Declarative, SAT-solver-based Scheduling for an Embedded Architecture with a Flexible Datapath

Nikita Frolov, Magnus Själander, Per Larsson-Edefors, Sally A. McKee
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Gothenburg, Sweden
Email: frolov@student.chalmers.se, {hms,perla,mckee}@chalmers.se

***Abstract*—Much like VLIW, statically scheduled architectures that expose all control signals to the compiler offer much potential for highly parallel, energy-efficient performance. 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. We first build a compiler construction library that allows scheduling and resource constraints to be expressed declaratively in a domain-specific language, and then use this library to implement a compiler that generates programs that are 1.2–1.5 times more compact than either a baseline MIPS R2K compiler or a basic-block-based, sequentially phased scheduler.**

I. Introduction

Design-time configurable datapaths may increase computational efficiency for certain applications, but such finely tuned microarchitectures require configurable compiler back-ends to generate code for individual architectural variants. Furthermore, an exposed datapath controlled by wide instruction words places the burden of assigning microinstructions (syllables) to units on the compiler, rather than on the instruction decoder. Datapaths with flexible interconnect templates may enjoy more routes between execution units than would a fixed architecture. In contrast to traditional pipelined architectures whose compiler back-ends perform instruction selection and register allocation in separate, consecutive phases, these exposed architectures force compilers to concurrently assign units and value locations that mutually depend on each other.

We present a method to divide the scheduling problem into subproblems by expressing them as logical constraints to be simultaneously considered by a SAT solver. We build a compiler construction library, Bau, that allows generic scheduling constraints and target-specific resource constraints to be expressed declaratively in a manner independent from both the set of available execution units and from the instruction decode logic. We use this library to implement a new compiler for an instance of an exposed architecture based on the FlexCore [1] processor architecture defined in the FlexSoC project [2]. We compare performance of three processor/compiler combinations: a reference MIPS using gcc, our FlexCore implementation using a sequentially phased scheduler, and our FlexCore instance using our solver-based scheduler.

II. Target architecture

FlexCore's exposed architecture lacks a conventional instruction set architecture (ISA) and has no fixed set of assembly instructions. Operations at the machine level can instead be expressed as register transfer notations (RTN) specifying operations to be performed on output port registers of the various datapath units (Figure 1). The output port from which a value is read represents the address of the interconnect multiplexer, and the operation represents the control signals (i.e., the op-code) to a specific datapath unit. 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 [3].)

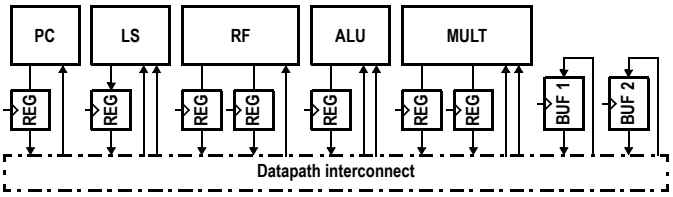

Fig. 1. A basic variant of FlexCore extended with a fast multiplier unit.

Architecture variant description has to be provided to the compiler as a list of valid operations and available resources. It is represented as a list of triples, where every triple denotes a type of execution unit, number of units of this type available, and the unit type's latency. For example, for a Viterbi accelerator, a configuration file entry might look like `(VITERBI, 1, 3)`, which means "this architecture variant has one Viterbi accelerator, and its latency is three cycles". The scheduler relies on this information to avoid hazards when allocating resources.

III. Compiler Challenges

We are using LLVM [4] for front- and middle-ends of the compiler. LLVM is a powerful set of tools for compiler construction, and rapidly gains popularity because of greater modularity than provided by still dominant GCC [5]. Many languages can already be compiled to LLVM bytecode, and many optimization techniques are implemented as LLVM passes. Although LLVM has many off-the-shelf components for developing RISC or CISC back-ends (e.g., instruction selectors and schedulers, register allocators, and peephole optimizers), these cannot be readily reused for an exposed architecture. Compiling LLVM bytecode to RTN assembly requires three steps: lowering the LLVM instructions to the RTN microoperations (*uops*) supported by a given variant of architecture, allocating resources to assign uops to execution unit instances and intermediate values to memory locations, and adjusting RTN microcode according to the produced schedule. Figure 2 illustrates the flow chart of activities implementing these steps.

