The *MOVE* project

Henk Corporaal

Delft University of Technology
The Netherlands

Abstract

The MOVE project of the Department of Electrical Engineering of the TU Delft aims at designing and building high-performance processors by means of a new class of architectures called *transport-triggered architectures*. Transport-triggered architectures comprise a new class of architectures that are programmed by specifying data-transports instead of operations; they are particulary suited for application specific purposes. In order to exploit the power of these architectures new code scheduling techniques are required.

1 Introduction

This document gives a short overview of the *MOVE* project at the Delft University of Technology. The *MOVE* project concerns the automation of application-specific processor system design providing optimal cost-performance tradeoffs. The project is based on three tenets: RISC, instruction-level parallelism, and massive message-passing parallelism. RISC principles tell us to carefully quantify cost and performance of extra hardware functionality, taking also into account the design turnaround time, which is influenced by the design complexity. Instruction-level parallelism allows for high-performance computing nodes, whose functionality and performance (both processing and i/o performance) is adapted to specific applications or application domains. If the application permits and requires the power of massive parallelism, the nodes should be easily configurable to allow massive MIMD processing. The nodes have local memory, and communicate through the exchange of messages. Although this organization exploits the locality of computing available in many massive parallel applications, it does not preclude the implementation of (logical) shared memory.

The *MOVE* project finds its direct roots in the SCARCE project [1, 2, 3, 4, 5, 6] and the design of communication processors for distributed computer systems [7, 8, 9]. In both projects flexible processor frameworks (for data processing and communication respectively) have been designed and implemented using the ASA Silicon compiler and NELSIS[1] VLSI design tools. The main goal of the project is to provide an efficient way for designing and building high-performance application-specific processor systems. These systems consist of one or more high-performance computational nodes. Within the project we distinguish two main threads: 1) a framework for the design of these computational nodes according to the application needs; 2) a framework for the design of distributed memory systems based on these nodes.

MOVE architectures are based on the transport-triggering concept (the *MOVE* concept for short). This concept and its main advantages are discussed next. Section 4 gives a high level description of the MOVE framework, on which the current research is focussed. A project status overview concludes this short overview.

[1]ASA is a trademark of SAGANTEC, NELSIS has been developed at the Delft University of Technology, The Netherlands.

2 The MOVE concept

Transport-triggered architectures (TTAs) [10] form a new class of architectures that differ from traditional operation-triggered architectures (OTAs) in the way they are programmed. Traditional OTAs are programmed by specifying operations. Each specified operation results in a few data-transports on the data path, e.g., an addition results in two data-transports of the operands to the ALU and one data-transport of the result from the ALU.

TTAs are programmed by specifying the data-transports, or *moves*, directly. All these moves are between internal registers. A TTA consists of a set of function units (FUs) and register units (RUs) connected by a fully or partially connected interconnection network. FUs provide functionality, RUs provide general purpose registers.

The register set is divided into general purpose, operand (O), trigger (T), and result (R) registers. General purpose registers (GPRs) are located in RUs and have the same function as the GPRs of OTAs. Operand registers are part of an FU; they are used to deliver operands. Similar to an operand register, a trigger register is also used to deliver an operand, however a move to a trigger register is also the signal for the FU that all operands have been delivered and the operation can start. After triggering, the operands are sent to a pipeline that does the operation. After the operation has completed the result is placed in the result register of the pipeline. With a move from a result register the result can be used for other operations or it can be stored in a GPR. Reading a result register is also the signal that the operation has ended and the result register can be used for the result of another operation on the same FU. As an example, the following move code is the translation of $m[p] = m[q] + a$, where p, q, and a are stored in the GPRs r_q, r_p, and r_a:

$r_q \to$ *ld-T*		// GPR $\to$ trigger load
...		// load in progress
...		// load in progress
ld-R $\to$ *add-O*	$r_a \to$ *add-T*	// result load $\to$ operand addition, GPR $\to$ trigger addition
...		// addition in progress
$r_q \to$ *st-O*	*add-R* $\to$ *st-T*	// GPR $\to$ operand store, result addition $\to$ trigger store

This example shows that TTAs allow multiple moves per cycle; similar to very long instruction word (VLIW) computers, which allow multiple operations per cycle.

