Automatic Synthesis of Transport Triggered Processors

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

A designer can chose from several options when mapping an application into a combination of hardware and software. Usage of an ASIP offers the advantage of a large design freedom, allowing optimal tuning of performance and costs. However there are two major problems related to the design of ASIPs: 1) the design trajectory is long, and 2) it is impossible to do a quantitative search of the whole design space.

The alleviate these problems we propose a design trajectory based on a *templated, transport triggered architecture*. Using a restricted, but still very large, design space we are able to automate the design trajectory based on a quantitative analysis of many design points.

This paper presents this design method and shows its results when the method is applied to two benchmarks.
Keywords: CoSynthesis, Design space exploration, ASIP design, Transport triggering.

1 Introduction

Designing ASICs based on templated application specific instruction set processors (ASIPs) is an attractive solution that offers flexibility and a short design time while still having most of the advantages of ASICs. The design process consists of finding the right architecture parameter values for the given application, such as the operation set, the amount of instruction level parallelism, and the sizes of the register files. The quality of a solution depends on the offered performance and the implementation costs.

In this paper we discuss an automated design process for ASIPs using the *MOVE framework*. This framework consists of a set of tools for hardware and software synthesis. Central to the MOVE framework is the the use of an architecture design space based on a *transport triggered architecture*, or TTA. Processors generated with this framework are therefore called transport triggered processors, or TTPs. These processors resemble well known VLIW processors, but a key difference is that TTPs are programmed by specifying data transports instead of operations. This gives an extra level of control to the code generator, and allows for a more efficient use of hardware resources.

Although we use a TTA template structure for designing TTPs, the design space for TTPs is still very large. Picking a proper solution (for a specific application) from this design space requires a quantitative analysis of many design points. This search process must largely be automated in order to reduce the design time. Therefore tools are needed, not only for making the quantitative analysis of hardware and software (generated code), but also for the automated search.

The paper is structured as follows: section 2 explains briefly how TTPs operate. Section 3 presents the synthesis process as used within our ASIP design framework, and describes the tools needed for this process. It also lists the major parameters of the architecture design space. In section 4 we describe our search methods, and the benchmarks used to demonstrate the search process. The results of the experiments, and their discussion, is given in section 5. In section 6 several conclusions are drawn.

2 Transport Triggered Architectures

Here we give a short overview of TTAs. More elaborate descriptions can be found in [1, 2]. TTAs can be viewed as a superset of traditional VLIW (very large instruction word) architectures. VLIW processors can perform multiple operations per cycle; they are of the SIMO type (single instruction, multiple operation). Their instructions are horizontally encoded; i.e. each instruction has a number of fields, where each field specifies a RISC type operation. Whereas VLIWs are programmed by specifying operations, TTAs are programmed by specifying the required data transports. These transports may trigger operations as side effect. TTAs are of the SIMT type (single instruction, multiple transports); one instruction may specify multiple concurrent transports. Programming transports adds an extra level of control to the code generator, or compiler. More compiler control means more code optimization opportunities.

The structure of a TTA is very simple; figure 1 shows an example architecture, containing 9 FUs and 6 transport buses. Each FU is connected to the transport network with one or more input and output *sockets*. In general, the network does not have to be fully connected, although full connectivity would relax the code generation process.

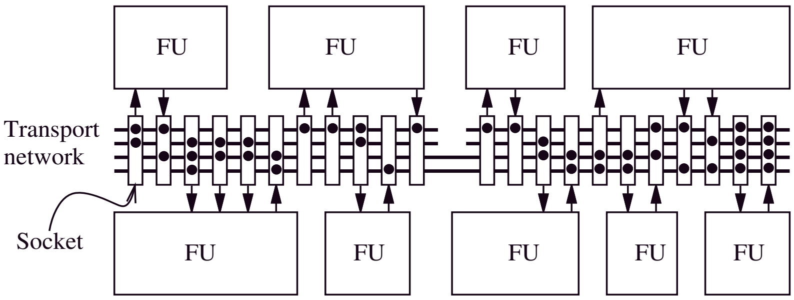

Figure 1: General structure of a TTA.

Each function unit (FU) includes one or more operand and result registers which are part of the total register address space (as seen by the programmer). We distinguish the registers according to their purpose: 1) FU-operand registers (O), 2) FU-trigger registers (T), 3) FU-result registers (R), and 4) general-purpose registers (r)[1]. Usually general-purpose registers are combined into one or more register files. A register file can have one or more *read and write ports*, where each read port connects to an output socket and each write port to an input socket.

