Register File Port Requirements of Transport Triggered Architectures

Jan Hoogerbrugge Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {janh,heco}@duteca.et.tudelft.nl

Abstract

Exploitation of large amounts of instruction level parallelism requires a large amount of connectivity between the shared register file and the function units; this connectivity is expensive and increases the cycle time.

This paper shows that the new class of transport triggered architectures requires less ports on the shared register file than traditional operation triggered architectures. This is achieved by programming data-transports instead of operations.

Experiments with our extended basic block scheduler have shown that the reduction of the required number of register file ports is substantial. The average requirement for scalar applications is 0.50 read and 0.35 write ports per operation instead of 2 read and 1 write ports. Due to this reduction it is possible to execute 2 operations per cycle with a two-ported register file and 3.6 operations per cycle with a six-ported register file.
Keywords: Transport triggered architectures, VLIW, ILP, register ports, scalability

1 Introduction

Traditional very long instruction word (VLIW) processors have a major problem that limits their scalability. The problem is the connection between the shared general purpose register file (GPRF) and the function units (FUs). Assuming a 2-operand 1-result operation set, each FU requires two read busses and two read ports on the GPRF to fetch operands and it requires one bus and one write port to write back produced results. When the number of FUs n becomes larger the $3n$ busses and ports become very expensive in terms of chip area and power consumption and will have a serious negative impact on the achievable cycle time. Due to pipelining there is also a bypass circuit required to reduce the delay between flow dependent operations. This bypass circuit has at least $O(n^2)$ complexity, again a serious scalability bottleneck.

An obvious way to solve this problem is partitioning the shared GPRF and FUs into clusters of a single GPRF with a few FUs and adding busses for communication between the clusters [CDN92]. This solution has several problems: (1) inter-cluster communication requires operations and time, (2) inter-cluster communication requires also busses and GPRF ports, (3) not all code is very well clusterable (most heavily unrolled numerical code loops are clusterable but for non-numerical is it questionable), and (4) it makes compilation for VLIWs more complicated.

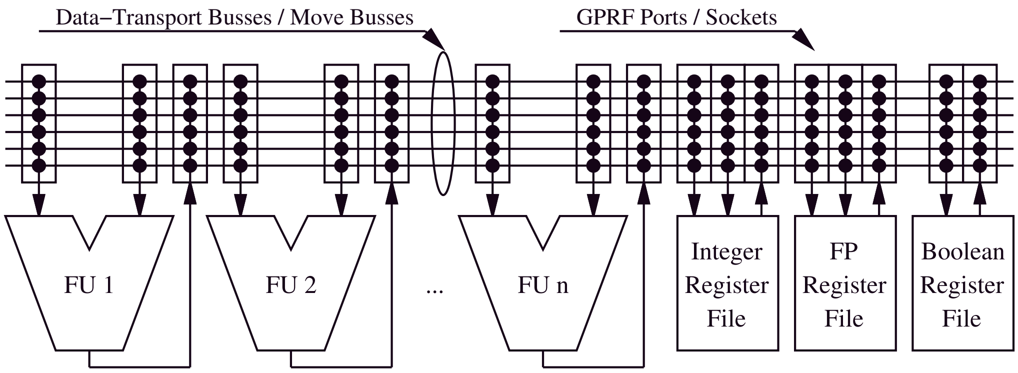

Figure 1: The general structure of a TTA.

Our solution to cut down the number of busses, GPRF ports, and bypass circuit complexity is called *transport triggering* as opposed to traditional *operation triggering*. The idea is that we do not program a processor by specifying operations, which cause implicit data-transports between GPRFs and FUs, but we specify the data-transports directly. In a recent paper we have shown that transport triggered architectures (TTAs) need less busses and do not need associative bypass circuitry [HC94]. In this paper we will show that TTAs need significantly less GPRF ports than traditional operation triggered architectures (OTAs) as well. We shall describe why TTA need less GPRF ports and report experiments that quantify the reduction.