The first step generates RTN in its template form, which includes references to types of execution units (but not to specific units, themselves) and specifiers for which dataport to use on a given unit type. The static single assignment form (SSA) of LLVM bytecode is preserved in this transformation, and lowering can be accomplished by instantiating instruction templates defined in terms of variant-specific uops. Bau is written in Haskell, so both LLVM and RTN code are represented with a hierarchy of abstract data types, and the

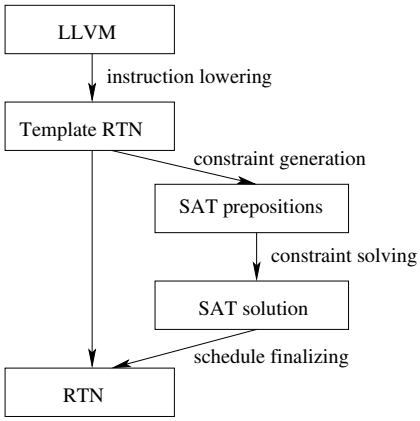

Fig. 2. Compilation flow

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 type system performs dispatch according to the fully specified type of target (Figure 3).

```
class Target arch where
lower :: (String, InstrDesc) -> [MicroOp arch]
```

Fig. 3. Interface for variant-specific lowering functions

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 interface shown on Figure 3. It is possible to generate code for a specific operation and operand type while still preserving the SSA form of LLVM (Figure 4).

```
instance Target core where
...
lower (v, IDBinOp BOAdd
                    (TDInt U 32) a b) =
 [ MO2 v ALUOp 0 AO_ADDU (DPU a) (DPU b) ]
...
```

Fig. 4. A LLVM-RTN lowering rule written in Haskell

The second step consists of resource allocation — specifying units at which to map uops and the dataports from which to read operands. The latter is required, because an exposed architecture imposes the burden of forwarding data between execution units on the compiler. Because every unit has its own register to store its result, the choice of dataports for a uop to read operands from depends on the unit used to compute a given operand. It means that the compiler cannot perform instruction scheduling and register allocation sequentially. In a system based on an exposed architecture, these operations must be performed concurrently and globally, which is reflected as the *phase sequence problem* [6]. 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 schedulers, due to both the simplicity of formulating and refining problems and to the impressive performance [7].

The third step is to substitute identifiers of unit instances and names of dataports into RTN code according to the schedule produced by the SAT solver. Spilling code generation is performed when the value schedule is interpreted — not only dataport names are supplied into a uop instead of value names, but additional uops are inserted to ensure that values are stored in memory after they are computed and read back when they are required by other uops. Different spilling strategies can be implemented to either optimize for access latency or for total number of transfers. The current version of Bau uses a simple strategy that allocates faster memory for values that are to be used sooner.

IV. SAT Interface

Issues of interaction with a SAT solver are abstracted away by the `satchmo` library [8]. `satchmo` provides 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. Constraints can be imposed on relations with the `assert` function that builds clauses out of variables included in the relation. After the solution is found, the relation data structure representing the schedule can be translated to the final RTN code.

`satchmo` provides the `SAT` monad that encodes the SAT problem and is based on the `State` monad. A constraint is then defined as a function that generates propositions in conjunctive normal form (CNF) accepted by the SAT solver by combining `asserts`. Composition of several constraints can thus be represented by composition of several monadic functions (Figure 5).

```
bbConstrs constrs res scheds bbs = do
  forM_ (zip bbs scheds)
    $ \ ((BBU label succs ops), s) -> do
      mapM_ ($ (res, ops, s)) constrs
```

Fig. 5. Local scheduling constraint combinator

Sec. V and VI formulate the scheduling problem as a constraint satisfaction problem. Constraints are defined over a set of triples of uop name (i.e., name of SSA value produced by it), resource name (execution unit or memory location) and cycle number. Every possible schedule entry (triple) can be encoded by a boolean variable with three indexes x_{orc}, where o corresponds to value name, r to resource 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 uop and execution unit instance are also given a numerical identifier with the maxima of O being equal to the number of uops in a basic block and U and L being equal to amount of all unit instances and memory locations accordingly, regardless of their type. The maximum location index L 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.

V. Instruction Placement

A. Problem Statement

After the program has been expressed in terms of uops 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 instruction ordering problem can be defined as follows [9]:

- 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;
- instruction ordering preserves data dependencies.