Triggering operations occurs by moving data into the appropriate FU-trigger (T) registers. All operations, including load/store and branch/jump/link occur solely through move instructions. FUs operate concurrently and may incorporate pipelined execution.

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

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

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

First the values of r1 and r2 are moved to the input registers of the adder. After some time (depending on the latency of the adder) the result can be moved to r3. At a first glance it looks like a typical RISC operation requires three move operations. However, a number of optimizations are possible which reduce the number of move operations to an average of about two (see e.g. [3]). This means that less transport capacity is needed.

In practice most FUs implement multiple operations (like having a single FU for both add and subtract). These operations are distinguished by mapping multiple trigger identifiers on the same physical trigger register.

Conditional execution is provided by means of predicated execution. Move operations can be predicated by an expression of booleans registers containing results of compares. The move operation is only executed when the expression evaluates true.

The main design advantages of TTPs are 1) extreme simplicity and therefore quick design time, 2) great flexibility and performance scalability (transport network and FUs are orthogonal, and can be optimized to the application independently), 3) short processor cycle times (cycle time is determined mainly by data transport), and 4) a substantial reduction on the number of register ports required [4].

Apart from design advantages, TTAs also have advantages with respect to the following issues: 1) better scheduling because the operations are more fine grain, 2) efficient usage of transport capacity, 3) extra fine grain parallelism as a result of separating ALU functionality into its individual components, and 4) reduced register usage.

3 The Synthesis Process

Within the synthesis process we use an *architecture template*, i.e. processors are built according to the pattern of a TTA. At first sight this suggests that we restrict ourselves and therefore obtain inferior solutions. In practice however, this turns out to be a big winner. First of all, the template covers a very large design space. Secondly, the template building blocks are pre-designed and can therefore be made very efficient, both in area and performance. Thirdly, the architecture pipelining is worked out very carefully, alleviating many timing bottlenecks; prototype realizations learned us exactly where these bottlenecks are. Finally, usage of a clearly defined design space allows the design of synthesis and evaluation tools, which not only generate a combined hardware and software solution, but also allow a quantitative analysis of the design space.

The main parameters of the template architecture are given in table 1. The number of sockets needed (see figure 1) is implied by the number and type of FUs, register files, and number of long immediates[2]. Connections between sockets and buses are specified individually.

The synthesis of hardware and software for a given application is done using the MOVE framework. An overview of this framework is shown in figure 2. It basicly consists of three components:

Optimizer which is responsible for searching the design space and the interaction with the designer.

Hardware subsystem generating processor layout, and giving information on timing, area, and power consumption.

[2]Putting immediates on the data buses requires output sockets as well.

Parameter	Range
#buses	≥ 1
#FUs	≥ 1
FU latencies	≥ 1
FU type	all types which are in the library
FU pipelining method	latching always or hybrid pipelining [1]
#registers, IRF[a]	$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, \ldots$[b]
#input ports, IRF	≥ 1
#output ports, IRF	≥ 1
#registers, FRF[c]	$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, \ldots$
#input ports, FRF	≥ 1
#output ports, FRF	≥ 1
#registers, BRF[d]	$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, \ldots$
#input ports, BRF	≥ 1
#long immediates	≥ 1
bus socket connection	yes, no

[a] Integer register file.
[b] These numbers correspond to 2^n, and $2^n + 2^{n+1}$, for $n \geq 0$.
[c] Floating point register file.
[d] Boolean register file.

Table 1: Main parameters of the architecture design space.

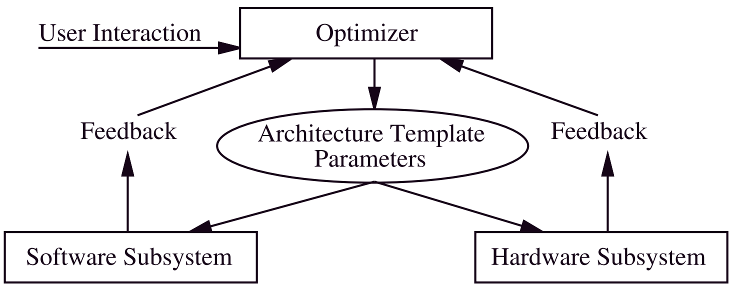

Figure 2: MOVE framework overview

Software subsystem generating instruction level parallel code, and giving statistical information on usage of hardware resources.

