pair)
defBeforeUse =
  bbConstr $ λ(s, res, (isch, _), ops, l_deps) →
    nest2
      l_deps
      (brokenDepSlots (allSlots isch))
      (neverBoth s isch)

defaultValueConstrs :: (Target arch) ⇒ [Constraint arch]
defaultValueConstrs = [storeAtMostOnce, storeAtLeastOnce, dontStoreForwards,
  respectLiveness]

-- Value properties

storeAtMostOnce =
  bbConstr $ λ(s, res, (_, vsch), ops, l_edges) →
    nest2
      (map fst l_edges)
      (taskSlotPairs (allSlots vsch))
      (neverBoth s vsch)

storeAtLeastOnce =
  bbConstr $ λ(s, res, (_, vsch), ops, l_edges) →
    nest2
      (map fst l_edges)
      (taskSlots (allSlots vsch))
      (atLeastOne s vsch)
```

```

-- Location properties

-- location 0 means "off-unit", i.e., direct forwarding or stack
dontStoreForwards =
  bbConstr $ \(s, res, (isch, vsch), ops, l_edges) →
    forM_ l_edges $ \(def, use) →
      forM_ (consDepSlots (allSlots isch) (def, use)) $ \((isd, isu)) →
        assert s [ neg $ isch ! isd, neg $ isch ! isu
                  , neg $ vsch ! (def, 0) ]

-- if values can be live at the same time, they don't share a location
-- computationally simpler, but bans valid schedules
-- !(def1, l) v !(def2, l) v !(def2, c1) v !(use1, c2), c1 < c2
respectLiveness =
  bbConstr $ \(s, res, (isch, vsch), ops, l_deps) →
    forM_ (filter (noto(edgesDependent ops)) $ defUsePairs l_deps) $
      \((def1, use1), (def2, use2)) →
        forM_ (tail $ allSlots vsch) $ \l →
          assert s [ neg $ vsch ! (def1, l), neg $ vsch ! (def2, l) ]

-- more precise, but the SAT problem becomes bigger
-- a location is not overwritten before stored value is used
-- !(def1, l) v !(def2, l) v !(def2, c1) v !(use1, c2), c1 < c2
-- readBeforeOverwrite =
-- bbConstr $ \(s, res, (isch, vsch), ops, l_deps) →
-- forM_ (filter (noto(edgesDependent ops)) $ defUsePairs l_deps) $
-- \((def1, use1), (def2, use2)) →
-- forM_ (tail $ allSlots vsch) $ \l →
-- forM_ (depSlots (allSlots isch) (use1, def2)) $ \(isu1, isd2) →
-- assert s [ neg $ vsch ! (def1, l), neg $ vsch ! (def2, l)
-- , neg $ isch ! isu1, neg $ isch ! isd2 ]

```

Listing A.5: Bau/Primitives.hs

```

{-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}

module Bau.Primitives where

import Control.Monad ( forM, forM_, void )
import Data.Array
import qualified Data.Map as M
import Data.List
import Data.Maybe
import MiniSat

import Bau.RTN

-- Schedules are relations between tasks and slots

```

```

relation s bnd =do
  let indices =range bnd
  pairs ←forM indices $ λindex →do
    lit ←newLit s
    return (index, lit)
  return $ array bnd pairs

-- Quantification scopes

-- define a constraint local to a basic block
bbConstr :: BBConstraint arch →Constraint arch
bbConstr constr (s, res, schs, bbs, bbedges) =do
  forM_ (zip bbs schs) $ λ((BBU label ops), sch) →do
    constr (s, res, sch, ops, bbedges M.! label)

type Range =[Index]

