# Resource Assignment in a Compiler for Transport Triggered Architectures

Jan Hoogerbrugge    Henk Corporaal

Delft University of Technology, Dept. of EE
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {janh,heco}@duteca.et.tudelft.nl

**Abstract**

The paper describes how resource assignment is performed in a compiler for transport triggered architectures, and in particular, how busses are assigned to data transports. This becomes an important issue when function units and register files are only partially interconnected via an irregular network of busses. Reducing this connectivity is important for processors exploiting instruction level parallelism; it may not only reduce the required chip area, but may also shorten the processor cycle time and therefore increase the performance.

The bus assignment problem is solved by transforming it to a bipartite matching problem for which efficient algorithms exists. Experiments have been performed to measure how the cycle count depends on the connectivity between function units and register files. Results show that a significant amount of connectivity from a fully connected configuration can be removed before the cycle count starts to increase.

**Keywords:** resource assignment, instruction-level parallelism, bipartite matching, reduced inter FU connectivity

## 1 Introduction

Today's VLSI technology makes it possible to implement a large number of functional units (FUs) on a chip which makes large amounts of instruction-level parallelism (ILP) possible. An increasingly important problem is the connectivity between the FUs and register files (RFs). $N$ FUs require $3N$ RF ports and a bypass network between the FUs and RFs of $O(N^2)$ paths [6], e.g., the two-issue DEC Alpha A21064 processor contains already 38 bypass paths [12]. This requires a significant amount of chip area and is a serious treat for the attainable cycle time; note that bypassing is part of the execution stage of the processor pipeline and therefore likely contributes to the critical timing path of a processor design. It makes therefore sense to focus on the data transports between the FUs and RFs, instead of the execution of operations on the FUs. This is exactly the motivation for *transport triggered architectures* (TTAs). TTAs are, unlike traditional *operation triggered architectures* (OTAs), not programmed by programming operations but by programming data transports. The data transports may trigger operations on FUs as side effect. By programming data transports, a compiler is able to suppress useless data transports and RF accesses that frequently occur in OTAs.
This paper describes one important aspect of compilation for TTAs: the assignment of resources, in particular, the assignment of busses to data transports. This task becomes non-trivial when FUs and RFs are partially connected to the busses. It will be shown that the method proposed for bus assignment makes it possible to generate efficient code for irregular connected TTAs.
This paper is organized as follows. Section 2 gives a brief description of TTAs. Section 3 describes how resource assignments are performed in our TTA compiler. Section 4 describes an experiment that measures how the cycle count depends on the connectivity between the FUs and RFs. Finally, section 5 summarizes the paper.

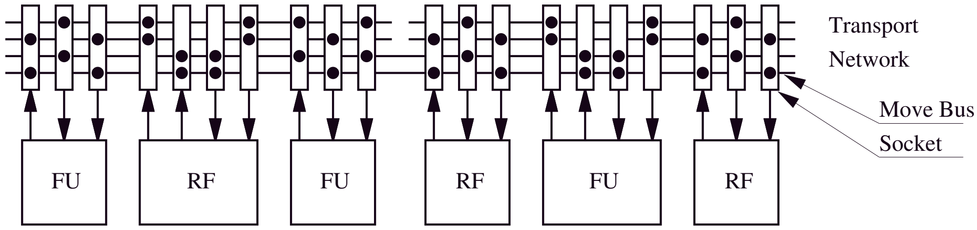


**Figure 1:** General structure of a TTA.

# 2 Transport Triggered Architectures

A compiler sees a TTA as a collection of FUs, RFs, *move busses*, and *sockets*; see figure 1. FUs perform operations, RFs provide temporary fast accessible storage, the network of move busses performs data transports between the FUs and RFs, and sockets interface FUs and RFs to move busses. Normally, each socket is connected to a different FU input/output or RF port. TTAs are programmed by specifying the required data transports over the move busses between the FUs and RFs. A TTA instruction consists of a number of move slots, each specifying a move operation that controls a move bus. To illustrate TTA programming, consider the following three operations:

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

These operations can be translated into the following two TTA instructions:

```
r2 -> add_i1; r3 -> add_i2; r2 -> sub_i1; r6 -> sub_i2
add_o -> st_i1; sub_o -> st_i2
```

