Using Transport Triggered Architectures for Embedded Processor Design

Henk Corporaal Marnix Arnold

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: {heco,marnix}@cs.et.tudelft.nl

Abstract

There exists a huge market for application specific processors used within embedded systems. This market is driven by consumer requirements (e.g., new products, more functionality and flexibility, product digitalization, better performance-cost ratio, portability) and processor design capabilities (i.e., what can we offer). Being successful within this market requires a short time-to-market; this necessitates usage of automated design tools. These tools should not only assist in the quick generation of a specific processor, but also enable the designer to investigate quickly, and quantitatively, a large set of alternative solutions. Therefore, these tools should be based on a flexible and programmable processor template.

In this paper we propose the usage of Transport Triggered Architectures (TTAs) for such a processor template. TTAs can be compared to VLIWs; both can exploit the compile-time available instruction level parallelism. However, TTAs are programmed differently. TTAs combine a set of interesting features; apart from being fully programmable, they have favorable scaling characteristics, they easily incorporate arbitrary functionality, and their organization is well structured, allowing simple and automatic design. The paper explains these features.

Based on this template a set of design tools has been developed; they include a parallelizing C/C++ compiler which exploits the available processor and application concurrency, a processor generator, simulators, profilers, and a tool for architecture exploration; these tools are integrated within a graphical user interface. Within the paper we shortly describe these tools and demonstrate how they can be applied to a particular application. This example application is taken from the image processing area. It will be shown how the tools assist in exploring many solutions, including those which incorporate application specific functionality.

Keywords: Transport triggered architectures, Cosynthesis, Design space exploration, ASIP, Neighborhood operations, VLIW

1 Introduction

Processors play an ever increasing role in all aspects of life. Every year, billions of them find their way in all kinds of equipment, ranging from highly dedicated systems to general purpose workstations. It will be clear that these systems have to satisfy largely different requirements: they are designed for completely different applications, with different cost and performance constraints. Because a short time-to-market becomes ever more important for successful commercial introduction of new equipment, the time which can be spent on development is very limited; in addition, the design effort should be low to reduce costs.

The computer architect is faced with an almost impossible job when designing and implementing an architecture for a processor based system. Numerous design decisions, whose effects on the final result are difficult to predict precisely, have to be taken. The effect on performance and cost of supporting a certain feature not only depends on the application for which the architecture has to be designed, the quality of the compiler to deal with this feature, and the quality of the implementation of this feature, but, most frustratingly, also on many of the other decisions taken so far. Implementing *each* feature efficiently in hardware, and within the compiler, such that a quantitative cost-performance analysis can be made, is infeasible.

A solution to this problem is to use a highly *automated design trajectory* which can handle a large range of applications. In section 3 we argue that this design trajectory must be based on a *templated* architecture which satisfies the following five requirements: *scalability, flexibility, programmability, modularity*, and finally, *software compatibility*. Scalability and flexibility are needed to cover a wide range of solutions; programmability is needed to handle any application written in a high level language; modularity is necessary for efficient and automatic synthesis; software compatibility guarantees that we can reuse existing, developed, code, by re-compilation for a different instantiation of the template.

Existing classes of architectures can not meet these requirements simultaneously. The most promising candidate is the class of VLIW architectures; VLIWs can offer support for the exploitation of large amounts of instruction level parallelism (ILP) by adding numerous function units (FUs). However, this class does not fully satisfy the scalability, flexibility and modularity requirements. It will be shown that realizations of these architectures are unnecessarily complex. Solutions created with high level synthesis systems usually do not satisfy the programmability and software compatibility requirements. Therefore a new class of architectures is proposed and described, the class of **transport triggered architectures**, or **TTAs**.