-- Universal slot quantifiers

allSlots sch =let ((t0, s0), (tN, sN)) =bounds sch
  in [s0..sN]

taskSlots :: Range →Index →[Entry]
taskSlots slots task =[ (task, s) |s ←slots ]

taskSlotPairs :: Range →Index →[(Entry, Entry)]
taskSlotPairs slots task =
  [ ((task, s1), (task, s2)) |s1 ←slots, s2 ←slots, s1 <s2 ]

depSlots :: Range →LEdge →[(Entry, Entry)]
depSlots cycles (def, use) =
  [ ((def, c1), (use, c2)) |c1 ←cycles, c2 ←cycles, c1 <c2 ]

brokenDepSlots :: Range →LEdge →[(Entry, Entry)]
brokenDepSlots cycles (def, use) =
  [ ((def, c1), (use, c2)) |c1 ←cycles, c2 ←cycles, c1 ≥c2 ]

ruinedDepSlots :: Range →(Index, Index, Index) →[(Entry, Entry, Entry)]
ruinedDepSlots icycles (def1, def2, use1) =
  [ ((def1, c1), (def2, c2), (use1, c3))
    |c1 ←icycles, c2 ←icycles, c3 ←icycles, c1 <c2, c2 <c3 ]

consDepSlots :: Range →LEdge →[(Entry, Entry)]
consDepSlots icycles (def, use) =
  [ ((def, c1), (use, c2))
    |c1 ←icycles, c2 ←icycles, c1 <c2, c1 +1 ==c2 ]

nonConsDepSlots :: Range →LEdge →[(Entry, Entry)]
nonConsDepSlots icycles (def, use) =

```

```

[ ((def, c1), (use, c2))
  |c1 ←icycles, c2 ←icycles, c1 <c2, c1 +1 /=c2 ]

-- Universal task quantifiers

allTasks sch =let ((t0, s0), (tN, sN)) =bounds sch
                 in [t0..tN]

slotTaskPairs :: Range →Index →[(Entry, Entry)]
slotTaskPairs ops slot =[(i1, slot), (i2, slot)]
                        |i1 ←ops, i2 ←ops, i2 >i1 ]

defUsePairs :: [LEdge] →[(LEdge, LEdge)]
defUsePairs l_edges =[(le1, le2) |le1 ←l_edges, le2 ←l_edges, le1 <le2 ]

-- Existential entry quantifiers

-- 'and' and similar from satchmo should be avoided because they create one
-- additional variable for *every* clause

-- 1) a clause of all-positive literals is an OR
-- 2) a clause of all-negative literals is a NAND

assert s =voido(addClause s)

-- at least one entry (OR)
atLeastOne :: Solver →Relation Index Index →[Entry] →IO ()
atLeastOne s sch =voido(addClause s)omap (sch !)

-- none entries (NOR)
noOne :: Solver →Relation Index Index →[Entry] →IO ()
noOne s sch =mapM_ $ λ(i, j) →void $ addClause s [ neg $ sch ! (i, j) ]

-- not all entries (NAND)
neverAll :: Solver →Relation Index Index →[Entry] →IO ()
neverAll s sch =voido(addClause s)omap (nego(sch !))

-- no more than one of two conflicting entry (2-NAND-n-OR)
neverBoth :: Solver →Relation Index Index →[(Entry, Entry)] →IO ()
neverBoth s sch =mapM_ $ λ((i1, j1), (i2, j2)) →
  void $ addClause s [ neg $ sch ! (i1, j1), neg $ sch ! (i2, j2) ]

-- Quantifier combinators

nest2 :: [a] →(a →b) →(b →IO ()) →IO ()
nest2 un ex constr =
  forM_ un $ λo →do
    constr (ex o)

-- Hjelpfunksjoner

```



```

subseqN :: Int -> [a] -> [[a]]
subseqN 1 xs =map (\ x ->[x]) xs
subseqN n (x:xs) =(map (x :) (subseqN (n-1) xs)) ++subseqN n xs
subseqN _ _ =[]

edgesDependent :: [MicroOp arch] -> (LEdge, LEdge) ->Bool
edgesDependent ops ((def1, use1), (def2, use2)) =
  if def2 ==use1
  then True
  else let args_def2 =map dpValueo(filter isDPValue)oopargs $ ops !! def2
        args_def2' =map fromJusto(filter (/= Nothing)oormap (\ x ->findIndex ((==
          x)oopname) ops) $ args_def2
        in if null args_def2'
          then False
          else oromap (edgesDependent opso\ x ->((def1, use1), (x, def2))) $ args_def2'

```

Listing A.6: Bau/Options.hs

