Evaluating *Transport Triggered Architectures* for scalar applications

Henk Corporaal

Delft University of Technology, Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: heco@duteca.et.tudelft.nl

Abstract

In transport triggered architectures (TTAs) the programming and operational model is mirrored when compared with regular RISC and VLIW architectures; instead of programming operations which cause data transports as side effects, in TTAs the transports are programmed, where a transport may trigger an operation if necessary. Transports are therefore visible at the architecture level, and are completely programmer controlled.

The main advantages of TTAs are its simplicity and flexibility. Transport triggered architectures also have certain advantages with respect to scheduling freedom and transport utilization. This paper concentrates on a quantification of advantages related to code generation for scalar (non-numeric) applications.

Keywords: VLIW, superpipelining, code scheduling, transport triggering, hybrid pipelines.

1 Introduction

CPU architectures can be classified according to the way instructions trigger the operations in the CPU data path: 1) Instructions can specify an actual operation (add r5,r3,r4) requiring this operation to trigger the transport of the operands ($\rightarrow$ fetch r3 and r4, $\rightarrow$ store in r5); or 2) instructions can specify a transport of operands and/or results (move r7,r8) requiring this transport to trigger operations on the data ($\rightarrow$ r10=r9+r8).

The first class we call *operation triggered* architectures (OTAs) and the second class we call *transport triggered* architectures (TTAs). All recent RISC, VLIW, and superscalar architectures can be classified as operation triggered. The idea of transport triggering, used at the memory level, has been described before. Lipovski [1] described a control processor; Levine and Sanders [2] built a single move TTL machine (the micro-move) in the mid seventies for speech synthesis. In [3] the class of MOVE architectures are introduced; they apply the concept of transport triggering to the register transfer level, permit multiple transports per cycle, and use a new mechanism for the pipelining of function units (FUs), called *hybrid* pipelining. This class of architectures can be viewed as a superset of traditional VLIW architectures.

This paper presents an initial quantification of our claims on scheduling freedom and resource usage. The results are restricted because of the limitations of the compiler; currently we do not schedule beyond the basic block level. Nevertheless, these initial results are very encouraging.

The structure of the paper is as follows. It starts in section 2 with a description of the concept of transport triggering, and their advantages. Section 3 outlines the measurement environment we used. The results of the measurements and their interpretation are given in section 4. Finally section 5 summarizes the results.

2 Transport triggering

Central to the idea of transport triggering is to have more control about what is going on within a central processing unit. In this sense the change from operation to transport triggering is a natural extension of the change from CISC to RISC architectures. While the application of RISC principles simplified the design and reduced the instruction set, transport triggering extends these principles: it reduces the number of instructions to one and simplifies the design even further; e.g. decoding logic is trivial.

More compiler control means more code optimization opportunities. In order to get more control, the data transport has to be separated from the operations.

Separating transport from operations

TTAs apply the concept of transport triggering to the register transfer level. The structure of a

TTA is very simple; Figure 1 shows an example architecture, containing 9 FUs and 6 transport busses. Each FU is connected to the transport network with one or more input and output sockets. In general, the network does not have to be fully connected, although full connectivity would relax the code generation process.

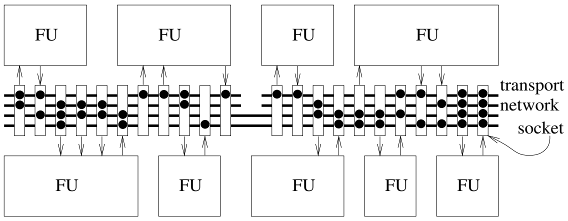


Figure 1: General structure of a MOVE architecture.

Each function unit (FU) includes one or more operand and result registers which are part of the total register address space (as seen by the programmer). We distinguish the registers according to their purpose: FU-operand registers (O), 2) FU-trigger registers (T), 3) FU-result registers (R), and 4) general-purpose registers (r)[1]. In principle one can use O, T, and R registers for general purpose; practically this happens mainly to O registers.

