Transport Triggered Architectures examined for general purpose applications

Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
P.O. Box 5031, Delft, The Netherlands
email: 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 transport triggered architectures are its simplicity and flexibility, allowing short processor cycle times and a quick design. Transport triggered architectures also have certain advantages with respect to scheduling freedom and transport utilization.
After introducing TTAs and discussing its advantages qualitatively, this paper concentrates on a quantification of those advantages related to code generation for general purpose applications, on a family of architectures, called MOVE architectures. Besides being transport triggered, these architectures use a new way of function unit pipelining, called hybrid pipelining, which further enhances scheduling freedom.

Keywords: VLIW, superpipelining, code scheduling, operation triggering, 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 [Lip76] described a control processor; Levine and Sanders [LS78] built a single move TTL machine (the micro-move) in the mid seventies for speech synthesis. In [CM91] 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.

As shown in figure 1, this class of architectures can be viewed as a superset of traditional VLIW architectures. RISC architectures are of type **SISO** (single instruction, single operation), because they execute one

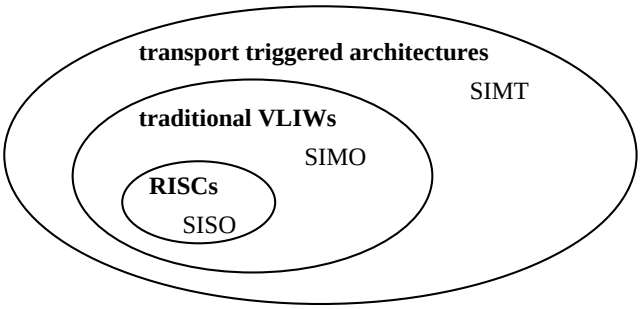


Figure 1: Architecture space of long instruction word machines.

instruction at a time; an instruction specifies one operation. VLIW architectures are of the **SIMO** type (single instruction, multiple operation). A VLIW instruction specifies multiple operations which are executed concurrently; each operation is comparable to a RISC operation. TTAs are of type **SIMT** (single instruction, multiple transports). One instruction may specify multiple concurrent transports. TTAs are more general, because they put fewer constraints on the scheduling of the data transports.

A qualitative analysis of transport triggered MOVE architectures indicates cost-performance advantages over OTAs on the following issues:

Design simplicity: MOVE architectures are very easy to design, and allow very short cycle times. This is a direct consequence of the clear separation of FUs and data transport.

Design flexibility: there are no architecture limits on type, number and implementation of FU, and on the data transport capacity. MOVE architectures are therefore particularly suitable for application specific processors.

Scheduling freedom: the scheduling of transports instead of operations allows for all kinds of extra compile time optimizations.

Efficient resource usage: available resources like data transport busses and FUs are used more efficiently. This results in a higher performance potential.

This paper provides an introduction to TTAs and presents an initial quantification of our qualitative 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 short overview of current architecture developments, considering the exploitation of instruction level parallelism. This section motivates the change of the programming model. Section 3 describes the concept of transport triggering, the class of MOVE architectures, and their advantages. Section 4 outlines the measurement environment we used. The results of the measurements and their interpretation are given in section 5. Finally section 6 summarizes the results, indicates further improvements and gives some concluding remarks with respect to the usefulness of transport triggering.

2 Recent developments in computer architecture

In order to increase execution speed of single processing nodes several architectural developments take place. They are based on two design techniques [JW89]:

Superpipelining: Change the architecture in order to decrease the achievable cycle time by subdividing the already existing pipeline stages. This is an extension of the pipelining principle used in RISC architectures. While in RISC machines the execution stage of most instructions takes 1 cycle, superpipelined machines split up this cycle. RISC principles brought down the CPI close to one. Superpipelining however, will more or less increase CPI, because of added delay slots. The idea is that this slight increase is more than compensated for by the decrease in cycle time (e.g., Cray, Intel I860). Examples are described in [Rus78, i8689].

