Transport-Triggering vs. Operation-Triggering

Jan Hoogerbrugge and Henk Corporaal

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

Abstract. Transport-triggered architectures are a new class of architectures that provide more scheduling freedom and have unique compiler optimizations. This paper reports experiments that quantify the advantages of transport-triggered architectures with respect to traditional operation-triggered architectures.
For the experiments we use an extended basic block scheduler that is currently being developed. The paper gives an description of the scheduling techniques used.
The results of our experiments show that the extra scheduling freedom together with the new optimizations are very welcome when resource constraints are limiting the performance. This is the case when resources are scarce or when the application contains a lot of instruction level parallelism.

1 Introduction

Improvements in current computer architectures are obtained by higher clock frequencies, exploitation of instruction level parallelism (ILP), and improvements in compiler technology.

Higher clock frequencies are a result of better VLSI technology, better VLSI design techniques, and superpipelining which reduces the number of logic levels between pipeline stages.

ILP is made possible due to higher transistor densities which allows for duplication of function units and data paths. The exploitation of ILP consists of mapping the ILP of the application onto the ILP of the target architecture as efficiently as possible. This mapping, which consists of scheduling and resource allocation, can be done in hardware or in software. The hardware approach is used by superscalar architectures, which have advanced hardware to detect dependences and resource conflicts between operations. A major problem of this approach is the scalability of the scheduling hardware. The software approach is used by Very Long Instruction Word (VLIW) architectures. These architectures have instructions with multiple operations that are issued simultaneously. It is the responsibility of the compiler to fill these instructions such that data dependence and resource conflicts will not occur. The major problem with this approach is that the static information used by the compiler is in some ways less powerful than the dynamic information used by the hardware scheduler.

The third factor that can improve performance significantly is compiler technology. By designing an architecture such that the compiler has more control and information about the target hardware it is possible to develop new compiler optimizations. An example of this are the exact specifications of function units. When the number of stages and the pipelining degree are known to the compiler, the compiler can generate code such that pipeline stalls and bubbles are minimized.

The class of transport-triggered architectures (TTAs) that we advocate fit nicely into these developments. It allows for superpipelining, exploitation of (VLIW) ILP, and more compiler control about the hardware. The major difference between TTAs and traditional operation-triggered architectures (OTAs) lies in the way they are programmed. OTAs are programmed by specifying operations which result in data-transports between internal registers and function units. TTAs are programmed at a lower level, a TTA program specifies the data-transports directly. This results in a number of advantages which together improve the total performance significantly.

The MOVE project researches a framework for the generation of application specific processors (ASPs) by means of TTAs. Central within the framework is a precisely specified architecture template and a set of architecture parameters. A compiler for this framework has been developed. The purpose of this paper is to present an evaluation of both TTAs and the (scheduling) techniques used by the compiler.

The next section will introduce TTAs and the advantages of TTAs. Section 3 discusses the developed compiler and the scheduling techniques it uses. Section 4 describes the performed experiments and evaluates their results. Finally, section 5 gives conclusions and indications for further research.

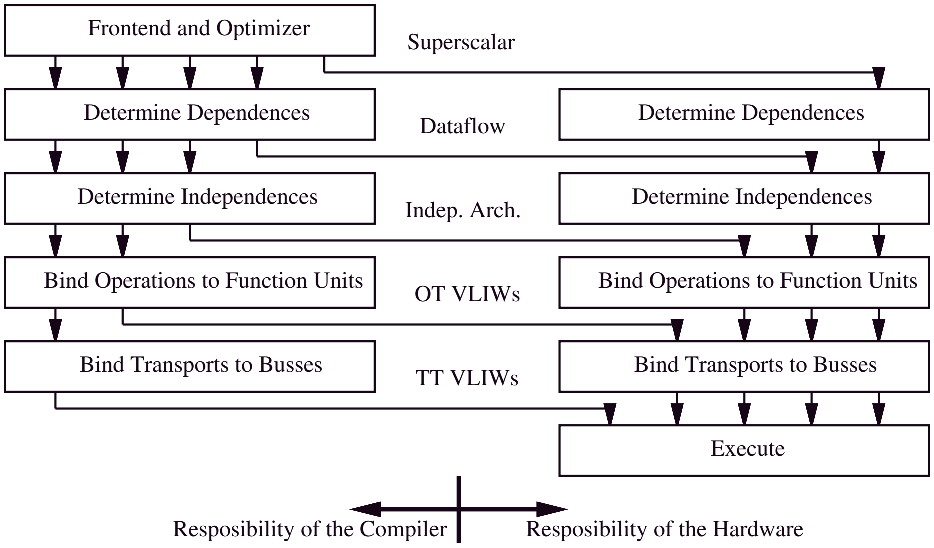


Fig. 1. The division of responsibilities between hardware and compiler.

