Cosynthesis with the MOVE framework

Henk Corporaal Jan Hoogerbrugge

Delft University of Technology
Department of Electrical Engineering
P.O. Box 5031, 2600 GA Delft, The Netherlands

Abstract

Due to the decreasing feature size of VLSI technology, the amount of hardware which can be integrated into a single chip increases. As a result, future processor chips may execute tens of operations concurrently. Many applications can profit from these huge amounts of hardware parallelism by designing an application specific instruction set processors (ASIP). Two problems emerge however: 1) the design space of ASIPs is very large; it is difficult to chose a satisfactory solution, and 2) the design complexity increases and therefore design cycle gets too long.

The alleviate these problems we propose a design trajectory based on a *templated, transport triggered architecture* (TTA). 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.

As side effect a key aspect of TTAs will be demonstrated: the reduction of the on-chip data transport requirements; this may result in a better cost-performance ratio of the realized ASIPs.

Keywords: CoDesign, 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. This architecture is of the *instruction level parallel* type; it resembles the well known VLIW architecture. However, a key difference is that TTAs 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 for designing ASIPs, the design space 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 the MOVE framework, and discusses briefly how TTAs operate. Section 3 lists the major parameters of the architecture design space and shows how this space is explored. The results of applying the MOVE framework to two benchmarks is given in section 4. Finally, in section 5 several conclusions are drawn and a comparison with related work is made.

2 The MOVE framework

Within the synthesis process we use an *architecture template*, i.e. processors are built according to the pattern of a TTA. A specific TTA is defined by a set of architecture parameters, like the number and type of function units, the number of register files and registers, etc. At first sight this suggests that we restrict ourselves and therefore obtain inferior solutions. In practice however, several advantages emerge. Firstly, the template building blocks are pre-designed and can therefore be made very efficient, both in area and performance. Secondly, 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. Note, that the template still covers a very large design space.

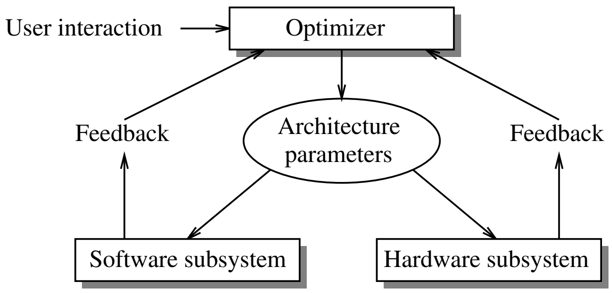

Figure 1: MOVE framework overview

The synthesis of hardware and software for a given application is done using the MOVE framework; this framework produces both the layout of an ASIP and the corresponding object code to be executed on this ASIP. An overview of this framework is shown in figure 1. It basically consists of three components:

- **Optimizer** which is responsible for searching the design space and the interaction with the designer. It determines the configuration (i.e., the set of architecture parameters) to be evaluated next.
- **Hardware subsystem** generating processor layout, and giving information on timing, area, and power consumption.
- **Software subsystem** generating instruction level parallel code, and giving statistical information on usage of hardware resources.

These components are detailed in following subsections. Before, we briefly describe how TTAs operate.

2.1 Transport triggered architectures

TTAs can be compared to VLIW architectures; their instructions are horizontally encoded; i.e. each instruction has a number of fields. Whereas fields for VLIWs specify RISC like operations, for TTAs they specify the required data transports. These transports may trigger operations as side effect. Programming transports adds an extra level of control to the code generator, and enables new optimizations; in particular, it allows us to get rid of many superfluous data transports to and from the register files and to reduce the on-chip connectivity[1].

A compiler views a TTA as a collection of function units (FUs), register files (RFs), *move buses*, and *sockets*; see figure 2. FUs perform operations, RFs provide temporary fast accessible storage, the network of move buses performs data transports between the FUs and RFs, and sockets interface FUs and RFs to move buses. Normally, each socket is connected to a different FU input/output or RF port.

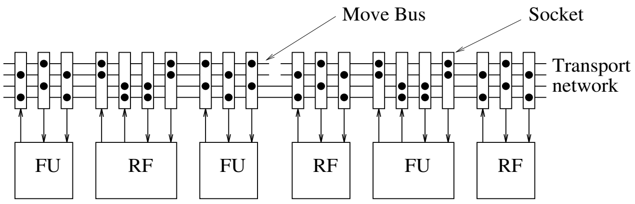

Figure 2: General structure of a TTA.

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 buses and RF ports directly to FUs, as is the case for many VLIW and superscalar architectures, 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 [1].

The interconnection network may be fully connected, in which case every socket is connected to all move buses, 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 buses 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.

