Compiler Scheduling for STA-Processors

Gordon Cichon, P. Robelly, H. Seidel, M. Bronzel, and Gerhard Fettweis
Vodafone Mobile Communications Chair
Technische Universität, D-01062 Dresden
cichon@ifn.et.tu-dresden.de

Abstract

This paper presents an adaptation of the list scheduling algorithm to generate code for processors of the Synchronous Transfer Architecture (STA) by applying techniques known from RISC and TTA. The proposed scheduling approach is based on informed, deterministic algorithms that can be implemented run-time efficiently. Although the presented compiler prototype does not generate optimized code, it provides a proof-of-concept of the feasibility of the proposed compiler architecture.

1 Introduction

Digital signal processing algorithms have a well predictable control- and data-flow. The hard real-time constraints does not allow for the flexibility that would justify to expend hardware resources for dynamic scheduling. Dynamic scheduling is particularly useful for coping with unpredictable events, such as cache-misses and branch-misprediction. However, these conditions do not occur very often in signal processing applications. Furthermore, avoiding all unnecessary hardware overhead is of paramount importance in battery powered operation.

VLIW processors are known to be compiler-friendly. However, they require large register files and complete bypass networks [9]. In particular, the implementation effort for register files is linearly dependent on both, the number of registers and the number of read / write ports for the register file. Increasing parallelism requires to increase the number of registers as well as the number of ports for concurrent accesses of different functional units. This creates a bottleneck at the register file.

Transport triggered architecture (TTA) [4] decouples the number of ports at the register file from the number of parallel functional units by providing explicit operations for reads from and writes to the register files and by specifying data transfers in the bypass network as *move* operations.

STA architecture is similar to TTA, however, it requires some adaption of compiler technology. This paper presents an adaptation of list scheduling that is suitable to generate code for STA processors. The resulting algorithm is similar to the TTA scheduling algorithm and uses some techniques from Tomasolu's hardware scheduling algorithm [5]. However, the proposed algorithm is part of the compiler and does not cause the hardware overhead associated with dynamic scheduling that is found in super-scalar processors.

Detailed background information on compiler construction is provided in standard literature [1, 8]. Additional information about parallelizing compiler technology can be obtained from [2, 14]. Before the paper becomes specific about compiler technology, a brief overview of the targeted STA architecture is given.

2 Synchronous Transfer Architecture (STA)

The Synchronous Transfer Architecture (STA) [3] is a novel architecture that enables the design of high-performance, low-power digital signal processors (DSP). STA processors aim to shift the effort for the execution of parallel operations from hardware to software.

STA is a simplification of TTA which removes additional bottlenecks and makes the overall architecture more efficient. TTA requires local queues for collecting operands and a controller that determines when exactly an operation is to be started. In the predictable execution environment of DSPs, the STA approach triggers the execution of operations explicitly by supplying control signals from the instruction word.

Figure 1 shows the architectural template of STA. The processor is split into an arbitrary number of functional units, each with arbitrary input and output ports. To facilitate synthesis and timing analysis, it is required that all output ports be buffered. Each input port is connected to an arbitrary number of output ports, as shown in Figure 2. For each computational resource, a segment of the STA instruction word contains the control signals (opcode) for the functional unit and the multiplexer controls the sources of all input ports and associated immediate fields.