Triggering operations occurs by moving data into the appropriate FU-trigger (T) registers. All operations, including load/store and branch/jump/link occur solely through move instructions. FUs operate concurrently and may incorporate pipelined execution.

As an example we show how a three address register add instruction translates into move operations:

ADD r3,r2,r1 $\Rightarrow$ r1$\rightarrow$O$_{ADD}$; r2$\rightarrow$T$_{ADD}$
R$_{ADD}$$\rightarrow$r3

An interesting feature of a TTA is the fact that bypassing (or forwarding) of data is programmed, instead of relying on expensive associative bypassing hardware. We call this *software bypassing*. The next example illustrates this:

ADD r3,r1,r2 $\Rightarrow$r1$\rightarrow$O$_{ADD}$; r2$\rightarrow$T$_{ADD}$
ADD r5,r3,r4 $\Rightarrow$R$_{ADD}$$\rightarrow$O$_{ADD}$; r4$\rightarrow$T$_{ADD}$; R$_{ADD}$$\rightarrow$r3
R$_{ADD}$$\rightarrow$r5

[1]General-purpose registers can also be expressed as monadic-identity FUs.

An FU-result required immediately as operand or trigger only does not have to be moved via a register ($R \rightarrow T$).

When subsequent uses of the same FU share the data of one of the operands, an additional saving in moves is possible. This is called *common operand elimination*. Instead of needing 3 moves, three-address operations can now be translated into 1.5 to 2 moves.

It also may happen that the register $r3$ is not live any more; this allows for the elimination of the move to $r3$ (which we call dead *write back elimination*). As shown in the following code:

ADD r3,r1,r2 $\Rightarrow$ r1$\rightarrow$O$_{ADD}$; r2$\rightarrow$T$_{ADD}$
ADD r5,r3,r1 $\Rightarrow$ R$_{ADD}$$\rightarrow$T$_{ADD}$
R$_{ADD}$$\rightarrow$r5

The minimal number of moves for an addition now becomes one.

Pipeline organizations in TTAs (as used in our project) use a so called *hybrid* mechanism. They are partly *continuous*, which means that they advance autonomously, and they are partly *push-pull*, which means the result is not overwritten by subsequent results, but has to be removed explicitly from the result register. Current high-performance architectures mainly implement *continuous* pipelines (e.g. in Multiflow) or *push-pull* ones (like in the i860). Hybrid pipeline stages always continue when the following stage is not in use and block otherwise. This organization allows us to implement a simple and efficient synchronization system to relax the scheduling constraints. Reading a pending result locks the processor until the result appears in the R register. One of the main advantages concerns the removal of instructions containing no-op moves only. Results with unpredictable timing (e.g., a cache load) can also be handled by this mechanism.

Advantages of TTAs

The main advantages of transport triggered architectures are its extreme simplicity and great flexibility, allowing short processor cycle times and a quick design. Apart from design advantages, transport triggered architectures also have advantages with respect to the following issues: 1) better scheduling because the operations are more fine grain, 2) efficient usage of transport capacity, 3) extra fine grain parallelism as a result of separating ALU functionality into its individual components (*FU-splitting*), 4) less register usage as a result of the mentioned *bypassing*, and 5) a higher clock frequency; data transport limits the cycle time only. Many of these advantages are quantified in section 4.

3 Experimental Framework

In order to research TTAs, we have developed a compiler which is parameterizable over a number of parameters; for example, FU functionality, number of FUs, FU delays, and the number of parallel move busses. For this paper we have certain parameters fixed, like the FU functionality. To be able to compare TTAs with OTAs, we can instruct the compiler to impose the constraints of operation triggering; for example, limiting the scheduling freedom of move operations. The architecture, compiler and benchmarks used in the experiments for this paper are discussed next.

3.1 Experimental RISC architecture

The experimental architecture we use for comparison purposes in this paper is a fully connected TTA, with a typical RISC functionality such as available on the MIPS processor. This architecture includes 64 general purpose registers (32 integer and 32 floating point registers). The main integer functionality and operand/trigger/result (OTR) registers are shown in Table 1. From this table we can see that the exploitable parallelism is limited. The available parallelism is caused by allowing the functional units to be accessed simultaneously.