Besides executing operations on data, TTAs need to provide immediate operands, conditional execution, and control flow changing operations. Details on these issues can be found in [2].

2.2 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[1]. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time. 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 (described in section 4). As shown, the solution space is bounded by a curve connecting so called *Pareto points*. These points are local optimal solutions; they are defined by [3]:

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

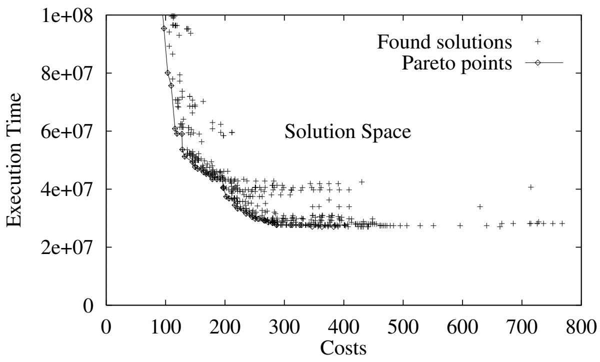

Figure 3: Possible solutions and Pareto points.

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.

2.3 Hardware subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of an application specific TTA 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 4 shows its organization.

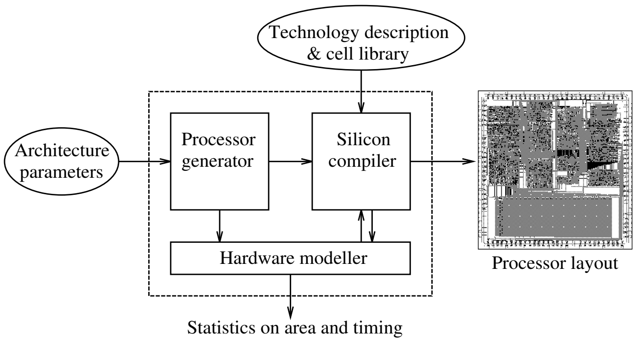

Figure 4: The hardware subsystem; the shown processor layout is of the Phoenix processor realized with the MOVE framework [4].

[1]Currently we included area and pins only.

The hardware subsystem makes use of the ASA silicon compiler[2], or, alternative relies on Synopsys and Cadense tools. Details of this system and realized processors are described in [5].

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 1. The costs of FUs are based on a 32-bit data path width, and relative to an integer FU. The model can freely be changed by the designer.

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]	$3.5 + W/16$
Registers[c]	$0.0125N \times W +$ $0.0031 \times RP$
Sockets[d]	$0.02 + 0.0125 \times W \times C$

[a] Including pads.
[b] Including pads for instruction fetch and on-chip wiring; each data bus corresponds to an instruction field; W equal the bus-width in bits.
[c] RP is equal to the number of register file ports; N the number of registers.
[d] C equal the number of connections.

Table 1: Example values and expressions of the hardware cost model, as used in this paper.

The minimum clock cycle time for a TTA realization 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 (its minimum value is taken 20 ns, the time to do an integer operation; C_b is the number of connections on bus b):

$$T_{clock-cycle} = \max\{20, \max_{\forall \text{buses } b}\{12 + 0.4 \times C_b\}\}ns$$

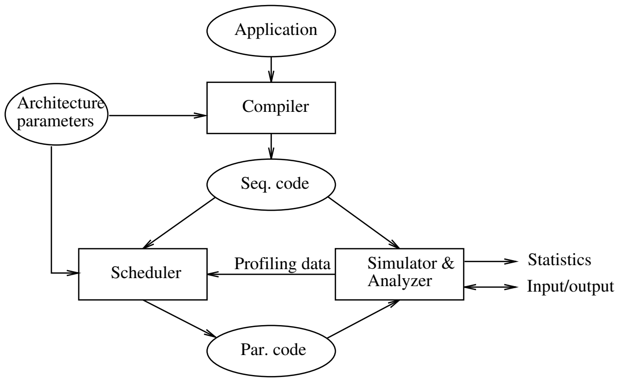

Figure 5: The software subsystem.

2.4 Software subsystem

The software subsystem is detailed in figure 5. 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 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 advanced techniques like extended basic block scheduling, software pipelining, and speculative execution, in order to enhance code motions [6].

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

3 Design space exploration

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

We divide the problem of finding a good application specific TTA 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 registers, 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.

