Comparing Software Pipelining for an Operation-Triggered and a Transport-Triggered Architecture

Jan Hoogerbrugge and Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems

Abstract. This paper reports the results of a comparison between a new class of architectures, called transport-triggered architectures, and traditional architectures, called operation-triggered architectures. It does this comparison by means of the MOVE-i860, which is a transport-triggered approximation of the i860. Several benchmarks are scheduled for the MOVE-i860 by a software pipelining scheduler. By scheduling according to three scheduling disciplines, we are able to measure the advantages of transport-triggered architectures.
The results of the experiments show that transport-triggered architectures perform better than operation-triggered architectures with similar hardware.

1 Introduction

Transport-triggered architectures comprise a new class of architectures that differ from traditional architectures in the way they are programmed. Like RISC architectures replace complex operations with complex addressing modes by simpler operations, transport-triggered architectures replace RISC operations by even simpler actions. Such actions are register-register data-transports. As a consequence the compiler can perform extra optimizations and has far better control over hardware usage.

With this paper we want to indicate the advantages of programming data-transports instead of operations. We do this by introducing a transport-triggered architecture called the MOVE-i860 which is based on Intel's i860. Like the i860, the MOVE-i860 is capable to do multiple operations per cycle. With a scheduler that applies software pipelining we did experiments to examine the gain of programming data-transports instead of operations.

The results of the experiments show the advantages of transport-triggered architectures. The extra scheduling freedom of transport-triggered architectures gives an improvement of about 25%, and two new optimizations give an improvement of 35%.

The remainder of this paper starts with an introduction of transport-triggered architectures. Section 3 introduces the MOVE-i860. Section 4 discusses our software pipelining algorithm. Section 5 describes the experiments we did with our scheduler and our MOVE-i860. Finally, Sect. 6 summarizes the results and gives indications for further research.

2 Transport-Triggered Architectures

This section will give an introduction to transport-triggered architectures. Its purpose is to give the reader an idea of the concept and to make the paper self-contained. More details about transport-triggered architectures can be found in [1,2].

The next subsection discusses the principles of transport-triggered architectures. Subsection 2.2 will describe the advantages of the concept.

2.1 Principles

Transport-triggered architectures (TTAs) from 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. Figure 1 shows the organization of a TTA. It 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 in Fig. 1), 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:

$r_q \rightarrow ld\text{-}T$	// GPR → trigger load
...	// load in progress
...	// load in progress
$ld\text{-}R \rightarrow add\text{-}O$; $r_a \rightarrow add\text{-}T$	// result load → operand addition, GPR → trigger addition
...	// addition in progress
$r_q \rightarrow st\text{-}O$; $add\text{-}R \rightarrow st\text{-}T$	// GPR → operand store, result addition → 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.

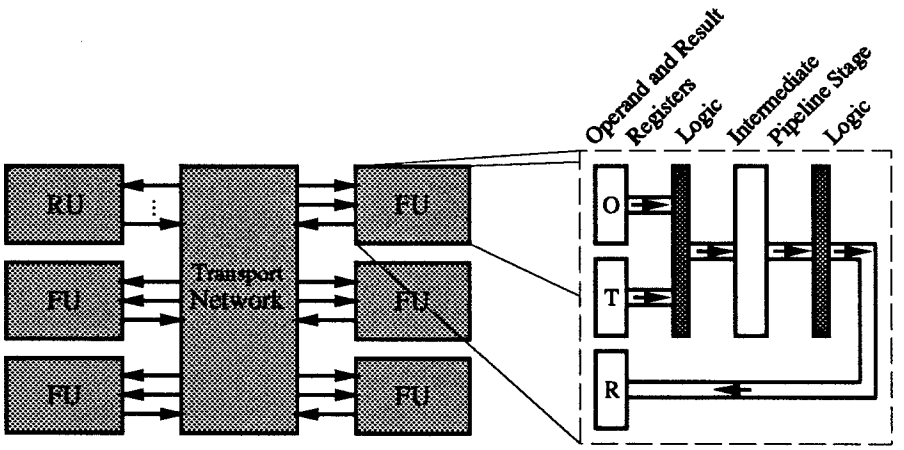

Fig. 1. The organization of a transport-triggered architecture.