The operating principle of a TTA is very simple. A TTA can be viewed as a set of FUs and register files (RFs), connected by an interconnection network. Just like VLIWs it offers support for ILP. A TTA is programmed by specifying the required data transports. FU operations occur as side effect of these data transports. For most traditional architectures a program has to specify the FU operations; the data transports required for these operations can not be controlled by the compiler, they are scheduled by hardware. For TTAs the compiler takes over this control. This results in numerous advantages, both at the hardware and the software level. More importantly, TTAs can serve as a template from which processors can be generated automatically. To this purpose a set of tools, called the *MOVE framework* has been developed. This framework allows for easy exploration of the architecture design space for any given application coded in C or C++. In this paper the framework is described and demonstrated by two example application from the image processing domain: edge detection and convolution.

This paper is structured as follows. Section 2 gives an overview of processor design goals, and argues that ASPs are useful but their design is problematic. Section 3 investigates the required template characteristics. Section 4 describes TTAs and their main hardware and software properties. It will be shown that TTAs match the template characteristics. Section 5 gives an overview of the MOVE framework. Section 6 introduces the example application used and demonstrates the operation of the MOVE framework. Finally, in section 7 we present several conclusions and recommendations.

2 Processor design goals

When designing a processor for a processor based embedded system a designer may have several optimizations in mind:

1. *Cost optimization.* Factors influencing costs are divided in recurrent and non-recurrent cost. Recurrent costs are influenced by: chip area, package and number of external connections, and the used

process technology; using a well established process technology is cheaper and gives a higher yield. The design time and the production of layout masks, if a full custom realization is used, largely determines the non-recurrent costs.

2. *Power optimization.* Low power consumption will have a high priority for battery powered equipment. Usage of this type of equipment will assume enormous proportions.
3. *Application optimization.* One may design dedicated processors which are highly customized for a given application, or one may try to design a more general applicable processor which can handle a larger range of applications. It will be clear that a dedicated processor may achieve more performance, and consume less VLSI area.
4. *Performance optimization.* If this is the main goal then the processor must give the highest performance for a specific application, while costs and power constraints are less important.

The above optimization requirements are often contradictory. Its up to marketing to target, given the technological and architectural possibilities, the right market spot, i.e. the right combination of design time, VLSI technology, design method (like the use of full custom chips, Sea-of-Gates or programmable gate arrays), performance, area, application domain, etc.

Instead of designing an application specific processor (ASP) for an embedded system one could also *buy* an existing general purpose processor (GPP), which is designed to execute many different applications with reasonable performance, or an available digital signal processor (DSP), whose design is tuned towards signal processing applications. These processors provide a cost-effective solution for applications with typical (i.e., not too divergent) computational requirements. E.g., a standard micro-controller type processor is well suited for managing the user interface (buttons and display) of video equipment. On the other hand, applications or application domains with requirements that do not map efficiently onto standard components call for ASPs.

Characteristics	**GPP**	**DSP**	**ASP**
Costs:			
• Non-recurrent	zero	zero	very high
• Recurrent	high	medium	low
Power consumption	high	medium	low
Functionality	high	medium	low
Performance	medium	high	very high

Table 1: Comparing characteristics of three different processor classes.

Those characteristics of these three processors, which are directly related to the mentioned optimizations, are given in table 1. The non-recurrent costs of GPPs and DSPs do hardly exist, because they can be taken off-the-shelve; for ASPs they can be very high. Recurrent costs may be low for ASPs, if the market volume is large enough, because they include only that functionality which is required by the application. This also holds for the embedded system itself, because an ASP is often cheaper to interface with the surrounding logic. Power consumption of high performance GPPs is very high; ASPs may consume less power because they do not include superfluous functionality. GPPs score much better on functionality; they are more general applicable. Also, their programming environment is usually much better than for DSPs and ASPs. Finally, ASPs may include special functionality which gives them a performance edge for those applications for which they are designed.