Functional parallelism: Add independent functional units in order to increase operation level parallelism and decrease effective CPI below one. Two representatives using functional parallelism are Superscalar and Very Long Instruction Word (VLIW) architectures. The former aims at dynamic detection of operation level parallelism. See e.g. [MS$^+$91, i9688, Do92]. The latter is totally dependent on compile time scheduling of operations. VLIW examples are described in [Ebc88, LS90, MP89, Tra87, BS88, R$^+$89]. In principle a Superscalar architecture can run unscheduled code and is therefore object code compatible with the same architecture without detection of runtime parallelism. However, getting a reasonable amount of parallelism requires hand coding or compiler support.

Superscalars are severly limited in the amount of exploitable parallelism. The hardware needed for runtime parallelism detection limits the allowable window detection size to a few instructions. Further hardware reduction is achieved by restricting the possible combinations of parallel instructions (e.g. one integer, one floating point, one load/store and one control instruction). However, still quite a bit of instruction control hardware is spent on work for which the compiler could do a far better job. A possible exception is the disambiguation of memory references. A favorable consequence of dynamic scheduling is that code size does not increase by unnecessary empty instruction slots. This means that superscalar techniques can certainly be used to upgrade existing architectures; they give moderate speed improvements, but need active compiler support in order to get this speedup.

VLIW and superpipelining architectures have a higher potential for parallel execution than superscalars. Note that from a compilers perspective a VLIW and superpipeline behave similar in the sense, that for both architectures, the compiler has to search for independent operations, which can be scheduled together into one (VLIW) instruction or in a pipelined fashion. The advantage of these architectures for general purpose computation will probably not be big, because of the large sequential code component of those computations. However, in vector and other special purpose applications they may show large speedups.

Ideally we would like to combine VLIW and superpipelining principles, which would lead to architectures which deliver compatible performance for vector applications (as compared to vector machines), may be tailored to special applications, and which also give adequate scalar performance (which is a must for vector applications too). However, these architectures exhibit some disadvantages which result in complex organizations, poor hardware utilization, difficult to change functionality, and poor performance scalability.

VLIW organization

Figure 2 shows the internal communication structure of a superpipelined VLIW architecture. A set of FUs is connected to a bypass unit containing operand and bypass latches. The bypass unit is connected to a register file. A small clock cycle provides a high communication throughput.

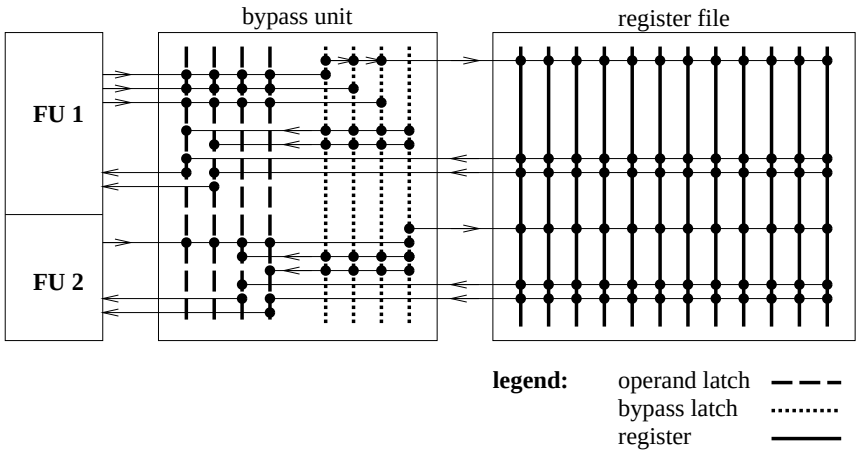


Figure 2: Communication between FUs and Register file in a VLIW architecture.

The bypass unit consists of operand latches, and bypass latches organized as a result FIFO for every FU. The operand latches can be written by the register file, by the FIFO, and by the FU output ports. The FU output ports also write the result FIFO. Writing can occur in all FIFO latches to allow FUs of different latency to deliver their result. Writing a result also implies writing the register identifier of the result. This identifier is used to deliver the result in the correct register en to be able to identify the result in the FIFO. A result which leaves the FIFO is written back in the register file. Reading the bypass is architecturally transparent: the identifier of the requested register is matched with the identifiers in the bypass; if a match occurs the bypass value is transferred to the operand latch instead of the value from the register file.