A move instruction specifies one or more move operations and up to one 16-bit immediate. Figure 2 shows 3 different move operation formats used for the experiments.

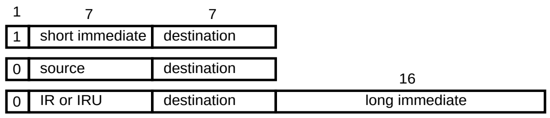


Figure 2: Different move operation formats.

3.2 The compiler trajectory

The compiler trajectory used for the measurements in this paper is shown in Figure 3. For comparison purposes we use the MIPS-C compiler, which delivers high quality code, as frontend[2]. The backend is retargetable; it translates arbitrary MIPS executables to sequential move code, and then parallelizes and schedules this code for the given MOVE architecture.

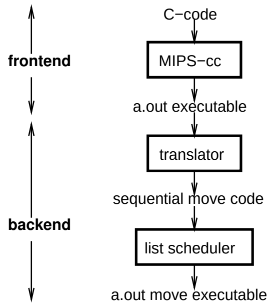


Figure 3: Overview of the compiler trajectory.

Before we move on to more aggressive and attractive scheduling techniques, we try to prove our MOVE expectations with a simple form of basic block scheduling. The results in this paper are obtained by a variant of list scheduling in a bottom-up fashion.

3.3 Benchmark characterization

The benchmarks are oriented towards general purpose applications (in [4, 5] we analyzed scientific code for TTAs). We use two familiar benchmarks: dhrystone and the c-preprocessor cpp. Table 2 lists their main characteristics (when compiled with cc, and run on a MIPS processor; cpp uses itself as input data). The small average basic block length is typical for general purpose applications.

One of the claims is that TTAs allow the exploitation of more fine grain parallelism. Therefore it is interesting to know how much parallelism is available in these two applications. Table 3 summarizes the maximal obtainable parallelism for different machine and compiler models. These figures are obtained using list scheduling of traces. In this table the machine parameter indicates the number of function units (FUs) of each type (each having single cycle latency).

Two compiler models are used: model C-1 assumes that branches are always predicted wrong; in model C-2 branches are always predicted correctly. For both models the compiler can distin-

[2] A different available frontend, not used for this paper, is based on the GNU-C software.

Function Unit (FU)	O	T	R	delay
load byte,half,word	La	Lb,Lbu,Lh,Lhu,Lw	Ld	2
load immediate (16b)	–	–	Ir	1
load upper immediate (16b)	–	–	Iru	1
store byte,half,word	Sa,So	Sb,Sh,Sw	-	1
addition, subtraction	Into	Add,Sub	Intr	1
multiplication	Mulo	Mul	Mulr	11
division	Divo	Div	Divr	32
logical	Logo	And,Or,Xor,Not	Logr	1
shift	Shto	Sll,Slr,Sar	Shtr	1
compare	Cmpo	Lt	Cmpr	1
branch, jump	Bo1,Bo2	Bra,Beq,Bne,Bgtz,Jmp	Pc	2

Table 1: The functionality of our experimental architecture.

benchmark	characteristics					
	static nr. of instructions	dynamic nr. of instructions	nr. of cycles	basic block length	NOPs	Interlocks
dhrystone	1358	386634	408851	4.8 instr.	7.9 %	5.7 %
cpp	5607	441799	467371	3.7 instr.	16.9 %	5.8 %

Table 2: Benchmark characteristics.

	dhrystone		**cpp**	
machine	**C-1**	**C-2**	**C-1**	**C-2**
FU-1	1.75	3.81	1.39	2.80
FU-2	2.03	7.60	1.48	5.59
FU-3	2.03	11.3	1.49	8.17

Table 3: Obtainable speedups for 3 different machine and 2 compiler models (C-1 and C-2).

guish all memory references except for those into the heap.

