Software Pipelining for Transport-Triggered Architectures

Jan Hoogerbrugge Henk Corporaal Hans Mulder

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

Abstract

This paper discusses software pipelining for a new class of architectures that we call *transport-triggered*. These architectures reduce the interconnection requirements between function units. They also exhibit code scheduling possibilities which are not available in traditional operation-triggered architectures. In addition the scheduling freedom is extended by the use of so-called *hybrid-pipelined* function units.

In order to exploit this freedom, existing scheduling techniques need to be extended. We present a software pipelining technique, based on Lam's algorithm, which exploits the potential of transport-triggered architectures.

Performance results are presented for several benchmark loops. Depending on the available transport capacity, MFLOP rates may increase significantly as compared to scheduling without the extra degrees of freedom.

1 Introduction

Software pipelining is a well known technique for optimizing loops for superpipelined and VLIW like architectures. In software pipelining the next iteration of a loop starts before the current one ends; it aims at minimizing the initiation interval of successive iterations. Several algorithms for automatic software pipelining exist [1,2,3,4]. They differ in complexity and flexibility.

This paper describes and investigates software pipelining for a class of architectures which we call *transport-triggered* architectures. In *operation-triggered* architectures, instructions specify operations and require this operation to trigger the transport of source and destination operands. In *transport-triggered* architectures instructions specify the transport of data only and, therefore, require the transport to trigger the operations.

All recent RISC, VLIW, and superpipelined architectures can be classified as operation-triggered. The class of MOVE architectures presented in [5] and certain micro architectures (e.g., [6]) can be classified as transport-triggered.

As stated in [5] transport-triggered MOVE architectures have extra instruction scheduling degrees of freedom. This paper investigates if and how those extra degrees influence the software pipelining iteration initiation interval. It therefore adapts the existing algorithms for software pipelining as developed by Lam [2]. It is shown that transport-triggering may lead to a significant reduction of the iteration initiation interval and therefore to an increase of the MIPS and/or MFLOPS rate.

The remainder of this paper starts with an introduction of the MOVE class of architectures; it clarifies the idea of transport-triggered architectures. Section 3 formulates the software pipelining problem and its algorithmic solution for transport-triggered architectures. Section 4 describes the architecture characteristics and benchmarks used for the measurements. In order to research the influence of the extra scheduling freedom, the algorithm has been applied to the benchmarks under different scheduling disciplines. The next section (5) compares and analysis the measurements. Finally section 6 gives several conclusions and indicates further research to be done.

2 The MOVE class of architectures

MOVE architectures can be viewed as a number of function units (FUs) connected by some kind of transport network (see figure 1). By changing the type and number of FUs and the connectivity and capacity of the transport network a whole range of architectures can be implemented; MOVE architectures are therefore ideal for being tailored to application specific needs. Currently we are implementing a single chip MOVE prototype architecture.

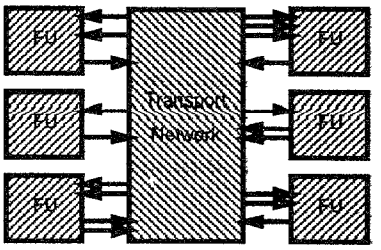


Figure 1: The MOVE architecture.

In the next subsections we first go into some more detail of the MOVE concept, explaining how to program this class of architec-

tures, and indicate some of its advantages (for a complete qualitative analysis see [5]). Second, the particular implementation of pipelined FUs is discussed. This implementation extends the code scheduling freedom.

2.1 The MOVE Concept

MOVE architectures are programmed by specifying the necessary transport of data values in the transport network. There is only one type of instruction: the move from FU to FU. In general, FUs are connected to the transport network by way of three types of registers, operand, trigger and result registers. All moves are than from register to register. General-purpose registers (GPRs) can be viewed as monadic-identity FUs. The function of the three register types is:

Operand registers: These act as normal general purpose registers, except when an operation is triggered (see below) on the accompanying FU its contents is used as an operand for the operation.

Trigger registers: A move to a trigger registers causes the start of an operation on the accompanying FU. The moved value is used as an operand of the operation. If more operands are needed, they are taken from operand registers of the FU. To handle different operations on a single FU, the trigger register is mapped into different logical address locations.

Result registers: Most FUs are fully pipelined, so an operation can be started in every cycle. When a result of an operation leaves the pipeline it is placed in the result register of the FU, from where it can be moved to other FUs.

All FUs contain at least one trigger type of register. Practical MOVE architectures handle multiple moves in parallel; in this sense they resemble VLIW architectures.

```
r_a → io        r_b → add       ; trigger addition
...                             ; space for other moves
r_c → sa        ir → sd         ; trigger store
```

Figure 2: The MOVE code for an addition followed by a store.