```

module Bau.Options where

import System.Console.GetOpt
import System.Exit
import System.Environment
import System.IO
import System.IO.Unsafe
import Data.Char
import Data.Maybe (isJust, fromJust, isNothing)
import System.Directory (doesFileExist)

import Control.Arrow ( (&&&) )

import Bau.RTN

unsafeArgs :: (Options a, [String])
unsafeArgs =unsafePerformIO $ do
  args <-getArgs

  -- Parse options, getting a list of option actions
  let (actions, files, _) =getOpt Permute options args

  startOptions' <-return startOptions

  return (startOptions', files)

files :: [String]
files =snd unsafeArgs

data Options a =Options

```

```

{ optDebug :: Bool
, optInterconnectFile :: Maybe FilePath
}

startOptions :: Options a
startOptions =
  Options
  { optDebug =False
  , optInterconnectFile =Nothing
  }

options :: [ OptDescr (Options a →(IO (Options a))) ]
options =
  [
    Option "i" ["interconnect"]
      (ReqArg (λarg opt →return opt { optInterconnectFile =Just arg })
        "FILE")
      "Load_interconnect_information_from_file"

    , Option "" ["debug"]
      (NoArg
        (λopt →return opt { optDebug =True }))
      "Disable_compilation_to_rtn,_used_to_produce_output_when_normal_compilation_
        fails"

    , Option "v" ["version"]
      (NoArg
        (λ_ →do
          hPutStrLn stderr "Version_0.01"
          exitWith ExitSuccess))
      "Print_version"

    , Option "h" ["help"]
      (NoArg
        (λ_ →do
          prg ←getProgName
          hPutStrLn stderr (usageInfo prg options)
          exitWith ExitSuccess))
      "Show_help"

  ]

```

Listing A.7: Bau/RTN.hs

```

module FlexCore.Arch where

-- we are using this type to create instances of the Target class
data FlexCore =FlexCore deriving Show

```

Listing A.8: Bau/RTN.hs

```

{-# LANGUAGE TypeFamilies #-}
module FlexCore.Target where

import LLVM.Core

import Bau.RTN
import Bau.Constraints
import FlexCore.Arch

-- The naming convention for multi-microop instruction is:
-- 1) every microop should have a non-empty name
-- 2) result names shouldNa be equal to original SSA names

instance Target FlexCore where
  data UnitType FlexCore =
    -- PC unit
    PC
    -- ALU
    |ALUOp
    -- AGU
    |AGUOp
    -- Multiplier
    |Mult |MultRegWrite
    -- Buffers
    |Buf
    -- Register bank
    |RegRead |RegWrite
    -- Load/store
    |LSRead |LSWrite -- addr value
    deriving (Read, Enum, Bounded, Ord, Eq)

  data Mode FlexCore =
    -- Immediate values (PC)
    Imm
    -- Control flow (PC)
    |GetPC |JumpSA |JumpSR |JumpDA
    |BEQZR -- Equal to zero (relative jump)
    |BNEZR -- Not equal to zero (relative jump)
    |BEQZA -- Equal to zero (absolute jump)
    |BNEZA -- Not equal to zero (absolute jump)
    -- ALU
    |ADD -- Add signed
    |ADDU -- Add unsigned
    |SUB -- Sub signed
    |SUBU -- Sub unsigned
    |AND -- Bitwise AND
    |OR -- Bitwise OR
    |NOR -- Bitwise NOR

```

APPENDIX A. BAU SOURCE CODE

```
|XOR -- Bitwise XOR
|SLL -- Shift left logical
|SRL -- Shift right logical
|SHR -- Shift right arithmetical signed
|SLT -- Set on less than
|SLE -- Set on less than or equal
|SEQ -- Set on equal
|SNE -- Set on not equal
|TEST -- Return operand 1
|DONTCARE -- Dont care. Useful for trinary output
-- Load/store
|LSW Int |LSWU Int |LSW_DONTCARE
-- Since amounts of register file read/write ports can differ, we choose
-- to make register number a "mode" instead of an "instance", so there
-- is no need in distinguishing between "computation" and "memory"
-- instructions.
|Reg Int
-- Quasi-mode bits
|Stall |None
deriving (Read, Eq)

isMemory Buf =True
isMemory RegRead =True
isMemory RegWrite =True
isMemory LSRead =True
isMemory LSWrite =True