In the first instruction the four operands of the add and subtract operations are moved from the RF(s) to the FU inputs of the FUs that perform the two operations. In the second instruction the results of the add and subtract operations are moved from the FUs that performed them to the FU that performs the store operation. From this small example we already observe a few advantages of TTAs. The results of the add and subtract operations are not written back to the RF and the operands of the store operation are not read from the RF. The former saves RF write accesses and data transports, the latter saves RF read accesses. Since TTAs do not couple move busses and RF ports directly to FUs, as is the case in OTAs, the freed resources can be used for other operations. This makes that TTAs have a better hardware utilization, which implies less hardware for the same performance or more performance with the same hardware [8].
The interconnection network may be fully connected, in which case every socket is connected to all move busses, or partially connected. A fully connected interconnection network simplifies the code generation task, but it likely results in a high bus load on the move busses which affects the achievable cycle time. Therefore, in practice the interconnection network will be partially connected and the compiler is responsible to use the available connections as well as possible.

# 3 Resource Assignment

As part of the *MOVE Project*, we have developed a parameterized compiler for TTAs. This compiler consists of a front-end based on GNU GCC and a back-end which includes an extended basic block scheduler, a software pipelining scheduler, a memory reference disambiguator, a register allocator, a function inliner, and a loop unroller. For more information about the used scheduling algorithms, the reader is referred to [7].
An important design decision in the development of a compiler for ILP processors is when to do resource assignments[1]. The back-end has to assign registers and RFs to pseudo registers, FUs to operations, immediate fields to immediate operands, and move busses and sockets to moves. Ideally, resource assignments

[1] In the high-level synthesis community one usually makes a distinction between *assignment* and *allocation*. The assignment problem deals with finding a mapping between resources and entities that need these resources, while the allocation problem deals with the existence of a mapping. The allocation problem is usually solved by trying to make an assignment.

should be postponed as long as possible and during scheduling only the possibility of an assignment should be checked. In reality this is hard to implement and it is computational expensive to do this for all resources of a TTA. Therefore, we made the following design decisions:

1. Register and RF assignment before scheduling.
2. Move bus assignment after scheduling.
3. FU, socket, and immediate field assignment during scheduling.

The following three subsections motivate these decisions and describe the assignments, in particular the move bus assignment, in more detail.

## 3.1 Register and Register File Assignment

Register and RF assignments to pseudo registers are made before scheduling. Making them during scheduling as proposed in [5] increases the engineering complexity significantly. Performing these assignments after scheduling as proposed in [3] (1) increases the register requirement, (2) is difficult in combination with predicated execution since predicated execution complicates live variable analysis required for register allocation [11], and (3) insertion of spill code into parallel (i.e., scheduled) code requires a postpass scheduling step.

Making register and RF assignments before scheduling has the problem of making assignments that result in false dependences and RF port resource conflicts during scheduling that limit parallelism. This problem can be solved by making the register allocator aware of how its assignment decisions affect the scheduling freedom of the scheduler. For register allocation this is realized by adding edges to the interference graph used by the coloring-based register allocator [2]. These edges prevent assignments that result in false dependences between operations that the scheduler may want to reorder.

For RF assignment we do something similar. The register allocator distributes the pseudo registers over the RFs such that pseudo registers that the scheduler may want to access in the same cycle are assigned to different RFs [10]. This will reduce RF port resource conflicts during scheduling.