So far we have only been talking about VLIWs and not about the other important class of instruction level parallel (ILP) processors namely superscalars. However superscalar processors suffer from the same problem as described above and it is unlikely that GPRF partitioning or transport triggering is applicable to superscalars.

The remainder of this paper is organized as follows. Section 2 introduces TTAs. Section 3 explains why TTAs need less GPRF ports than OTAs. Section 4 describes our extended basic block scheduler briefly. Section 5 reports experimental results. Section 6 describes related work. And finally, section 7 gives the conclusions of this paper.

2 Transport Triggered Architectures

A transport triggered architecture [CM91] consists of a set of FUs, GPRFs, *move busses*, and *sockets*. The FUs perform the operations, the GPRFs provide storage, the move busses perform the transports between the FUs and the GPRFs, and the sockets interface FUs and GPRFs to move busses. All these components are directly visible for the compiler. Figure 1 shows the general structure of a TTA, having n FUs, six move busses, three-ported integer and floating point GPRFs, and a two-ported boolean register file.

Programming an operation on a TTA consists of moving operands to the input registers of an FU that is capable to perform the operation, and moving the result from the result register of the FU to a GPRF or an FU after the operation has been performed. The input registers of FUs are divided into operand and trigger registers. The purpose of operand registers is only to deliver operands; writing to a trigger register also signals the FU that all operands are available and that the operation can start. Reading from a result register is also a signal to the FU that the operation is finished and the result of a pending operation can enter the result register. To make

TTA programming more clear let us consider how to translate the following code into *move code* and how to schedule it.

```
add  r1, r2, r3       /* r1 = r2 + r3 */
sub  r4, r5, r6       /* r4 = r5 - r6 */
st   r4, r1           /* store r4 at address r1 */
```

First we translate each n-operand m-result operation into $n-1$ operand moves (moves to an operand register), one trigger move (a move to a trigger register), and m result moves (moves from a result register):

```
r2 -> add_o; r3 -> add_t; add_r -> r1
r5 -> sub_o; r6 -> sub_t; sub_r -> r4
r1 -> st_o; r4 -> st_t
```

The suffixes '_o', '_t', and '_r' indicate the register type. Scheduling consists of assigning FUs to operations and assigning cycles, sockets, and move busses to moves. Lets assume we have two FUs named `alu1` and `alu2` for ALU operations, and one FU named `ls` for load-store operations.

```
r2 -> alu1.add_o; r3 -> alu2.add_t; r5 -> alu2.sub_o; r6 -> alu2.sub_t
alu1.add_r -> r1; alu2.sub_r -> r4; r1 -> ls.st_o; r4 -> ls.st_t
```

The prefixes '`alu1.`', '`alu2.`', and '`ls.`' indicate the FU on which the operation is executed. The code that we have produced so far is not correct. The problem is that the last two moves in the second instruction need bypassing. This consists of modifying the source fields. Due to this bypassing the first two moves of the second instruction become dead since they write values to the GPRF that are not used anymore. After bypassing and removing dead result moves the final code becomes:

```
r2 -> alu1.add_o; r3 -> alu2.add_t; r5 -> alu2.sub_o; r6 -> alu2.sub_t
alu1.add_r -> ls.st_o; alu2.sub_r -> ls.st_t
```

From this small example we can make an important observation. We did not need 3 moves and 3 GPRF accesses for each operation. In this example we totally needed 6 moves and 4 GPRF accesses for three operations instead of 9 transports and 9 GPRF accesses.

Besides executing operations on register operands we need three other mechanisms to be able to compile realistic applications. These are immediate operands, non-sequential control flow, and conditional execution. Small immediates are provided by adding an immediate bit to each move slot of the instruction format that indicates whether the source field of the move is an immediate or a register specifier. Longer immediates are provided by making the instruction register partially accessible. Non-sequential control flow is provided by making the program counter accessible; writing to it causes a jump, reading it is useful for obtaining the return address of a function call. Conditional execution is provided by means of predicated execution. A GPRF of booleans is used to store boolean results of comparisons. Each move can be predicated by an expression of booleans. Only if the boolean expression evaluates *true* the accompanying move is executed.