In order to make the MOVE concept more clear, an example is given in figure 2. The example shows the addition of GPRs r_a and r_b and a store of the result at memory address r_c. The code starts with moving r_a and r_b to the operand and trigger registers of the integer unit (registers *io* and *add*). The trigger register is accessed at the location that causes an addition on the integer unit. The operand move(s) must always be done before or in the same cycle as the trigger move. When the result of the addition leaves the pipeline it is available in the result register of the integer unit, in our example the latency of the integer unit is two, so the result move can take place two cycles after the trigger move. The result move passes the result directly to the trigger register of the memory unit. At the same time the address of the store is placed in the operand register of the memory unit. Thus a store is triggered with the data to be stored, and the address of the store is supplied via the operand register. Since a store has no result, no result move is needed. On average between 1.5 and 2 moves are necessary per operation.

Flow control operations are also done by moves. An absolute jump is simply writing the target address into the program counter, which is visible in the register address space. Relative jumps are provided by writing a displacement to the program counter displacement register. Both jumps have delay slots before instructions at the target address are executed.

Conditional execution is supported by guards. Each move can be guarded. A guarded move $g : r_a \rightarrow r_b$ only takes place when the guard g evaluates to true. In the prototype MOVE architecture g can be a boolean expression of two boolean values produced by the compare FU.

Advantages of the MOVE concept

The MOVE concept has several advantages both for implementation in VLSI and for generating high quality code. It will be clear that splitting traditional operations into more fundamental transport operations will offer extra opportunities for generating high quality code. The benefits for the scheduler in relation to software pipelining will become clear in the remainder of this paper. The most important implementation advantages are:

- Efficient transport capacity. On average we need far less than 3 data transports per operation; this means that with an equal amount of metal on a chip (transport busses require a large amount of chip area!) we can keep more function units busy.
- Very short cycle time. The cycle time of MOVE architectures is lower-bounded by the register to register transport time, which can be very short.
- Flexible architecture. It is very easy to tailor MOVE architectures to specific applications by changing the transport capacity (e.g. the number of busses), type of FUs, and the number of FUs.
- Simplicity. Despite its flexibility MOVE architectures are very easy to design. They are therefore suitable for automatic generation within a silicon compiler environment.

2.2 FU Pipelines

Most FUs within MOVE architectures are implemented using a so called *hybrid* pipelining mechanism. Figure 3 shows a typical integer pipeline with a latency of 3 clock cycles.

Current high-performance architectures generally implement pipelines where either intermediate pipeline stages continue always on each clock cycle to the next stage (e.g. [7,8]) or intermediate stages continue only in case a new operation is issued (e.g. [9]). In hybrid pipelines intermediate values continue to the next stage on each clock cycle, as far as they do not overwrite results from previous operations. This offers extra scheduling freedom; results may be moved from the FUs at any time after triggering the operation[1]. Hybrid pipelines also reduce register usage by using intermediate stages as temporary storage.

[1] Even before the result has been calculated; MOVE architectures implement a locking mechanism which avoids the insertion of no-op moves.

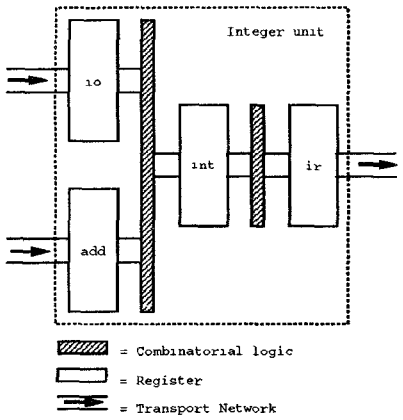


Figure 3: The pipeline of the integer unit.

3 Software Pipelining

Software pipelining is scheduling a loop in such a way that a next iteration starts before preceding iterations have finished. First, the conditions for a correct schedule and the basic algorithm for finding the optimal schedule are presented. The algorithm is based on the work of Lam [2]. Because MOVE architectures are transport-triggered and make the pipelines visible in the architecture, certain provisions have to be made to guarantee the correct pipeline usage. These provisions are described next.

3.1 Problem Formulation and Basic Algorithm

The instructions from the loop that needs to be software pipelined are represented as nodes in a directed graph $G = (V, E)$. The nodes represent instructions, or in our case moves. It is also possible that a node represents a collection of scheduled nodes. In that case we speak of hierarchical reduction (see [2]). Every node is supplied with a description of its resource needs. Between the nodes are arcs that describe precedence relations. Each arc (u, v) has two labels $d(u, v)$ and $p(u, v)$. The meaning of these two labels is that v needs to be scheduled at least $d(u, v)$ cycles after u of the $p(u, v)^{th}$ preceding iteration, that is:

$$\sigma(v) \geq \sigma(u) + d(u, v) - sp(u, v),$$

where $\sigma(v)$ is the cycle in which v is triggered, and s is the iteration initiation interval, this is the time between the initiation of two successive iterations.