Looking at figure 2 we make the following observations:

1. Values written from the bypass into the register file may be dead. Measurements indicate that many generated results are consumed immediately. Deeper FIFOs enlarge this number. The bandwidth of the write port into the register file is therefore largely redundant.
2. Operations requiring one or two operands under utilize the register-bypass bandwidth (e.g., monadic operations and branches).
3. The bypass can become rather complex when we increase the number of FUs and/or the superpipelining degree.
4. Adding FUS requires a complete redesign of the architecture.

We conclude that the communication requirements are high, and that the communication network is under-utilized. However, the compiler is able to detect all cases of under-utilization, and knows which temporary results it needs from the bypass. This motivates us to change the programming paradigm, and making all transports visible in the architecture. This will be explained in the next section.

3 Transport triggering and MOVE

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. The remainder of this section explains this separation and its consequences; it further elaborates on additional features necessary to get a universal programmable architecture.

3.1 Separating transport from operations

TTAs separate transport from operations. MOVE architectures (which are further detailed in the next subsection) apply the concept of transport triggering to the register transfer level. The transport model of MOVE is very simple; Figure 3 shows a single set of registers connected in some way (full connectivity makes code compilation easier, but it is VLSI area inefficient and it slows down the cycle time). Programming this machine occurs by register-register *move* operations only ($r_x \rightarrow r_y$). Similar to register sets having multiple ports, a MOVE architecture may execute multiple concurrent moves. A multiple-MOVE architecture is comparable to a VLIW architecture.

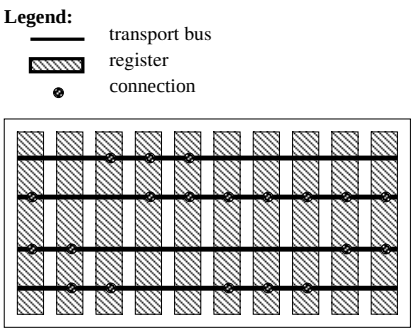


Figure 3: MOVE: transport model.

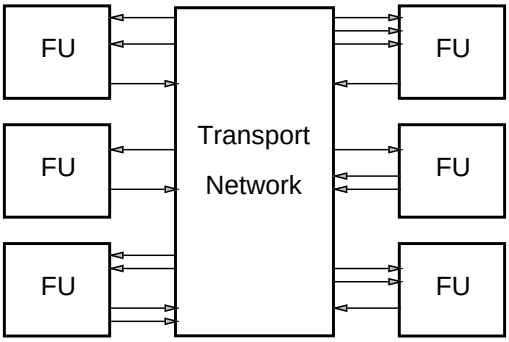


Figure 4: MOVE: operation model.

The operation model of MOVE is shown in Figure 4. All function unit (FU) operand and result registers are part of the register file. 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. The functionality to access external data and instruction memory is also implemented in FUs controlled similar to, for example, an arithmetic FU. FUs may all operate concurrently and may incorporate pipelined execution.

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

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

Note that many RISC-like instructions can be modeled with less than 3 moves, e.g. monadic operations, jumps, and register copies.

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

An interesting feature of a MOVE machine 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        ⇒   r1→O_ADD; r2→T_ADD
ADD r5,r3,r4        ⇒   R_ADD→O_ADD; r4→T_ADD; R_ADD→r3
                        R_ADD→r5
```

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        ⇒   r1→O_ADD; r2→T_ADD
ADD r5,r3,r1        ⇒   R_ADD→T_ADD
                        R_ADD→r5
```

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

3.2 The MOVE architecture

To make an efficient transport triggered architecture we need to execute conditionally, deal with pipelining and interlocks, and be able to handle exceptions. Only the first two mechanisms will be presented briefly (see [CM91] for further details). The choices we made establish what we call the family of MOVE architectures.

In our first MOVE prototype, which is build in 1.6μ CMOS Sea of Gates technology, conditional execution is implemented by means of guards. Every move is guarded, which implies that a move of the form $g : r_x \rightarrow r_y$ is only executed when g evaluates to true. The prototype includes simple guarded expressions of two condition bits which are settable by means of compare operations in the guard FU. The model used for the measurements in this paper does not use guards; for comparison purposes a traditional compare and branch mechanism has been used (see section 4).