Parameter	Range
#buses	≥ 1
#FUs	≥ 1
FU latencies	≥ 1
FU type	all library supported types
FU pipelining method	virtual-time or hybrid[2]
#registers, IRF[a]	≥ 1
#input ports, IRF	≥ 1
#output ports, IRF	≥ 1
#registers, FRF[b]	≥ 0
#input ports, FRF	≥ 1
#output ports, FRF	≥ 1
#registers, BRF[c]	≥ 1
#input ports, BRF	≥ 1
#long immediates	≥ 1
bus socket connection	yes, no[d]

[a] Integer register file.
[b] Floating point register file.
[c] Boolean register file.
[d] Any connection can be specified individually.

Table 2: Main parameters of the architecture design space.

3.1 Resource Optimization

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

The local search algorithm is described in figure 6. It starts with an initial configuration that corresponds with a oversized TTA, e.g. one with 64 registers, 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:

$$Quality = \frac{1}{ExecutionTime^{\alpha}.Costs^{\beta}}$$

where α and β are constants which express the relative importance of performance and costs. Since connectivity optimization, which takes care of the bus load, is performed after resource optimization resource optimization ignores the influence of the high bus load on the execution time.

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 α and β constants.

During the remove-sweep we want to stay as close as possible to the cost axis. This is achieved by $\beta > \alpha$. Similarly, during the add-sweep we want to stay close to the execution time axis which is achieved by $\alpha > \beta$. The current version of the optimizer makes default five remove/add sweeps with α and β values between one and three. The set of used α and β values was obtained by means of experimentation. Adding more pairs to it did not lead to finding significantly more Pareto points, it only increased the exploration time.

```
proc ResourceOptimization(init_config)
beginproc
  curr = init_config
  for each α ∈ {1, 1.5, 2, 2.5, 3} do        /* quality parameters */
    repeat                                   /* reduce sweep */
      neig = {n ∈ P(init_config) | n = curr − {r_i},
              r_i ∈ curr, IsValid(n)}
      cand = {n ∈ neig | ∀n' ∈ neig,
              Quality(n, 1, α) ≥ Quality(n', 1, α)}
      if cand ≠ ∅ then
        curr = Select(cand)
      endif
    until cand = ∅
    repeat                                   /* extend sweep */
      neig = {n ∈ P(init_config) | n = curr ∪ {r_i},
              r_i ∉ curr}
      cand = {n ∈ neig | ∀n' ∈ neig,
              Quality(n, α, 1) ≥ Quality(n', α, 1)}
      if cand ≠ ∅ then
        curr = Select(cand)
      endif
    until cand = ∅
  endfor
endproc

proc Quality(config, α, β)
beginproc
  return Costs(config)^−α · CycleCount(config)^−β · CycleTime(config)^−β
endproc
```

Figure 6: The algorithm for resource optimization. A configuration is represented as a set of resources r_i. $\mathcal{P}(init\_config)$ represents all subsets of the initial configuration. IsValid(c) determines whether configuration c has sufficient resources to compile the application successfully. Select(s) returns a member of s, either using a first fit or best fit approach. Costs(c), CycleCount(c), and CycleTime(c) return the costs, cycle count, and cycle time, respectively, by invoking the compiler back-end and hardware modeler.

The resources that are considered by the explorer are currently: (1) FUs, (2) move buses, (3) registers (integer, floating point, and boolean), and (4) RF ports. Removing a move bus from a configuration is done incrementally by reducing its width in steps from 64 bits to 32 bits to 1 bit to 0 bits. Adding move buses proceeds in the other direction. This can lead to configurations such as two 64-bit buses for floating point numbers, six 32-bit buses for integers, and one 1-bit bus for booleans. Registers are removed/added in groups such that the number of registers per RF is always 2^n or $2^n + 2^{n+1}$, $n \geq 0$. This is because the difference in cost and performance of, for instance, a 32 and a 31 integer RF configuration is very small.

The exploration time of the algorithm shown in figure 6 can be fairly long, especially if the initial configuration is very large and the evaluation time of a configuration is long. Several methods to reduce the exploration time are applied:

1. Evaluating the configuration only for the most critical procedures of the application. For example, the set of procedures where 99% of the execution time will be spent. This often excludes possibly long procedures which are seldomly executed.
2. Using a rule-based hardware model, as discussed in section 2.3 instead of a processor generator to estimate cycle time and costs.
3. Removing resources from the initial configuration that have no effect on the cycle count before the actual exploration begins. These resources will not be added during the extend-sweep.
4. Memorizing all configurations together with their evaluation results in a hash table to prevent unnecessary evaluations.
5. Selecting a neighbor configuration immediately if it has a better quality than the current configuration instead of evaluating all neighbor configurations and selecting the best one. We call this *first fit* instead of *best fit* exploration.

3.2 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.