These components are detailed in following subsections.

3.1 Optimizer

Two main design evaluation criteria are cost and performance, where performance is defined as the inverse of execution time. Costs may include the amount of chip area, number of pins, power dissipation, and code size[3]. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time.

[3] Currently we included area and pins only.

The *solution space* is given by all possible design points in the 2-dimensional cost-performance space. Figure 3 shows many generated solutions for the Crypt application (as described in section 4).

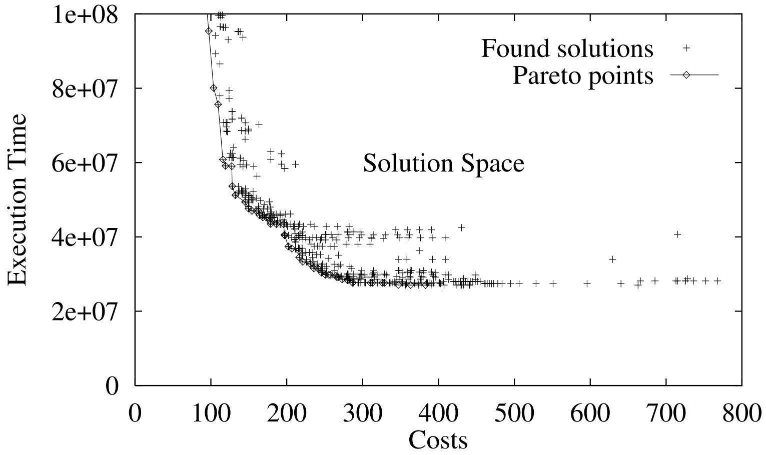

Figure 3: Possible solutions and Pareto points.

As shown in this figure, the solution space is bounded by a curve connecting so called *Pareto points*. These points are local optimal solutions; they are defined by [5]:

Definition: *A point* (t_p, c_p) *is a Pareto point iff* $\forall (t, c) \in$ *Solution-Space*, $(t \geq t_p \vee c \geq t_c)$.

The optimizer finds its way through this search space by iteratively trying different architecture solutions, and letting the software and the hardware subsystems produce relevant information about these solutions, like cycle time, costs and number of cycles needed to run the application. Based on this information a next design point is chosen by updating the parameters. This iterative design process is shown in figure 4.

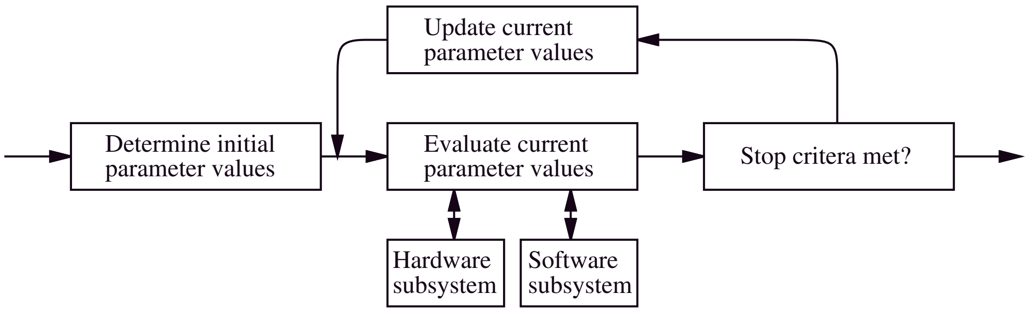

Figure 4: The iterative design process.

The initial architecture parameter values can be chosen freely by the user. He can also specify an evaluation function (e.g. minimize the product of costs and execution time), and the stop criteria. In section 4 we will discuss several methods for updating the current parameter set.