2.2 Advantages

Advantages of the concept of transport-triggered architectures can be split into implementation advantages and new compilation optimizations. The most important advantages for VLSI implementation are:

1. Perfect for superpipelining. FU pipelines can be stretched to make shorter cycle times possible. The only lower bound on the clock cycle the is register-register transfer time across a move bus, which can be very short.
2. Perfect for functional parallelism. FUs and transport capacity can be added to increase parallelism. Unlike operation-triggered VLIWs, TTAs 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 (see [3]). The three busses/ports worst case assumption is not needed for TTAs. This reduces area and speeds up transport time.

3. Perfect for application specific processors (ASPs). FU parameters (operations, latencies, throughput) and interconnection network parameters (topology, number of busses) can be set according to the needs of an application domain.
4. Perfect for automatic generation. Due to its simplicity, it becomes possible to use a silicon compiler for automatic layout generation. The input of the silicon compiler are a template description, VLSI building blocks (e.g. FUs), and values for the architecture parameters.

Besides the traditional compiler optimizations, TTAs offer the following unique possibilities to produce high quality code:

1. More scheduling freedom. TTAs divide operations into smaller components. This makes parallelism more fine-grained, and thus better schedulable.
2. 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* → *ld-T* uses the result of a result move *add-R* → *r0* in the same cycle as the result move, then the operand move needs to be changed into *add-R* → *ld-T* so that the result is taken from the FU instead of the RU.
3. Operand sharing. When two successive operations on the same FU have guaranteed 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 body.
4. 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.
5. Reduced GPR demand. TTAs need less 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 operation, this makes GPR lifetimes shorter.

3 The Target Architecture

To make a comparison between OTAs and TTAs we use Intel's i860 as starting-point. The i860 can operate in VLIW mode; it issues 64-bit instructions on each clock cycle each containing one integer operation and one floating point operation. From the operation-triggered i860 we derive an TTA, called the MOVE-i860, with approximately the same hardware. We have made this derivation based on the internal organization and the instruction set of the i860 (see [4]).

Figure 2 shows the organization of the MOVE-i860. Like the i860 it has two RUs; the floating point RU is connected by three 64-bit wide busses with the two floating point FUs, and the integer RU is connected by three 32-bit wide busses with integer FUs. The load/store FU is connected to both RUs; where the connection to the floating point RU

is 128-bit wide. This makes it possible to transfer four single precision or two double precision floating point numbers in one cycle.

Similar to the i860, the MOVE-i860 has extra move busses between FU pairs that require more connectivity; we call these busses bypasses. The bypass between the integer unit and the load/store unit is because the i860 supports loads with indexed addressing mode. This requires two busses to transfer the two address components, one bypass to transfer the effective address to the load/store unit, and one bus to transfer the loaded data to its destination.

The two bypasses between the floating point FUs are there to model the (complicated) interconnection between the floating point FUs of the i860. Each result of the two FU can be fed into the operand and trigger register of itself or the other floating point FU. All bypasses of the MOVE-i860 have a multi-cast possibility. This means that a result that is placed on a bypass can be used by multiple operand and trigger registers that are connected to the bypass.

There is a separate 128-bit wide load/store bus between load/store unit and floating point FU because the i860 has the possibility to do a floating point operation and a load/store simultaneously. This requires four register ports on the floating point RU. The bus is 128-bit wide since the i860 supports 32-bit, 64-bit, and 128-bit loads and stores of floating point numbers.

All FU latencies of the MOVE-i860 are chosen equal to their corresponding latencies of the i860. Like the FUs of the i860, all FUs have a throughput of one operation per clock cycle. Table 1 shows the latencies of all MOVE-i860 FUs.

We want to emphasize that MOVE-i860 does not need to be an exact copy of the i860 to make a fair comparison between OTAs and TTAs.

