2009 IEEE Computer Society Annual Symposium on VLSI

Scheduling for an Embedded Architecture with a Flexible Datapath

Thomas Schilling, Magnus Själander, Per Larsson-Edefors

Chalmers University of Technology
nominolo@gmail.com, hms@chalmers.se, perla@chalmers.se

Abstract

Embedded systems put stringent demands on post-fabrication flexibility as well as computing performance efficiency. The FlexSoC scheme approaches the implementation of embedded systems from a general-purpose processor point of view: The FlexCore processor has a datapath whose configuration is under instruction control; in its minimal configuration, the processor represents a simple 5-stage pipeline. However, thanks to a flexible processor interconnect, the FlexCore datapath configuration can be changed at run-time to boost performance for the currently executed code. The consequence of this flexibility is that pipelining is not hard-coded into the datapath, but all instruction scheduling needs to be done by software at compile time. We present a scheduling technique for the FlexCore processor allowing for efficient use of datapath resources over a flexible interconnect. The flexible interconnect indeed offers plenty of opportunities for parallel operations, but it also makes the analysis of instruction dependencies difficult. Thus, we propose to use a SAT-solver to enable the scheduler to efficiently check constraints on computing and communication resources. In an evaluation on four different benchmarks, our scheduler is shown to produce schedules that are as efficient as fine-tuned, manual schedules.

1 Introduction

Embedded electronic designs such as mobile devices must provide high computing performance at low power and energy dissipation. Although a general-purpose processor (GPP) is useful for many computing tasks, it is very inefficient for certain performance-demanding tasks. This is especially true in the embedded domain, for which the executed applications are not very diverse but belong to a limited number of categories. For these reasons, a contemporary embedded design typically consists of a simple GPP that is paired with a number of special-purpose units (accelerators) tailored to the intended set of applications.

Since accelerators are external to the GPP and its datapath, the prevalent design methodology has two serious shortcomings: First, instructions for accelerators are not part of the GPP instruction set, so they often have to be awkwardly programmed by writing to special registers or memory addresses. If the GPP can only issue one instruction at a time, accelerators should preferably perform complex, multi-cycle instructions to achieve notable speedups. Second, since it is difficult to integrate accelerators into an existing datapath, accelerators can rarely share computing resources with the GPP and need their own ALU-like resources. As a result, the hardware grows quickly with the addition of accelerators, incurring an area and power overhead.

The FlexSoC scheme [7] aims to alleviate the listed shortcomings using two key ideas. *i) Flexible interconnect:* Instead of having a fixed datapath and data bus, a baseline FlexCore processor contains a fully-connected datapath interconnect, which allows each unit to receive inputs from the output of any unit, including itself. *ii) Pipeline on demand:* The control bits of all datapath units of a FlexCore as well as to its interconnect are exposed to the software layer. This allows for an extremely flexible dataflow in which the pipeline depth varies.

The FlexSoC scheme facilitates the integration of hardware accelerators but moves the implementation complexity into the software layer. In particular, in a FlexCore processor it is no longer possible to use the conventional separation of instruction scheduling and register allocation. Resource conflicts between instructions are highly dependent on the order of instructions and have to be resolved at compile time. We describe the FlexCore scheduler, which has been designed with the following key features:

- To achieve high utilization of existing resources and forwarding capabilities of the interconnect.
- To use local buffers, rather than the register file, to store temporary results.

- To support communication-path constraints of the flexible interconnect, i.e., to create a schedule that does not use certain, "prohibited" interconnect paths.

The remainder of this paper is organized as follows: Section 2 gives an overview of the FlexCore architecture. Section 3 describes our scheduler in detail. In Section 4 we evaluate the quality of the generated schedules, and Section 5 concludes this paper.

2 The FlexCore Architecture

In modern conventional Instruction Set Architectures (ISAs) most instructions are self-contained in the way that they perform an operation on a set of input operands and write back the results to a register file. The assumptions that the input operands are values from a register file[1] and that each instruction is always executed in the same way, regardless of prior and subsequent instructions, greatly simplify scheduling of instructions: Now each instruction can be scheduled without considering neighboring instructions—only data and a few control constraints, e.g. read after write, limit the ordering of instructions. There is, however, one disadvantage associated with the strategy above; it becomes necessary to add hardware resources to perform forwarding and to check for hazards at runtime. This is due to the fact that most ISAs do not provide means to consider neighboring instructions and their interaction with each other. Since instruction interactions are monitored and controlled by hardware, the data resulting from an instruction is always routed along similar routes and interact with other instructions in a specific way. This certainly keeps the complexity of the control logic down, since allowing data to flow through a datapath along all possible routes would require excessive amounts of logic circuits, and lead to an extremely complex processor design. A modern conventional ISA therefore imposes strict limitations on how resources of a datapath implementing the ISA can be utilized.