2 Transport-Triggered Architectures

Transport-triggered architectures [CM91] trust the task of translating operations into data-transports to the compiler. This can be depicted as an extension of a scheme presented in [RF93]. Figure1 shows a division between the responsibilities of the compiler and the hardware. It shows five classes of architectures: superscalar, data flow machines, independence architectures [TS88], operation-triggered (OT) VLIWs, and transport-triggered (TT)

VLIWs. Superscalars shift most responsibilities to the hardware, while VLIWs shift most responsibilities to the compiler. The figure clearly shows that TT VLIWs depend even more on compilers than OT VLIWs.

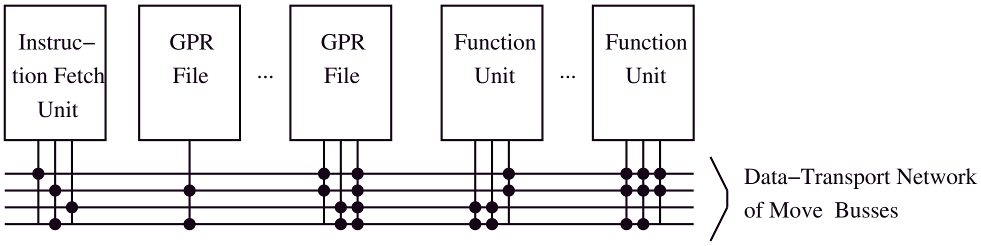


Fig. 2. The general structure of a TTA.

2.1 Principles

The compiler sees a TTA as a set of function units connected by a data-transport network of one or more move busses (see figure 2). To perform an operation the compiler generates moves to move the operands of the operation to registers of the function unit that will perform the operation, and a move to move the result of the operation from the function unit to a place where it will be stored or further processed. For example, an addition "$r4 = r2 + r3$" is translated into three moves:

$r2 \rightarrow add\_o$	/* Operand move from GPR file to FU */
$r3 \rightarrow add\_t$	/* Trigger move from GPR file to FU */
$add\_r \rightarrow r4$	/* Result move from FU to GPR file */

The first two moves specify that the contents of general purpose registers (GPRs) *r2* and *r3* are transported to input registers of the adder function unit. The third move transports the result of the addition to GPR *r4*.

One of the moves that move the operands to the function unit should signal the function unit that all operands are available and the operation can begin. This is called the *trigger move*, and the other moves that provide operands are called *operand moves*. The move that fetches the result is called a *result move*. A trigger move should be scheduled in the same or in a later cycle than the operand moves that belong to the same operation. The distance between a trigger and a corresponding result move should be at least the latency of the function unit.

A move instruction consists of multiple *move slots*; each slot specifies a separate transport. Immediate operands are provided by an (optional) immediate bit per move. This bit indicates whether the source field of the move is an immediate or it is a register specifier. Longer immediates are provided by making the instruction register (partly) accessible. Long immediates may or may not take the place of one or more move slots.

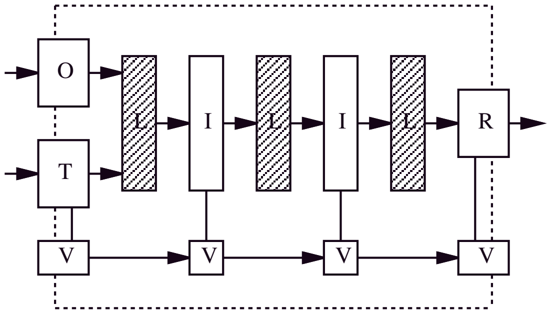
Control flow is provided by making the program counter accessible. Writing to the program counter causes a (possible delayed) jump. Reading the program counter is useful for saving the program counter during a function call.

Conditional execution is provided by means of predicated execution [AKPW83]. Moves can be predicated be a (simple) boolean expression of predicate registers. The move is only executed whenever the boolean expression evaluates to *true*. The predicate registers are stored in a special register file of single bit registers and are set by boolean values produced by compare function units. For example, the conditional jump "*if* $r2 = r3$ *goto label*" is translated into the following moves:

$r2 \rightarrow eq\_o$; $r3 \rightarrow eq\_t$ /* Operand and trigger move of compare */
$eq\_r \rightarrow b4$ /* Result move of compare to predicate register *b4* */
$b4$? $label \rightarrow pc$ /* Predicated jump to *label* */

2.2 Hybrid Pipelines

Transport-triggered architectures give the compiler more freedom to schedule moves from and to function units by means of *hybrid pipelines*. Each pipeline stage is accompanied with a valid bit that indicates whether the stage contains valid data or not. An operation proceeds to the next stage only when it does not overwrite data of a previous operation, i.e., the valid bit is off. The first pipeline stage consists of operand and trigger registers, the last stage corresponds to the result register(s). Figure 3 shows the organization of a function unit with a four stage pipeline.