3.2 Hardware subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of an application specific TTP in silicon. It accepts architecture parameter values, technology information and a cell library as input, and produces a VLSI layout (e.g. in CIF format) of the generated processor as output. Figure 5 shows its organization. The hardware subsystem makes use of the ASA silicon

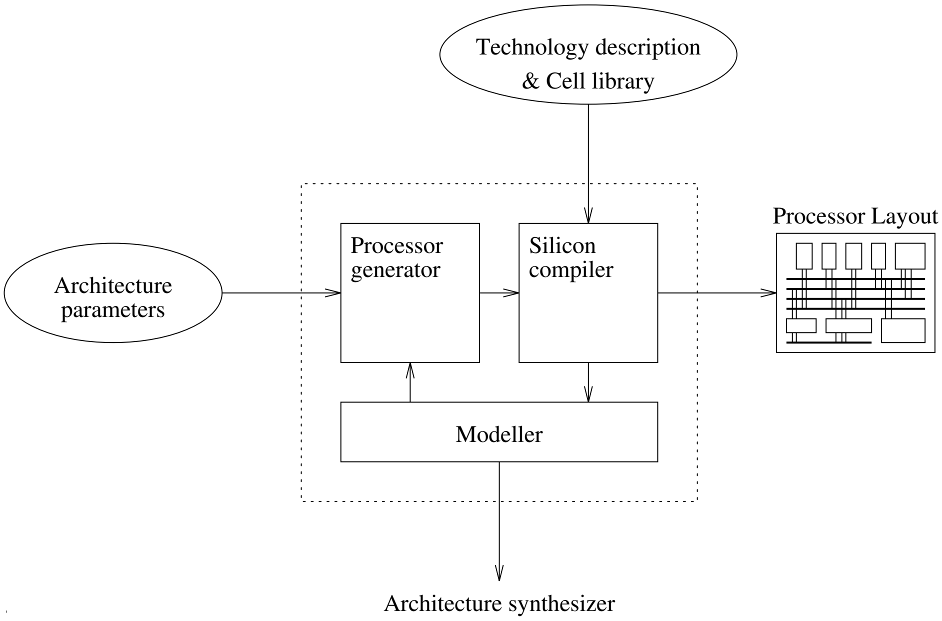

Figure 5: The hardware subsystem.

compiler[4]. Details of this system are described in [6].

The experiments presented in this paper make use of a *hardware model*. This model is based on realizations in a CMOS 1.0μ technology. The values and expressions used for costs (area, including number of pins) are shown in table 2. The costs of FUs are based on a 32-bit data path width[5], and relative to an integer FU.

The minimum clock cycle time for TTPs is largely determined by the time needed to perform (and control) data transport. This depends on the number of connections to a bus. It is estimated to be (in nanoseconds):

$$T_{clock-cycle} = \max\{20, \max_{\forall \text{buses } b}\{12 + 0.4 \times \#\text{connections at bus } b\}\}$$

Its minimum value is taken 20 ns, the time to do an integer operation.

3.3 Software subsystem

The software subsystem is detailed in figure 6. It provides the user with three main tools to develop code for TTAs. These tools are 1) a compiler (based on GCC) to translate HLL (high level language) code to sequential move code, 2) a scheduler to schedule the sequential code and produce parallel code for a target TTA, and 3) a simulator and analyzer to verify and evaluate both

[4]ASA is a trademark of SAGANTEC Europe b.v., Eindhoven.

[5]Although the hardware subsystem can generate processors for any data width, the software subsystem currently requires 32 bit integers.

Parameter	Value or Expression
Integer unit	1
Full shifter	3
Logic unit	1
Ifetch unit	0.5
Load-store unit[a]	11.0
Multiplier (fixed point)	6
Divider (fixed point)	6
Compare unit (fixed point)	1
ALU (including multiply)	10
Floating point unit	20
Other Pins	4
Buses[b]	$3.5 + \text{Width}/16$
Registers[c]	$0.0125N \times \text{Width} + 0.0031 \times (\#RP + \#WP)$
Sockets	$0.02 + 0.0125 \times \text{Width} \times \#\text{Connections}$

[a] Including pads.

[b] Including pads for instruction fetch and on-chip wiring; each data bus corresponds to an instruction field.

[c] $\#RP$ and $\#WP$ indicate the number of read and write ports.

Table 2: Main values and expressions of the hardware cost model.