The problem is finding a schedule σ with the lowest possible s that satisfies the following two constraints:

1. **Precedence constraints:** $\forall(u, v) \in E \ [\sigma(v) \geq \sigma(u) + d(u, v) - sp(u, v)]$.
2. **Resource constraints:** At every moment no more resources are used than there are available.

These two constraints give two lower bounds on the initiation interval.

The algorithm for finding a schedule for a given initiation interval is a variant of list scheduling; its basic form is given in figure 4.

```
ScheduleNodes(V, E, s)
begin
    v := AnElement(V)
    S := {v}
    σ(v) := 0
    return ScheduleNextNodes(V, E, S, s)
end

ScheduleNextNodes(V, E, S, s)
begin
    if V = S then return success
    v := AnElement(V − S)
    l := LowerBound(V, E, S, v, s)
    u := UpperBound(V, E, S, v, s)
    S := S ∪ {v}
    for σ(v) := l to u do
        if NoResourceConflicts(S, s) then
            if ScheduleNextNodes(V, E, S, s) = success then
                return success
    return fail
end
```

Figure 4: The basic algorithm for finding a schedule.

The differences with list scheduling are:

- If a resource is used at cycle t, it is also used at $t + ks$, where k is an integer number.
- In list scheduling a node may have a lower bound on σ (or an upper bound in case of bottom-up scheduling). However in software pipelining a node may also have an upper bound since G is cyclic. Each node has to be scheduled between its lower and upper bound.
- It is not always possible to schedule each node between its bounds due to resource constraints.

The algorithm starts with scheduling a node at cycle zero. Next a second node is taken and it is scheduled at a place between its lower and upper bounds where enough resources are available. The remaining nodes are scheduled in a similar fashion.

The two bounds of a node v are determined by the following two functions:

$$LowerBound(V, E, S, v, s) =$$

$$\max\{\sigma(u) + \sum_{e \in l}(d(e) - sp(e)) \mid l \text{ is a path from } u \in S \text{ to } v\}$$

$$UpperBound(V, E, S, v, s) =$$

$$\min\{\sigma(u) - \sum_{e \in l}(d(e) - sp(e)) \mid l \text{ is a path from } v \text{ to } u \in S\},$$

where S is the set of scheduled nodes.

In contrast to Lam we backtrack (to a certain level) if a node could not be placed, trying one of the earlier placed nodes at a later cycle. We found out that without backtracking schedules for transport-triggered architectures are too often non-optimal. Current research aims at improving the heuristics in order to avoid backtracking.

The function *AnElement*() in Lam's algorithm does not take a random element but uses a heuristic. The element that is selected is the one from the ready list with the lowest upper bound. The ready list is the following subset of $V - S$:

$$ReadyList(V, E, S) =$$

$$\{v \in V - S \mid \neg\exists u \in V - S : (u, v) \in E \wedge p(u, v) = 0\}$$

We have experimented with several other heuristics, and we have found a better one that uses the same ready list but another selection function. The selection function chooses the node in the ready list with the highest priority. This priority is determined by:

$$Priority(v) = \alpha\frac{r(v)}{s} - \beta(u(v) - l(v)),$$

where $r(v)$ is an indication of the scarceness of the resources that v uses, $u(v)$ and $l(v)$ are the bounds of v, and α and β are weight factors that should be determined with some experimentation. With this heuristic we also consider resource constraints. Since as s becomes larger the change for a resource conflict becomes less frequent, $r(v)$ is divided by s. With this heuristic we achieved an improvement of more than 5% (the average s/s_{min}) in comparison with Lam's heuristic when we do not apply backtracking.

When a graph is not strongly connected (there is no path between every pair of nodes) one or both bounds may not exist. To prevent this we add arcs to E that have a low $d - sp$. By doing this we can guarantee that all nodes have finite lower and upper bounds. Due to the low $d - sp$ we do not severly limit our scheduling freedom.

In Lam's algorithm, software pipelining takes place in two steps. In the first step all strongly connected components (SCCs) are scheduled, and in the second step the scheduled SCCs are reduced to nodes that form an acyclic graph. This graph is scheduled by a modified list scheduling algorithm. We did not use this two-step method because when a SCC is scheduled, it does not make use of the information in previously scheduled SCCs. By using more information it is possible to make better schedules. The drawback of our method is the increasing scheduling time. In practice there is not much difference since many loops contain only a single SCC.