From this table we may conclude that although both applications show a reasonable amount of parallelism when branches are well predicted, the available parallelism at the basic block level is rather low[3]. Having multiple FUs per type barely speeds up the execution in the latter case.

4 Measurement results

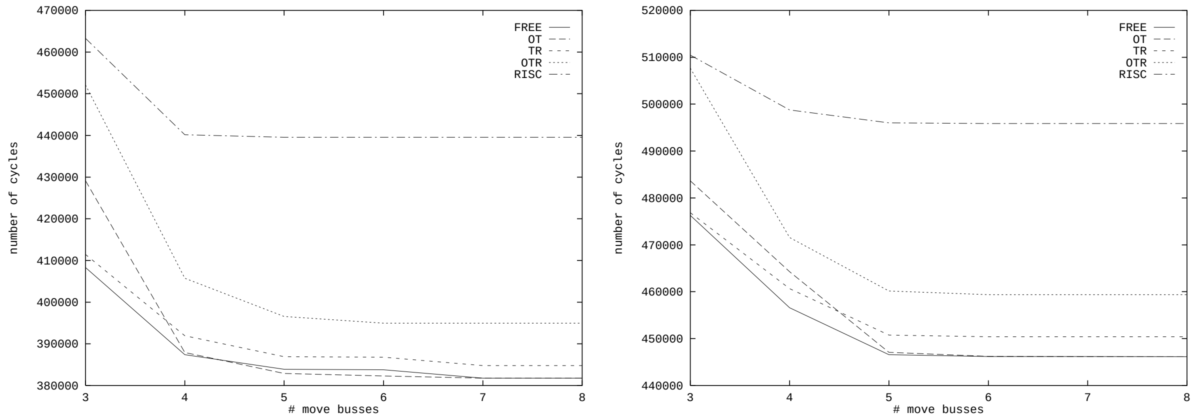
The measurements presented in this section give an initial quantification of the claims made about transport triggered architectures. In the next subsections we will look at different topics: influence of scheduling freedom, transport usage, the effect of FU splitting, and the influence of more optimal pipelining of MOVE architectures.

4.1 Scheduling freedom and optimizations at the transport level

When starting from a 3-address RISC architecture, we can increase the scheduling freedom in four steps to reach the MOVE model. The five resulting models are defined as follows:

1. **RISC** This is the most restricted case where moves to O and T have to be scheduled in the same cycle, while the move from R should take place exactly *FU delay* cycles afterwards. On top of that the number of triggers per cycle is reduced to one; each cycle can only initiate one operation. This model combined with 3 transport busses closely emulates RISC hardware.
2. **OTR** This model corresponds to the former one, except that multiple triggers per cycle are allowed.
3. **TR** O and T are decoupled, but T is to be scheduled after or with O. The fixed timing between T and R remains.

[3] In fact the figures shown even are a little bit too optimistic, because the trace analysis tool only assumes basic block boundaries at jumps; this results in larger basic blocks.



a. dhrystone b. cpp

Figure 4: Number of dynamic-instructions versus scheduling freedom and number of move busses

4. **OT** Scheduling O and T remains fixed as in model 1. Scheduling T and R is decoupled. R may be read later than *FU delay* cycles after T.
5. **FREE** Now both O and R are decoupled from T. T must be scheduled with or after O, and R may be scheduled any cycle past the delay of the FU.

Using the above models, we have compared the dynamic cycle counts after scheduling. Figure 4 shows the results for dhrystone and cpp as a function of both scheduling freedom and the number of moves per instruction. Note that these numbers include the effect of software bypassing and write back elimination.

For the M-3 model (3 transport busses) we note a difference of 7% (cpp) till 13% (dhrystone) between the FREE and RISC scheduling disciplines. This difference is mainly caused by the scheduling restriction of coupling the operand trigger and result moves (fixed relative positions); using this discipline with 3 busses there is not much room for multiple triggers per cycle.