The FlexCore processor abandons the conventional, fixed ISA to offer a more fine-grained and unlimited control of the datapath resources. All control signals for the entire FlexCore datapath and all address bits of the flexible interconnect are exposed to the compiler. This allows for very powerful scheduling, since the interaction between different instructions can and must be handled by the compiler. However, the scheduling phase becomes more complex and has to deal with completely new requirements. The FlexCore scheduling will be presented in Section 3.

An added benefit of not having a fixed ISA is that this allows for easy inclusion of new datapath units, which is important in the embedded domain. The addition of a new datapath unit is straightforward: *i)* Add routing resources between the FlexCore interconnect and the new unit. *ii)* Add the new unit's control signals to the existing wide control word of the FlexCore processor. Once the compiler has been updated with the notion of the new datapath unit and its routing limitations, if any, the new unit can be scheduled in the same way as any other unit of the datapath.

The simplest FlexCore processor (our baseline processor) consists of the same datapath units found in a simple five-stage 32-bit GPP pipeline. The datapath units are connected to a fully-connected interconnect that allows data to be routed freely between the different units (see Figure 1). Two data buffers (RegA and RegB) are attached to the interconnect, since two buffers are used in conventional five-stage GPP pipelines to forward data past the ALU (when performing store operations) and past the load-store unit (when performing arithmetic or logical operations). With respect to datapath units, the baseline processor is a minimum configuration that allows a FlexCore processor to act as if it were a GPP and ensures that an application can be executed on a FlexCore with at least similar performance as if executed on a five-stage GPP pipeline. Performance can be improved in later design stages by adding application-specific datapath units (accelerators), simply by connecting them to the interconnect. There is no need for any complex modifications of the control logic, since data movements and forwarding is handled by the scheduler, in software.

A fully-connected interconnect is resource demanding and does not scale well to a large number of interconnect ports. However, it is possible to identify which interconnect routes are utilized by a particular application, since all routing of data is statically determined during the scheduling phase [5, 7]. The used routes are identified by observing the combined set of interconnect routes for the set of applications to be executed on the FlexCore processor. All unused routes are subsequently removed from the interconnect. As long as we do not remove any paths that exist in the five-stage pipeline of a GPP, any application (also the ones that are not benchmarked for the design of the interconnect) can

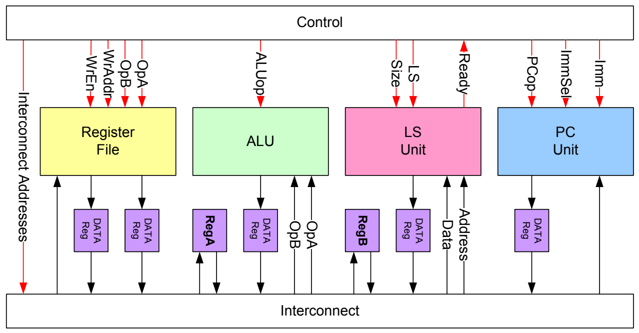

Figure 1: Illustration of the baseline FlexCore processor.

[1]Immediates passed through the instruction can also be used as input operands.

be executed on the FlexCore processor.

For the development of the scheduler of this work, the baseline FlexCore processor was extended with a multiplier unit. The multiplier was included to support a fast Fourier transform application, which is used as one representative of typical embedded applications. The control signals of the multiplier unit together with the datapath units of the baseline FlexCore[2] and the address signals of a fully-connected interconnect form a 109-bit wide control word. In the FlexSoC scheme, the generation of the 109-bit wide instructions is the task of the scheduling engine of the compiler.