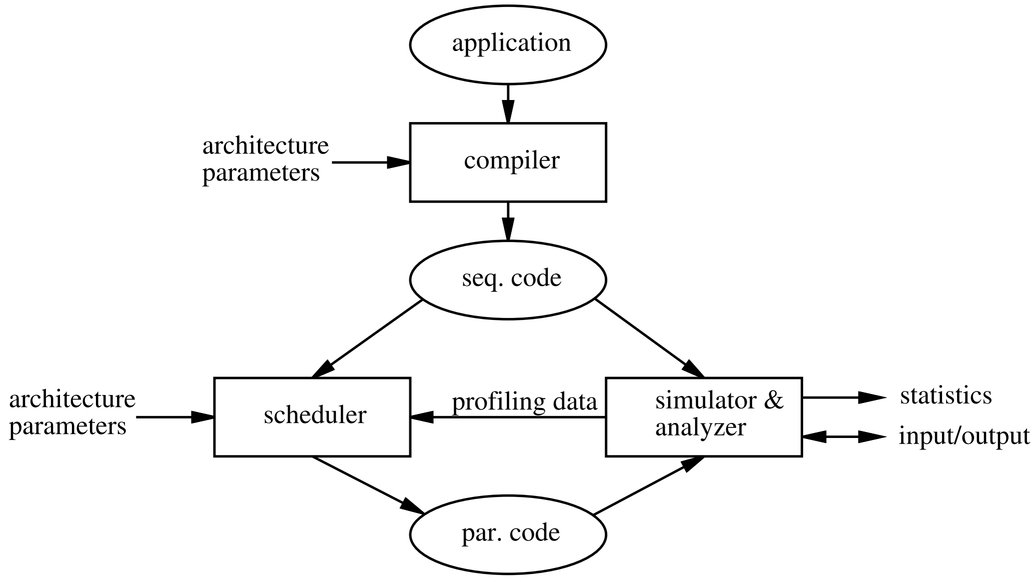

Figure 6: The software subsystem.

the sequential and parallel code. The software subsystem accepts any application coded in C or C++. The scheduler is the most important part of this subsystem. It has to generate instruction level parallel code, while exploiting all the available hardware resources. To this purpose it uses profiling data (like execution frequencies) from the simulator. The scheduler uses extended basic block scheduling and speculative execution in order to enhance code motions [3].

4 Experiments

We divide the problem of finding a good application specific TTP into two subtasks, resource optimization and connectivity optimization. Resource optimization consists of finding the right match of resources, such as which FUs to use, how many buses, how many GPRs, and how many ports on the register files. The goal of resource optimization is to reduce costs. Connectivity optimization determines the connectivity between the buses and the sockets. The resulting connectivity is based on the communication requirements between the FUs and the register files. The primary goal of connectivity optimization is to reduce the bus load and thus the cycle time.

4.1 Resource Optimization

The optimizer of the MOVE framework wants to present the curve of Pareto points to the designer so that he can decide what solution matches his requirements. It does that by means of a local search algorithm.

The local search algorithm starts with a design space point that corresponds with a oversized TTP, e.g., 64 GPRs, 16 buses, 4 load-store FUs, 4 adders, etc., and full connectivity between the buses and the sockets. From this configuration we go to a *neighbor* point by removing one resource. We repeat this until we reach a minimum configuration which is needed to compile the application successfully. Which resource is removed is determined by the following quality function that evaluates a design space point:

$$Quality = \frac{1}{ExecutionTime^a.Costs^b}$$

where a and b are constants which express the relative importance of performance and costs. Since connectivity optimization takes place after resource optimization we ignore the influence of the high bus load on the execution time.

We have two methods for choosing between neighbor points. The *best fit* method takes the neighbor with the highest quality; the *first fit* method takes the first neighbor that has a higher quality than the current point. The first fit method requires fewer points to be evaluated but it can be expected that the results are not as good as the best fit method.

After the local search algorithm has reached the minimum configuration we reverse the process and add resources until we reach the point where we started the search process. With this sweep back we hope to find new Pareto points. We make a few remove/add sweeps with different values for the a and b constants. The current version of the optimizer makes default five remove/add sweeps with a and b values between one and three.

To prevent unnecessary invocations of the hard and software subsystems all design points together with there execution time and costs are memorized in a hash table.

4.2 Connectivity Optimization

Connectivity optimization transforms the fully connected configuration found by resource optimization into a partially connected configuration that has less load on the buses and therefore a shorter cycle time. This is accomplished by removing bus socket connections from the buses in a round robin fashion. The bus socket connection that is removed from a bus is the first connection that has no influence on the cycle count. If no such connection exists we take the connection with the lowest influence on the cycle count. We repeat this process until the cycle time remains constant (limited by other factors than bus load) and the cycle count starts to increase.