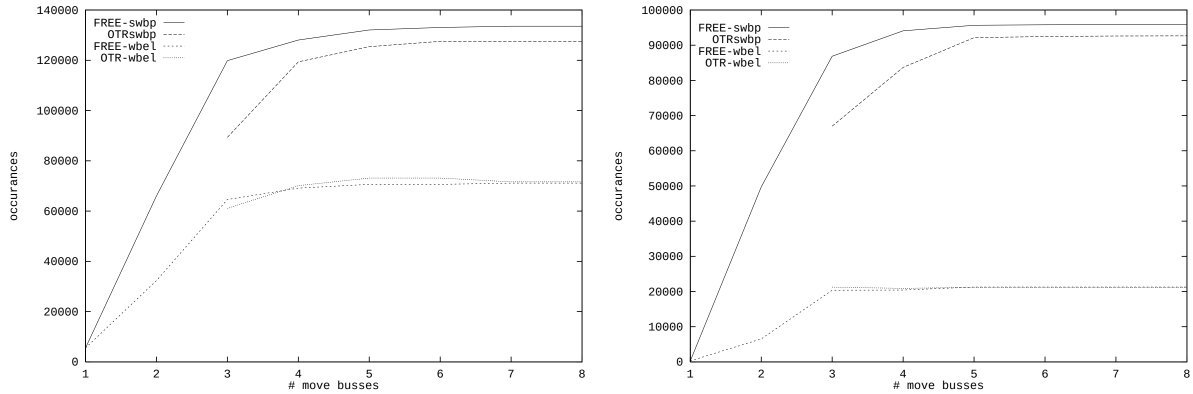
Closer examination shows that the decoupling of the operand from the trigger is especially important. In the OT model it often appears that operand and trigger can not be scheduled in the same cycle. This effect disappears when enough transport capacity is available, because scheduling yields fewer move-slot constraints and therefore less need for additional scheduling freedom. However the OTR and FREE models still differ about 3% for many move busses. This is partly due to scheduling difficulties. When doing bottom up scheduling in these latter models, we have to look ahead when scheduling a result move, in order to check if trigger (and possibly operand) move can be scheduled. In some cases our scheduler does not take the optimal decision; we need to incorporate a better heuristic, or a limited form of back tracking. The RISC model performs relatively worse for a large number of busses. It can not exploit the parallelism caused by FU splitting. Section 4.3 will elaborate on this aspect.

Figure 5 shows the number of *software bypasses* and *write back eliminations* as a function of the scheduling freedom (FREE and OTR models) and the number of parallel moves. When the constraints on scheduling moves are relieved, because more move busses are available, the bypassing frequency quickly increases, however there is no noticeable difference in the number of write back eliminations. Currently our compiler does not perform a global register live analysis. As a consequence the number of write back eliminations is rather pessimistic and will increase in future.

Note that this figure includes M-1 and M-2 models for the FREE scheduling discipline. In the OTR model it is not possible to schedule for less than 3 busses. The interesting fact is that move architectures can also be scaled down to 1 or 2 busses.

4.2 Usage of transport capacity

In this section we investigate the usage of the available transport capacity. There are several reasons why an operation triggered architecture



a. dhrystone   b. cpp

Figure 5: Software bypassing and write back elimination occurrences

does not utilize its transport capacity efficiently. First of all, many operations do not require 3 operands. Second, several specific compiler optimizations for TTAs reduce the number of transports. In section 4.1 we mentioned already the write back elimination optimization. Another optimization results from common operand elimination; this happens when succeeding operations on the same function unit happen to use the same operand. Currently our scheduler does not use this latter optimization, however we are able to identify these operand moves and count the potential number of eliminations.

Table 4 shows the transport requirements for move architectures with 1 to 4 transport busses (using the FREE scheduling model), and compares this with a RISC architecture. All architectures have MIPS functionality (with one function unit of each type) and corresponding operation latencies. The RISC data is calculated from table 2.