A 109-bit wide instruction format would require an excessive amount of instruction bandwidth, as well as a large memory footprint for the applications encoded in this instruction format. To remedy this, a compression technique was developed within the FlexSoC project, by Thuresson *et al.* [8]. This technique is based on partitioned look-up tables and generates, for the selected set of applications, information on the appropriate look-up table configuration as well as on how these tables should be updated at run time. The proposed technique was shown to reduce the 109-bit wide instructions to 64 bits (that is, by 40%) for the benchmarks used in our work, i.e., Autocor, FFT, and Viterbi.

3 Scheduling for FlexCore

The FlexCore scheduler processes each basic-block separately using an algorithm similar to list scheduling. The main difficulty is that data routing constraints directly affect the placement of instructions, thus the scheduler has to perform both at the same time. The basic idea of our scheduler is to select instructions greedily, but ensure that all already placed instructions can receive their inputs. Datapath constraints are tracked, checked for satisfiability, and solved using a SAT-solver as outlined in Section 3.1.

The input to the scheduler is a dependency graph of *primitive operations*, i.e., operations that can be executed on a single unit. ALU instructions correspond to exactly one primitive operation, but branch instructions and memory accesses can be mapped to multiple primitive operations (e.g., for memory instructions: address calculation and memory load or store).

The schedule is built up cycle by cycle, from *last* cycle to first cycle. The widespread technique to schedule the branch last and subsequently move instructions into delay slots does not work for the FlexCore processor, since moving instructions is hindered by data routing constraints. Scheduling backwards makes it easier to place instructions in delay slots—when placing the branch instructions, all instructions scheduled before automatically wind up in delay slots. The FlexCore processor has at least one delay slot, possibly even two, depending on instruction decoder overhead.

[2]ALU, register file, load-store unit, program-counter unit, and two 32-bit buffers.

Like list scheduling, the algorithm maintains a ready-list and greedily picks instructions from this list as long as enough computing resources are available. The algorithm then validates that each of these instructions can also obtain all its inputs, adjusting the list of chosen instructions where necessary. This process requires making assumptions about which instructions will be scheduled next, because outputs of these instructions can be forwarded directly and reduce pressure on the datapath. Consider the following partial schedule:

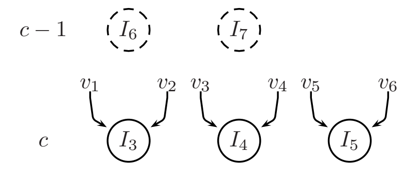

Figure 2: Example (partial) schedule

Here c is the cycle currently being scheduled. Three instructions could potentially be scheduled in this cycle as there are enough execution units available to execute them. These three instructions require six values as inputs. Even with only two register read ports, all instructions may still be able to receive all their inputs if a sufficient number of the following conditions are met:

- Two or more instructions use the same input. The same value can be routed to all units that need it.
- Inputs are read from buffers (RegA or RegB in Figure 1). Due to the flexible interconnect, the two buffers can be used as two additional registers to store intermediate results.
- Values can be forwarded directly. If the needed value is produced in the preceding cycle it can be read directly from the output of the execution unit.

To find out which outputs may be produced in the preceding cycle the algorithm must make a guess. This guess is made using a heuristic and needs to be verified when the algorithm processes the respective cycle. If the guess turns out to be wrong, the list of selected instructions will have to be adjusted and the algorithm has to backtrack if already scheduled instructions depended on a removed instruction.

For example, in Figure 2 above, assume that I_5 could only be scheduled in cycle c if I_7 can be scheduled in cycle $c-1$. If, upon scheduling cycle $c-1$, it turns out that I_7 cannot be scheduled in that cycle, I_5 cannot be scheduled in c, either. Removing I_5 from cycle c may again result in instructions being removed from cycle $c+1$ and so on.

This need for backtracking makes the algorithm's run time non-linear (potentially exponential) in the number of instructions. Fortunately, in our tests backtracking was rare and did not affect performance noticeably. The frequency of backtracking, however, depends on the heuristics used for guessing as well as the balance of functional units and storage units (i.e., register ports).

In order to handle execution units with multi-cycle latency, the checking of constraints for a single instruction is spread out even further. Figure 3 shows the four steps performed for each cycle and Figure 4 describes the top-level structure of the algorithm in pseudo-code.