Legend:
O = Operand register
T = Trigger register
R = Result register
V = Valid bit
I = Intermediate stage
L = Combinatorial logic

Fig. 3. A function unit with a four stage pipeline.

Hybrid pipelines allow that operations stay longer in the pipeline than is strictly needed to perform the operation. This extra scheduling freedom is useful for operations that are not on the critical path and that can be performed by function units that are not fully utilized.

In order to be able to perform different operations on a single function unit (e.g., addition and subtraction on an integer unit), a trigger register may be mapped at multiple address locations. This makes it possible to select an operation depending on the used address by the trigger move.

2.3 Advantages

The main advantages of TTAs are its extreme simplicity and great flexibility, allowing short processor cycle times and a quick design. Apart from design advantages [CM91], TTAs also have advantages with respect to the following issues:

1. **More scheduling freedom:** Dividing operations into individual transports of operands and results allows for better scheduling opportunities and therefore speeding up execution.
2. **Transport capacity:** Because the required transport capacity is designed on desired performance instead of worst case usage (three busses per RISC type operation), its utilization can be significantly higher.
3. **Register usage:** Many values can be bypassed from one function unit to another without needing to be stored in GPRs.
4. **Operand swapping:** Operands of commutative operations can be swapped by the compiler whenever that turns out to be useful. The operand that is moved last to the function unit will trigger the operation.
5. **Software bypassing:** Bypassing is scheduled by the compiler instead of scheduled by a hardware bypass unit.
6. **Dead result move elimination:** When all usages of a value are in the same cycle as the value is defined by a result move the result move to a GPR becomes dead code and can be eliminated.

Section 4 describes experiments that quantify some of these advantages using an extended basic block scheduler that is described in the next section.

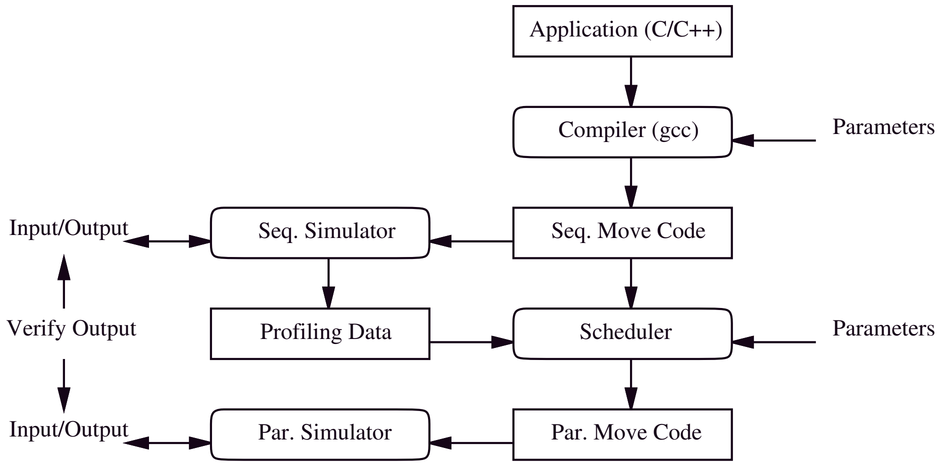


Fig. 4. The compiler trajectory.

3 The Extended Basic Block Scheduler

Our compiler trajectory consists of the GNU gcc compiler, a scheduler, and a simulator (see figure 4). The gcc compiler is ported to a generic sequential TTA. It has parameters that allow the user to strip the target operation set whenever that target ASP does not support certain operations (e.g., byte and half word loads and stores).

The scheduler is controlled by a machine description file that describes the target ASP. This description includes available function units, the operations that each function unit supports, latencies, GPR file sizes, and a description of the interconnection network.

A simulator for both sequential and parallel code is provided (1) to obtain profiling information, (2) to verify the compiler, (3) to verify the scheduler, and (4) to obtain application characteristics.

3.1 Scheduling an Executable

The input of our scheduler is a complete executable of generic MOVE code. The scheduler starts with reading the executable and its profiling data when present. After that the following steps are done to convert a sequential executable into a parallel executable:

```
schedule an executable
{
    perform inlining, unrolling, and peeling
    for each procedure do
    {
        perform data and control flow analysis
        perform memory reference disambiguation
        make a data dependence graph
        schedule each region
        flatten control flow graph
        perform register allocation
        reschedule modified basic blocks
    }
    relocate the executable
}
```

Function inlining, loop unrolling, and loop peeling are done in order to expose more parallelism for the scheduler. These code expansions are controlled by profiling data, the amount of code duplication, and a user specified parameter.