3 Why Do TTAs Need Less GPRF Ports?

There are several reasons why TTAs need less than $3n$ GPRF ports to keep n FUs busy. It is a superset of the reasons why TTAs need less than $3n$ busses.

1. **Not all operations require two GPR operands**, examples are register to register copies, operations with immediate operands, jumps, and calls.

2. **Not all operations produce a result**, examples are jumps, calls, and stores.

3. **Bypassing** replaces a GPR in a source field of a move by the result register of an FU. Bypassing occurs when a value is used in the cycle in which it is defined.

4. **Dead result move elimination** removes useless write backs to a GPRF. It occurs when all usages of a value are bypassed, i.e., all usages of a value are in the same cycle in which it is defined.

5. **Operand sharing** means that an operand move can be eliminated because the value that it moves is already present in the destination operand register. For example, in the following code fragment the second operand move is useless and can be eliminated since the value of `sp` is already in `alu.add_o`.

   ```
   sp -> alu.add_o; 4 -> alu.add_t
   sp -> alu.add_o; 8 -> alu.add_t; alu.add_r -> ls.ld_t
   alu.add_r -> ls.ld_t; ls.ld_r -> ...
   ls.ld_r -> ...
   ```

   Most of the shared operands are small immediates (1, 0, -1, 4, etc.) or the stack pointer. When the shared operand is a GPR a GPRF read access is eliminated.

6. **Socket sharing** means that the usage of a GPRF read port is shared by multiple reads. It occurs when multiple moves read the same GPR in the same cycle. Sockets are designed that they can be selected from multiple busses as long as these busses access the same register. For example, in the following instruction a socket is shared between the first and the third move since they both access the GPR `r1`.

   ```
   r1 -> alu1.add_o; 4 -> alu1.add_t; r1 -> alu2.add_o; 8 -> alu2.add_t
   ```

   Since our current pipeline scheme selects the sockets before predicates are available and evaluated it is not possible to share a socket between moves that access different GPRs but have mutual exclusive predicates (at most one of the predicates can evaluate *true*), for example

   ```
   b0: r1 -> alu1.add_o; !b0: r2 -> alu2.add_o
   ```

   This form of socket sharing requires another pipeline scheme that would increase the latency between the define and the use of a boolean for predication by one cycle.

As an extreme example that shows the difference between the actual and the worst case requirements let us consider the following software pipelined [Lam88] loop[1] that increments the elements of an integer vector:

[1] For simplicity we have omitted the loop control operations.

```
    ld r1, r2; add r2, r2, 4
    ld r1, r2; add r2, r2, 4; add r3, r1, 1
 L: ld r1, r2; add r2, r2, 4; add r3, r1, 1; st r3, r4; add r4, r4, 4
                              add r3, r1, 1; st r3, r4; add r4, r4, 4
                                             st r3, r4; add r4, r4, 4
```

An OTA would need at least 5×3 GPRF ports. However during the steady state of the loop (the third instruction) all operands are immediates or bypassed register operands and all write backs are dead. Therefore the actual GPRF port requirement during the steady state of the loop is null!

By explicitly controlling bypassing and write backs we have created another level in the memory hierarchy. A compiler should try to move all values directly from one FU to another without using GPRs. When this fails it uses GPRs for spilling.

4 Our Extended Basic Block Scheduler

Our code generation software for TTAs consists of the GNU C/C++ compiler which produces sequential move code and a parameterized extended basic block scheduler which transforms the sequential code into parallel code (see figure 2). Our scheduler, which took most of our effort, has the following properties:

- Like [BCK91, ME92] our scheduler has a region as scheduling scope, which is more general than other scheduling scopes such as superblocks [HMC$^+$93], traces [Fis81], and decision trees [HD86].

- It performs 'smart' register allocation before scheduling [Pin93]. It is called smart because the register allocator attempts to assign registers without creating false dependences for the scheduler.

- It uses profiling information when available. Profiling information is used for speculative execution, controlling function inlining and loop unrolling, and to compute spilling costs during register allocation.