Characteristic	Compress	Crypt
Percentage add/subtract operations	22%	25%
Percentage memory operations	26%	32%
Percentage shift operations	9%	2%
Percentage compare operations	10%	3%
Percentage logic operations	6%	34%
Parallelism upper bound based on trace analysis	6.6 ops/cycle	5.2 ops/cycle
Parallelism exploitable by our scheduler	2.5 ops/cycle	3.4 ops/cycle
Average number of live integer registers	11.3	19.7

Table 3: Characteristics of Compress and Crypt.

By removing connections in a round robin fashion we balance the bus load, i.e., all buses have approximately the same number of connections.

4.3 Benchmarks

For our experiments we shall use two applications, Compress and Crypt. Compress is the standard Unix tool for data compression. Crypt is a program for encrypting data based on the DES algorithm. Table 3 gives characteristics of these applications.

5 Results

In this section we will show how resource and connectivity optimization work out in practice for the Compress and Crypt benchmarks.

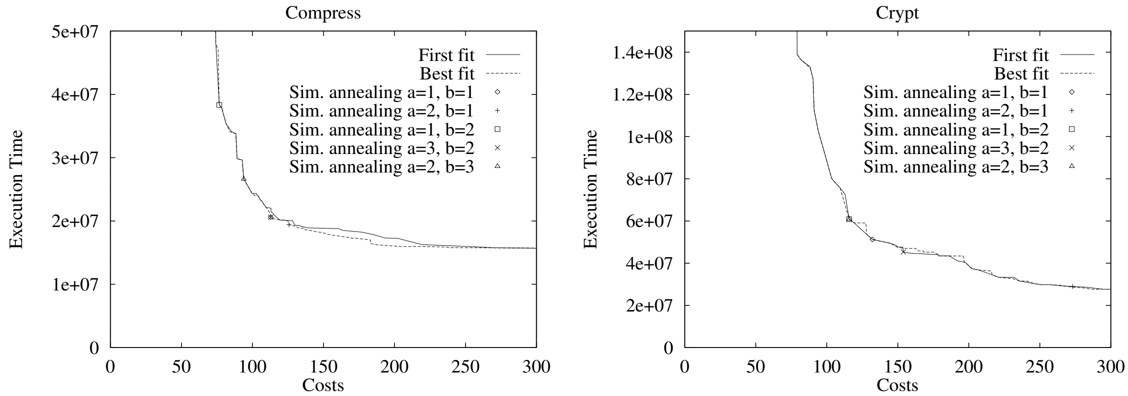

Figure 7: Resource optimization. Only the interesting parts of the curves are shown.

5.1 Resource Optimization

Figure 7 shows the curve of Pareto points for both benchmarks for the best fit and first fit methods. The curve of Compress confirms our intuition that best fit performs better than the first fit method. However, the Crypt benchmark shows some points were best fit performs worse.

In order to get confidence in our local search algorithm we have implemented a search algorithm based on simulated annealing [7] that tries to maximize the above mentioned quality function for given a and b. The results of searching by means of simulated annealing for five different values for a and b are also shown in figure 7. All points found by simulated annealing are also found by the local search algorithm which means that our local search algorithm performs well.

After the curve of Pareto points has been computed the designer selects a point on the curve that meets his requirements. In practice he wants to examine other points that are laying near to the selected point in the design space and reevaluate these points with other data sets and a more precise hardware modeller, or with the actual layout generator.

Let us assume that the designer takes the point on the curve that is also found by means of simulated annealing for $a = 2$ and $b = 1$. Figure 9 shows the TTPs that correspond with these points for Compress and Crypt with the difference that the interconnection network is still fully connected. The found configurations correspond with what could be expected on the basis of the characteristics of table 3. Crypt has more exploitable parallelism and therefore more buses, more registers simultaneously live and therefore more integer registers, less compares and therefore fewer boolean registers, and a lot of logic operations and therefore two logic FUs.

5.2 Connectivity Optimization

The results of connectivity optimization for the two benchmarks are shown in figure 9. We have stopped the process of removing connections at a point were the cycle count started to increase (see figure 8). At this point we had already reached the minimum clock cycle of 20 ns.