3.2 Handling Pipelines

In transport-triggered architectures traditional instructions are split into their transport components. This, combined with the hybrid pipeline mechanism of MOVE architectures makes that the resource management of FUs has to be changed. It is no longer possible to see a pipeline as a single resource.

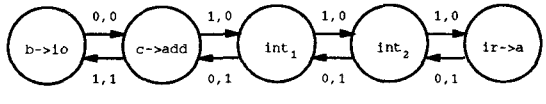


Figure 5: Precedence constraints between moves. Each precedence constraint is labeled with its d and p.

node	resources
$b \rightarrow io$	use move bus, claim operand register
$c \rightarrow add$	use move bus, claim trigger register
int_1	release operand and trigger register, claim internal stage
int_2	release internal stage, claim result register
$ir \rightarrow a$	use move bus, release result register

Table 1: The resources.

Figure 5 shows how we model an *Add* a, b, c operation for a 3-stage integer FU (as shown in figure 3). In the MOVE architecture $b \rightarrow io$ is a move from GPR b to the integer operand register io. Besides using one move slot in the instruction stream, and using a transport bus during one cycle, this move also claims the io register for one or more cycles. The io register is released one cycle after the move $c \rightarrow add$ to the trigger register. On its turn, this move claims the internal stage of the integer pipeline. Thus nodes can use, claim, or release resources.

Pipelines have properties that have to be modeled in precedence and/or resource constraints. The two properties are: (1) pipelines have a FIFO structure, and (2) pipelines have a finite storage capacity. We model these properties by introducing internal moves, and by treating pipeline stages as resources. Internal moves are moves from an operand register, a trigger register or a pipeline stage to a result register or a pipeline stage.

All four registers inside the unit are resources that need to be claimed and released afterwards. The precedence constraints between the moves for an addition are shown in figure 5, and the resources used, claimed and released by each node are summarized in table 1. For software pipelining we add the indicated backward arrows. They can be seen as an inter iteration WaR-dependency, and a forward edge can be seen as a intra interation RaW-dependency. We have added these backward edges to model some resource constraints by precedence constraints. These precedence constraints are used by the heuristic in *AnElement*(). Thus these extra arcs are not used for preventing incorrect schedules but for a better performing heuristic.

4 Benchmark results

The purpose of our measurements is to research the influence of transport-triggering and its extra scheduling degrees of freedom on the initiation interval for software pipelined loops. The next subsections first define the architecture used for experimentation, second the different scheduling disciplines and finally the measurement results for a couple of benchmarks.

4.1 Architecture Parameters

The MOVE architecture has parameters that are relative easy to change to the requirements of specific applications; e.g. transport capacity, number and type of FUs, latency of FUs, number of guards, and the number of GPRs. For the measurements in this paper we assume FU-parameters as listed in table 2.

With a latency of n cycles, we mean that if an operation is triggered in cycle t, the result can be read in cycle $t+n$. We assume 1

FU	operand	trigger	result	latency
load/store	*sa*	*la, sd*	*ld*	3
load immediate (16b/32b)	–	–	*ir1, ir2*	1
int addition/subtraction	*io*	*add, sub*	*ir*	2
fp multiplication	*fmo*	*fmul*	*fmr*	5
fp addition/subtraction	*fao*	*fadd, fsub*	*far*	4
logical	*lo*	*and, or, xor, not*	*lr*	1
shift	*sho*	*sll, slr, sar*	*shr*	2
compare (fast)	*gcof*	*geq, gne*	*X*[a]	1
compare (slow)	*gcos*	*gt, ge, hi, hs*	*X*[a]	2
jump, branch	–	*pc, pcd*	*pc, pcd*	2, 3[b]

[a] Not a result register, but a boolean used in the evaluation of a guard.

[b] A branch delay is one cycle longer than a jump delay.

Table 2: The functionality of our experimental model.

FU of each type, and a sufficient number of registers. The registers are 32 bit, and combined for both integer and floating point values. All registers are fully interconnected by a transport network containing 2 or more busses. Each move instruction uses 1 bus during 1 cycle. Three immediate formats are supported: 6, 16 and 32 bit. Short immediates of 6 bit are part of the regular move instructions (which are sized 16 bit in the MOVE prototype); longer immediates have to be read from the instruction registers (*ir*1 and *ir*2), which means that they occupy 1 or 2 move instruction slots.

4.2 Scheduling Disciplines

In order to study the effect of the extra scheduling freedom of the MOVE architecture we introduce six scheduling disciplines with increasing degrees of freedom. The first two disciplines are derived from traditional VLIWs; that means that every traditional instruction has been replaced by its corresponding moves, without applying possible extra optimizations which are introduced by this replacement. The remaining disciplines are transport-triggered models. They differ in the allowed scheduling freedom. This allows us to determine exactly where the extra speedup has to be contributed to. The disciplines are:

VLIW: Every instruction is projected on a fixed move pattern. In this pattern, operand and trigger moves are in the same cycle, and a result move has to take place exactly a fixed number of cycles later (depending on the latency of the FU).