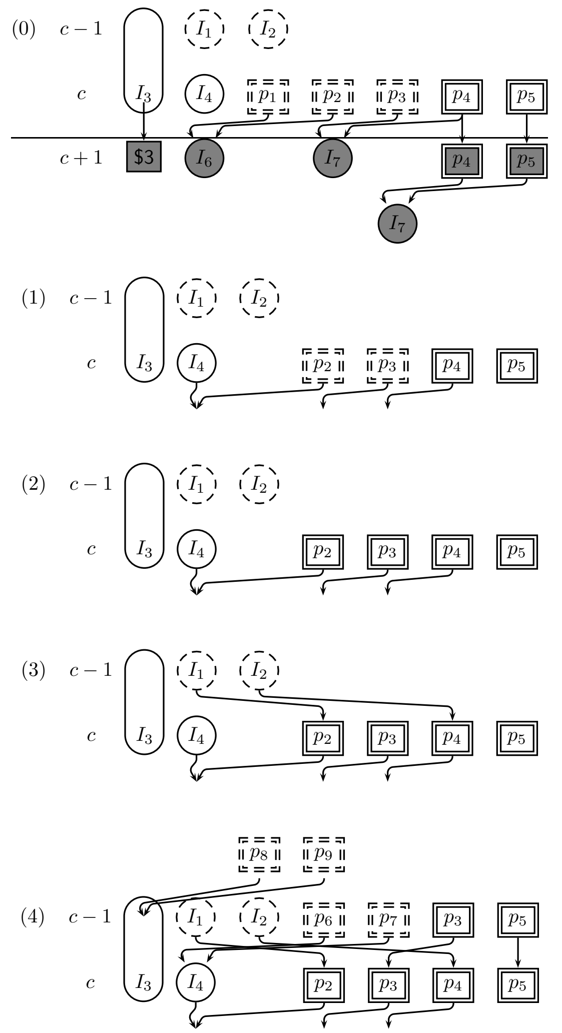

Figure 3: Core steps of the algorithm.

The scheduler deals mainly with three sets: The list of selected instructions I (I_3 and I_4 in Figure 3), a list of values that need to be read in the current cycle $N(c)$[3] (p_1–p_4), and a list of variables that are live L (p_4, p_5), i.e., variables that need to be stored in some temporary location. The SCHEDULECYCLE function takes three additional parameters: The current cycle c, a graph U of remaining instructions with their dependencies, and the current schedule S.

[3] N is in fact a map of sets, indexed by cycle.

```
1:  function SCHEDULECYCLE($c$,$U$,$S$,$I$,$N$,$L$)
2:      if $U = \emptyset \wedge N = \emptyset$ then return $S$
3:      $N' \leftarrow ForwardDirectly^*(c, I, N, S)$
4:      $AddConstraintsReadable^*(c, N')$
5:      $L' \leftarrow AddNewLives(N', L)$
6:      $(I', U') \leftarrow GuessInstructions(c-1, U, S)$
7:      $L'' \leftarrow AddConstraintsWriteback^*(I', L')$
8:      $L''' \leftarrow ExtendLiveRange^*(L'')$
9:      $N'' \leftarrow AddNextNewLives(S, I, N')$
10:     $S' \leftarrow AddToSchedule(S, I)$
11:     $R \leftarrow ScheduleCycle(c-1, U', S', I', N'', L''')$
12:     if $R$ is an error then
13:         try again with smaller $I'$ or fail if $I' = \emptyset$
14:     return $R$
15:
16: function SCHEDULEBASICBLOCK($G$)
17:     $S \leftarrow ScheduleCycle(1, G, \emptyset, \emptyset, \emptyset, \emptyset)$
18:     $R \leftarrow GetSATSolution$
19:     return $GenerateSchedule(S, R)$
```

Figure 4: Core Algorithm

In Figure 3, I_3 and I_4 have been selected to be scheduled in cycle c. Values p_1 through p_4 need to be readable in cycle c because they are inputs to instructions starting in cycle $c + 1$. Finally, p_4 and p_5 are live variables, i.e., it is known that these values cannot be forwarded directly and have to be stored in a register or buffer in cycle c.

Note that at this point p_1 to p_5 are not assigned to any register or buffer, yet. Instead the SAT-solver keeps track of possible locations for each value and ensures that there exists at least one solution to find a valid location for each.

1. *ForwardDirectly*. The first step detects which values can be forwarded directly because they are outputs of instructions selected for the *current* cycle c.