From this table we can draw several interesting conclusions. First of all, note that it is possible to schedule for a single bus MOVE architecture, with a very efficient usage of this single bus. However, the data transport usage drops quickly when enlarging the data transport capacity. This is mainly a consequence of insufficient exploitable parallelism within a basic block. As was shown in table 3 the achievable parallelism for dhrystone and cpp (for one FU/type, single cycle latency) is 1.75 and 1.39 respectively. However these amounts of parallelism are hardly sufficient to fill the delay slots of multicycle operations. Clearly the scheduling scope has to be extended beyond basic blocks.

program	model	10^3 cycles	10^3 cycles	moves/ cycle	% usage
dhrystone	M-1	967	916	0.95	95
	M-2	533	889	1.67	83
	M-3	408	857	2.10	70
	M-4	387	852	2.20	55
	risc	408	1068	2.62	87
cpp	M-1	964	862	0.89	89
	M-2	584	856	1.47	73
	M-3	476	842	1.77	59
	M-4	457	842	1.84	46
	RISC	467	1101	2.36	79

Table 4: Transport usage for MOVE architectures with 1 to 4 busses and for RISC

A second observation relates to the comparison of a RISC architecture and a corresponding MOVE architecture with 3 transport busses (MOVE-3). A RISC requires 25 - 31 % more transport capacity. This is caused by three effects: common operand elimination, write back elimination and many operations requiring less than 3 operands. Although table 3 clearly shows the superiority of MOVE-3 w.r.t. required transport capacity, our current compiler can not effectively exploit this extra available capacity. Execution times of RISC and MOVE-3 are similar (within 2 %). This is again caused by the limited scheduling scope.

Besides insufficient parallelism within a basic block, the restriction to basic block scheduling leads to another disadvantage for MOVE architectures. It turns out that at the end of a basic block often an extra cycle is needed to flush the pipelines of function units, and store the results in general purpose registers. The current scheduler requires results to be in general purpose registers between basic blocks; this restriction disappears when we schedule beyond basic blocks.

Taking these limitations into account it is rather surprising that we could achieve the same performance. We expect that in extending the scheduling scope, MOVE architectures are able to effectively exploit these extra transport resources[4]. Note that all unused move slots are available to more aggressive scheduling, while for the RISC only no-op instructions are available.

4.3 FU splitting

One of the advantages of TTAs is that different functionality (like arithmetic and logical operations) does not have to be combined into one FU (e.g. an ALU); it is easier to have separate FUs for nonrelated operations. This results both into design and performance advantages. The performance is enhanced because multiple operations can be started in a single cycle, resulting in a better FU utilization. Table 5 demonstrates the effect of reducing the numbers of triggers per cycle to one (using the FREE scheduling discipline). For 3 busses this restrictions results in a performance loss of 2 to 6 %. It will be evident that this restriction gets worse for larger transport capacities. We expect the effect of FU splitting to

model	dhrystone		cpp	
	M-3	M-4	M-3	M-4
1 trigger/cycle	434	423	484	481
more triggers/cycle	408	387	476	457

Table 5: Performance (measured in 10^3 cycles) influence of FU splitting

increase when extending the scheduling scope.

4.4 Cycle time

Previous sections showed several scheduling and resource usage advantages of MOVE architectures. There may be a catch however. When we look at the following code

ADD r1, r2, r3	r2 $\rightarrow O_{add}$, r3 $\rightarrow T_{add}$
SUB r4, r5, r1	....
	$R_{add} \rightarrow O_{sub}$

we see that in general the MOVE operations require an additional cycle (as compared to a RISC) to cover the transport of a result. We can view this as if a MOVE machine is inherently superpipelined. Because of this superpipelining nature the slowdown in number of executed cycles can be offset by a higher clock frequency and by more aggressive scheduling. Of course this *lost cycle* disappears when the trigger register (T) is removed and the first operation (pipeline) stage is integrated with the transport stage; it is also possible to remove the result register. Removing the trigger or result register means that part of the FU logic will be active within the data transport cycle, and therefore stretches the minimal achievable cycle time.