The data and control flow analysis step is pretty standard. It partitions a procedure into natural loops [ASU85] which correspond to *regions*. It performs live-analysis to detect off-liveness during inter-basic block code motions and to build an interference graph during register allocation. The data flow analysis step also assigns each live-range to an unique GPR. This should eliminate most of the false data-dependences.

The currently used memory disambiguator is very limited by the choice of using gcc[1]. It consists of making address derivations by means of reaching definitions produced by the data flow analysis phase. These derivations are used for the gcd-test in order to try to proof independence between two memory references.

After building a data-dependences graph for the whole procedure, the scheduler schedules each region of the procedure individually. Scheduling a region will be discussed in the next section.

The intermediate representation of the procedure consists of a control flow graph that does not require that some control flow edges have to be fall through. This makes inlining, unrolling, peeling, and some control flow transformations during scheduling easier to perform. However the basic blocks of the final code have to be placed sequentially into memory. It is therefore necessary to flatten the control flow graph. This step may introduce extra jumps.

Since the procedure is fully renamed by assigning each live-range to a unique GPR a large number of GPRs are in use. The register allocation step is used map these live-ranges back to physical GPRs. The register allocator inserts save/restore code whenever it runs out of GPRs. The basic blocks that are modified are rescheduled by a (simple) basic block scheduler. This scheduling before register allocation approach has the advantage that it decouples the two complex problems of scheduling and register allocation. The drawback is that it requires that all GPRs of the same type (integer, floating point, and predicate) should be in a single register file.

Since the size of the code is changed during scheduling, the executable has to be relocated. This consists of modifying addresses in the data and text segments of the executable.

3.2 Scheduling a Region

In order to find sufficient ILP to justify the cost of multiple function units and data paths, a scheduler should have a larger scope than a single basic block at a time. Various scheduling scope types that are larger than a single basic block are known from the literature:

1. **Trace:** A sequence of basic blocks that are likely to be executed after each other. A trace may have side entries/exits from/to other traces [Fis81].
2. **Decision tree:** A tree of basic blocks. A decision tree may have side exits, however only the root of the decision tree is allowed to have entries from other decision trees [HD86].
3. **Superblock:** A trace with the limitation that only the first basic block is allowed to have entries from other superblocks [MLC+92].
4. **Hyperblock:** A superblock created from an acyclic control flow graph by means of predicated execution [HMC+93].
5. **Region:** A set of basic blocks that corresponds to the body of a natural loop. Since loops can be nested, regions can also be nested in each other. Like natural loops, regions have a single entry point (the loop header) and may have multiple exits [BR91].

Our scheduler operates on regions which is the most general scheduling scope from the five mentioned above. More general means a larger scheduling scope, and potentially more and better exploitation of ILP.

[1] A lot of information that can be used by the disambiguator is not available in the current backend of gcc. This will become better in future releases of gcc.

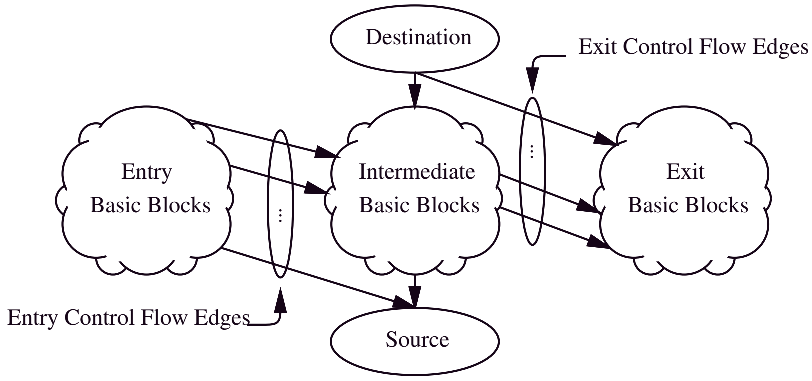


Fig. 5. A fragment of a region.

The scheduler schedules a region by visiting the basic blocks of the region in a topological order. Each basic block is scheduled using a basic block list scheduling algorithm, and after that the basic blocks that are reachable from the current basic block are examined for operations that can be scheduled in the current basic block. Thus the global algorithm for scheduling a region looks like:

```
schedule a region
{
    for each basic block b of the region in topological order do
    {
        1: schedule b's operations by means of list scheduling
        2: try to import operations from reachable basic blocks into b
    }
}
```

The first step is basically list scheduling modified for TTAs. A data-dependence graph (DDG) and a set of *ready* operations are used. An operation is ready whenever all its predecessors in the DDG are scheduled. The list scheduler repeatly takes an operation from the ready set and schedules it at the first cycle where data-dependences and resource constraints are satisfied. The selection from the ready list is based on the critical path heuristic used by almost every other list scheduler.