The previous discussion suggests that ASPs are interesting in several aspects. However, doing a dedicated design for some application comes at a cost. Using a traditional design trajectory takes significant investment and effort. This may induce the following ASP design problems:

1. *Long time-to-market*, and therefore use of outdated technology. The long design trajectory causes ASP performance, as compared to available GPPs and DSPs, to be far lower than expected, because the latter currently see an annual performance improvement of more than 50% [HP95].
2. *High non-recurrent engineering cost*: no reuse of developed hardware and software.
3. Only *limited design space exploration* possible, and therefore limited opportunity for applying the discussed optimizations.
4. *High design risk*: the design may never achieve correctness or the required level of performance.
5. Tendency towards *limited programmability* due to design constraints. This leads to difficulty with running complete applications besides the performance-critical routines, and inability to adapt to changes of algorithms or application requirements.

The solution to these design problems is discussed in the next section.

3 Template characteristics

To overcome the ASP design problems we would like to define a class of architectures from which ASPs can be automatically generated. This class should cover a large range of applications with very different performance and functionality demands. This results into the following class requirements:

1. *Scalability*, such that the performance of the processor can be increased to high performance levels without changing the architecture concept underlying the processor.
2. *Modularity*, making automatic synthesis of many different instantiations tractable. Processor building blocks have to be designed and implemented once, but can be reused many times.
3. *Programmability*, to handle large applications and become more immune towards algorithmic changes.
4. *Flexibility*, which means that the functionality of the processor can be easily adapted to changing market requirements.
5. *Software compatibility*, allowing to reuse software. Software compatibility can be achieved either at the architectural level, or at the high level language (HLL) level. The former is impossible when designing ASPs for completely different applications. However, it should be possible to compile HLL code, without first having to write a new compiler.

In the following two subsections we will concentrate on the first four requirements. The fifth requirement can be met by having a *retargetable compiler*; such a compiler can be designed if the modularity requirement has been fulfilled [Hoo96]. A potential template to fulfill these requirements is a VLIW. It will be shown however that VLIWs have problems with scalability and modularity.

3.1 Scalability and Modularity

The computer architect has several techniques at his disposal to scale the processor to higher levels of performance [Cor95]. One of the most important techniques, which is currently generally applied, is to add more, concurrently operating FUs. FUs are responsible for the execution of the operations. Two main classes of architectures use this technique: superscalars and VLIWs.

In order to keep the FUs busy the parallelism contained in a program has to be detected and exploited. Superscalars use hardware to detect this parallelism, while VLIWs rely on compile-time detection of

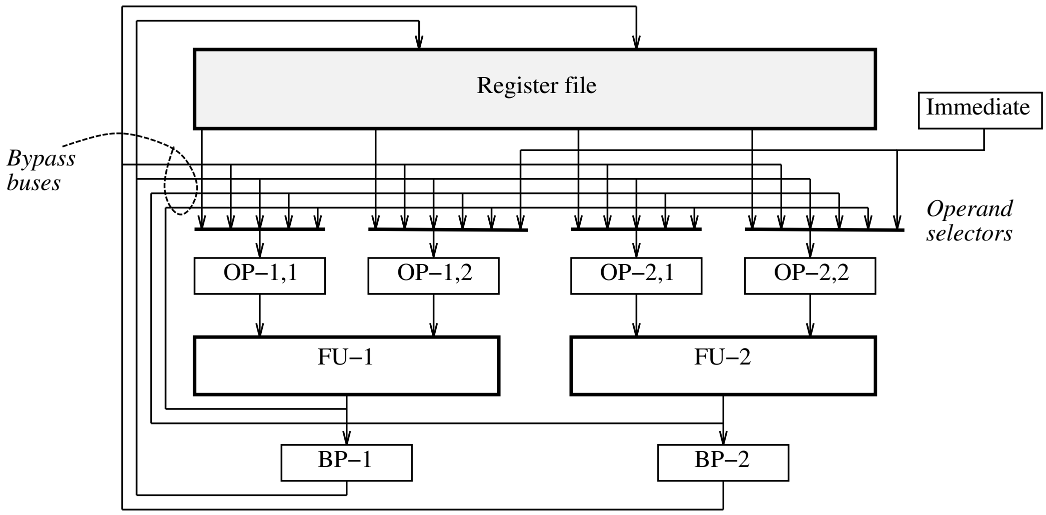

Figure 1: Possible data path of a VLIW processor with 2 function units.