Most pipeline organizations in MOVE FUs 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* [RYYT89, Tra87] pipelines or *push-pull* [i8689] ones. MOVE 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.

3.3 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:

Instruction scheduling: dividing operations into individual transports of operands and results allows for better scheduling opportunities and therefore speeding up execution.

Transport capacity: because the required transport capacity is designed on desired performance instead of worst case usage, its utilization can be significantly higher. Viewed from a technological point of view, this implies that, given a desired performance, TTAs are less "metal" hungry than OTAs. Alternatively, with an equal amount of "metal" MOVE architectures outperform their operation triggered counterparts.

Fine-grain parallelism: separating ALU functionality in its individual components (add/sub, or, and, xor, shift) allows for a finer grain of parallelism to be exploited.

Register usage: scheduling transports also allows for the use of FU operand, intermediate stage, and result registers for temporary storage, resulting in a reduced general purpose register usage.

Higher clock frequency: in a MOVE, the execute cycle is separated from the operation cycle, yielding a higher degree of pipelining than a RISC machine. The consequences are longer latencies in cycles, but higher clock frequencies.

Code Size Coding data transports explicitly results in a less dense encoding, when compared to 3-address RISC encodings. However, in practice many moves disappear as a result of compiler optimizations. Depending on the quality of the compiler the density of the resulting MOVE code may be comparable to RISC code.

Section 5 will quantify many of the above advantages; a more complete qualitative analysis can be found in [CM91]. The main disadvantage of MOVE architectures is that the compiler backend becomes more complex. Scheduling at the transport level, and exploitation of the extra scheduling degrees of freedom introduces more algorithmic complexity. This aspect will not be treated further in this paper.

4 Experimental Framework

In order to research the MOVE concept, 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.

4.1 Experimental RISC architecture

The experimental architecture we use for comparison purposes in this paper is a fully connected MOVE architecture with a typical RISC functionality like 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

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.

parallelism in this MOVE model is limited. The available parallelism is caused by allowing the functional units to be accessed simultaneously. This effect, which is intrinsic to the move architecture, is what we call *FU splitting*.

A move instruction specifies one or more move operations and up to one 16-bit immediate. Figure 5 shows 3 different move operation formats used for the experiments. Short 7-bit immediates can be put in the source

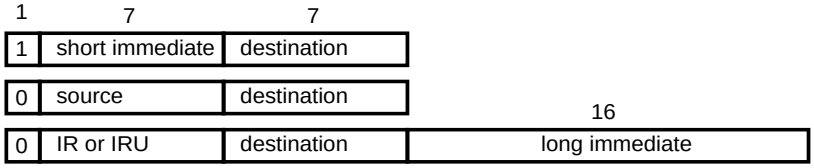


Figure 5: Different move operation formats.

field and are indicated by a 1-bit flag. Larger 16-bit immediates take an extra field, and the *source* is set to denote the Instruction Register Ir or Iru, which will hold the immediate at that point in the execution. Reading from Iru will put the immediate in the upper half of the word.

4.2 The compiler trajectory

The compiler trajectory used for the measurements in this paper is shown in Figure 6. 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.

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. When analyzing the dataflow in a basic-block, no information on

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

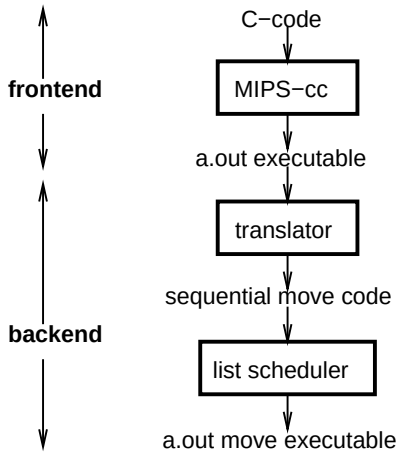


Figure 6: Overview of the compiler trajectory.

memory-reference disambiguation is being used. The scheduling restriction to basic blocks appears to be a major bottleneck for fully exploiting the capabilities of transport triggered architectures. We will return to this issue in section 5.