The second step examens all operations from basic blocks in the current region reachable from the current basic block. The priority function that is used to select a ready operation looks like:

$$global\_priority = probability * local\_priority - duplication\_cost$$

where $probability$ is the probability (estimated or based on profiling data) that the operation will be executed after the current basic block is executed, $local\_priority$ is the critical path priority function scaled between 0 and 100, and $duplication\_cost$ is a parameter that is subtracted whenever the inter-basic block code motion requires code duplication. The scheduler tries to schedule the selected operation in the current basic block. This is repeated until all ready operations from reachable basic blocks have been tried.

Code motion from one basic block *source* to another basic block *destination* requires code duplication whenever *destination* does not dominate [ASU85] *source*, and requires off-lives checks whenever *source* does not post-dominate *destination*.

Figure 5 shows what needs to be done when an operation is taken from *source* and placed in *destination*. First the scheduler determines all intermediate basic blocks in the current region between *source* and *destination*. All immediate predecessors of *source* and of the intermediate basic blocks that are not reachable from *destination* require a duplication. This is the set of *entry basic blocks* in figure 5. Whenever it is not possible to place one of these duplications the whole code motion from *source* to *destination* is aborted.

The off-liveness checks consists of testing the immediate successors of *destination* and the immediate successors of the intermediate basic blocks from which *source* is not reachable. These blocks are shown in figure 5 as *exit basic blocks*. Whenever the value that an operation defines is live at the beginning of one of the exit basic blocks the operation needs to be predicated by the predicate register that controls the corresponding exit control flow edge. Since our scheduler assumes only predicate expressions of a single predicate register (b and $\overline{b}$), the number of exit basic blocks which require predicated execution is limited to one. If predicated execution is needed an extra (*flow*) dependence is introduced between the operation and the (compare) operation that defines the predicate. A memory store operation can be viewed as a normal operation that defines *memory*, which is always live.

3.3 Scheduling an Operation

Scheduling an operation[2] consists of scheduling the moves that perform the operation. Each operation consists of one trigger move, zero or more operand moves, and zero or one result moves (only a store has no result).

The scheduler starts with scheduling the trigger move. It has to find the first cycle where (1) a suitable function unit is available, (2) a suitable move bus is available, and (3) an immediate can be provided when needed.

After the trigger is scheduled the scheduler proceeds with scheduling the operand and result moves. This consists of finding a cycle where a suitable move bus is available and when needed an immediate for an operand move. During this step the scheduler has to make the following checks in order to use the hybrid pipelines correctly:

1. Trigger and result moves should be scheduled in first-in first-out order.
2. The number of outstanding operations on a function unit may not exceed the capacity of the function unit.
3. An operand register is in use until the operation proceeds to the second pipeline stage.

All these checks are verified by means of a resource table that keeps track of the state of each function unit. When one of the checks fails the (operand or result) move has to be placed in an earlier (in case of an operand move) or later cycle (in case of a result move).

[2] For example, an addition, multiply, load, or store. Jumps, function calls, OS calls, and copy operations are handled differently.

There are cases where this does not help. In these situations the scheduler has to unschedule the operation and delay the trigger move by one cycle.

Whenever the scheduler schedules an operand or trigger move the scheduler checks whether the value used by the move needs to be bypassed from the function unit that produced it instead of getting the value from a GPR. If bypassing is required the source field of the move is changed. Furthermore, a check is made to find out whether the result move of the operation that produced the value has become dead and can be eliminated. This happens when all usages of a value are in the same cycle as the result move that produced the value. A dead result move is marked as being dead and its move bus is released. When the scheduler has to delay the operand or trigger move it restores the source field and brings a killed result move back to live.

When an operation is imported from a reachable basic block into the current basic block it is not allowed to increase the length of the current basic block. This guarantees that extended basic block scheduling is always better than basic block scheduling. However, when a result move reaches the "bottom" of the current basic block it is allowed to move to the immediate successor(s) of the current basic block. Similarly, operand moves are allowed to move to immediate predecessor(s) of the current basic block when they hit the "top" of the current basic block. Due to these two code motions it is possible that operations are in progress during the transition of one basic block to another.

4 Experimental Results

The goal of our experiments is to measure the quality of (the current state of) our scheduler, and evaluate some aspects of transport-triggering. We shall perform three experiments. The first experiment will measure the difference between basic block and extended basic block scheduling, the second experiment will measure the effect of scheduling freedom, and the third experiment compares transport-triggering with operation-triggering.

We shall focus on workstation type applications, which are characterized by a limited amount of ILP and an irregular data driven control flow. For a discussion of scientific code and TTAs see [HCM91, HC92].