Besides doing operations, the architecture should also provide immediate operands, control flow, and conditional execution. Short immediates are stored in the source specifier of a move. Longer immediates are provided by reading parts of the instruction stream; the instruction register is visible in the architecture. When it is read, the immediate is taken from the instruction stream.

Control flow is done by simply writing a target address to the program counter. Reading the program counter can be used to save it for function calls. For the purpose of relative jumps, a special register is provided. Writing a displacement to this register causes a relative jump.

Conditional execution is done by means of guards. Compare FUs produce boolean results; each move can be guarded by a boolean expression of these booleans. Only when a guard expression evaluates true the guarded move takes place. Guards allow conditional execution of both data and flow control operations. Guards also incorporate multi-way branching.

3 Advantages of transport-triggered architectures

Advantages of TTAs can be split into implementation advantages and new software compilation optimization possibilities. The most important implementation advantages are:

- Ideal for employing superpipelining at both operation and data transport level. FU pipelines can be stretched to make shorter cycle times possible. The only lower bound on the clock cycle is register-register transfer time across the network. The network can be superpipelined itself; in that case the achievable clock cycle time reduces to the register-register transfer time within one network cluster; this time can be very short.

- Ideal for employing functional parallelism at both operation and data transport level. FUs and transport capacity can be added to increase parallelism. Unlike OTAs, CPUs using a TTA do not need three busses and three register ports for each operation per instruction; e.g. twelve ports for a four operation per instruction VLIW. Three busses and ports per operation is a worst case assumption, since many operations do not need them. For example, some operations need a single source operand, or do not produce a result. Also a lot of results are directly bypassed to the next FU, without needing to be stored in a GPR. Further the compiler can perform many optimizations specific for TTAs which reduce the needed transport capacity, and therefore the network requirements (like number of transport busses and network connectivity), even further. As a consequence, the required VLSI area is reduced and the cycle time is further improved.

- Ideal for the design of application specific processors (ASPs). FU parameters (like number of FUs, supported operations, latencies, throughput and pipelining degree) and interconnection network parameters (like topology, number of busses and pipelining degree) can be set according to the needs of an application domain.

- Ideal for automatic generation. The CPU has a very simple design; this is a consequence of having independent FUs and of the reduced data transport requirements. Furthermore, the network has less complexity; e.g. no complex bypassing hardware is needed, bypassing is done in software. Due to its simplicity, it becomes possible to use a silicon compiler for automatic layout generation. The inputs to the silicon compiler are a template description, VLSI building blocks (e.g. FUs), and values for the architecture parameters. The output is a CPU silicon layout ready for fabrication of the CPU.

- Perfect suitable for incorporating register mapped interprocessor communication support; the concept of register mapped functionality is inherently supported by a TTA. Register mapped communication enables very short latencies and a high communication bandwidth. This opens the possibility of integrating systolic communication within a general purpose processor framework.

Besides the traditional compiler optimizations, the TTAs offer the following unique optimizations:

- More scheduling freedom. TTAs divide operations into smaller data transport components. This makes parallelism more fine-grained, and the resulting code schedules are more efficient and achieve higher CPU execution performance.

- Software bypassing. A result of an operation can be used for another operation in the same cycle if software bypassing is applied. Bypassing means getting the value from the FU that produced it instead of the RU where it will be stored. Bypassing reduces the delay between (true) dependent operations.

Unlike OTAs, TTAs do not need special (associative) hardware to do bypassing, but all bypassing can be done in software under control of the compiler. For example, when an operand move *r0* $\rightarrow$ *add-O* uses the result of a result move *add-T* $\rightarrow$ *r0* in the same cycle as the result move, then the operand move needs to be changed into *add-T* $\rightarrow$ *add-O* so that the result is taken from the FU instead of the RU.