## 3.2 Move Bus Assignment

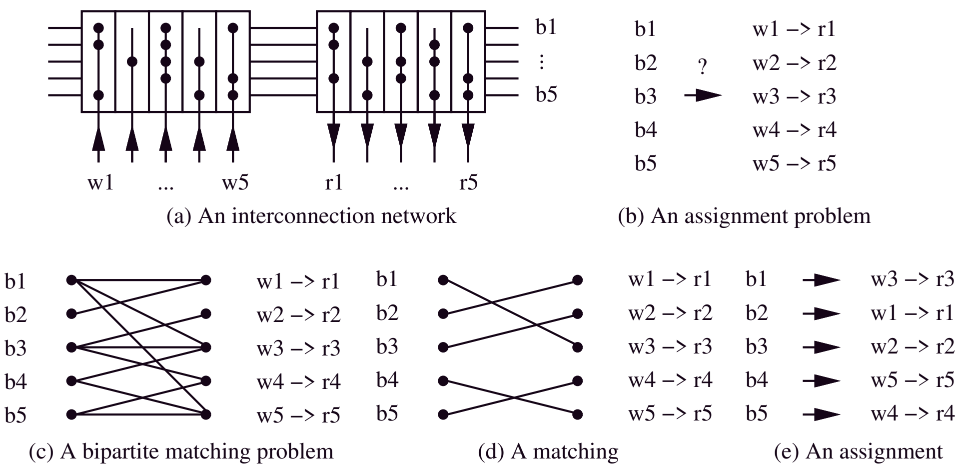


**Figure 2:** Move bus assignment via bipartite matching

Move bus assignment is performed after scheduling; during scheduling only the possibility of an assignment is checked. This is necessary if we want to be able to generate efficient code for partially connected TTAs. Furthermore, it is also a necessity if we want that quasi full connectivity performs as well as full connectivity [7]. Figure 2 shows how move bus assignment is checked during scheduling and how the actual assignment is done after scheduling. The problem to be solved is: given a set of moves between socket pairs

| Parameter | Value | | | |
|---|---|---|---|---|
| Move busses | 8, 64-bits wide | | | |
| Function units: | **Number** | **Latency** | **Operations** | |
| • LSU | 2 | 2 | load and store operations | |
| • ALU | 2 | 1 | integer operations | |
| • FPU | 1 | 3 | floating point operations | |
| Register files: | **Number** | **Registers** | **Ports** | **Width** |
| • predicates | 1 | 4 | 1W | 1 |
| • integer | 1 | 48 | 3W+3R | 32 |
| • floating point | 1 | 48 | 2W+2R | 64 |
| Immediates fields | 2 × 32-bits | | | |
| Jump latency | 2 cycles (1 delay slot) | | | |
| Interconnection network | varying connectivity | | | |
| Memory system | perfect (no cache and TLB misses) | | | |

**Table 1:** The main parameters of the used architecture

that have to be performed (w1 $\rightarrow$ r1,. . . , w5 $\rightarrow$ r5 in figure 2) and an interconnection network of move busses (b1,. . . , b5), can a different move bus be assigned to each move that is capable to do the data transport? A move bus is capable to perform a data transport between two sockets if it is connected to both sockets and it is wide enough (move busses may have different widths). The move bus assignment problem is solved by transforming it to a bipartite matching problem [1]. An undirected bipartite graph $G = (V_m \cup V_b, E)$ is built were $V_m$ corresponds to the data transports, $V_b$ to the move busses, and $E$ contains an edge from an $m \in V_m$ to a $b \in V_b$ if move bus $b$ is capable to do perform data transport $m$. An assignment can be made if there is a matching $M$ with $|M| = |V_m|$. A matching $M$ is a subset of $E$ such that for all nodes $v \in V_m \cup V_B$, at most one edge of $M$ is incident on $v$. Clearly, a matching $M$ of size $|V_m|$ gives us directly the assignment we were looking for. The bipartite matching problem can be solved exactly by an $O(m\sqrt{n})$ time complexity algorithm, where $n$ is $|V_m \cup V_B|$ and $m$ is $|E|$ [1].
Due to the large number of times that the assignment question needs to be answered, solving it by means of a bipartite matching problem is computational too expensive. Therefore, the bipartite matching algorithm is preceded by a simple, fast but inexact first fit type assignment algorithm that can handle most cases successfully ($\geq 95\%$ for irregular interconnection networks). Only if the first fit type assignment algorithm cannot find an assignment the bipartite matching algorithm is used.
To illustrate the need for postponing move bus assignment until scheduling has been performed, assume that w1 $\rightarrow$ r1 is scheduled first, followed by w2 $\rightarrow$ r2, w3 $\rightarrow$ r3, etc., and first fit move bus assignment. In this case b1 will be assigned to w1 $\rightarrow$ r1, b3 to w2 $\rightarrow$ r2, b4 to w3 $\rightarrow$ r3, and b5 to w4 $\rightarrow$ r4. Scheduling w5 $\rightarrow$ r5 will fail since b2, the only free move bus left, cannot be assigned to it.