4.1 Architecture and Benchmarks

Since we are focussing on workstation code we shall assume an architecture with short latencies. Table 1 shows which function units are available in our experimental architecture together with their number and latency. Other assumed architecture parameters are:

1. A fully connected interconnection network. For our experiments we shall vary the number of move busses between one and eight.
2. No branch delay and predicate registers can be used in the same cycle as they are defined. This corresponds to same conditional branch latencies as most RISC CPUs (e.g., MIPS R2000). For example, a branch-if-equal looks like:

   $r2 \rightarrow eq\_o$; $r3 \rightarrow eq\_t$ /* Operand and trigger move of compare */
   eq_r ? $label \rightarrow pc$ /* Predicated jump without delay slots */

3. Sufficient GPRs. We have to assume this since our register allocator is not fully operational.
4. No base-offset memory addressing mode. Since many memory addresses are the sum of two components (about 40–60%), the actual load latency is about 2.5 cycles when compared with an architecture that supports base-offset addressing.
5. 6-bit short immediates and one 32-bit long immediate per instruction.

The following set of twenty benchmarks is used for our experiments. It should reflect a typical workstation load:

a68, bison, cksum, cmp, compress, cpp, dhrystone, diff, expand, flex, gnuplot, gunzip, gzip, od, sed, stanford, uncompress, unexpand, uniq, and *virtex*.

Most of the benchmarks are GNU utilities, the exceptions are *compress, uncompress, virtex, dhrystone*, and *stanford*.

Function unit	Number	Latency
Load/Store	2	2
Integer	2	1
Shift, sign-extend	1	1
Integer compare	1	1
Logic	1	1
Floating point	1	3
Floating point compare	1	3
Floating point convert	1	3

Table 1. Assumed function unit parameters.

4.2 Effect of Extended Basic Block Scheduling

Figure 6 shows the effect of a larger scheduling scope. The values plotted are the average speedups of the twenty benchmarks relative to basic block scheduling for a single move bus machine. The figure clearly shows that more ILP in the hardware requires extended basic block scheduling in order to use the hardware effectively. The difference between basic block and extended basic block is about 40% for an eight move bus machine. The main reasons we observed why the speedup is sub-linear are:

1. Due to speculative execution some moves are not always useful.
2. ILP of workstation code is limited [Wal91].
3. ILP of workstation code is not very smoothable [TGH92].
4. Memory references of workstation code are hard to disambiguate.

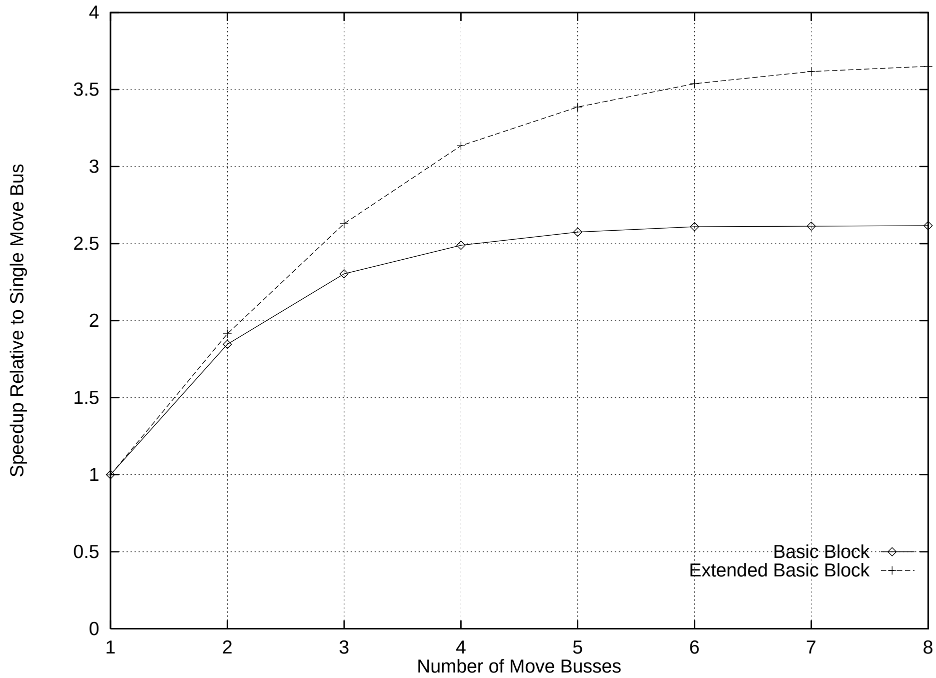


Fig. 6. Basic block vs. extended basic block scheduling.

4.3 Effect of Scheduling Freedom

OTAs deliver all operands of an operation in the same cycle and fetch the result as soon as it is available. TTAs have more scheduling freedom, a trigger move can be scheduled after an operand move and operations can stay longer in the pipeline if the scheduler "thinks" that is wise. To investigate the effect of the extra scheduling freedom of TTA we introduce four scheduling models:

1. **OTR:** The distance between all moves of the same operations is fixed.
2. **OT:** The distance between operand and trigger moves of the same operation is fixed.
3. **TR:** The distance between trigger and result moves of the same operation is fixed.
4. **FREE:** No restrictions on the distance between the moves of an operation.