2. *AddConstraintsReadable*. All remaining values need to be readable from a temporary storage location. The second step generates the proper constraints and ensures that they can be satisfied. Values that are not outputs of any instruction in the current basic-block must either be read from registers or stand for constants and must be read from the immediate port.

3. *AddConstraintsWriteback*. The third step takes care of write-backs. Outputs of instructions ending in the preceding cycle $c-1$ may define live variables, including variables that are only live during one cycle (e.g., p_2).

Values that will be needed outside the current block must to be written back to the register file.

The constraints are again expressed and checked via the SAT-solver. This step requires the guess of which instructions will be scheduled in cycle $c-1$ which will be validated when the next cycle is scheduled.

4. The final step prepares the input sets for scheduling the next cycle. Values that are inputs to instructions in I with latency l need to be read in cycle $c - l$ and are accordingly added to the set $N(c - l)$. Values corresponding to outputs of instructions in the current basic-block that have not yet been scheduled become live variables. An important detail is that we allow these values to change locations between cycles, or even be duplicated, if that turns out to be useful. To do so, we merely generate constraints for how values may move from one location to another; for example, if a value is in the register file in cycle $c - 1$ then it may be written to a buffer in cycle c provided there is a read port available in cycle $c - 1$. It is then up to the SAT-solver to decide where the value will actually be stored.

   Finally, the algorithm advances to the next cycle and the assumed instructions in this cycle (I') become the selected instructions for the next cycle (I).

Each step above may fail. Either there is no direct route on the datapath (recall that we may enforce some restrictions on the datapath to improve overall efficiency) or there is no way to fulfill the datapath constraints. If that is the case, the algorithm backtracks to a previous cycle, removes the least important instruction (according to the list-scheduling priority function) from the list of selected instructions and tries again.

We did not implement more sophisticated backtracking since the scheduler run time was negligible, but for less well-balanced datapath designs more sophisticated backtracking may be necessary.

3.1 Enforcement of Datapath Constraints

An important goal of our scheduler was to take best possible advantage of the flexibility of the interconnect. Assigning fixed locations to values on the datapath is reasonable, if there exists only one register file in which all temporary values are stored. The FlexCore's buffers are best used for short-lived values. However, the lifetime of a value is only known after all instructions have been scheduled, which in turn depends on *where* values are stored.

Modern SAT-solvers are remarkably efficient and well-suited for expressing routing constraints. A SAT-solver is a constraint solver for propositional formulas, i.e., logical formulas involving only Boolean operators and Boolean variables. A Boolean variable for a value on the datapath expresses the decision "In cycle c, store value v in location l" and is written (c, v, l). If in cycle c, a value v can be in any of the locations l_1 or l_2, the constraint $(c, v, l_1) \vee (c, v, l_2)$ is added to the SAT-solver. More constraints are needed to ensure valid schedules; for example, only one value may be read from a read port at any time. Due to space reasons we have to refer to [4] for further details. Our implementation uses the (unmodified) MiniSAT solver [1].

Once all instructions of a basic-block have been scheduled, the SAT-solver is queried for a solution. This solution is then used to generate the proper control bits for the interconnect configuration. The SAT-solver will return *one* solution, not necessarily the best solution; it may contain unnecessary moves, but these can easily be cleaned up by adding further constraints as long as a solution can be found (e.g., if a value switches locations, add the constraint that this is not the case.)

3.2 Register Allocation

The SAT-solver does not perform register allocation. Instead, conventional (global or local) techniques such as graph-coloring or linear-scan register allocation can be used with usual spilling and rewriting steps. (Our implementation actually takes MIPS code as input and uses linear-scan inside one basic-block while reusing most of the original allocation.)

3.3 Block-Boundary Optimizations

Because of missing (implicit) pipelining, a basic-block schedule generated by the described algorithm will always contain operand reads in the first cycle and possibly write-backs in the last cycle. For basic-blocks where all predecessors are known, those reads can be moved into the last cycle of all preceding blocks. For the case of conditional branches this is a form of speculative execution. This is safe because the read values will be discarded if the alternative execution path is taken.

This optimization can be done after all basic-blocks have been scheduled and benefits from information about the most common execution traces. We have not implemented this optimization, yet, but have performed it manually for the inner loops of the benchmarks in Section 4.