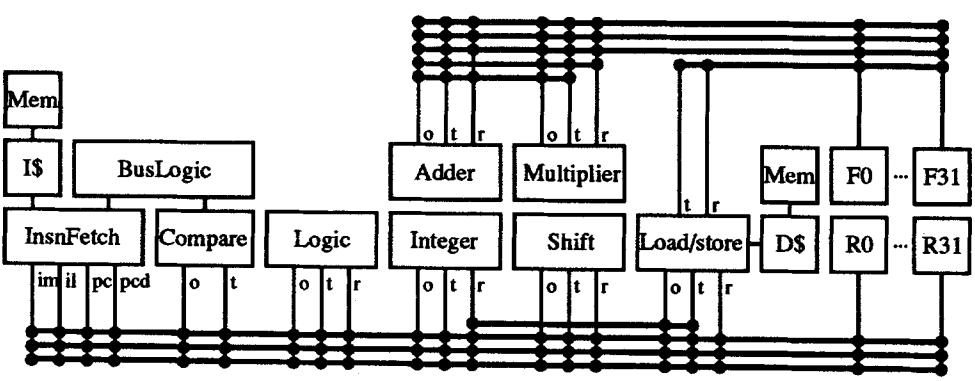

Fig. 2. The organization of MOVE-i860.

Table 1. The latencies of the MOVE-i860 FUs.

Operation	Latency
Integer operations	1
Loads (8, 16, 32, 64, 128 bits)	3
Single precision addition/subtraction	3
Single precision multiplication	3
Double precision addition/subtraction	3
Double precision multiplication	2
Number of branch delay slots	1

4 The Software Pipelining Algorithm

Software pipelining is a scheduling technique that transforms loops into semantically equivalent loops such that iterations of the original loop start before preceding iterations have finished. It processes different iterations of the original loop in different stages of the software pipeline in parallel. Figure 3 shows the principle of software pipelining; the original loop takes three cycles per iteration, the transformed loop executes one iteration per cycle. The software pipeline starts with a prologue of three instructions that fills the pipeline with iterations. Then a steady state of one instruction is repeated until enough iterations have been executed. The software pipeline ends with an epilogue that flushes the pipeline.

There are several software pipelining algorithms that differ in complexity and quality of the produced code. Our software pipelining algorithm is based on Lam's software pipelining algorithm (see [5,6]) with several modifications for TTAs (see [7]).

Lam's algorithm is can be viewed as a modification of the well-known list scheduling algorithm. The modification is needed to schedule cyclic data-dependency graphs (DDGs). Nodes of the DDG represent the moves that need to be scheduled, and the edges represent data-dependencies between the moves that must be satisfied to guarantee proper semantics. The DDG is cyclic when moves are dependent on moves of preceding iterations.

Besides the precedence constraints due to data-dependencies, the scheduler has to obey resource constraints. Resource constraints guarantee correct hardware usage. TTAs have the following resource constraints:

1. No more moves per cycle than move busses are available.
2. No more operations in a pipeline than the pipeline capacity permits.
3. Not more than one trigger and one result move per cycle on the same FU.
4. Trigger and result moves on the same FU in first-in first-out order.
5. No operand register overwritting by another operand move before it is used.

Precedence and resource constraints result in a lower bound on the size of the steady state of a software pipeline. The scheduler starts with trying to schedule (assigning a cycle) all moves with a steady state size equal to this lower bound. When this fails the scheduler increments the steady state size and tries again. This is repeated until a schedule is found.

Like list scheduling, Lam's algorithm is non-backtracking. This results in a polynomial time complexity and near-optimal schedules. Our scheduler has a backtrack option to find

optimal schedules at the cost of a exponential time complexity. This option is included to measure what can be reached with software pipelining when heuristics are perfect.

```
loop:   ld f0, r0++;
        fmul f1, f0, f2;
        st f1, r1++;        beqz r2--, loop;

                            ⇓

        ld f0, r0++;
        ld f0, r0++;        fmul f1, f0, f2;
loop:   ld f0, r0++;        fmul f1, f0, f2;  st f1, r1++;    beqz r2--, loop;
                            fmul f1, f0, f2;  st f1, r1++;
                                              st f1, r1++;
```

Fig. 3. The principle of software pipelining. Software pipelining reduces the steady state from three to one cycle.