We can actually calculate the worst case frequency increase to cover the *lost cycle* by means of the degree of superpipelining as introduced by Jouppi [6]. The superpipelining factor (sf) is defined to be the average operation delay:

$$sf = \sum_{all\ operations\ o} f(o) \times l(o)$$

where $f(o)$ is the frequency and $l(o)$ the latency of operation o.

Table 6 shows the FU latencies as introduced in Section 3 (RISC model) and the FU latencies including the *lost cycle* (MOVE model; this model does not have logic within the transport cycle).

Furthermore we define the clock factor, being the factor the clock needs to be faster for the MOVE versus the RISC model. In our case (using the indicated instruction mixture) this factor is approximately 1.52.

This factor however only holds for those situations where no exploitable parallelism remains. This implies that for few move busses the factor will be lower; only for infinite resources and a perfect match of operations within the critical path we will reach this factor. Table 7 shows the calculation for the two latency models.

As expected, for one move bus the clock factor is close to one. For increasing number of move busses the factor increases to maximal 35 %, still far below the theoretical value of 52 %. This is caused by a nonperfect match of the distribution

[4] This belief is strengthened by our positive experience of scheduling scientific code using software pipelining, see [4, 5].

operations	r-r	load	lim	stor	add	mul	div	logi	shft	cmp	branch	jmp	
mix	2%	21%	10%	9%	25%	0%	0%	1%	5%	4%	18%	4%	sf
RISC	1	2	1	1	1	11	32	1	1	1	2	2	1.43
MOVE	1	3	1	1	2	12	33	2	2	2	3	2	2.18
	Clock factor $\frac{MOVE}{RISC}$												1.52

Table 6: Calculating exploitable parallelism and cycle factor.

	MOVE Models				
program	M-1	M-2	M-3	M-4	M-8
dhrystone	1.02	1.12	1.29	1.33	1.35
cpp	1.04	1.10	1.21	1.25	1.28

Table 7: Calculated clock factor for different transport capacities

of operations in the critical path, and by remaining resource limitations, like a limited number of registers (resulting in many antidependencies).

It appears that an M-3 model MOVE machine including the *lost cycle* only needs to have a clock frequency which is 21 to 29% higher than a RISC. Initial measurements on a VLSI prototype design makes us believe that more than 50% is easily achievable.

5 Conclusions

In this paper we have set the first step in verifying the claimed advantages of transport triggered architectures over traditional operation triggered architectures for general purpose applications. We have investigated the utilization of the inter-FU transport structures and conclude that for a RISC with 3 register ports the effective utilization is very low. In operation triggered architectures we cannot control the data transport in such a way that the available transport capacity is used more optimally. Transport triggered architectures do offer this necessary control. However, when having 3 or more transport busses, more aggressive scheduling techniques are needed to fully exploit the available transport capacity; the scheduling scope of a basic block is too narrow.

Looking into the future, we have to research several topics: extending the scheduling scope, implementing Common Operand Elimination, better exploitation of the dead write back elimination optimization, and anti dependency removal.

Based on the facts as presented in this paper we have a firm belief that transport triggering opens a whole new architectural area with lots of new scheduling opportunities.

Currently we are completing a first implementation of a MOVE architecture in 2μ (1.6μ gate length) CMOS Sea of Gates technology, operating at a 12.5 ns cycle time [7]. This architecture allows for four concurrent transports per cycle. The design allows us to verify the hardware related advantages of MOVE architectures.

References

[1] G. J. Lipovski. Architecture of a simple, effective control processor. In *Second Symposium on Micro Architecture*, pages 187–194, 1976.

[2] Arvin Levine and William R. Sanders. The miss speech synthesis system. Technical Report TR299, Inst. for mathematical studies in the social sciences, Stanford University, December 1978.

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

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

[5] 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.

[6] N.P. Jouppi and D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proc. 3th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, April 1989.

[7] Henk Corporaal and Paul van der Arend. Move32int, a sea of gates realization of a high performance transport triggered architecture. In *EuroMicro-93*, Barcelona, September 1993.