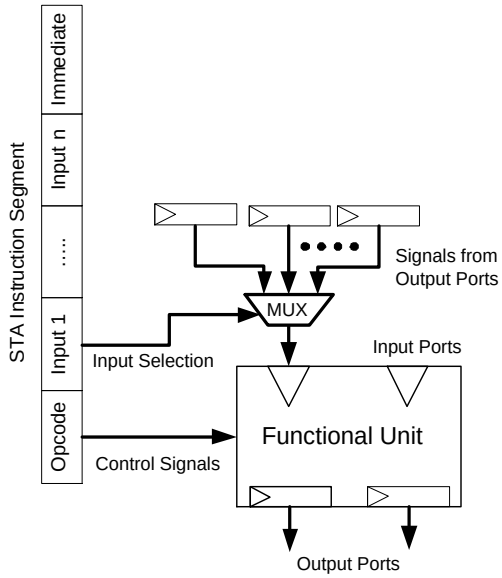

Figure 1. STA: Modules

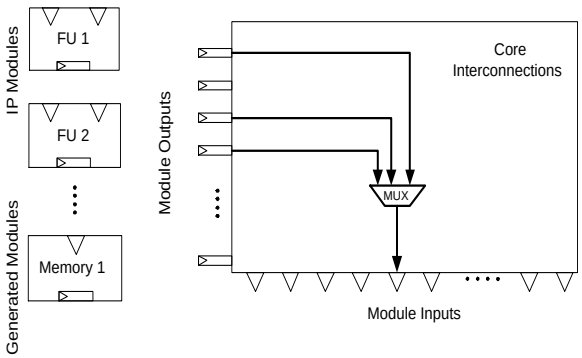

Figure 2. STA: Interconnection Network

Figure 2 shows all input multiplexers together forming a switching matrix between the output and input ports. This system constitutes a synchronous data flow network. The switching matrix may implement arbitrary connections depending on the application, performance, and power-saving requirements.

3 Related Work

In order to provide a motivation for a novel compiler backend architecture, this section presents an overview of the related work in DSP compiler technology. It will be shown, that the technology of classical DSP compilers is not applicable for STA.

A compiler backend consists of three phases: instruction selection, scheduling, and register allocation. ISAs which allow arbitrary instructions to be used in conjunction with arbitrary registers are called orthogonal. The orthogonal ISA allows a compiler to perform the phases separately while generating near-optimal code, e.g. GCC [11].

The instruction set architecture (ISA) of classical DSP architectures are not orthogonal, i.e., they allow only specific combinations of instructions and registers to be used in combination. Therefore, instruction selection cannot be performed without information about register allocation. The compiler phases cannot be performed independently of each other. To tackle this problem, some compiler frameworks propose an integrated approach which performs those three phases simultaneously. It is called phase-coupling [7].

A phase-coupled compiler backend is a search problem in a high-dimensional search space. It is an NP-hard problem. All proposed solutions rely on heuristic search algorithms, originating from the field of artificial intelligence (AI). The idea of those approaches is to create a great number of arbitrary sub-optimal solutions and then transform them into a more optimized solution based on specific heuristics.

While the phase coupled compiler backend is an NP-hard problem, each individual phase of the compiler backend is well understood, and informed heuristics help solve each one isolatedly in a near-optimal way. Therefore, this paper proposes to keep the compiler phases separated.

4 STA Compiler Backend Architecture

Phase-decoupling in the compiler backend requires an orthogonal ISA. It is achieved through separation of processor resources into two types of modules: These functional units perform computations and are not allowed to contain intrinsic states other than pipelining. Register files and memories store states of computations but are note allowed to perform any computations. Canonical STA denotes the class of architectures that realize this separation.

Consequently, the classical RISC compiler approach applies to canonical STA. Due to the application of informed algorithms, the algorithms can be made very efficient.

Figure 3 shows the proposed adaptation of the RISC approach to STA. The proposed compilation process consists of the following phases:

- The compiler backend starts with a generic medium-level intermediate representation (MIR) of the target program. We call this representation linear or sequential instruction list (insn).
- In a first step, this generic MIR is transformed to a target specific MIR by mapping generic instructions to the operations available on the target architecture. This state-of-the-art procedure is called instruction-selection and is described in detail in [1, 8].

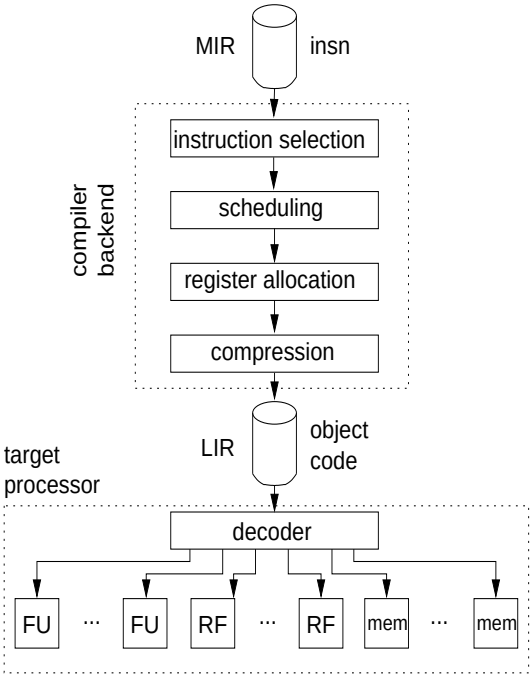

Figure 3. Compiler Backend Overview

- The scheduling phase determines the point of time in which each specific operation is executed. The result of this phase is a low-level intermediate representation (LIR). This LIR contains a time-table for each processor resource: the operations of each functional unit, the connections used to produce input operands, and the read/write addresses for the register files.
- After scheduling, a register allocation phase takes place. It maps generic virtual registers, so-called locations, to hardware registers. This phase uses standard algorithms [1, 8] again.