constraints =defaultInstrConstrs ++defaultValueConstrs

-- control flow
-- don't have to implement 'switch' because LLVM can lower it to branches with opt
-- lowerswitch

lower (v, IDCall t f args) =[ MO (v++"_0") RegRead sp [] 0 []
, MO (v++"_1") PC Imm [DPU $ AI 32] 0 []
-- push $fp
, MO (v++"_2") ALUOp SUB [d v 0, d v 1] 0 []
, MO (v++"_4") RegRead fp [] 0 []
, MO (v++"_6") LSWrite (LSW 4) [d v 2, d v 4] 0 []
-- push $ra
, MO (v++"_3") ALUOp SUB [d v 0, d v 2] 0 []
, MO (v++"_5") RegRead ra [] 0 []
, MO (v++"_7") LSWrite (LSW 4) [d v 3, d v 5] 0 []
-- save $sp in $fp
, MO (v++"_8") RegWrite fp [d v 0] 0 []
-- update $ra
, MO (v++"_9") PC GetPC [] 0 []
, MO (v++"_10") PC Imm [DPU $ AI 96] 0 []
, MO (v++"_11") ALUOp ADD [d v 9, d v 1] 0 []
, MO (v++"_12") RegWrite fp [d v 0] 0 []
```

APPENDIX A. BAU SOURCE CODE

```

-- jump to entry point
, MO (v++"_13") PC JumpSA [DPU f] 0 [v++"_12"]
, MO (v) RegRead v1 [] 0 [v++"_13"]

lower (v, IDRet t r) =[ MO (v++"_0") RegWrite v1 [DPU r] 0 [] ]
++lower (v, IDRetVoid)
lower (v, IDRetVoid) =[ MO (v++"_10") RegRead ra [] 0 []
, MO (v++"_11") PC JumpSA [d v 10] 0 [] ]

lower (v, IDBrCond c l1 l2) =[ MO (v++"_t") PC BNEZA [DPU l1, DPU c] 0 []
, MO (v++"_f") PC BEQZA [DPU l1, DPU c] 0 [v++"_t"] ]
lower (v, IDBrUncond l) =[ MO v PC JumpSA [DPU l] 0 [] ]

-- arithmetic
-- FIXME: implement signed
lower (v, IDBinOp BOMul (TDInt False _) a b) =[ MO v Mult None [DPU a, DPU b] 0 []
]
lower (v, IDBinOp m (TDInt False _) a b) =[ MO v ALUOp (mode m) [DPU a, DPU b] 0
[] ]
where mode m =case m of {BOAdd →ADDU ; BOSub →SUBU ; BOAnd →AND ; BOOr →OR ;
BOXor →XOR ; BOShL →SLL ; BOLShR →SRL ; BOAshR →SHR }
lower (v, IDICmp IntEQ a b) =[ MO v ALUOp SEQ [DPU a, DPU b] 0 [] ]
lower (v, IDICmp IntNE a b) =[ MO v ALUOp SNE [DPU a, DPU b] 0 [] ]
lower (v, IDICmp IntSLT a b) =[ MO v ALUOp SLT [DPU a, DPU b] 0 [] ]
lower (v, IDICmp IntULT a b) =[ MO v ALUOp SLT [DPU a, DPU b] 0 [] ]
lower (v, IDICmp IntSGT a b) =
[ MO (v++"_0") PC Imm [DPU $ AI 4294967295] 0 []
, MO (v++"_1") ALUOp SLE [DPU a, DPU b] 0 []
, MO v ALUOp XOR [d v 0, d v 1] 0 [] ]
lower (v, IDICmp IntUGT a b) =
[ MO (v++"_0") PC Imm [DPU $ AI 4294967295] 0 []
, MO (v++"_1") ALUOp SLE [DPU a, DPU b] 0 []
, MO v ALUOp XOR [d v 0, d v 1] 0 [] ]