4.3 Benchmark characterization

The benchmarks are oriented towards general purpose applications (in [HCM91, HC92] we analyzed scientific code for TTAs). We use two familiar benchmarks: dhrystone and the c-preprocessor cpp[3]. 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.

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.

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 Comp-1 assumes that branches are always predicted wrong; in model Comp-2 branches are always predicted correctly. For both models the compiler can distinguish all memory references except for those into the heap.

[3]Although we can schedule any MIPS executable, the amount of presented detail limits us to two applications.

	dhrystone		cpp	
machine \ compiler	**Comp-1**	**Comp-2**	**Comp-1**	**Comp-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.

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[4]. Having multiple FUs per type barely speeds up the execution in the latter case.

5 Measurements and 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, the influence of more optimal pipelining of MOVE architectures, and code size.

5.1 Scheduling freedom and optimizations at the transport level

Because all moves representing a RISC instruction are decoupled from each other, MOVE architectures inherently provide additional degrees of scheduling freedom. 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.

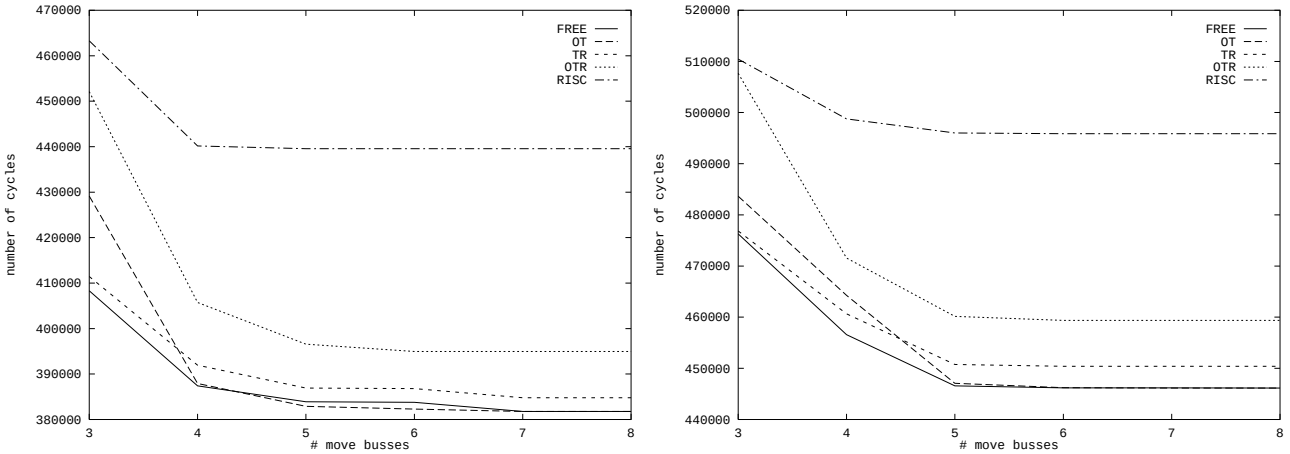
4. **OT** Scheduling *O* and *T* remains fixed as in model 1. Scheduling *T* and *R* is decoupled. *R* may be read later than *FU delay* cycles after *T*.

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.

[4] 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 RISC processor can be viewed as a MOVE architecture with 3 transport busses, which is scheduled according to the RISC scheduling model. A VLIW can be considered as a MOVE architecture with multiple FUs, having ample transport capacity to handle worst case transport situations (many FUs require 3 transport busses each), and which is scheduled according to OTR model.