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 Experiments

To demonstrate the operation of the MOVE framework two applications, Compress and Crypt, are used. 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. The parallelism upper bound is measured with trace analysis, while the exploitable parallelism is measured with our compiler. We will show how resource and connectivity optimization work out in practice for the Compress and Crypt benchmarks.

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

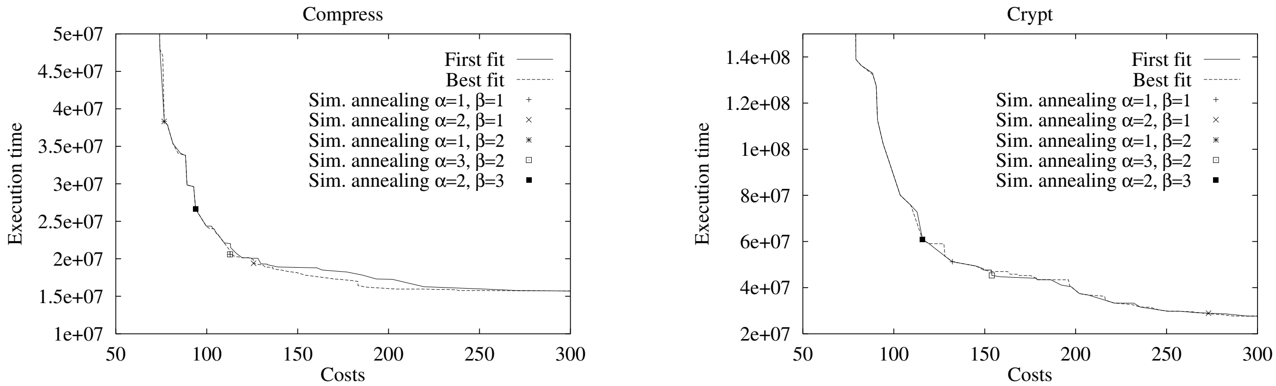

Figure 7: Resource optimization. Only the interesting parts of the curves are shown.

Characteristic	**Compress**	**Crypt**
% add/subtract operations	22%	25%
% memory operations	26%	32%
% shift operations	9%	2%
% compare operations	10%	3%
% logic operations	6%	34%
Parallelism upper bound	6.6 ops/cycle	5.2 ops/cycle
Parallelism exploitable	2.5 ops/cycle	3.4 ops/cycle
#integer registers live (average)	11.3	19.7

Table 3: Characteristics of Compress and Crypt.

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 that tries to maximize the above mentioned quality function for given α and β. The results of searching by means of simulated annealing for five different values for α and β 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 $\alpha = 2$ and $\beta = 1$. Figure 9 shows the TTAs 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.

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.

The connection count reduced from 120 to 61 and from 340 to 129 connections for Compress and Crypt respectively; so a reduction of 50% or more is readily achievable. 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.

Table 4 summarizes the results of the optimizations. For the configuration shown in figure 8 Compress achieve an average number of 2.0 operations per cycle. A VLIW would require at least 3 FUs in order to achieve this parallelism, which means that its (integer) register file requires at least 9 read and write ports. Our TTA requires only 4 ports. Similarly, the selected TTA for Crypt has 8 ports on its integer register file. A comparable VLIW would require at least 15 ports to achieve the same average amount of parallelism (3.2 operations per cycle). This demonstrates that TTAs require far less register ports.

Compress	cycle time	#cycles	ops/cycle	costs
Initial large conf.	42.4 ns	763k	2.5	774
After resource opt.	22.8 ns	970k	2.0	125
After connect. opt.	20.0 ns	970k	2.0	112
Crypt	cycle time	#cycles	ops/cycle	costs
Initial large conf.	42.4 ns	1352k	3.4	774
After resource opt.	26.4 ns	1446k	3.2	273
After connect. opt.	20.0 ns	1446k	3.2	229

Table 4: Summary of configurations found by the search process.

5 Conclusions and related work

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.
- 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. As side effect it was shown that the synthesized TTAs require far