VLIW-bp: This is VLIW with software bypassing. Software bypassing is overlapping result and trigger (or operand) moves of RaW-dependent operations. E.g. the result move $ir \rightarrow r_0$ can take place in the same cycle with the trigger move $r_0 \rightarrow sd$ when the source field of the trigger move is changed into *ir*. Software bypassing is the counter part of hardware bypassing which has the same goal, namely reducing the latency between RaW-dependent operations.

OTR-1: Extra dead code removal. Splitting instructions into their transport components offers extra optimization opportunities. Software bypassing introduces dead code. Many results which go directly to an operand or trigger register do not have to be moved to GPRs. The removal of this dead code both reduces transport and GPR requirements.

OTR-2: Loop invariant code motion. Operand moves which are invariant within the body of the loop may be placed in the loop prologue. This optimization is unique for transport-triggered architectures. To make this optimization more applicable, we always place immediates of commutative operations in the operand register.

TR: In contrast to the OTR-1/2 models, where the operand, trigger and result moves are scheduled at fixed relative positions (like in the VLIW models), the TR model allows all operand moves to be scheduled earlier than their corresponding trigger moves. Trigger and result move are still at fixed distance.

FREE: This model allows full scheduling freedom of both the operand and result moves. It makes use of the capabilities of the hybrid pipelined FUs which allow result moves to be scheduled at any time after the trigger move. Remark that for software pipelining it does not make sense to schedule a result move *earlier* than the latency of a FU would permit, except for code density.

Since we listed the models in an order of increasing scheduling freedom, the following relation holds for the software pipeline initiation interval *s*:

$$s_{VLIW} \geq s_{VLIW-bp} \geq s_{OTR-1} \geq s_{OTR-2} \geq s_{TR} \geq s_{FREE}$$

discipline	*b*	kernel 3	kernel 5	kernel 11	kernel 12	SAXPY	vec-inc
VLIW	2	10 (10)[a]	13 (13)	10 (9)	9 (9)	12 (12)	12 (10)
VLIW	3	7 (7)	9 (9)	6 (6)	6 (6)	8 (8)	10 (10)
VLIW	4	5 (5)	8 (8)	6 (5)	5 (5)	6 (6)	10 (10)
VLIW	5	5 (5)	8 (8)	5 (5)	4 (4)	5 (5)	10 (10)
VLIW-bp	2	10 (10)	13 (13)	10 (9)	9 (9)	12 (12)	11 (7)
VLIW-bp	3	7 (7)	9 (9)	6 (6)	6 (6)	8 (8)	8 (7)
VLIW-bp	4	5 (5)	7 (7)	5 (5)	5 (5)	6 (6)	8 (7)
VLIW-bp	5	4 (4)	7 (6)	5 (5)	4 (4)	5 (5)	8 (7)
OTR-1	2	10 (8)	12 (11)	9 (8)	9 (8)	10 (9)	11 (7)
OTR-1	3	6 (6)	8 (7)	6 (5)	6 (5)	8 (6)	8 (7)
OTR-1	4	5 (4)	7 (6)	5 (4)	5 (4)	6 (5)	8 (7)
OTR-1	5	4 (4)	7 (6)	4 (4)	4 (3)	5 (4)	8 (7)
OTR-2	2	7 (7)	10 (9)	7 (6)	7 (6)	8 (7)	8 (7)
OTR-2	3	5 (5)	7 (6)	5 (4)	5 (4)	7 (5)	8 (7)
OTR-2	4	4 (4)	7 (6)	4 (4)	4 (3)	6 (4)	8 (7)
OTR-2	5	4 (4)	7 (6)	4 (4)	3 (3)	5 (3)	8 (7)
TR	2	7 (7)	10 (9)	6 (6)	7 (6)	8 (7)	5 (4)
TR	3	5 (5)	6 (6)	4 (4)	5 (4)	6 (5)	4 (4)
TR	4	4 (4)	6 (5)	4 (3)	4 (3)	4 (4)	4 (4)
TR	5	4 (4)	6 (5)	3 (3)	3 (3)	4 (3)	4 (4)
FREE	2	7 (7)	10 (9)	6 (6)	7 (6)	7 (7)	5 (4)
FREE	3	5 (5)	6 (6)	4 (4)	5 (4)	5 (5)	4 (4)
FREE	4	4 (4)	6 (5)	3 (3)	3 (3)	4 (4)	4 (4)
FREE	5	4 (4)	6 (5)	3 (3)	3 (3)	4 (3)	4 (4)

[a] Minimum initiation interval s_{min} is shown between parentheses.