Using the above models, we have compared the dynamic cycle counts after scheduling. Figure 7 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.



a. dhrystone b. cpp

Figure 7: Number of dynamic-instructions versus scheduling freedom and number of move busses

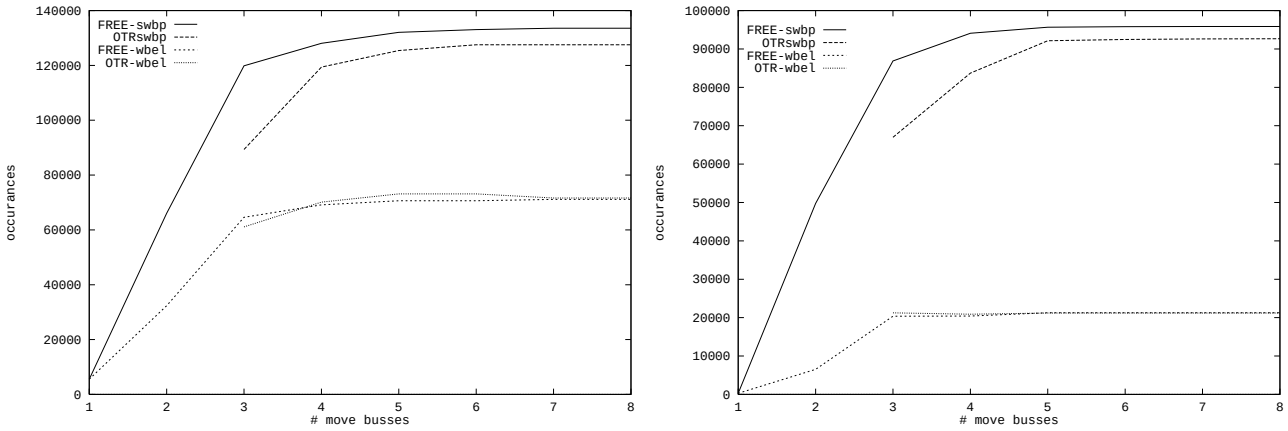
For the M-3 model (3 transport busses) we note a difference of 7% (cpp) till 13% (dhrystone) between the FREE and RISC scheduling disciplines. This difference is mainly caused by the scheduling restriction of coupling the operand trigger and result moves (fixed relative positions); using this discipline 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 worse for a large number of busses. It can not exploit the parallelism caused by FU splitting. Section 5.3 will elaborate on this aspect.

Figure 8 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.



a. dhrystone b. cpp

Figure 8: Software bypassing and write back elimination occurrences

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 down scaled downward to 1 or 2 busses.

5.2 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 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 5.1 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 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. In section 5.4 we will come back to this point.

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

program	model	10^3 cycles	10^3 moves	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

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[5]. Note that all unused move slots are available to more aggressive scheduling, while for the RISC only no-op instructions are available.

5.3 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 increase when extending the scheduling scope.

[5]This belief is strengthened by our positive experience of scheduling scientific code using software pipelining, see [HCM91, HC92].

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

5.4 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 timing in Figure 9 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

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

Figure 9: Example of the *lost cycle*

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 [JW89]. The superpipelining factor (sf) is defined to be the average operation delay:

$sf = \sum_{alloperationso} f(o) \times l(o)$, where $f(o)$ is the frequency and $l(o)$ the latency of operation o.

Table 6 shows the FU latencies as introduced in Section 4 (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 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 of operations in the critical path, and by remaining resource limitations, like a limited number of registers (resulting in many antidependencies).

	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

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.

5.5 Code size

As mentioned before, explicit specification of transports results in a less dense encoding. For example, a 3 bus MOVE as used in our experiments requires 61 bits per instruction (3 x 15 bits for the transport specification, and 16 bits for a half word immediate). Considering the fact that our compiler currently achieves similar performance (within 2%), it will be clear that MOVE programs are larger and require more instruction bandwidth. Currently our compiler produces between 2.3 and 2.5 moves per RISC instruction (not counting no-ops of course). Better exploitation of the MOVE specific compiler optimizations (write back and common operand elimination) will lower this number. Further we noted that many half word immediates could be replaced by short ones (which are encoded in the move source field). When we are able to fill the move slots of an instruction more efficiently and enhance the exploitation of the move optimizations, we expect that the dynamic and static code size will not dramatically differ between MOVE and RISC.

Comparing larger MOVE architectures (multiple FUs per type, more transport capacity) with corresponding VLIWs we even expect MOVE code to be denser. This is a consequence of the fact that all move slots within an instruction are equal and easier to fill, in contrast to VLIW instructions, where each slot specifies an operation for a specific FU.