- Operand sharing. When two successive operations on the same FU are guaranteed to have the same value in the operand register, one operand move can be saved. We call this operand sharing, and it can be viewed as a special form of common subexpression elimination. When one operand move is shared among all iterations of a loop, then the operand move is loop invariant and can be placed before the loop.

- Dead result move elimination. When all uses of a result are used via software bypassing, then the result move can be eliminated. This saves one move and the usage of one GPR. Since most results are only used once or twice (e.g. temporaries), dead result move elimination occurs frequently.

- Reduced GPR demand. TTAs need fewer GPRs since (1) results are directly bypassed from FU to FU, and (2) operations can stay longer in a pipeline than is needed to do the operations, this makes GPR lifetimes shorter.

Initial experiments [11] have shown that the results of these new optimizations are very promising. The experiments show that TTAs perform 20–50% better than OTAs with similar hardware for scientific code using adapted software pipelining techniques.

In [12] TTAs are analysed for general purpose applications, using a basic block scheduling compiler. It is shown that a operation based architectures require up to 30% more transport capacity, however in order to efficiently exploit this capacity by a TTA we need a scheduling scope far beyond basic block boundaries. The latter is currently under development. First results will be publiced in [13]. It is shown that scheduling beyond basic blocks gives a 40% performance improvement.

For a more extensive description and evaluation of the *MOVE* concept see e.g. [10, 14, 11, 15].

4 MOVE framework

The ultimate goal of the MOVE project is to establish a framework for the semi-automatic generation of application (domain) specific MOVE processors. The structure of this framework is depicted in figure 1. The framework consists of three main components:

Software framework This framework contains a tool for trace driven application analysis, and generators for front end and back end compiler, assembler, architecture simulator and debugger.

Hardware framework The main part of this framework is a VLSI (CMOS) processor generator, build on top of a library of generic VLSI components. Further it should be able to answer questions about needed VLSI area, critical circuit time, and power consumption.

Optimizer This tool interacts with the user and the other components. It coordinates the synthesis process. Iteratively the architecture parameter values are determined based on user constraints, and on information from the former components.

Short overviews of these components are presented in the next sections.

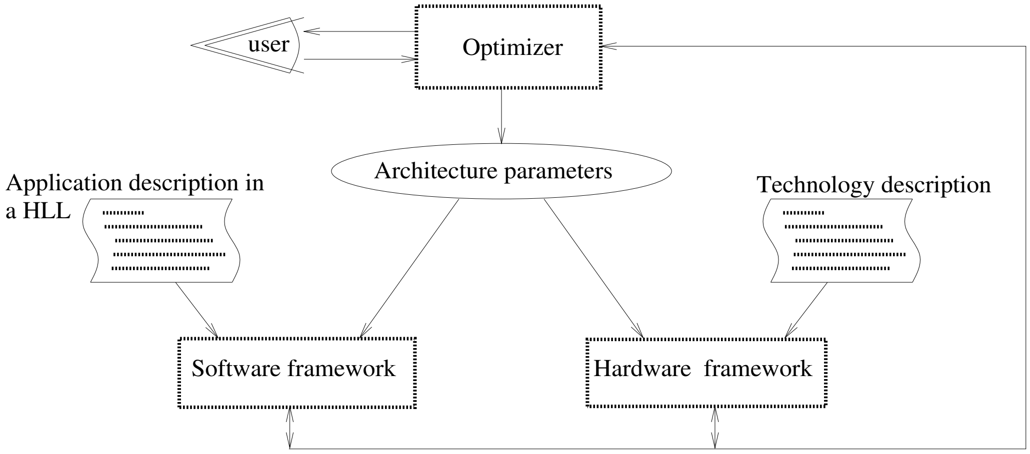

Figure 1: MOVE framework.