- It applies predicated execution for speculation of stores and operations that overwrite off-live registers. Due to predicated execution it occurs frequently that small basic blocks disappear because all its operations are moved upwards.

- It applies a limited form of multi-way branching. With our current predicated execution scheme, which limits boolean expressions to expressions of one or two booleans, we can only handle three-way branches (two taken paths and one fall through path).

- It is parameterized. All information about the target TTA is read from a machines description file. This information includes FU latencies, FU functionality, the number of booleans, the number of GPRs, the number of sockets, the number of move busses, and the available connections between sockets and move busses.

- It assumes one GPRF for integers, one for floating point numbers, and one for booleans. Scheduling for multiple GPRFs of the same type (partitioned GPRFs) remains a research point.

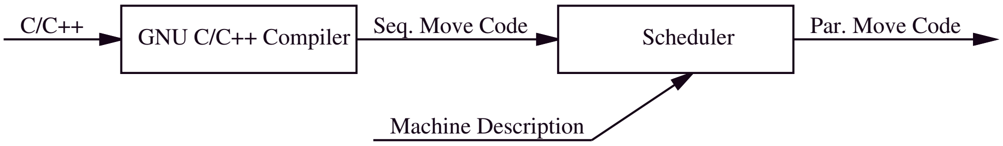

Figure 2: The code generation tools.

TTAs give more control and therefore more responsibilities to the compiler. This makes scheduling for TTAs more complicated than scheduling for OTAs. However by developing and implementing a scheduler for TTAs that is able to generate efficient code for realistic programs we have proven that the extra complexity can be managed.

The most important differences between scheduling for TTAs and OTAs are related to the correct usage of FU pipelines and the detection of bypassing, dead result moves, operand sharing, and socket sharing. Correct FU pipeline usage consists of: (1) obeying the first-in-first-out property of pipelines, (2) preventing pipeline overflows by pushing too many operations into a pipeline, and (3) assuring that what goes into a pipeline will also get out even when trigger and result moves are located in different basic blocks or have different predicates. All these checks are performed by means of a resource table.

5 Experiments

The goal of our experiments is to measure how performance depends on the number of GPRF ports for TTAs. To make this experiment surveyable we shall restrict our measurements to integer code. We shall use the following benchmarks:

> *bison, cjpeg, cpp, dhrystone, diff, djpeg, expand, od, sed, sort, stanford, tr, unexpand, uniq,* and *virtex*

The architecture that we shall use for our experiments consists of 6 FUs, 12 move busses, a GPRF of 32 integer registers, a GPRF of 8 booleans, and predicate expressions of one or two booleans. The FUs are single cycle and can perform each operation (an uniform VLIW). The interconnection network between the GPRFs and FUs is fully connected (like in figure 1). The number of GPRF ports is varied.

The motivation for this architecture is that we do not want to limit the performance by other resources than GPRF ports. For the same reasons we apply *perfect* memory reference disambiguation. The sequential code of benchmarks is simulated before scheduling and the addresses of memory references are used during scheduling by the memory reference disambiguator. By doing this we do not limit the performance by a poor memory reference disambiguator.

Our current code generation software does not support base-offset addressing mode. So a base-offset memory reference requires two cycles, one for the addition and one for the actual memory reference.

5.1 What is the Actual GPRF Port Requirement?

In our first experiment we want to measure the actual GPRF port requirement of TTAs and the factors that contribute to the difference between actual and worst case requirements.

Measurement	Result
Average number of GPR source operands for sequential code	0.97
Average number of GPR results for sequential code	0.60
Average number of GPR source operands for parallel code	0.50
Average number of GPR results for parallel code	0.35
Percentage of operand and trigger moves that are bypassed	50%
Percentage of result moves that are eliminated by dead result move elimination	35%
Percentage of operand moves that are eliminated by operand sharing	8.9%
Average number of moves that share the same socket (GPRF read port)	1.23

Table 1: Results of various measurements.