3.4 Related Work

Our scheduler was inspired by Rimey's [3] ASIC scheduler which also uses a greedy algorithm. Rimey models the datapath constraints with a network-flow graph, which is tricky to implement and has some limitations. While a SAT-solver is more complex, modern SAT-solver implementations are highly optimized and are therefore still likely to be more efficient than a special-purpose implementation.

A scheduler with very similar requirements was developed for the NISC architecture [2]. It partitions the dataflow graph into trees of instructions contributing to the same output and then schedules one tree at a time. Instructions obtain their inputs either directly from an execution unit or from the register file. This depth-first processing of the dataflow graph tries to create a maximum number of opportunities for direct forwarding. However, if that is not possible the value needs to be stored in the register file, which in turn requires many read ports if parallel execution is desired. For the FlexCore processor we wanted to take advantage of the two data buffers as cheap temporary storage locations. If all temporary values are read from the register file, exact storage locations (register number) can be assigned in a separate pass. With heterogeneous storage, allocation has to be done in parallel with instruction selection.

4 Results

The main goal of the scheduler was to get as close as possible to manual, fine-tuned schedules. Table 1 shows that this is indeed possible.

	Auto-cor	FFT	Viterbi	Rgb-hpg
MIPS	10	37	15,9,13,15,10	41
Manual	8	21	14,10,12,15,10	37
Autom.	8 (7)	21 (19)	14,10,13,15,11	38

Table 1: Number of control words for the inner loops of various benchmarks.

Table 1 shows the number of control words for the basic-blocks of the inner loops of three EEMBC benchmarks—the numbers are equivalent to the number of cycles executed. The results are for a minimal FlexCore processor with a multiplier and include the optimization mentioned in Section 3.3[4].

The numbers in parentheses can be obtained by allowing write-backs after the delay slot(s). This is possible if the write-back is moved to the beginning of all successors of the basic-block. This transformation, however, may require additional instructions in blocks preceding a loop, because the written value has to be read from the same output port. This optimization may lead to better results than manual schedules, because it apparently had not been considered when those were done.

In one special case (a long sequence of load instructions) our scheduler produces a schedule worse than MIPS, because it does not perform the address calculations in parallel to the memory accesses. A possible fix would be to fine-tune the priority function to jointly schedule instructions that contribute inputs to the same instruction, as described in [6]. Another solution would be to just fall back to MIPS emulation if the automatic schedule turns out to be longer.

[4]Otherwise most blocks would be one cycle longer

The (unoptimised) scheduler itself is reasonably fast. Depending on the basic-block size, it schedules about 500 MIPS instructions per second. About 20-40% of the time is taken up by the SAT solver.

5 Conclusion

In this paper we described an algorithm that can automatically generate efficient code for the FlexCore architecture. The generated code is comparable in quality to manually scheduled code, but can be generated in a fraction of the time.

The performance of the FlexCore design relies on sophisticated software tools to utilize its flexibility. The existence of a good instruction scheduler is an important step toward such a set of tools.

References

[1] N. Een and N. Sörensson. An Extensible SAT-solver. SAT 2003, 2003.

[2] M. Reshadi and D. Gajski. A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 21–26, New York, NY, USA, 2005. ACM.

[3] K. E. Rimey. *A compiler for application-specific signal processors*. PhD thesis, 1989. Chair-Paul Hilfinger.

[4] T. Schilling. Instruction Scheduling for an Exposed Control Architecture. Master's thesis, Chalmers University of Technology, 2008.

[5] M. Själander, P. Larsson-Edefors, and M. Björk. A Flexible Datapath Interconnect for Embedded Applications. In *IEEE Computer Society Annual Symposium on VLSI*, pages 15–20, May 2007.

[6] A. M. Sllame and V. Drabek. An efficient list-based scheduling algorithm for high-level synthesis. In *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, pages 316–323, 2002.

[7] M. Thuresson, M. Själander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom. FlexCore: Utilizing Exposed Datapath Control for Efficient Computing. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pages 18–25, July 2007.

[8] M. Thuresson, M. Själander, L. Svensson, and P. Stenstrom. Flexible Instruction Decoding Based on Partitioned Look-Up Tables. Technical Report 2008-15, Department of Computer Science and Engineering, Division of Computer Engineering, Chalmers University of Technology, July 2008.