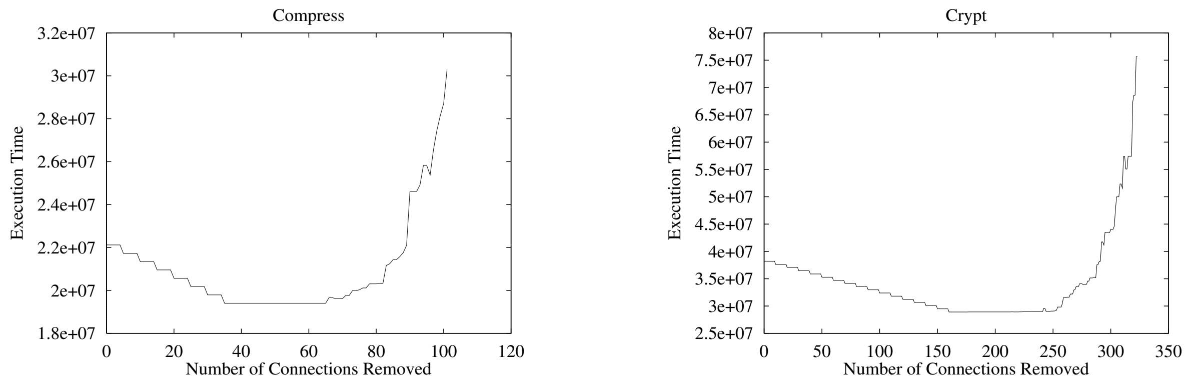

Figure 8: Connectivity optimization.

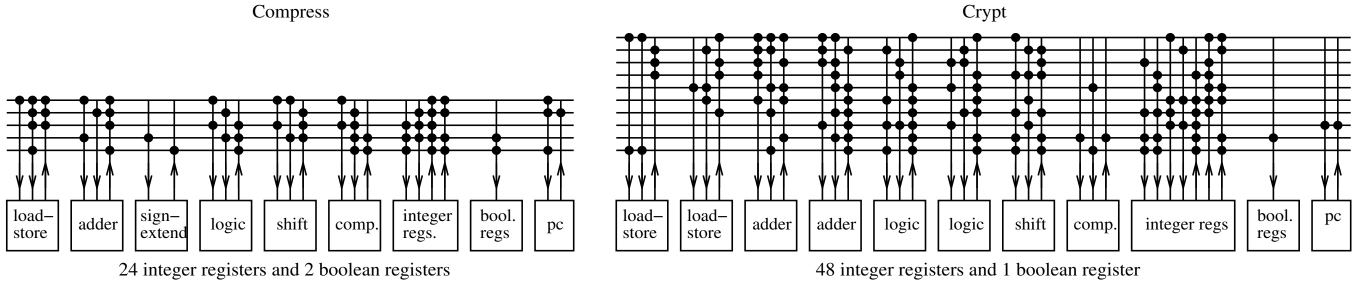

Figure 9: The resulting configurations after connectivity optimization.

less connectivity and fewer register ports than more traditional instruction level parallel architectures.

Related work

The synthesis process used within the MOVE framework has much in common with high level synthesis (HLS) systems which have been developed (see e.g. [7, 8, 9, 10]); they both aim at the automatic mapping of a functional specification into a structural description. There are several differences, though:

- Most HLS systems do not accept 'full blown' applications written in commonly used imperative languages. Usually their usage is restricted to input from small applications written in a special language [11], containing only one or a limited number of functions.
- The MOVE framework limits the design space by using a templated TTA. As indicated in section 2 this has several advantages. It also does not make sense to include design points within the design space for which we can not compile and synthesize efficiently.
- Our template TTA is fully programmable, pipelined, and if required, interruptable. Although TTPs can be optimized to run critical code parts of an application, non critical parts can run on the same processor as well. On the other hand, many HLS systems are meant to be used for synthesizing highly application specific data paths, which are controlled by either FSMs or microcode. It is therefore difficult to adapt the functional specification once a processor has been synthesized, let alone that completely unrelated functions could be processed. In this sense our approach is more flexible.

In short, the MOVE framework aims at a fully programmable solution for large applications written in commonly used HLLs, while HLS systems are more oriented towards the synthesis of special function units (SFUs), or of processors which are dedicated to process a given function. In fact a synergy between both approaches is recommended. With this synergy HLS systems are used to design SFUs which are then automatically incorporated within the MOVE framework. The MOVE framework is already prepared to handle those SFUs [12].

References

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

[2] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

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

[4] Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, pages 18–27, Heijen, The Netherlands, May 1995.

[5] Reinoud Lamberts. A MOVE Processor Generator. Technical report, Delft Univ. of Technology, November 1993.

[6] Jan Hoogerbrugge. *Code generation for Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996.

[7] Raul Camposano and Wayne Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[8] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-level Synthesis; introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[9] Petra Michel, Ulrich Lauther, and Peter Duzy, editors. *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992.

[10] Robert A. Walker and Raul Camposano, editors. *A survey of high-level synthesis systems*. Kluwer academic publishers, 1991.

[11] Jan Vanhoof et al. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer academic publishers, 1993.

[12] Henk Corporaal and Jan Hoogerbrugge. Code generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, pages 240–259. Kluwer Academic Publishers, 1995. Chapter 14.