Figure 7 shows the DDGs of a n cycle dyadic operation for the four different models. It shows that OTR has more dependences than OT and TR, and FREE has less dependences. Less dependences means more freedom and potentially better results.

Figure 8 shows the result of scheduling the benchmarks according to the four scheduling models. It shows that the extra scheduling freedom is profitable when the transport capacity is small (less than five busses), in other words when resource constraints are limiting the performance. It can be expected that for applications with more parallelism, and when the scheduling techniques are improved, the point where the four lines of figure 8 merge will shift to the right. In other words, the extra scheduling freedom will become more important.

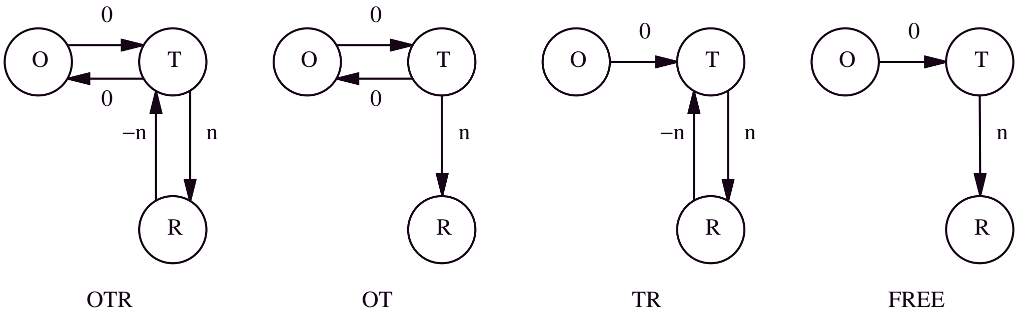


Fig. 7. The DDGs of a dyadic operation for the four scheduling models.

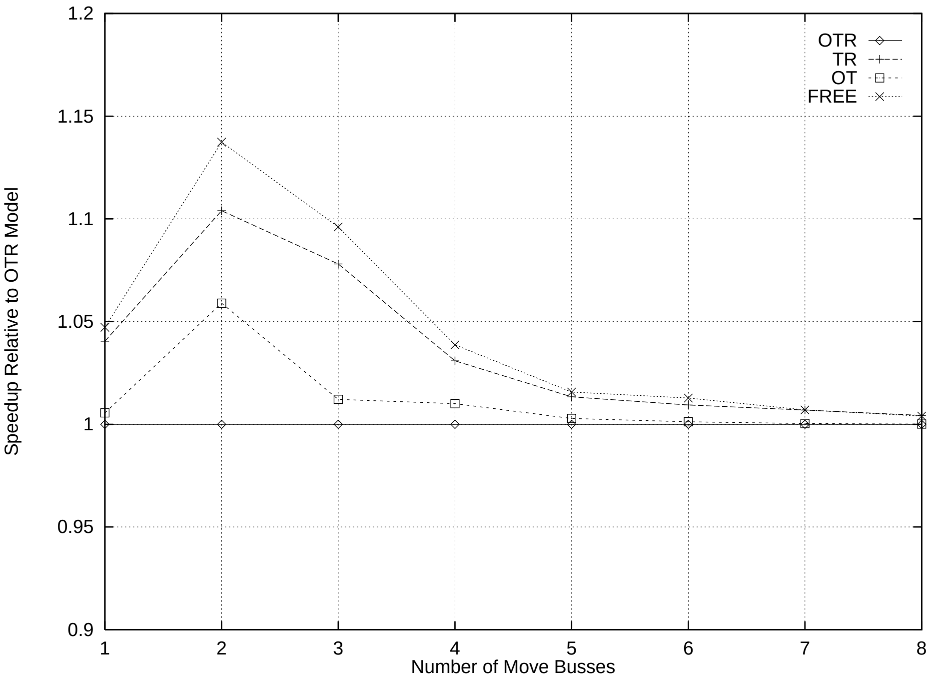


Fig. 8. Four different degrees of scheduling freedom.

4.4 Effect of Transport-Triggering vs. Operation-Triggering

Our last experiment will compare an n-issue TTA with an m-issue OTA for $1 \leq n \leq 8$ and $1 \leq m \leq 3$ (n is the number of concurrent transports, m is the number of concurrent operations). Since n-issue OTA needs $3n$ busses it is interesting to compare a $3n$-issue TTA with an n-issue OTA. We model an n-issue OTA for our scheduler as a TTA with an infinite number of busses and an extra restriction of at most n operation per cycle. Figure 9 shows the results. The three horizontal lines correspond to one, two, and three issue OTAs. It is interesting where they cross the curved line that represents an one to eight issue TTA. These crossing let us make the following conclusions:

1. A $3n$-issue TTA performs better than an n-issue OTA.
2. A TTA needs less than $3n$-issue to compete with an n-issue OTA.