  Since the number of hardware registers is limited, the register allocator may fail to assign registers for some specific time-table. In this case, additional spill code has to be inserted to store some values in main memory. This technique is applied in the GNU Compiler Collection [11], and it is reported to work well.
- In order to save space in the instruction memory, the compiled code is subject to a tailorable compression phase. The output of this phase results in object code.

  Similar to the register allocation phase, the compression phase may fail to represent the LIR. In this case, additional constraints have to be introduced and the scheduling phase has to be restarted.

In the STA processor hardware, the following steps take place to execute the code:

- The STA processor fetches compressed code from its instruction memory.
- A hardware decoder decompresses this code. The result of decompression is a set of instruction segments that describe the operation of each functional unit and register file at each clock cycle, as shown in Fig. 1.

  In this terminology, super-scalar scheduling [5] can be regarded as one instance of a specific decompression technique. Since STA aims to provide minimal hardware overhead to implement this decompression phase, the TVLIW encoding [13] represents a more appropriate compression method, a 2D run-length encoding approach.
- Finally, the operations are executed on a set of functional units and interconnection resources.

5 Scheduling

This section presents an instruction scheduling approach similar to the Tomasulo scheduling for super-scalar processors presented in [5]. Both, STA scheduling and Tomasulo algorithm, perform list scheduling of primitive operations and have to handle additional data transports to and from register files. Since hardware scheduling is timing critical and very expensive, it has to be implemented with rather limited resources. For the proposed software approach, also more complex scheduling techniques are applicable, e.g. as presented in [10].

The STA scheduler performs the following steps:

- priority sorting
- launch candidate selection
- operation launch

As a decision aid for the following steps, a priority sorting of the operations within a basic block is performed, as described for list scheduling in [8].

STA is designed to have little overhead for instruction processing. Hence, there is no hardware overhead in the processor that is dedicated to access the register files automatically, as in VLIW or super-scalar processors. Consequently, in addition to traditional list scheduling, the STA scheduler needs to create these register file access operations associated with the data processing operations explicitly .

As in list scheduling, the scheduler maintains a timetable of already scheduled operations. In each step, it determines a set of launch candidates. A launch candidate is composed of a data processing operation and the earliest clock tick at which the operation may be started according to data dependencies and available resources. E.g., an instruction `add`

`R1, R2` would be a launch candidate for a VLIW scheduler if `R1` and `R2` have been computed, and the adder resource is available.

The STA scheduler has additional prerequisites: A data processing operation can only be launched if the operands and the results can be routed. This routing can take place directly or indirectly.

5.1 Operand Routing

Direct routing can take place if there is a connection from the source port computing a value and the destination port using a value. In addition, this routing method is only possible if the following operation uses the value in the clock cycle directly after it has been computed.

Indirect routing means that the value is stored temporarily inside a register file. This routing method requires a connection from the source port to a register file with an available write resource, and from an available read resource of a register file to the destination port.

By using this routing method, the following operation can be delayed arbitrarily. This delay guarantees generality, i.e., the scheduler is able to process any sequential instruction list. In the worst case, each operation is performed sequentially one after another.

In order to support canonical STA architectures with a reduced interconnection network, the case has to be dealt with in which the processor does not contain a direct connection from a specific output to a specific input port. These omissions are performed by hardware architects in order to reduce the complexity of the interconnection network, resulting in higher clock speeds and lower power consumption. If direct routing cannot be performed because a direct connection is missing, indirect routing has to be performed instead.

For this reason, canonical STA calls for a direct connection from every output port to a register file, through which indirect routing may be performed. Therefore, generality can be guaranteed even in the case of reduced interconnection networks. As indirect routing slows down the computation by the latencies of the register file accesses, a tradeoff between the cycle count and the interconnection network density is possible.

If multiple register files are available for a given data type, the compiler scheduler must be able to compute a route from the register file containing the particular value to the input port of a functional unit requiring that value. This routing generates additional operations copying the value though several register files to the destination functional unit.