5 Experiments

This section describes the experiments we did to make the comparison. It starts with introducing the used benchmarks. After that we introduce three scheduling disciplines. Subsection 5.3 reports the results of our experiments, and Subsection 5.4 analyzes the results.

5.1 Benchmarks

We have used eleven well-known benchmark loops for the comparison between OTAs and TTAs. Four of them are SAXPY, DAXPY, SDOT, and DDOT from the LINPACK library. The remaining loops are kernels 1, 3, 5, 7, 9, 11, and 12 from the Livermore Loops benchmark. All these loops were translated to three-address intermediate code by hand and fed to our scheduler.

5.2 Scheduling Disciplines

To make a comparison between OTAs and TTAs we define three scheduling disciplines, ranging from transport scheduling to operation scheduling:

1. FREE: nodes of the DDG represent moves, and the two TTA-unique optimizations (operand sharing and dead result move elimination) are enabled.
2. FIXED: nodes of the DDG represent operations, however, the two optimizations remain enabled.

3. OPER: nodes of the DDG represent operations, and the two optimizations are disabled.

The effect of the increased scheduling freedom of TTAs will be expressed by the difference in results of the FREE and FIXED discipline. The differences between the FIXED and OPER discipline will tell us the effect of the two TTA-unique optimizations. The total effect of scheduling transports instead of operations will be expressed by the differences between FREE and OPER.

5.3 The Results

The eleven loops were scheduled by our scheduler according to the three scheduling disciplines with and without backtracking. Scheduling with backtracking gives optimal schedules, without backtracking gives schedules that could be expected from a realistic compiler. Due to the size of Livermore loops 7 and 9, we were not able to schedule them with backtracking enabled.

The results of the experiments are given is Table 2 and are graphically represented in Fig. 4. An analysis of the results will be made in the next subsection.

Table 2. Results (steady state size) for MOVE-i860.

Name of Benchmark	Near-optimal Software Pipelining			Optimal Software Pipelining		
	FREE	FIXED	OPER	FREE	FIXED	OPER
SAXPY	5	7	13	5	5	8
DAXPY	5	6	11	3	3	5
SDOT	7	10	11	5	7	8
DDOT	3	7	8	3	4	4
LL1	5	5	10	3	4	5
LL3	3	5	5	3	3	4
LL5	6	6	7	6	6	6
LL7	18	19	22	-	-	-
LL9	16	22	24	-	-	-
LL11	3	3	4	3	3	4
LL12	3	3	4	2	3	4

5.4 Analysis of the Results

The results shown in Table 2 and Fig. 4 let us draw the following conclusions:

1. For near-optimal scheduling, the steady state sizes for the FIXED scheduling discipline are on average 31% longer than for the FREE scheduling discipline. For optimal scheduling the average difference is 17%. This improvement is due to the extra scheduling freedom of TTAs.
2. For near-optimal scheduling, the steady state sizes for the OPER scheduling discipline are on average 39% longer than for the FIXED scheduling discipline. For optimal

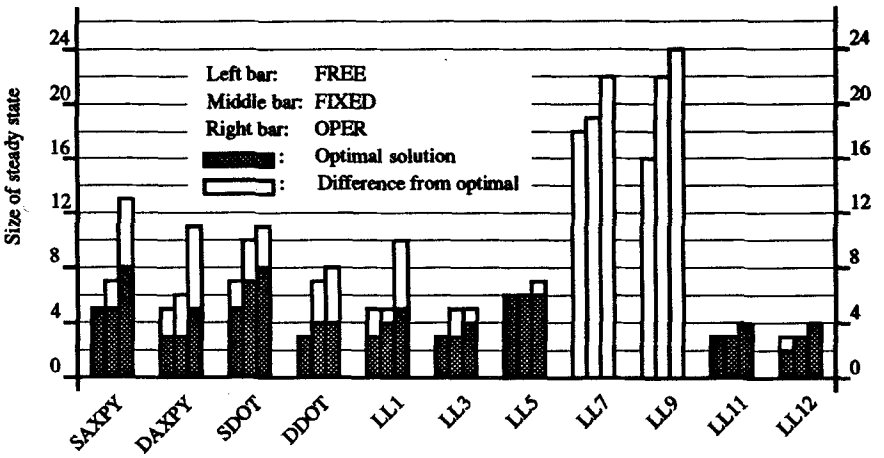

Fig. 4. Results for MOVE-i860.