Table 3: The measured initiation intervals.

4.3 Benchmarks

We use six loops for studying the effect of different scheduling disciplines. Four of them are kernels from the Livermore loops[2]; the fifth loop is the SAXPY loop; the sixth loop is a loop that increments each element of a floating point vector by one (see figure 6). Because of its size the latter loop is included for analysis purposes presented in section 5. We shall call this loop 'vector-increment'.

Table 3 shows the initiation intervals of the six loops, for the six scheduling disciplines with full backtracking. The number of move busses *b* varies from 2 to 5. In addition to the measured initiation interval *s*, the minimal interval s_{min} (determined by the resource and predecence constraints) is shown between parentheses. It will be clear that for small *b*, s_{min} will be determined by the available transport capacity, while for large *b* the number of FUs or the precedence constraints bound s_{min}.

[2] We assume single precision floating point calculations.

```
for r0 := r10 to r11 do
begin
        r2 := memory[r0]
        r3 := r2 + 1
        memory[r0] := r3
end
```

Figure 6: The vector-increment example.

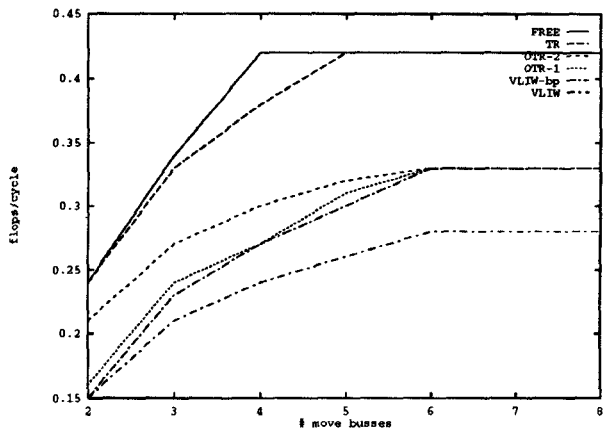


Figure 7: Flops/cycle for different scheduling disciplines and transport capacities.

To make the results more clear, a graphic summary of the table is shown in figure 7 (for b ranging from 2 to 8). This graph displays how the harmonic mean number of floating point operations (that is equal to the arithmetic mean of the number of FLOPS/cycle) increases as the number of move busses grows from two to eight. The graph is in correspondence with the inequality of the last subsection.

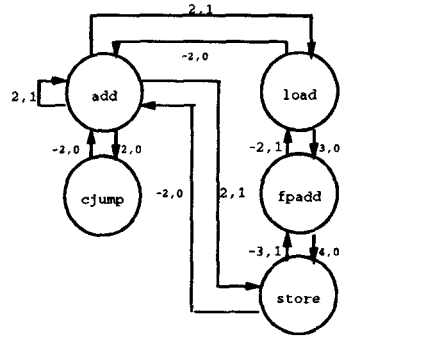
b	kernel 3	kernel 5	kernel 11	kernel 12	SAXPY	vec-inc
2	7 (7)	10 (9)	7 (6)	7 (6)	13 (7)	5 (4)
3	5 (5)	9 (6)	5 (4)	5 (4)	7 (5)	4 (4)
4	5 (4)	8 (5)	5 (3)	5 (3)	5 (4)	4 (4)
5	5 (4)	6 (5)	3 (3)	3 (3)	5 (3)	4 (4)

Table 4: The measured initiation intervals without backtracking for the FREE scheduling discipline.

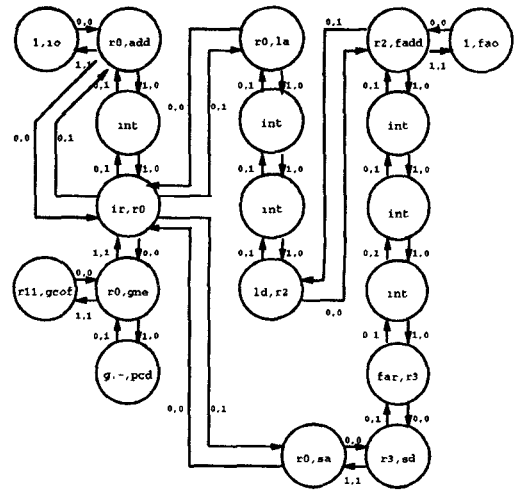
To indicate the effect of backtracking, we included table 4 that contains the results for the FREE discipline without any backtracking. The effect of backtracking on our benchmarks for the FREE discipline is that the average s/s_{min} decreases from 1.27 to 1.06. When we use Lam's heuristics for our benchmarks the average s/s_{min} would be 1.32.