		Number of Read Ports					
		1	2	3	4	5	6
Number of Write Ports	1	1.98	2.26	2.31	2.34	2.35	2.35
	2	2.48	3.11	3.32	3.37	3.38	3.39
	3	2.65	3.37	3.63	3.68	3.71	3.72
	4	2.63	3.47	3.68	3.75	3.78	3.78
	5	2.63	3.47	3.70	3.76	3.79	3.79
	6	2.64	3.48	3.70	3.76	3.79	3.80

Table 2: Operations per cycle as function of the number of GPRF ports.

Table 1 shows the results of scheduling the 15 benchmarks for the above described architecture with 3 read ports and 3 write ports on the integer GPRF.

The first two lines of table 1 show that a lot of operations do not need two GPR operands or produce a result. The average number is 0.97 GPR operands and 0.60 results per operation. After the scheduler schedules the code and applies bypassing, dead result move elimination, operand sharing, and socket sharing these two numbers become 0.50 and 0.35. Compare this with 2 and 1 which is what an OTA would need.

The last four lines of table 1 show that the contribution of bypassing, dead result move elimination, and socket sharing is substantial. The contribution of operand sharing is low since: (1) it does not occur very frequently, (2) about half of the common operand moves have an immediate source, and (3) about a quarter of all moves are operand moves.

5.2 How Does ILP Depend on the Number of GPRF Ports?

In our second experiment we shall vary the number of read and write ports between 1 and 6, giving 36 different GPRF configurations. Table 2 shows how many operations are executed per cycle for each of the 36 GPRF configurations. This experiment shows some very good results. For example, even with the one read and one write port two operations per cycle can be executed. Another example, with 3 read and 3 write ports 3.6 operations per cycle are executed. For this achievement an OTA would need at least 8 read and 4 write ports.

This experiment also tells us the optimal division between read and write ports for a certain

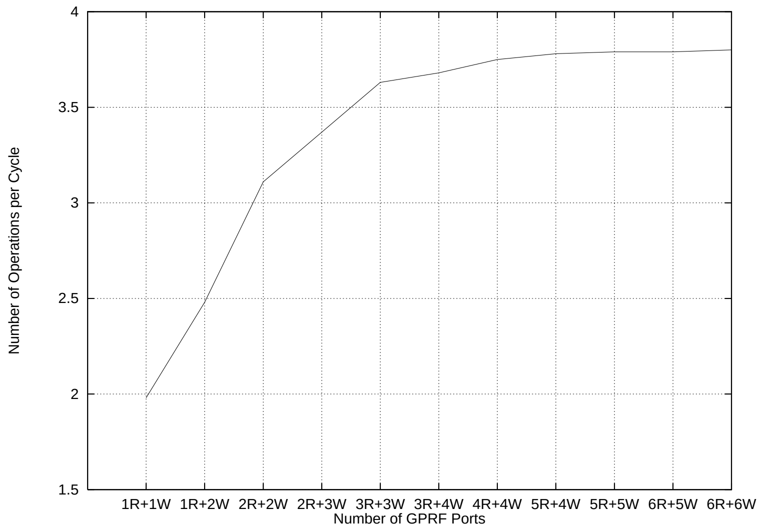

Figure 3: Operations per cycle as function of the number of GPRF ports.

number of ports. For example the optimal configuration for a 7 ported GPRF consists of 4 write ports and 3 read ports. Figure 3 shows the number of operations per cycle for the optimal configurations.

It is remarkable that some of the optimal configurations with a low number of ports have more write ports than read ports. This is due to dead result move elimination. When a result move is scheduled a write port is claimed. However after scheduling the operations that are flow dependent on it, the result move can become dead and the claimed write port is released. Therefore the scheduler sometimes needs an extra write port during scheduling that is not needed in the final code.

5.3 How Many Ports are Needed for a Certain Issue Rate?

It should be clear that TTAs with less move busses (a lower issue-rate) need less GPRF ports. In order to measure this we have scheduled the benchmarks for the most interesting GPRF configurations for one to twelve move busses. Figure 4 shows the results. For small TTAs (4–6 move busses) a two read two write ports GPRF provides sufficient GPRF bandwidth. For larger TTAs (7–12 move busses) a three read three write ports GPRF is a good solution.