scheduling the average difference is 29%. This improvement is due to the two extra TTA-unique optimizations.

3. For near-optimal scheduling, the steady state sizes for the OPER scheduling discipline are on average 74% longer than for the FREE scheduling discipline. For optimal scheduling the average difference is 50%. This is the total gain of scheduling transports instead of operations.
4. Near optimal scheduling approaches optimal scheduling for the FREE discipline. The difference between near-optimal and optimal is larger for the other disciplines. This makes transport scheduling less dependent on heuristics.

An example of a steady state is shown in Fig. 5. It is the result of scheduling the SAXPY loop according to the FREE discipline. Each instruction contains the moves across the seven move busses and the three bypasses. The example shows that some operand moves are loop invariant code and are placed before the steady state, e.g. the move that moves the '*a*' of SAXPY's '$a.x + y$' to the multiplier is loop invariant.

Since the steady state contains eight floating point operations, the execution rate for this loop is 1.6 floating point operations per cycle.

```
loop:  r4 → add-T; add-R → r3; fadd-R → f12; f10 → fadd-T; ld-R → f0; fmul-R → fadd-O; add-R → ld-T
       add-R → r4; r4 → st-O; r0 → add-T; f9 → fadd-T; f3 → fmul-T; f12 → st-T; fmul-R → fadd-O
       r0 → ne-T; add-R → r0; fadd-R → f15; f8 → fadd-T; f2 → fmul-T; fmul-R → fadd-O; add-R → ld-T
       g: loop → pcd; fadd-R → f14; f1 → fmul-T; ld-R → f8
       r3 → add-T; fadd-R → f13; f11 → fadd-T; f0 → fmul-T; fmul-R → fadd-O
```

Fig. 5. The steady state of SAXPY scheduled according to the FREE discipline.

6 Conclusions

This paper describes the results of a comparison between OTAs and TTAs. The comparison was made by introducing a TTA named MOVE-i860 which is based on Intel's i860 and a software pipelining algorithm based on Lam's algorithm.

The results show that the effect of extra scheduling freedom of TTAs is 17–31%, and the effect of two TTA-unique optimizations is 29–39%. The total effect of transport triggering is 50–71%. Besides lower dynamic instruction counts, TTAs have also several implementation advantages that reduce the clock cycle.

The figures based in this report were made with a software pipelining scheduler and benchmarks that are ideal for software pipelining. Currently we are doing similar experiments with an extended basic block scheduler and workstation-code. We hope to find similar results.

The current state of our project is the integration of our scheduling algorithm into a compiler. For this purpose we want to use the GNU C/C++ compiler of the Free Software Foundation.

References

1. Henk Corporaal and Hans (J.M.) Mulder. MOVE: A Framework for High-Performance Processor Design. In *Supercomputing-91, Albaquerque*, November 1991.
2. Jan Hoogerbrugge. Software Pipelining for Transport-Triggered Architectures. Master's thesis, Delft University of Technology, Delft, The Netherlands, December 1991.
3. John L. Hennessy and David A. Patterson. *Computer Architecture, a Quantitative Approach.* Morgan Kaufmann publishers, 1990.
4. Intel, Santa Clara, CA. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.
5. Monica S. Lam. *A Systolic Array Optimizing Compiler.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.
6. Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.
7. Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software Pipelining for Transport-Triggered Architectures. In *Proceedings of the 24th Annual Workshop on Microprogramming*, pages 74–81, Albaquerque, New Mexico, November 1991.