5 Analysis of the results

As shown in figure 7, depending on the transport capacity, the MFLOP rates may increase more than 50% for a sample of livermore benchmarks. There are several independent factors contributing to this increase. In order to analyze these factors we look in detail at the vector-increment loop as shown in figure 6.



a: operation-triggered graph.



b: transport-triggered graph.

Figure 8: Data dependency graph for the vector-increment loop.

Figure 8 shows the data dependency graph for this loop: (a) for an operation-triggered architecture and (b) for a transport-triggered one. In these graphs it is assumed that the index addition is done after the store instruction. In figuring out the dependencies you should carefully consider all RaW and WaR hazards.

In going from the VLIW discipline to the FREE discipline the interval s for 2 move busses reduces from 12 to 5 cycles. The corresponding code schedules for VLIW and FREE are displayed in figure 9. We distinguish four important factors contributing to this drastic cycle reduction: (1) software bypassing, (2) dead code removal, (3) common subexpression elimination (CSE), and (4) code motion. They are discussed next.

Software Bypassing The difference between VLIW and VLIW-bp is the result of software bypassing. If there exists a RaW-

$r_{10} \rightarrow r_0$		
$r_0 \rightarrow la$		
$ld \rightarrow r_2$		
$1 \rightarrow fao$	$r_2 \rightarrow fadd$	
...		↑
...		
...		
$far \rightarrow r_3$		
$1 \rightarrow io$	$r_0 \rightarrow add$	
$r_0 \rightarrow sa$	$r_3 \rightarrow sd$	steady state
$ir \rightarrow r_0$		
$r_0 \rightarrow la$		
$r_{11} \rightarrow gcof$	$r_0 \rightarrow gne$	
$g : -9 \rightarrow pcd$		
$ld \rightarrow r_2$		
$1 \rightarrow fao$	$r_2 \rightarrow fadd$	↓
$far \rightarrow r_3$		
$1 \rightarrow io$	$r_0 \rightarrow add$	
$r_0 \rightarrow sa$	$r_3 \rightarrow sd$	
$ir \rightarrow r_0$		

a: VLIW discipline.

$r_{10} \rightarrow r_0$	$1 \rightarrow io$	
$r_0 \rightarrow add$	$1 \rightarrow fao$	
$r_0 \rightarrow la$		
$r_{11} \rightarrow gcof$		
$r_0 \rightarrow sa$	$ld \rightarrow fadd$	
$ir \rightarrow add$	$ir \rightarrow r_0$	↑
$r_0 \rightarrow la$	$r_0 \rightarrow gne$	
$g : -2 \rightarrow pcd$		steady state
$far \rightarrow sd$		
$r_0 \rightarrow sa$	$ld \rightarrow fadd$	↓
$ir \rightarrow add$	$ir \rightarrow r_0$	
$r_0 \rightarrow la$		
$far \rightarrow sd$		
$r_0 \rightarrow sa$	$ld \rightarrow fadd$	
$ir \rightarrow r_0$		
$far \rightarrow sd$		

b: FREE discipline.

Figure 9: The code for the vector-increment loop for 2 different disciplines.

dependency between a result move and an operand or trigger move, then without software bypassing the usage must be after the result move. So the d-label of the arc that describes this dependency is 1. However with software bypassing usage and result move may overlap, so the d-label now becomes 0; look e.g. in figure 8 between the moves $far \rightarrow r_3$ and $r_3 \rightarrow sd$. From the precedence constraints it follows that

$$s \geq \sum_{e \in c} d(e) / \sum_{e \in c} p(e) \qquad \forall \text{cycles } c \in G$$

The effect of software bypassing is a reduction of the 'length' ($\sum d_i / \sum p_i$) of the cycles in the graph. This means a lower s_{min} due to precedence constraints. In RISCs this bypassing is also available, however not in software but in hardware. For large VLIWs this hardware would be very complex. The transport-triggered architectures however do not need this hardware, they obtain the same result in software.

Removing Dead Code As a result of software bypassing dead code may result. E.g. in replacing $far \rightarrow r_3$ and $r_3 \rightarrow sd$ by $far \rightarrow sd$, it may happen that r_3 is no longer live. This occurs frequently when a the result move produces a temporary result that is used immediately by a few other operations. In each case a result move and a register usage is saved. In operation-triggered architectures there is no possibility to prevent a value to be written back to the register file when all its usages are through the bypassing hardware. In OTR-1 dead code is removed. In the schedule of figure 9b we see 2 examples of this optimalization. The $far \rightarrow r_3$ and $ld \rightarrow r_2$ moves have been removed.