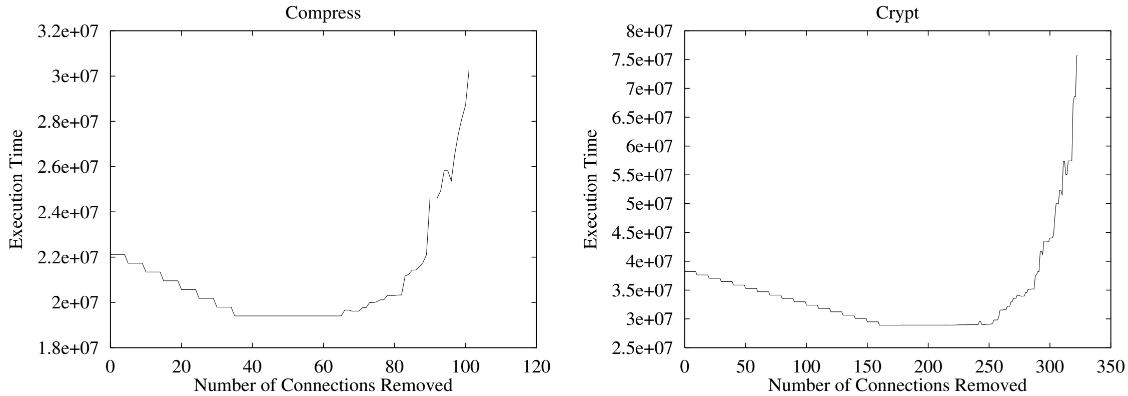

Figure 8: Connectivity optimization.

The connection count reduced from 120 to 61 and from 340 to 129 connections for Compress and Crypt respectively. The two configurations clearly show that the more important FUs and register files are better connected than the less important ones. For example the sign-extend FU of Compress is only connected to two buses, while the integer register files for both configurations are strongly connected.

6 Conclusions

In this paper we showed an automated design trajectory for ASIPs based on transport triggered architectures. This trajectory has two fundamental capabilities:

- It maps arbitrary applications, written in C/C++, into a combination of hardware and software.

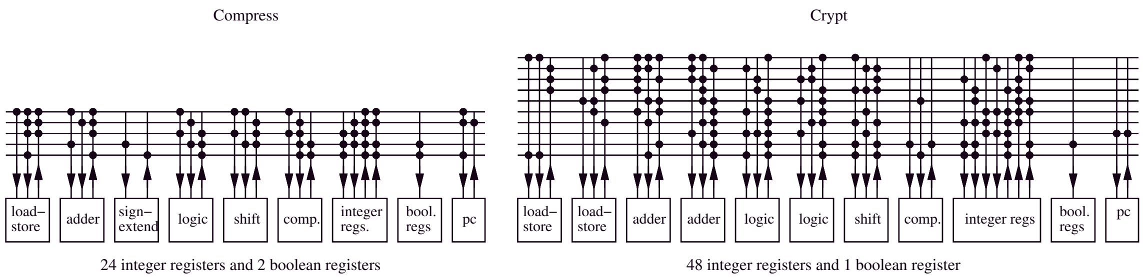

Figure 9: The resulting configurations after connectivity optimization.

Stage	Compress cycle time	Compress cycle count	Compress costs	Crypt cycle time	Crypt cycle count	Crypt costs
Initial oversized conf.	42.4 ns	763k	774	42.4 ns	1352k	774
After resource opt.	22.8 ns	970k	125	26.4 ns	1446k	273
After connectivity opt.	20.0 ns	970k	112	20.0 ns	1446k	229

Table 4: Summary of the search process.

- It offers the possibility to do a quantitative analysis of large parts of the design space.

The search process to find a proper solution consists of resource and connectivity optimization. Resource optimization attempts to find the cost effective set of resources. Connectivity optimization reduces the connectivity in order to reduce bus load and cycle time. We have applied this method for two benchmarks. Table 4 summarizes the results.

Still a lot of research and implementation has to be done. To mention a few items: 1) improving heuristics for faster design space exploration, 2) enlarge the design space, e.g., including caches, and 3) supporting multiple data widths simultaneously for hardware resources.

References

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

[2] Henk Corporaal and Paul van der Arend. MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.

[3] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Lecture Notes in Computer Science 786, Compiler Construction*, pages 435–449. Springer-Verlag, 1994.

[4] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *MICRO-27*, Santa Clara, December 1994.

[5] R. Brayton and R. Spence. *Sensitivity and Optimization*. Elsevier, 1980.

[6] Reinoud Lamberts. A move processor generator. Technical report, Delft Univ. of Technology, November 1993.

[7] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley and Sons, 1989.