parallelism. This has consequences for scalability. Superscalars require much hardware for the run-time detection of parallelism, in case many FUs are added. VLIWs do not need that hardware and are therefore much more scalable. However, VLIWs may run into complexity problems as well. Let us illustrate these problems by looking at figure 1, showing a VLIW with 2 FUs. In order to let FU results bypass the register file 4 bypass buses, 2 bypass registers (BP-i), and input selectors at each FU input operand register (OP-i,j) are required[1]. Furthermore, the register file, which is shared between the 2 FUs, requires 6 ports (4 read and 2 write ports) in order to let each FU operate at single cycle throughput. This figure makes clear that in the general case, when having N FUs, the bypass complexity of a VLIW is $\theta(N^2)$ and the number of register ports is $\theta(N)^2$. So, VLIWs need much data transport capacity (i.e. a large number of buses and ports) for mutual FU traffic and for transports between FUs and the register file(s). This has a large impact on both area and timing characteristics of VLIW realizations.

In the worst case (code) scheduling situations this capacity may be necessary, however, average usage is much lower. The reason is twofold:

1. Many results can directly be bypassed to (other) FUs. This results in a reduction of both read and write accesses on the register file; it also reduces somewhat the number of required registers. Figure 2 illustrates this effect. It shows, for the well known Stanford benchmark the number of reads and last usages of FU results as a function of their life-time (when scheduled for a large VLIW); e.g., a value produced (and thus defined) in cycle 200 and (only) used in cycles 200 and 203 is counted as a read at life-times 0 and 3 and a last-usage at life-time 3. It turns out that many FU results are only used in the same cycle as they are produced (about 40%), and that many results (about 45%) have their last usage in the first cycle; these values do not have to be written in the register file.
2. Not all connections between FU inputs and outputs are equally important. E.g., table 2 shows the inter FU data transfer top 20 when running an MPEG2 audio decoder. This table demonstrates that for specific applications the inter FU traffic is not uniform, but heavily biassed. This suggests that the inter FU connectivity may be reduced without sacrificing performance.

[1]Note that bypassing is highly recommended; it reduces the experienced latency of each operation and therefore the critical path in the data dependency graph.

[2]In fact, as shown in [Cor95], the actual bypass complexity is even worse, namely $\theta(N^2 \log(N) log(log(N)))$.

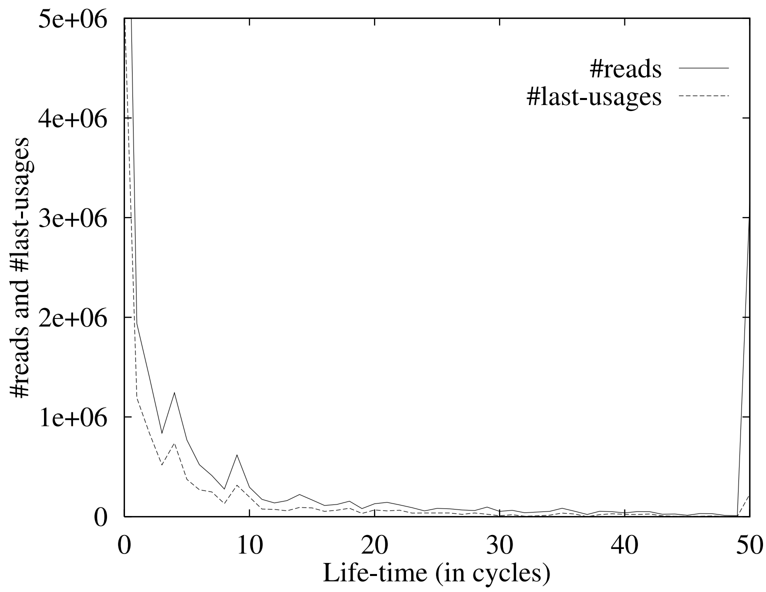

Figure 2: Counts of the number of reads and last usages of FU results as a function of their life-time.