As in section 4.3 the benefit of transport-triggering is most expressive when resource constraints are limiting the performance.

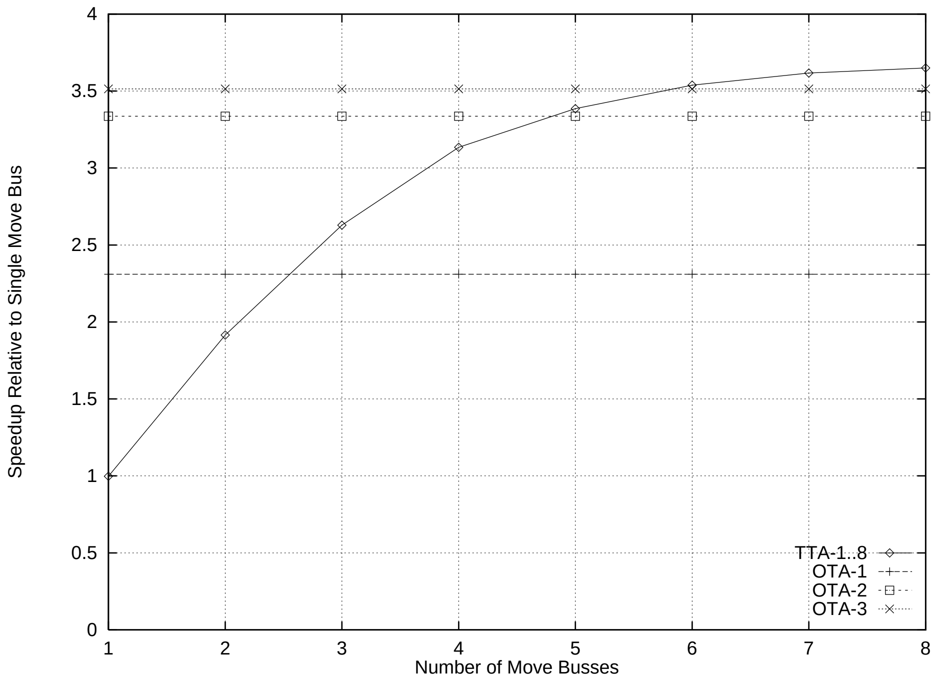


Fig. 9. Transport-Triggering vs. Operation-Triggering.

5 Conclusions

We described (1) a new class of architectures called TTAs with their advantages, and (2) an extended basic block scheduler for these TTAs. With this scheduler we did some experiments to measure how it compares to basic block scheduling and how TTAs compare to traditional OTAs.

We observed a 40% higher speedup for extended basic block scheduling instead of basic block scheduling for an eight move bus TTA. Further research should find out how this can be improved by better memory reference disambiguation, more flexible code motions, and software pipelining instead of loop unrolling.

We also observed significant better results due to transport-triggering instead of operation-triggering. This is especially true for the cases where resources are scarce and it is hard for the scheduler to map the ILP of the application on the ILP of the hardware. These are precisely the architectures that are most cost-effective and for which instruction scheduling is most challenging.

References

[AKPW83] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Januari 1983.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[BR91] David Bernstein and Michael Rodey. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[CM91] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A Framework for High-Performance Processor Design. In *Supercomputing-91, Albaquerque*, November 1991.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[HC92] Jan Hoogerbrugge and Henk Corporaal. Comparing Software Pipelining for an Operation-Triggered and a Transport-Triggered Architecture. In *International Workshop on Compiler Construction*, Paderborn, Germany, October 1992.

[HCM91] Jan Hoogerbrugge, Henk Corporaal, and Hans Mulder. Software Pipelining for Transport-Triggered Architectures. In *Proceedings of the 24th Annual Workshop on Microprogramming*, pages 74–81, Albaquerque, New Mexico, November 1991.

[HD86] Peter Y.T. Hsu and Edwards S. Davidson. Higly Concurrent Scalar Processing. In *Proceedings of ISCA 13*, pages 386–395, June 1986.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique fro VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–249, May 1993.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual Workshop on Microprogramming*, pages 45–54, Portland, Oregon, December 1992.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7(1/2):9–50, May 1993.

[TGH92] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. One the Limits of Program Parallelism and its Smoothability. In *Proceedings of the 25th Annual Workshop on Microprogramming*, pages 10–19, Portland, Oregon, December 1992.

[TS88] M.R. Thistle and B.J Smith. A Processor Architecture fo Horizon. In *Proceedings of Supercomputing '88*, pages 35–41, Orlando, Florida, November 1988.

[Wal91] D.W. Wall. Limits of Instruction-Level Parallelism. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.