4.1 Software framework

Main parts of the software framework are a trace analysis tool and a compiler generator. In fact these two parts share much code. Using these two parts we are able to schedule (producing instruction level parallel code) both traces and sequential code. Figure 2 shows a block diagram showing the main parts of this framework.

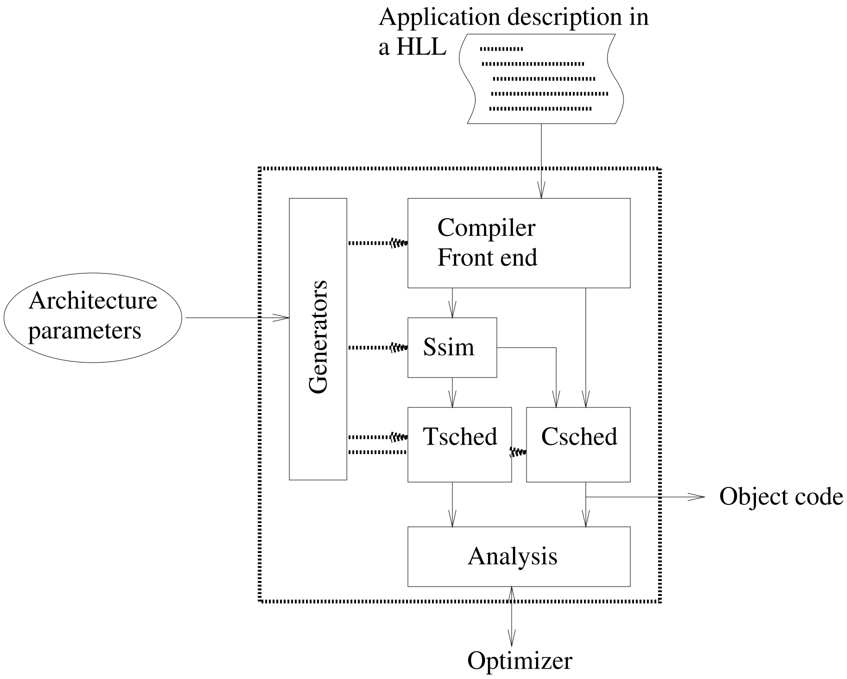

Figure 2: Software framework overview.

The software framework accepts an application coded in C or C++ [2], and architecture parameter values, like available functionality (type of operations), and latency of operations, as input. Based on

[2]In the future Fortran will also be supported, using the GNU fortran compiler as front end.

the architecture inputs it generates a compiler front end (based on GNU-CC), architecture simulators, back end code schedulers, and debuggers. The figure does not show all components. The front end produces sequential move code. Using one or more data sets (not shown in the figure), Ssim, which is a sequential simulator, generates traces for the trace scheduler (Tsched) and profile information for the code scheduler (Csched). The schedulers compact the trace or code, according to specified architecture constraints.

The analysis part is able to produce all kinds of statistics based on the scheduled traces or object code, and communicates this information with the optimizer. Among the statistics are, amount of parallelism, distribution of parallelism, schedule length, and resource usage. Based on these statistics, the optimizer can check if user constraints are still met, and make decisions how to further optimize the architecture.

4.2 Hardware framework

The hardware framework is responsible for the realization of a application specific, single chip MOVE processor. In figure 3 the hardware framework is drawn as a box which accepts architecture parameter values, and a technology description as input and produces a VLSI layout (e.g. in CIF format) of the generated processor as output. Besides that, it interfaces directly, via the layour analyser, with the