The former suggests that there are opportunities for improvement; however this requires that all the inter FU data transports and transports between FUs and register file(s) are compile-time controlled. This is not the case for VLIWs and other traditional architectures. Therefore a different architecture is needed.

Modularity requires that the generated processors are build out of a restricted number of building blocks. These building blocks are pre-designed and efficiently implemented. Apart from application specific function units, which have to be special designed, any required ASP can be designed by combining more or fewer of these building blocks.

A VLIW architecture is a good candidate for satisfying the modularity requirement; e.g., FUs can be added to a VLIW without change of their architecture concept. However, as discussed in the scalability paragraph above, their required transport capacity does not scale well when adding FUs; in fact it is directly related to the number of FUs. This means that buses and FUs can not be added independently.

In case the FUs share the same register file, the problem even gets worse: adding a FU means a complete redesign of the register file, because it needs more ports. Further, the addition of a FU may add a field to the instruction format as well, and therefore change the instruction cache and/or the external instruction data path. For the same reason, the addition of a SFU, which is only occasionally (within one routine) used, may lead to sparse instruction encoding. These problems can partly be solved by adding crossbars between FUs and register files, as done in the TriMedia processor [Cas94], however, this may turn out to be a costly solution.

3.2 Programmability and Flexibility

Usually ASPs have a limited *programmability*; they only support a restricted set of operations. Full programmability requires that besides an instruction fetch unit the regular set of RISC (reduced instruction set computers) operations is supported, such that it is possible to efficiently compile HLL programs for these processors; ASPs often lack this functionality. A standard way to solve this problem and make an embedded system both programmable, such that it can run large compiled HLL programs, and application specific, such that it can run critical routines fast enough, is to use a combination of a GPP and a ASP; the ASP acts as a *coprocessor* of the GPP [Koc94]. This way the programming environment of the

Transport	Transport count	Perc.%
add-r →add-o	362,715	29.9
add-r →ld-t	179,660	14.8
ld-r →mulfx-o	79,488	6.5
ld-r →mulfx-t	78,792	6.5
mulfx-r →add-t	70,272	5.8
shl-r →add-t	53,790	4.5
add-r →gt-o	52,319	4.3
add-r →gt-t	48,616	4.0
and-r →shl-o	38,088	3.1
add-r →mul-o	36,864	3.0
mul-r →and-o	36,864	3.0
sub-r →gt-o	36,693	3.0
add-r →mul-t	34,560	2.8
shl-r →add-o	23,090	1.9
add-r →st-o	20,139	1.7
add-r →add-t	18,648	1.6
sub-r →sub-o	11,376	0.9
add-r →st-t	11,105	0.9
add-r →shl-o	10,241	0.8
and-r →gt-o	9,740	0.8
All	1.213.060	100

Table 2: Inter FU data transfer top 20 measured for MPEG2 audio decoding. Register to register copies and traffic of immediate operands is not included; add →mul-1 indicates a transfer from an add operation to the first input operand of a multiply operation; similar for the other transfers.

GPP can be used. Since we can implement arbitrary functionality within the ASP the flexibility of this solution is guaranteed. However, this coprocessor approach has four disadvantages:

1. An extra component is needed, and the interface of the ASP should be adapted to the GPP coprocessor interface. This makes this solution more complex and expensive.
2. Communication of instructions and/or data is required between the GPP and the coprocessor.
3. The coprocessor is used only to speedup some critical routine(s); it is not used for the other, more serial code.
4. Functionality has to be duplicated. E.g., a coprocessor may require addressing and load-store functionality, which is already available in the GPP, for performance purposes.