B. Translation to Propositions

1) Every instruction has exactly one entry in the schedule: As 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 corresponding type UT. First, an instruction o should have no more than one entry in the schedule. This is achieved by demanding impossibility of every unique pair of variables to be assigned with true:

$$\forall o, u_1, u_2, c_1, c_2 : u_1, u_2 \in UT(o), \overline{x_{o,u_1,c_2} \wedge x_{o,u_1,c_2}} \quad (1)$$

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:

$$\forall o : \bigwedge_{\substack{i=1 \\ k=1}}^{\substack{i=U \\ k=C}} x_{i,u,k} \quad (2)$$

2) 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:

$$\forall o_1, o_2, u, c : o_1 \neq o_2, \overline{x_{o_1,u,c} \wedge x_{o_2,u,c}} \quad (3)$$

3) Instruction types match: If types of an instruction and an execution unit do not match (e.g., an ALU operation cannot be assigned to a load-store unit), the corresponding variables should never be assigned with true:

$$\forall u : u \notin UT_o, \bigwedge_{\substack{i=1 \\ k=1}}^{\substack{i=O \\ k=C}} \bar{x}_{i,u,k} \quad (4)$$

4) Data dependencies are not broken: Given an instruction c dependent on the result of instruction p and a latency d_u, it could be assumed that c should never be scheduled before cycle d_u after p:

$$u_p \in UT_p, u_c \in UT_c, x_{p,u,c_p} \rightarrow \neg \bigvee_{c_c \leq c_p + d_u} x_{c,u,c_c} \quad (5)$$

VI. Value Placement

A. Problem Statement

The SSA form of RTN code defines paths between value producers and value consumers, and for values with many consumers those paths may overlap. *Value paths* define where a value will reside at a given point in time. A value can travel between two units in a number (and a combination) of ways: directly through the interconnect or through a pipeline buffer, a register or memory. The constraints can be summarized as following:

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

It should be noted that priorities of different memory levels (e.g., prefer a register to a RAM location) are not considered at the constraint resolution layer. It is the solution interpreter who has to select locations with faster access times based on amount of accesses to a values or some other criteria (Section IV).

B. Translation to Propositions

Every basic block has its independent schedule, because uops cannot be moved between basic blocks without 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 BB, 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.

1) Every location stores just one value at a time: For every memory location l there should not be any pair of variables assigned with true in the same cycle:

$$\forall bb \in BB : \forall o_1, o_2 \in O_{bb}, (u, c_{bb}) : \overline{y_{o_1,u,c_{bb}} \wedge y_{o_2,u,c_{bb}}} \quad (6)$$

2) Location types match: If types of a value and a memory location (e.g., a register) do not match, corresponding variables should never be assigned with true:

$$\forall bb \in BB : \forall u : u \notin UT_o, \bigwedge_{\substack{i=1 \\ k=1}}^{\substack{i=O_{bb} \\ k=C_{bb}}} \bar{y}_{i,u,k} \quad (7)$$

3) Value paths are not broken and have a start and an end: A value should be stored during all cycles between the ones on which the producer and the consumer instructions are scheduled, not inclusive (if the path has zero length, it would mean direct forwarding), and not on the others:

$$\forall bb \in BB : \forall o_p \in O_{bb} : (x_{o_p,u,c} \wedge x_{o_c,u,c+d_u}) \oplus \bigoplus \bigwedge_{\substack{j=1 \\ k=1}}^{\substack{j=L \\ k=c_{bb}}} y_{o,j,k} \quad (8)$$

VII. Results

We use the FlexCore [1] processor as a representative for design-time configurable and exposed architectures to compare performance of different schedulers. Three benchmarks from the EEMBC benchmark suite [10] were compiled for this architecture.

Table I compares size of benchmark assembly code as produced by a sequentially phased scheduler [11] and Bau. Note that the two schedulers accept different input representations — the sequential scheduler works on MIPS assembly, and Bau works on LLVM bytecode. Since LLVM has SSA form and never reuses value names, an LLVM program is lexically longer than an equivalent MIPS program.

	Autcor	FFT	Viterbi
MIPS	346	691	617
seq.	402	827	641
LLVM	465	698	690
Bau	204	295	401

TABLE I
Total Number of Instructions

RTN assembly produced by Bau is twice as short as RTN assembly produced by the sequential scheduler, but it doesn't say much about performance. EEMBC benchmarks have similar organization — the setup and output sections that are largely sequential but perform many

function calls for result output, and the actual computation routine that has no to few function calls but has many inner loops. Table II shows code size for computationally intensive functions that lie at the core of the benchmarks.

	Autcor	FFT	Viterbi
MIPS	38	337	324
seq.	42	392	307
LLVM	70	569	617
Bau	35	266	338

TABLE II
NUMBER OF INSTRUCTIONS IN INNERMOST FUNCTIONS OF VARIOUS BENCHMARKS

Bau produces code that is 20-50% smaller for the autocorrelation and FFT transform benchmarks, but on the Viterbi algorithm implementation Bau performs 10% worse. The reason for this difference is that `autcor` and `fft` benchmarks do not perform any function calls at all during the computation, and `viterbi` depends on a recursive function, which imposes high pressure on the call stack in the current version of Bau.