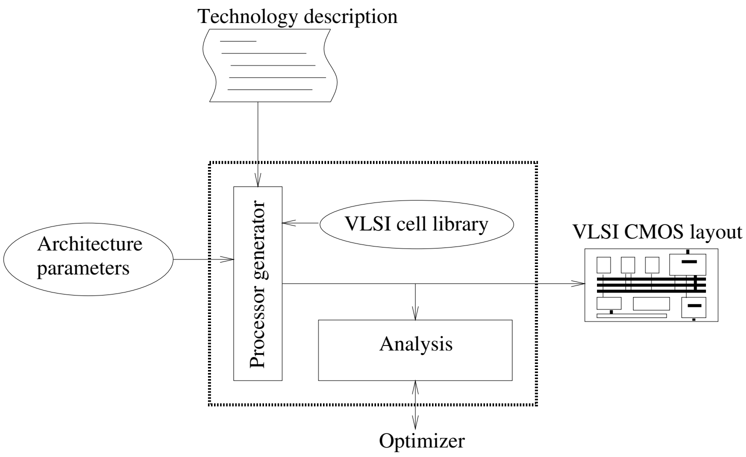

Figure 3: Hardware framework overview.

optimizer, to exchange information about area, speed and power consumption requirements.

The hardware framework uses a VLSI library of predesigned cells (at symbolic layout level) to generate a processor. These cells can either be standard cells used by standard cell routing tools, or user designed basic cells used for generating regular structures (like memories).

4.3 Optimizer

The user may specify constraints on maximal chip area, maximal power dissipation, minimal performance (of whole application, or of e.g. a critical loop within the application), and on maximal pin count. Based on these constraints the optimizer steers the software and hardware frameworks to produce valid architecture solutions. Although the MOVE framework severely restricts the allowable architecture solutions offered for a specific application, in practise the specified user constraints still leave a very large search space of allowable solutions.

The optimizer finds its way through this search space by iteratively trying different architecture solutions, and letting compiler and hardware framework produce relevant information about these solutions.

Ideally the user must be able to account costs to area, power dissipation, performance and pin count in such a way, that the optimizer finds its way in the architecture search space indepent from further user input. This process can be very time consuming however, because many solutions must be processed by the software and hardware framework. The search time may be reduced by:

1. *Using user knowledge*. In practize we expect the user to be knowledgable about the MOVE hardware and software frameworks, and is able to shorten the search trajectory to a satisfactory architecture solution, by interactively giving further hints (regarding changes to architecture parameters) to the optimizer. The user may be guided in this process by statistical information on solutions produced so far.

2. *Using builtin knowledge*. For certain architecture parameters, the software and hardware frameworks may be able to predict the influence of changing their values, without actually trying these changed values. These predictions can be used by the optimizer to cut off large parts of the search space.

3. *Linear algorithms*. In general, when iteratively a solution has to be found, it is essential keep the processing time per iteration minimal. Therefore software and hardware framework (in particular the code and processor layout generators) should aim at using algorithms which depend linear on the size of their input domain (like number of operations to be scheduled, or number of VLSI cells to position). In many cases however, linear algorithms are only possible in conjunction with smart heuristics.

5 Project status and research

Currently the following major research and development activities take place in the *MOVE* project.

1. Design of a hardware framework for the semi-automatic generation of a VLSI implementation of data processor and MIMD communication processor.

2. Optimizing the compiler and simulator generator. The compiler frontend is based on the GNU C/C++ compiler. The backend contains a *MOVE* specific code scheduler, in order to exploit the available instruction level parallelism. Both compiler and simulator are retargetable to application specific *MOVE* processors.

3. Further research into efficient scheduling for extended basic blocks and of loops (software pipelining) [14, 16, 17].

4. Finishing the realization of a test chip, which implements a *MOVE* architecture with limited functionality. This chip has been implemented on a Sea of Gates pattern in 2.0μ CMOS (gate length 1.6μ), designed at the Delft University of Technology and fabricated by Philips. Metalization will be done at DIMES in Delft (see [18, 19]).

5. Development of an application analysis and architecture synthesis tool. This tool currently accepts C code and generates among others information about required functionality and achievable performance speedups.