-- memory
lower (v, IDAlloca t tsize n) =
[ MO (v) RegRead sp [] 0 []
, MO (v++"_0") PC Imm [(DPU $ AI $ tsize*n)] 0 []
, MO (v++"_1") ALUOp SUBU [(DPU $ AV v), d v 0] 0 []
, MO (v++"_2") RegWrite sp [d v 1] 0 [] ]
-- FIXME: implement different sizes
lower (v, IDLoad t a) =[ MO v LSRead (LSW 4) [DPU a] 0 [] ]
lower (v, IDStore t a b) =[ MO v LSWrite (LSW 4) [DPU b, DPU a] 0 [] ]

lower (v, i) =[ ]

-- special registers
zero =Reg 0
v0 =Reg 26
v1 =Reg 27

```

```

gp =Reg 28 -- global pointer
sp =Reg 29 -- stack pointer
fp =Reg 30 -- frame pointer
ra =Reg 31 -- return address

-- syntactic sugar for naming uops
d :: String ->Int ->DataPort arch
d v n =DPU $ AV (v++"_"++show n)

```

Listing A.9: Bau/RTN.hs

```

{-# LANGUAGE StandaloneDeriving, FlexibleInstances, TypeFamilies #-}

module FlexCore.AsmWriter where

import LLVM.Core (GlobalDesc(..), Field(..))

import Bau.RTN
import FlexCore.Arch
import FlexCore.Target

instance Show (Assembly FlexCore) where
  show (A filename funcs globals) =
    "_____ofile_____,_\λ"
    _____++_filename_++_"\λ"\n\n"
    ++"_____otextλn"
    ++concatMap show funcs ++"λn"
    ++"_____odataλn"
    ++concatMap show globals

instance Show (Function FlexCore) where
  show (F funcname funcargs bbs _) =
    "_____oalign_2λn"
    ++"_____oglobl____" ++funcname ++"λn"
    ++funcname ++":λn" ++concatMap show bbs

instance Show (BasicBlock FlexCore) where
  show (BBU bname mops) =bname ++":λn" ++concatMap (("_____"++)o(++ "λn")oshow)
    mops
  show (BBS bname iws) =bname ++":λn" ++concatMap show iws

instance Show (InstrWord FlexCore) where
  -- show (IW mops) = " " ++show (length mops) ++" microopsλn"
  show (IW mops) ="_____rtn_____" ++show mops ++"λn"

instance Show (MicroOp FlexCore) where
  show (MO v PC mode os _ _) ="PC" ++show mode ++"_" ++concatMap ((++ " ")oshow)
    os
  show (MO v ALUOp mode os n _) ="ALUOp" ++show n ++"_" ++show mode ++"_" ++
    concatMap ((++ " ")oshow) os

```

```

show (MO v LSRead mode os n _) = "LSRead" ++ show n ++ "_" ++ show mode ++ "_" ++
  concatMap ((++ "_" ) ◦ show) os
show (MO v LSWrite mode os n _) = "LSWrite" ++ show n ++ "_" ++ show mode ++ "_" ++
  concatMap ((++ "_" ) ◦ show) os
show (MO v RegRead mode os n _) = "RegRead" ++ show n ++ "_" ++ show mode ++ "_" ++
  concatMap ((++ "_" ) ◦ show) os
show (MO v RegWrite mode os n _) = "RegWrite" ++ show n ++ "_" ++ show mode ++ "_" ++
  + concatMap ((++ "_" ) ◦ show) os
show (MO v t mode os n _) = show t ++ show mode ++ show n ++ "_" ++ concatMap ((++
  "_" ) ◦ show) os

deriving instance Show (UnitType FlexCore)

instance Show (Mode FlexCore) where
  show (Reg n) = "Reg" ++ show n
  show (LSW n) = "LSW_" ++ show n
  show (LSWU n) = "LSWU" ++ show n
  show Stall = "Stall"
  show None = ""
  show m = case m of { Imm → "Imm" ; GetPC → "GetPC" ; JumpSA → "JumpSA" ; JumpSR →
    "JumpSR" ;
    JumpDA → "JumpDA" ; BEQZR → "EQZR" ; BNEZR → "NEZR" ; BEQZA → "EQZA"
    ;
    BNEZA → "NEZA" ; ADD → "ADD" ; ADDU → "ADDU" ; SUB → "SUB" ; SUBU →
    "SUBU" ;
    AND → "AND" ; OR → "OR" ; NOR → "NOR" ; XOR → "XOR" ; SLL → "SLL" ;
    SRL → "SRL" ; SHR → "SHR" ; SLT → "SLT" ; SLE → "SLE" ; SEQ → "SEQ" ;
    SNE → "SNE" ; TEST → "TEST" ; DONTCARE → "DONTCARE" }