6 Conclusions and Further Research

In this paper we have set the first step in verifying the claimed advantages of transport triggered architectures over traditional operation triggered architectures, like RISCs and VLIWs, 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.

Because MOVE architectures use the hybrid pipelining concept, it adds more degrees of freedom to the scheduling process; a MOVE is able to exploit more parallelism than a RISC with comparable hardware.

Unlike a RISC, a MOVE exploits software bypassing, implying that results are directly moved to operation or trigger registers under program control. A direct consequence is a reduced hardware usage. Note that especially for VLIW and superscalar architectures this associative hardware can become very expensive.

Although not intrinsic to the MOVE concept, it is possible that most operations incur an additional cycle latency. This effect (called the *lost cycle*) is caused by the separation of transport and operation. Measuring two different latency models and calculating their relative performance as a function of the number of MOVE slots gives us the relative clock frequency needed to balance the performance of the two models. We are convinced that the increase in latency will be more than compensated for by the higher achievable clock rates.

Looking into the future, we have to research the following topics:

1. Extending the scheduling scope of the backend scheduler. The scheduler has to use branch profiling information in order to adequately predict branches.

2. Implement the Common Operand Elimination (COE). Scheduling heuristics have to be improved in order to optimize the amount of COE. The architecture could also implement multiple operand registers per FU. A typical case is the load/store unit, where a special operand register could be used for the stack pointer. For general purpose applications this leads to a significant reduction of the required transport capacity.

3. Better encoding of immediates. Currently we did not put much emphasis onto instruction encoding density. Measurements by Pixie (a profiling tool) indicate that many 16 bit immediates can be replaced by short ones (e.g. for dhrystone 90 % of the immediates requires less than 8 bits).

4. Better exploitation of the dead write back elimination optimization. Looking carefully at the generated code of our compiler, we see that many more write backs are dead. Eliminating all of them requires full register live analysis.

5. Anti dependency removal. Parallelism increases if we can remove anti dependencies by register renaming. Currently we were not able to do this because we used the register allocator of the frontend.

6. Exploitation of guards. In our first prototype MOVE processor chip, every data transport is guarded by a boolean expression. Usage of guards results in 1) more scheduling freedom, 2) support of multiway branching, and 3) branch removal, which results in longer basic blocks. We are looking at compiler techniques which effectively exploit these guards.

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 1.6μ CMOS Sea of Gates technology, operating at a 10 ns cycle time. This architecture allows for four concurrent transports per cycle. The design allows us to verify the hardware related advantages of MOVE architectures.

References

[BS88] e.a. Borkar S. iwarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, 1988.

[CM91] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albaquerque, November 1991.

[Do92] Daniel W. Dobberpuhl and other. A 200-MHz 64-b Dual-Issue CMOS Microprocessor. *IEEE journal of solid-state circuits*, 27(11), November 1992.

[Ebc88] Kemal Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, pages 1–21, Pisa, Italy, April 1988.

[HC92] 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.

[HCM91] Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software pipelining for transport-triggered architectures. In *MICRO-24*, Albaquerque, November 1991.

[i8689] Intel. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[i9688] intel. *80960KB Programmer's Reference Manual*, 1988.

[JW89] 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.

[Lip76] G. J. Lipovski. Architecture of a simple, effective control processor. In *Second Symposium on Micro Architecture*, pages 187–194, 1976.

[LS78] 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.

[LS90] Junien Labrousse and Gerrit A. Slavenburg. A 50MHz microprocessor with a Very Long Instruction Word architecture. In *ISSCC '90*, February 1990.

[MP89] 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.

[MS$^+$91] William Mangione-Smith et al. A performance comparison of the ibm rs/6000 and the astronautics zs-1. *IEEE computer*, January 1991.

[R$^+$89] B.R. Rau et al. The cydra 5 departmental supercomputer, design philosophies, decisions, and trade-offs. *IEEE Computer*, January 89.

[Rus78] Richard M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[RYYT89] R. Ramakrishna Rau, David W.L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 Departmental Supercomputer. Design Philosophies, Decisions and Trade-offs. *IEEE Computer*, pages 12–35, January 1989.

[Tra87] Trace: Technical summary. Multiflow Computer Inc., June 1987.