6. Research into efficient MIMD communication support [7, 8, 9, 20].

TTAs are also used at the Electronal Engineering faculty at TUD, for the design of a customized navigation receiver (see [21]), and in ADOSS project (a cooperation with the Physics Department of the TUD), which aims at the automatic design of massive parallel simulation systems.

The *MOVE* project team consists of 1 staff member, 3 PhD students, 4 Research Assistants and several Master Degree students.

References

[1] J.M. Mulder, R.J. Portier, A. Srivastava, and R. in 't Velt. An efficient emulation mechanism for hardwired pipelined processors. In *Proceedings of the 21nd workshop on microprogramming and microarchitectures*, November 1988.

[2] J.M. Mulder et al. A framework for application-specific architecture design. In *Proceedings of the 16th International Symphosium on Computer Architecture*, May 1989.

[3] J.M. Mulder and R.J. Portier. Cost-effective design of application-specific VLIW processors using the SCARCE framework. In *Proceedings of the 22nd workshop on microprogramming and microarchitectures*, August 1989.

[4] J.M. Mulder and P. Stravers. A Flexible VLSI Core for an Adaptable Architecture. In *Proceedings of the 22nd workshop on micrprogramming and microarchitectures*, August 1989.

[5] J.M. Mulder, R. Portier, and A. Srivastava. A framework for high-speed controller design. In *Proceedings of the 23th workshop on microprogramming and microarchitectures*, November 1990.

[6] J.M. Mulder. Application-Specific Controllers as Regular Structures in a Silicon Compiler. In *Proceedings of the Silicon Design Conference*, November 1990.

[7] H. Corporaal and J.G.E. Olk. A Scalable Communication Processor Design Supporting Systolic Communication. In *EDMCC-2 Conference proceedings*, München, April 1991.

[8] H. Corporaal and J.G.E. Olk. Design and Evaluation of Communication Processors supporting Message Passing in Distributed Memory Systems. In *DMCC-6 conference proceedings*, Portland, April 1991.

[9] H. Corporaal and J.G.E. Olk. A Scalable VLSI MIMD Routing Cell. In *DMCC-6 conference proceedings*, Portland, April 1991.

[10] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[11] Jan Hoogerbrugge and Henk Corporaal. Comparing software pipelining for an operation-triggered and a transport-triggered architecture. In *Lecture Notes in Computer Science 641, Compiler Construction*, pages 219–228. Springer-Verlag, 1992.

[12] Henk Corporaal. Evaluating Transport Triggered Architectures for scalar applications. *Microprocessing and Microprogramming*, 38:45–52, September 1993.

[13] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Compiler Construction conference CC-94*, 1994.

[14] Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software pipelining for transport-triggered architectures. In *MICRO-24*, Albuquerque, November 1991.

[15] Henk Corporaal. Transport triggered architectures examined for general purpose applications. In *Sixth Workshop Computer Systems, Delft*, pages 55–71, January 1993.

[16] Andy Verberne and Henk Corporaal. Towards efficient code scheduling for transport triggered architectures. In *Fourth symposium on Computer Systems*, Amsterdam, October 1991. UvA.

[17] Robert Portier. GEPS, global enhanced pipeline scheduling. Technical Report submitted for publication, Delft University of Technology, Electr. Eng. Department, The Netherlands, August 1993.

[18] Henk Corporaal. Move32int, architecture and programmer's reference manual. Technical Report 1-68340-44-(1993)01, Delft University of Technology, Electr. Eng. Department, The Netherlands, 1993.

[19] Henk Corporaal and Paul van der Arend. Move32int, a sea of gates realization of a high performance transport triggered architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.

[20] Eddy Olk and Henk Corporaal. The osi model applied to mimd communication processor design. In *PARCO'93*, Grenoble, September 1993.

[21] Paul Stravers and Eric Aardoom. A procesor framework customized for navigation computations. *INTEGRATION, the VLSI journal*, 14(2):197–214, December 1992.