VIII. Related Work

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 choose an optimal scheduling phase sequence were summarized by Norris and Pollock [6]. They have also proposed several strategies for phase communication and making instruction scheduler and register allocator mutually sensitive. More recent works on the topic [12], [13] have not departed from the scheme of separate but iteratively communicating phases but barely proposed alternative communication strategies. A notable exception is [14] where serialization of CFG is performed incrementally. However, neither approach is directly applicable to an exposed architecture where the compiler has control over the instances of executions units and the forwarding paths a value can take.

A major reason not to unite instruction reordering and register allocation is the NP-hardness of a combined problem [15]. 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 a solution quickly [7], [16], 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 [17]. Improving performance of SAT solvers is a topic of ongoing research, with proposed support for parallelization of solvers to run on multicore and multinode architectures [18], [19], GPUs [20] and FPGAs [21].

IX. Conclusion

The Bau library reduces required effors to develop compilers for new variants of configurable architectures by separating architecture-specific resource constraints from the generic scheduling constraints. Furthermore, it enables separating scheduling algorithm from the scheduling engine (in this case, the SAT solver) and reduce the code base that developers must maintain. We have demonstrated that modularized design of the scheduler establishes clear boundaries between loosely-related properties of the schedule and allows better utilization of hardware resources. In many cases, Bau generates schedules that are 1.2–1.5 times smaller than sequentially phased scheduler. This has important implications for the design of future, energy-efficient, application-specific embedded systems.

References

[1] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "FlexCore: Utilizing exposed datapath control for efficient computing," *Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, 2009.

[2] 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*, Nov. 2003, pp. 52–55.

[3] M. Thuresson, M. Själander, and P. Stenstrom, "A flexible code compression scheme using partitioned look-up tables," in *Proc. High Performance Embedded Architectures and Compilers*, Jan. 2009, pp. 95–109.

[4] 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*, Mar. 2004, pp. 75–86. [Online]. Available: http://llvm.org

[5] "GNU Compiler Collection," http://gcc.gnu.org.

[6] C. Norris and L. Pollock, "Experiences with cooperating register allocation and instruction scheduling," *International Journal of Parallel Programming*, vol. 26, no. 3, pp. 241–284, 1998.

[7] N. Een and N. Sörensson, "An extensible SAT-solver," in *Proc. International Conference on Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, May 2003, no. 2919, pp. 333–336.

[8] "satchmo: SAT encoding monad," http://dfa.imn.htwk-leipzig.de/satchmo/.

[9] S. Memik and F. Fallah, "Accelerated SAT-based scheduling of control/data flow graphs," in *Proc. IEEE International Conference on Computer Design*, Sep. 2002, pp. 395–400.

[10] J. Poovey, T. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the EEMBC benchmark suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, September/October 2009.

[11] T. Schilling, M. Själander, and P. Larsson-Edefors, "Scheduling for an embedded architecture with a flexible datapath," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, May 2009, pp. 151–156.

[12] I. Cutcutache and W.-F. Wong, "Fast, frequency-based, integrated register allocation and instruction scheduling," *Software: Practice and Experience*, vol. 38, no. 11, pp. 1105–1126, Sep. 2008.

[13] D. Koes, "Register allocation aware instruction selection," Carnegie Mellon University School of Computer Science, Tech. Rep. CMU-CS-09-169, Oct. 2009.

[14] 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*, Apr. 2003, pp. 1–16.

[15] R. Motwani, K. Palem, V. Sarkar, and S. Reyen, "Combining register allocation and instruction scheduling," Courant Institute, Tech. Rep. TR 698, Jul. 1995.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 38th ACM/IEEE Design Automation Conference*, Jun. 2001, pp. 530–535.

[17] J. Crawford and A. Baker, "Experimental results on the application of satisfiability algorithms to scheduling problems," in *Proc. Conference on Artificial Intelligence (AAAI)*, Jul. 1994, pp. 1092–1097.

[18] M. Lewis, T. Schubert, and B. Becker, "Multithreaded SAT solving," in *Proc. of the 12th Asia and South Pacific Design Automation Conference*, Jan. 2007, pp. 926–931.

[19] Y. Hamadi and L. Sais, "ManySAT: a parallel SAT solver," *Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 12, pp. 245–262, Jun. 2009.

[20] C. 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*, Nov. 2002, pp. 306–317.

[21] A. Dandalis and V. Prasanna, "Run-time performance optimization of an FPGA-based deduction engine for SAT solvers," *ACM Transactions on Automation of Electronic Systems*, vol. 7, no. 4, pp. 547–562, Oct. 2002.