This approach allows for a trade-off between performance and number of ports at the register file, and the amount of connectivity in the processor. [6] has shown that reducing the interconnectivity and number of ports at the register file does not significantly impact computational performance.

5.2 Operation Launch

After evaluation of the requirements, a specific launch candidate has to be selected for launch. A straightforward approach used in the Tomasulo algorithm is to select an arbitrary operation which has the highest priority.

Tomasulo has to select and launch a set of operations on a clock-by-clock basis. This way, it can react efficiently on operations with a variable latency. However, it cannot undo a scheduling decision once an operation is launched.

Since STA scheduling is performed offline, the STA scheduler can launch groups of operations at differing points of time. In some situations, it turns out to be useful being able to perform backtracking on scheduling decisions.

After the launch candidate selection, the operation is launched by inserting the following information into the time-table: The launched operation, register file read operations for operands and register file write operations for results, as well as switching information for the interconnection network. This way, the instruction is launched, and the results are available after the time periods specified in the STA machine description.

6 Reducing Register Accesses

In Tomasulo scheduling, the register file serves as a central communication hub for data values that flow from one functional unit to another. While read accesses to the register file might be short cut by the bypass network for speed reasons, all computation results are constantly written back to the register file.

This is cause by the fact that Tomasulo does not know in advance whether any result is reused or not. Additionally, the processor has to be able to provide a state snapshot at any time when an interrupt occurs.

On the other side, the offline STA compiler can determine whether a specific value will be reused. Therefore, it can remove any unnecessary register file write accesses and save valuable register file access resources: If a value is transported exclusively by direct routing, the register file write access can be omitted. As no hardware register has to be allocated in the register file, this approach free additional valuable resources in the register files and facilitates the construction of register files with a small number of registers.

With the presented list scheduler approach, it is impossible to determine which of the output values require direct or indirect routing. This is determined by the launch time of the dependent operations, which are not launch candidates

yet because they are waiting on the operation being considered.

As a simple solution, the author proposes the following two-phase approach: First, to generate all potential write operations to the register file, and – after having scheduled all dependent instructions – to remove unnecessary register writes again. This introduces some sort of phase-coupling between the sub-phases of the scheduler and is not implemented yet.

It remains to be investigates whether better algorithms for that problem exist. Potentially, launch candidate selection should try to build chunks of directly routed operation chains that are contiguous and as long as possible.

The presented compiler prototype is optimized for SIMD architectures which achieve their degree of parallelism by replicating a simple processing element. As a working assumption for compiler development, it may contain a single item of each individual resource. Therefore, no features are currently implemented in the presented prototype that support scheduling for a multiplicity of identical functional units or register files.

7 Register Allocation

After the scheduling is completed, register allocation takes place. The STA approach allows the use of standard RISC register allocation. It assumes a sufficient number of registers in order to work efficiently. If the register allocator runs out of registers, it has to insert spill code and restart the scheduling.

In some cases, there will not be enough hardware registers to accommodate all necessary values. In this case, some of them will have to be stored in an external memory. The additional operations required to store and load values from such an external memory are called spills. After inserting spill operations, the scheduling phase has to be restarted with the resulting new sequential instruction list.

After register allocation, the resulting code is compressed. It may not be possible to represent an arbitrary piece of code with a specific compression technique. If compression fails, scheduling may be restarted introducing additional constraints. Additionally, launch candidate selection can help fulfill compression requirements a priori. In the case of TVLIW compression, the scheduler observes a given maximum of operations to select per cycle for non-loop code.

In the presented version of the compiler, the register allocator does not perform well. This is due to that fact that it does not perform proper spill insertion yet. Additionally, it seems to contain software bugs. Therefore, a complex algorithms, currently fail in the register allocation phase.

8 Results

The compiler prototype is capable of compiling two programs: *addv*, which adds to vectors of numbers, and *fir*, which computes a real-valued FIR filter.

8.1 addv

This is the Matlab™ program of addv:

```
function c = addv (a, b, n)
  for i = 1:n
    c(i) = a(i) + b(i);
  end
end
```

The program “addv” is successfully vectorized by the compiler, and compiled to 53 clock cycles of assembly code. The inner loop of this program takes 11 clock cycles to execute. This is caused, first of all, by the unoptimized compiler that does not perform common subexpression elimination. Secondly, the processor architecture contains only a single scalar ALU for performing address computation of three address pointers and maintaining a loop counter. Overall, the program takes 352 clock cycles to add two vectors of 256 elements.