## 3.3 Function Unit, Socket, and Immediate Field Assignment

Function unit, immediate field, and socket assignments are performed during scheduling. The motivation for this, instead of postponing the assignment decision, is that for these resources there is usually not much to choose. There are usually one or two FUs per FU type, one or two immediate fields, and a small number of read or write ports per RF, especially when there are multiple RFs per RF type.
FU, socket, and immediate field assignment is performed by means of a simple first fit assignment algorithm; the first free resource that fits is selected. If there are multiple candidates with a different but overlapping functionality, it is preferable that the most specialized resource is selected, and more universal resources are kept for other operations. For example, if there are two FUs, $f_1$ and $f_2$, where the operation set of $f_1$ consists of load and store operations, the operation set of $f_2$ consists only of load operations, and both are available for a load operation, $f_2$ is preferred. The order in which resources are examined for assignment corresponds to the order in which they are listed in the machine description file that describes the target TTA. The user should therefore specify the resources in increasing generality, e.g., $f_2$ before $f_1$.

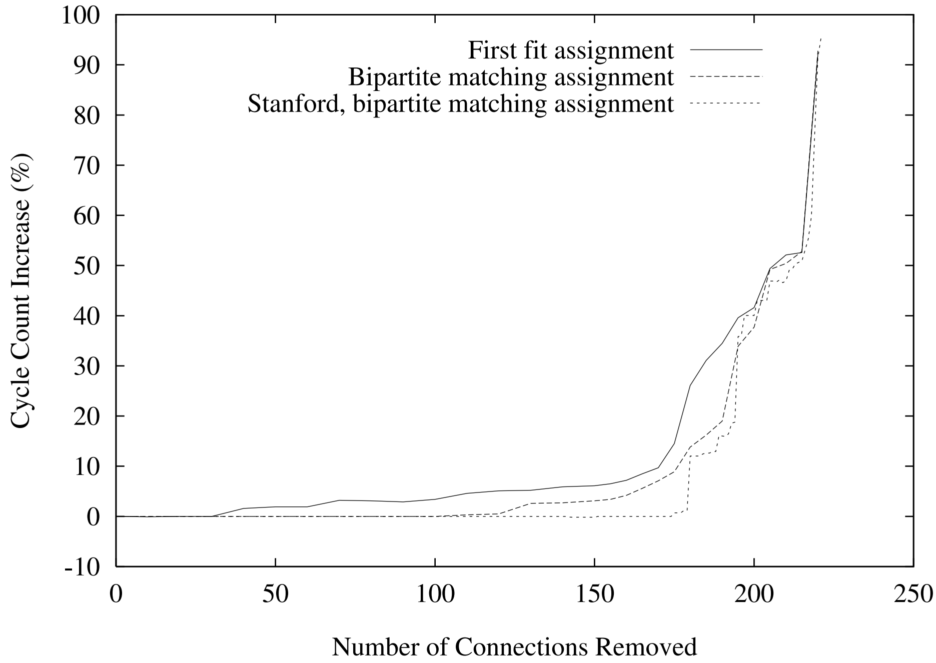


**Figure 3:** The influence of connectivity on the cycle count

# 4 Influence of Connectivity on Cycle Count

In this section we will describe the measurements that we have performed to measure the influence of the connectivity between FUs and RFs on the cycle count. For this experiment we will use the following benchmark set:

> *a68, compress, mpeg_play, virtex, bison, diff, uniq, sed, flex, gs, gzip, sum, expand, djpeg, cpp, wc, sort, cmp, od, tr, arfreq, equaliz, g722, instf, interp3, mulaw, music, radproc, rfast,* and *rtpse*

The first twenty benchmarks are commonly used UNIX workstation applications; the last ten are DSP applications taken from [4]. The used architectural parameters for the experiment are listed in table 1.

In order to measure the effect of partial connectivity on the cycle count we need a sequence of configurations with a connectivity varying from full connectivity to the minimal connectivity required for successful compilation. To obtain this sequence we use a TTA processor synthesis tool as described in [9]. This tool is capable to find an almost optimal (in terms of performance) TTA for given cost (in terms of chip area), or, vice versa, the cheapest TTA for given performance needs. With this tool we determine a sequence of TTAs with reduced connectivity for one particular application. For this purpose we use an independent benchmark, the stanford benchmark, which is not part of the above set of benchmarks. This benchmark is a collection of ten small benchmarks including *queen, hanoi, FFT,* and *quicksort*. Two of these benchmarks use floating point operations, the others use only integer operations. Using an independent benchmark for obtaining a connectivity reduction sequence means that this sequence is not biased to the above set of benchmarks.