6 Related Work

There are several other ways to reduce the number of GPRF ports by making use of the fact that the average requirement is lower than the worst case requirement. Several approaches are discussed (without being exhaustive).

- Different instruction encoding. In his thesis [Joh91] about superscalar processor design Johnson recognized the difficulty of supporting many access ports to the register file. He proposes a different instruction encoding format specifying 2 or more instructions; within the instruction scope of this format bypassing and socket sharing can be specified explicitly, therefore reducing the number of read access ports on the GPRF. Apart from giving up

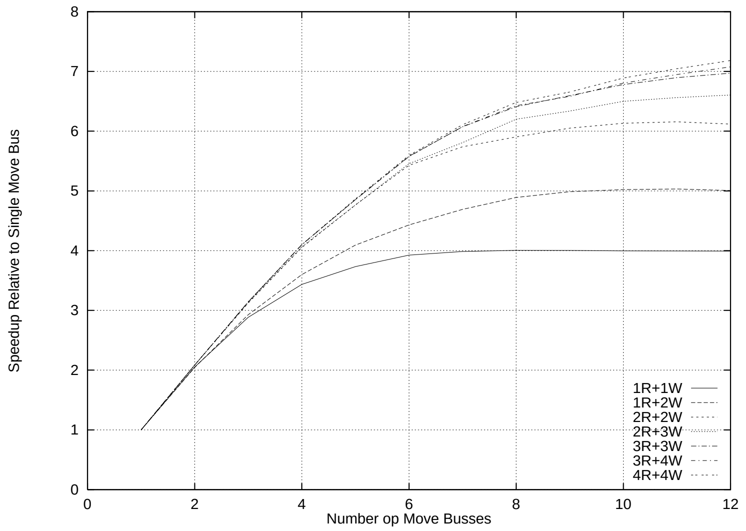

Figure 4: Speedup curves for various GPRF configurations.

one of the biggest advantages of superscalars, binary code compatibility[2], this proposal has some other disadvantages, 1) the bypassing scope is very limited, and 2) bypassing across join points can not be specified.

- Shared busses between FUs. The CDC 6600, which used scoreboarding to allow out-of-order issue of instructions, had 16 FUs which could require parallel access to read and write ports of the GPRF. In order to reduce the number of access ports the FUs were grouped into 4 groups, where each group only had one set of access ports to the register file scheduled by hardware. A similar approach can be used by VLIWs where the scheduling is done by the compiler.

- Shared write back busses. This is a special case of the former approach were only write back busses are shared among multiple FUs. It is used in e.g. the i960 [Int92], M88110 [DA92]. The M88110 is a 2 issue superscalar architecture having 2 write back busses. Worst case more than 2 operations require simultaneously access to these busses. Shortest latency operations have highest priority. The i960 also has 2 write busses on the register file, one for the execution units, and one for the load/store units. Both busses are hardware scheduled where in this case shortest latency operations have lower priority.

- Common data bus. The IBM 360/91 used the Tomasulo algorithm for out-of-order issue and completion applied to floating point operations. It relies on the use of a shared bus (the common data bus) for writing results to the GPRF and bypassing them to the reservation stations.

- Complex operations. Using operations with more complex functionality also may reduce register port requirements. A typical example is the combined multiply-accumulate instruction. The intermediate product is directly bypassed to the adder without being written to the GPRF. Other examples are CISC instructions using a more complex addressing mode.

[2] Unless the format change is only visible in the instruction cache.

- Accumulator machine. Well known from early microprocessors is the usage of an accumulator register which can be used to bypass a result to the input of the ALU or to share operands. Although reducing register ports was not an issue, usage of an accumulator might reduce their number. In fact the way TTAs reduce register ports has a lot in common with this approach.
- Multiple identical register files. To reduce the implementation complexity of many read ports one might implement N copies of the register file and broadcast write backs to all these copies. The number of read ports is reduced with factor N, at the expensive of a large area overhead. This method is applied in the Cydrome VLIW processor [RYYT89]. Essentially this is an implementation variation, and not a reduction of ports at the architectural level; therefore it is not further discussed.