8.2 fir

This is the Matlab™ program of the FIR filter:

```
function y = fir (c, x, m, n)
  for i = 1:n
    y(i) = 0.0;
  end
  for i = 1:n-m
    for k = 1:m
      y(i+k) = c(k) * x(i) + y(i+k);
    end
  end
end
```

The “fir” program is also successfully vectorized by the compiler, and compiled to 123 clock cycles of assembly code. The innermost loop over ’k’ takes 25 clock cycles to execute one instance of the loop body. In order to perform a real-valued FIR filter of order 32 on 256 data samples, the program takes 27.167 clock cycles.

9 Discussion

The compiler’s sequential instruction lists are still suboptimal and perform many redundant computations. However, apart from the unoptimized instruction lists and unnecessary register file accesses, the presented scheduler produces an optimal scheduling for the given list of operations.

The proposed approach is an extension to proven approaches for scheduling general-purpose RISC and VLIW processors. These algorithms map complex algorithms efficiently on simple primitive operations. Furthermore, the proposed scheduling approach for STA is based on informed, deterministic algorithms that can be implemented run-time efficiently.

In contrast to VLIW, STA allows fast-path computations with neither register file transactions, nor having to provide a full bypass interconnection network as a crossbar switch. In addition, it allows to reduce the number of read and write ports of the register files.

In addition to that, the STA approach allows the exploitation of the full parallelism available in the architecture without requiring application-tailored composite instructions, like [12]. Thus, our approach can be realized efficiently while maintaining phase decoupling for instruction-selection, scheduling, and register-allocation.

10 Future Work

A lot of algorithms of the remain to be implemented: Classical RISC optimization techniques (e.g. common subexpression elimination, loop code hoisting, and alike) and modulo scheduling for efficient loop execution without the overhead for unrolling. By applying these algorithms, the STA compiler should be able to match the performance of RISC and TTA compilers.

STA specific optimization strategies, like explained in the text might lead to additional performance improvements: Removal of unnecessary register access, composite operand routing for multiple register files, support of multiple functional units of the same functionality, etc. This should lead to more results of the STA compiler for more complex signal processing algorithms can be published.

Due to its efficient nature based on phase-decoupling and informed algorithms, the presented scheduler can be made very run-time efficient. In a mobile communications device, such a compiler backend can be used to provide compatibility to a standard ISA, e.g. ARM, x86, Java byte-codes, etc. The backend may either run on a compilation server in a network, or it might run directly on the stream processor of the target STA device.

This would allow additional features, like recompiling the code when the number and speed of the functional units change. Thus, a generic device may contain an arbitrary number of functional units which are dynamically coupled to different processor cores. As soon as the configuration of a processor core changes, the code may be recompiled. Additionally, dynamic code morphing allows adaptive remeasuring of wave-pipelines and to compensate dynamic latency variations, e.g., due to supply-voltage (power-saving), temperature, etc.

Acknowledgments

This work has been sponsored in part by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) within SFB358–A6, and the DFG–Priority Program VIVA.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Redding, MA, 1985.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, Burlington, MA, 2001.

[3] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis. Synchronous transfer architecture (STA). In *Proc. of Fourth International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04)*, pages 126–130, Samos, Greece, July 2004.

[4] H. Corporaal. *Microprocessor Architecture from VLIW to TTA*. John Wiley & Sons, 1997.

[5] J. Hennessy and D. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition edition, 1996.

[6] J. Hoogerbrugge and H. Corporaal. Register file port requirements of transport triggered architectures. pages 191–195.

[7] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[8] S. Muchnik. *Advanced compiler design and Implementation*. Morgan Kaufmann Publishers, 1997.

[9] G. F. Pfister and et al. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*. IEEE Computer Society Press, August 1985.

[10] A. Römer. *Beiträge zur Optimierten Code-Erzeugung*. PhD thesis, Technische Universität Dresden, 2004. In German.

[11] R. Stallman. GNU compiler collection internals. `http://gcc.gnu.org/onlinedocs/gccint/`. A GNU Manual.

[12] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proc. DAC 2001*.

[13] M. Weiß and G. P. Fettweis. Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processors. In *3rd. Int. Workshop in Signal and Image Processing (IWSIP '96)*, pages 517–520, Jan. 1996.

[14] Zima and Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Redding, MA, 1990.