Common Subexpression Elimination If source operands do not change in two successive calculations (using the same FU), the second operand move may be removed. In our example this results in 3 moves being placed out of the loop (loop invariant code motion is a special case of CSE), saving 3 move slots in the steady state (these moves are: $1 \rightarrow fao$, $1 \rightarrow io$ and $r_{11} \rightarrow gcof$). The effect of this optimization is measured in the difference between ORT-1 and OTR-2. As shown in figure 7 the savings are significant (more than 25% in case of a small transport capacity). To make this optimization more frequently applicable, constant operands (immediates in general) of commutative operations are supplied via the operand register.

Code Motion In going from OTR-2 to the FREE scheduling discipline we further relax the scheduling constraints for the operand and result moves. These moves no longer have to be put at fixed distances of the trigger move (as is the case in operation-triggered architectures). As shown in figure 7 there is an important gain by decoupling the operand move from the trigger move. Decoupling operand and trigger moves can even lower the minimal s due to predecence constraints. This can be explained by examining the graphs of vector increment in figure 8. The cycle that gives a lower bound for s is from the integer addition, via the load, fp addition, and store, back to the integer addition. In the top graph this cycle gives a lower bound of $7/1 = 7$ cycles. However for transport-triggered architectures with decoupled operand moves this bound is $8/2 = 4$ cycles. Thus decoupling causes a lower minimum s, and therefore a better s. Besides relaxing the precedence constraints, decoupling of operand and trigger moves reduces the resource requirements of both the transport network and the GPRs; it may eliminate extra moves to temporary registers, because results may stay longer in the pipeline, or may be directly put into operand registers i.s.o. GPRs.

Remark that precedence constraints can also be relaxed by using register renaming or a second pointer. However this introduces extra register and transport requirements (extra moves). This may increase s_{min} due to the resource constraints.

As will be clear from looking at the absolute MFLOP values in figure 7, the function units are underutilized. This means that we have to apply some combination of loop unrolling and register renaming in order to further reduce precedence constraints and to keep the FUs more busy. This is a topic of further research. Anyhow, the amount of required renaming is lower for transport-triggered architectures.

6 Conclusions and Research

In this paper we presented a method for software pipelining on transport-triggered architectures and compared the results with operation-triggered architectures. Measured in MFLOPS we observed speedups up to 50% depending on the transport capacity of the transport network connecting the function units and general purpose registers. The main factors contributing to this speedup are:

- Scheduling transport instead of operations gives extra scheduling freedom and allows for extra optimizations on the transport level like: operand and trigger move code motion, common subexpression elimination and loop-invariant code motion.
- On average the number of transport operations per operation is far less than 3. This means that if 3 busses are available per operation, their usage is rather low. In transport triggered architectures we have full control over bus usage. This means that, given a certain transport capacity, we can keep more function units busy.
- Modeling pipelines with internal nodes is an effective method for modeling hybrid-pipelined function units. It allows for decoupling operand and result moves from the trigger move, leading to reduced resource requirements on transport and general purpose register requirements, and also reducing precedence constraints.
- Bypassing is important for reducing precedence constraints. Transport-triggered architectures can realize bypassing in software; they do not need the bypassing hardware, which can become complex for large VLIWs.

Based on our experiments so far, we are rather optimistic about the use of transport-triggered architectures. We like to extend our research about software pipelining to larger architectures with more inhomogeneous transport networks. E.g. it is obvious that between integer and floating point units we do not need a large connectivity.

Currently we are developing a prototype transport-triggered MOVE architecture. A retargetable basic block scheduling compiler (based on GCC) and a simulator are already operational. The functionality, number of function units, and transport capacity can easily be changed. At present we try to integrate the software pipelining tool, as described in this paper, into this general compiler framework.

Research is going to find better heuristics that eliminate the need for backtracking. Another important research topic is the inclusion of register renaming and loop unrolling in order to lower precedence constraints and to keep function units busy.

References

[1] Alexander Aiken and Alexandru Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 308–317, Atlanta, Georgia, June 1988.

[2] Monica Lam. *A Systolic Array Optimizing Compiler. The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, Norwell, Massachusetts, 1989.

[3] Kemal Ebcioğlu and Toshio Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, University of Illinois at Urbana-Champaign, 1989.

[4] B. Su and J. Wang. Loop-carried dependence and the general urpr software pipelining approach. In *Proceedings of HICSS-24, Vol. 2*, pages 366–372, January 1991.

[5] Henk Corporaal and Hans (J.M.) Mulder. Move: a framework for high-performance processor design. In *Supercomputing-91, Albaquerque*, November 1991.

[6] Elliott I. Organick and James A. Hinds. *Interpreting Machines: Architecture and Programming of the B1700/B1800 Series. Operating and Programming systems series*, North-Holland, 1978.

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

[8] Trace: technical summary. Multiflow Computer Inc., June 1987.

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