Former approaches reduce register read and/or write ports using one or more of the methods as described in section 3. Table 3 indicates which methods are exploited. It will be clear to the reader that the approach used in TTAs to reduce register port requirements supersedes all the former listed approaches, and applies them in a much more uniform manner, because the compiler controls any data transport.

Approach	Exploited method	Port reduction
Different instruction encoding	1, 3, 6	read ports
Shared write back busses	2	write ports
Shared busses between FUs	1, 2	read and write ports
Common data bus	1, 2, 3	read and write ports
Complex operations	1, 2, 3, 4, 5, 6	read and write ports
Accumulator machine	1, 2, 3, 4, 5	read and write ports

Table 3: Different approaches to register file port reduction.

A completely different solution is to partition the GPRF (see e.g. [CDN92, BS91, Ell86]). Partitioning reduces register port requirements by exploiting FU data locality; its method is orthogonal to the approaches discussed in this paper. Most processors apply this principle already to separate the integer and floating point registers. This works well because there are only few data transports necessary between these different register files. Further partitioning is less straightforward because the fine grain nature of instruction level parallelism requires much inter FU communication. As a result extra register file move operations have to be inserted.

7 Conclusions

We have described transport triggered architectures (TTAs) and have shown that they have a substantial lower GPRF port requirement than traditional operation triggered architectures (OTAs). We have performed experiments to quantify this reduction. The results have shown that the average requirement is 0.50 read and 0.35 write ports per operation. Furthermore, it is possible to execute two operations per cycle with a small two-ported GRPF and 3.6 operations per cycle with a six-ported GPRF. This reduction has the following important implications:

- TTAs are less expensive than comparable OTAs (less chip area and power consumption).
- TTAs are better scalable that OTAs. More FUs can be served by a single GPRF. When a single GPRF becomes the bottleneck it is still possible to partition it.

The current version of our scheduler has no special heuristics that encourage scheduling of flow dependent moves in the same cycle so that bypassing and possible dead result move elimination occurs. Incorporating them could reduce the GPRF port requirement even further. The development of such a heuristic is one of the things we want to research in the future.

References

[BCK91] David Bernstein, Doron Cohen, and Hugo Krawczyk. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of the 24th Annual International Workshop on Microprogramming*, pages 103–113, Albuquerque, New Mexico, November 1991.

[BS91] Mauricio Breternitz, Jr. and John Paul Shen. Implementation Optimization Techniques for Architecture Synthesis of Application-Specific Processors. In *Proceedings of the 24th Annual International Workshop on Microprogramming*, pages 114–123, Albuquerque, New Mexico, November 1991.

[CDN92] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolua. Partitioned Register Files for VLIWs: A Preliminary Analisys of Tradeoffs. In *Proceedings of the 25th Annual International Workshop on Microprogramming*, pages 292–300, Portland, Oregon, December 1992.

[CM91] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A Framework for High-Performance Processor Design. In *Supercomputing-91, Albuquerque*, pages 692–701, November 1991.

[DA92] K. Diefendorf and M. Allen. Organization of the motorola 88110 superscalar risc microporcessor. *IEEE micro*, April 1992.

[Ell86] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1986.

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

[HC94] Jan Hoogerbrugge and Henk Corporaal. Transport-Triggering vs. Operation-Triggering. In *International Conference on Compiler Construction*, Edinburgh, Scotland, April 1994.

[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 for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–249, May 1993.

[Int92] Intel. *80960 CA Microprocessors Users Manual*, 1992.

[Joh91] William M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[Lam88] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[ME92] Soo-Mook Moon and Kemal Ebcioğlu. An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors. In *Proceedings of the 25th Annual International Workshop on Microprogramming*, pages 55–71, Portland, Oregon, December 1992.

[Pin93] Shlomit S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 248–257, June 1993.

[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.