Figure 3 shows the results of the experiment. Three curves are shown. The bottom curve shows the cycle count of the stanford benchmark relative to the cycle count of a fully connected network as function of the number of connections removed from a fully connected configuration. The cycle count of the stanford benchmark starts to increase after 175 of the connections have been removed. The initial fully connected configuration has 240 connections (8 move busses $\times$ 30 sockets). The middle curve shows the arithmetic average of the relative cycle count for the 30 benchmarks. This curve starts to raise after 120 connections have been removed. The top curve shows the average relative cycle count when move bus assignment would be performed during scheduling by a first fit assignment algorithm. This performs clearly worse than move bus assignment after scheduling, as proposed in this paper. The cycle count starts already to increase after 40 connections have been removed. This also demonstrates that first fit assignment is not capable of optimal exploitation of a quasi fully connected architecture for which 56 ($8 \times 7$) connections have been removed.

# 5 Conclusions

This paper describes an important issue in the design of a compiler for transport triggered architectures: the assignment of resources. The compiler has to assign registers and register files to pseudo registers, function units to operations, sockets (and register file ports) to moves, immediate fields to immediate operands, and move busses to moves. The latter is has been described in more detail. The move bus assignment problem is solved by transforming it to a bipartite matching problem. Furthermore, move bus assignment is performed after scheduling; during scheduling only the possibility of an assignment is checked. This makes it possible to schedule efficiently for partially interconnected architectures. Partial connectivity between function units and register files becomes important in future processors which support large amounts of instruction level parallelism; it (1) reduces the required processor chip area, (2) improves the scalability to more function units, and (3) makes shorter cycle times possible which improves the processor performance.
The scheduling efficiency for partially connected architectures has been verified by an experiment that compares move bus assignment after scheduling with assignment during scheduling by means of a simple first fit assignment algorithm. The results clearly show the superiority of assignment of move busses after scheduling using the method described in this paper. Furthermore, it has been demonstrated that the connectivity between function units and register files can be reduced with at least 50% without increasing the cycle count.


# Acknowledgements

The authors wish to thank Wiebe Cnossen for mentioning the words ‘it looks like a matching problem’.


# References


[1] AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] BRIGGS, P. *Register Allocation via Graph Coloring*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, Apr. 1992.

[3] CHANG, P. P., ET AL. The Importance of Prepass Code Scheduling for Superscalar and Superpiplined Processors. *IEEE Transactions on Computers 44*, 3 (Mar. 1995), 353–370.

[4] EMBREE, P. M. *C Language Algorithms for Real-Time DSP*. Prentice Hall, 1995.

[5] FREUDENBERGER, S. M., AND RUTTENBERG, J. C. Place Ordering of Register Allocation and Instruction Scheduling. In *Code Generation-Concepts, Tools and Techniques* (1991).

[6] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann publishers, 1990.

[7] HOOGERBRUGGE, J. *Code Generation for Transport Triggered Architectures*. PhD thesis, Department of Electrical Engineering, Delft University of Technology, Delft, The Netherlands, 1996.

[8] HOOGERBRUGGE, J., AND CORPORAAL, H. Register File Port Requirements of Transport Triggered Architectures. In *Proceedings of the 27th Annual International Workshop on Microprogramming* (San Jose, California, Nov. 1994), pp. 191–195.

[9] HOOGERBRUGGE, J., AND CORPORAAL, H. Automatic Synthesis of Transport Triggered Processors. In *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging* (Heijen, The Netherlands, May 1995), J. van Katwijk, J. J. Gerbrands, M. R. van Steen, and J. F. M. Tonino, Eds.

[10] JANSSEN, J., AND CORPORAAL, H. Partitioned Register Files for TTAs. In *Proceedings of the 28th Annual International Workshop on Microprogramming* (Ann Arbor, Michigan, Nov. 1995), pp. 303–312.

[11] MAHLKE, S. A., ET AL. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th Annual International Workshop on Microprogramming* (Portland, Oregon, Dec. 1992), pp. 45–54.

[12] SMITH, J. E., AND WEISS, S. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *IEEE Computer 27*, 6 (June 1995), 46–58.