instance Show (DataPort FlexCore) where
  show (DP inout unittype n name) = show unittype ++ show n ++ show name
  show (DPU arg) = show arg

instance Show GlobalDesc where
  show (Collection "" fields) = "\n" ++ concatMap show fields
  show (Collection name fields) = name ++ "\n" ++ concatMap show fields
  show (Constant name field) = show field
  show (Zeroes name n) = name ++ "\n_0comm_" ++ name ++ ",_" ++ show n ++ ",_4\n"
  show (Ascii "" s) = "_0ascii_" ++ show s ++ "\n"
  show (Ascii name s) = name ++ "\n" ++ "_0ascii_" ++ show s ++ "\n"

instance Show Field where
  show (Byte f) = "_0byte_" ++ show f ++ "\n"
  show (Half f) = "_0half_" ++ show f ++ "\n"
  show (Word f) = "_0word_" ++ show f ++ "\n"
  show Undef = "_0undef\n"

```

Listing A.10: Main.hs

```
import System.IO
```

```
import System.Exit
import System.Environment
import System.Console.GetOpt
import qualified Data.Map as M

import Bau.Options(files)
import Bau.RTN
import Bau.Translate
import Bau.Schedule

import FlexCore.Arch
import FlexCore.Target
import FlexCore.AsmWriter

main :: IO ()
main =do
  -- Command arguments are being parsed with unsafeIO in Options.hs
  if (null files)
  then do
    prg ←getProgName
    hPutStrLn stderr "No_arguments_given"
    --hPutStrLn stderr (usageInfo prg options)
    exitWith (ExitFailure 1)
  else do
    let res =M.fromList [ (PC,(1,2)), (ALUOp,(1,1)), (AGUOp,(1,1))
                        , (Mult,(1,2)), (MultRegWrite,(1,1))
                        , (Buf,(2,1)), (RegRead,(2,1)), (RegWrite,(1,1))
                        , (LSWrite,(1,1)), (LSRead,(1,1)) ]
        assemblies ←mapM (translateModule res) files :: IO [Assembly FlexCore]
    mapM_ print assemblies
    exitWith ExitSuccess
```


B

Integration with LLVM

Listing B.1: An example of a Makefile to compile a benchmark to bitcode before scheduling

```
BENCHMARK=fft00

CC=llvm-gcc
AS=llvm-as
CCFLAGS=-fplugin-arg-dragonegg-emit-ir -S
INCLUDE=-Ith_lite -Ial -Idatasets

common =al/usr.bc al/thal.bc th_lite/crc.bc th_lite/heap.bc th_lite/thlib.bc
benchmark =${BENCHMARK}.bc bmark_lite.bc

all: bmark-${BENCHMARK}.bc bmark-${BENCHMARK}.ll bmark-${BENCHMARK}.rtn

bmark-${BENCHMARK}.bc: $(common) $(benchmark)
    llvm-lc $(common) $(benchmark) -o bmark-${BENCHMARK}
    # instnamer is required so temporary values will have names
    # lowerswitch is required because compiler writer wouldn't have implement
    # IDSwitch
    opt -instnamer -instcombine -lowerswitch -mergereturn -o $@ $@

bmark-${BENCHMARK}.ll:
    llvm-dis bmark-${BENCHMARK}.bc

bmark-${BENCHMARK}.rtn:
    scheduler bmark-${BENCHMARK}.bc >bmark-${BENCHMARK}.rtn

%.S: %.c
    $(CC) $(INCLUDE) -o $@ $< $(CCFLAGS)
```

```
%.bc: %.S
    $(AS) -o $@ $<

clean:
    rm -f *.S
    rm -f *.bc
    rm -f al/*.bc
    rm -f th_lite/*.bc
    rm -f bmark-${BENCHMARK}.ll
    rm -f bmark-${BENCHMARK}
```