A far better solution is to use an application specific instruction set processor (ASIP) and have the *flexibility* to add *special function units* (*SFUs*). These units offer special hardware functionality (i.e., anything which goes beyond the commonly supported RISC like operations) that speeds up the execution of the critical routines. This solution does not suffer from the communication overhead because SFUs have, in contrast to coprocessors, access to the same register set as regular FUs. SFUs can also, although to a lesser extend, be used for serial code parts. The case study presented in this paper demonstrates this effect.

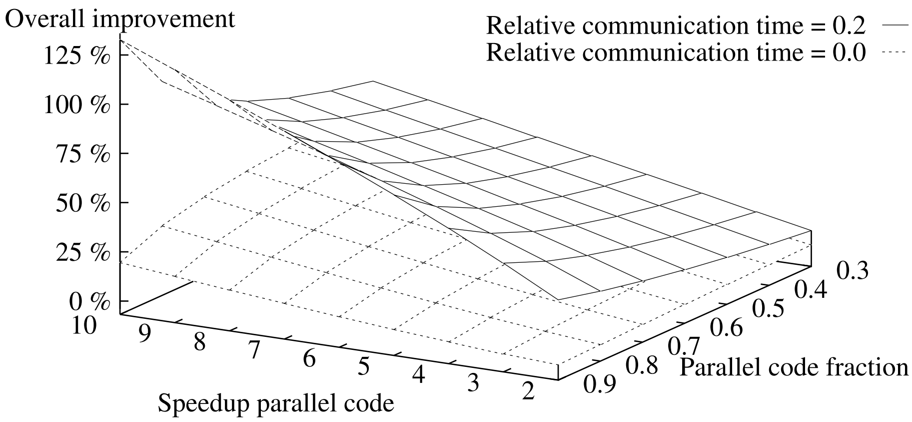

Figure 3: Performance improvement of a processor with SFU compared to one with a coprocessor.

It may also occur, that, as soon as we have the capability of adding FUs (because of the scalability requirement), it is more efficient, and easier, to add more regular FUs instead of a SFU; the reasons are well known to people designing RISC processors [HP95]. In this case the serial code part profits even more from these extra FUs.

To gain some insight, the performance of a processor with an extra SFU is modelled and compared to a GPP processor with a coprocessor. The coprocessor approach is discussed first. Assume a fraction f of a program, called the parallel part, is down-loaded to and executed by the coprocessor; the execution time of this fraction decreases with a speedup factor S_{par}. The other fraction $1 - f$, the serial part, is executed on the GPP processor. The execution time for a processor with coprocessor configuration is then estimated by:

$$t_{coproc} = (1 - f) \times t_{proc} + \frac{f \times t_{proc}}{S_{par}} + t_{comm} \tag{1}$$

where t_{proc} is the execution time of the program on the same processor without a coprocessor, and t_{comm} is the communication time required between processor and coprocessor. The latter is modelled to be proportional to execution of the number of instructions within the down-loaded code part on the processor. It is given by:

$$t_{comm} = rct \times f \times t_{proc} \tag{2}$$

where rct (the relative communication time) is a proportionality constant.

If a SFU is incorporated within the processor we assume again that the same speedup of the parallel fraction. In this case the serial part also shows a little speedup S_{serial}. There is no communication overhead. The execution time is then given by:

$$t_{SFU} = \frac{(1 - f) \times t_{proc}}{S_{serial}} + \frac{f \times t_{proc}}{S_{par}} \tag{3}$$

In figure 3 the performance improvement when using a SFU instead of a coprocessor is drawn as a function of S_{par} and f for two rct values: one with zero communication overhead, $rct = 0$, the other with $rct = 0.2$. The improvement of the serial code fraction, in case a SFU is used, is pessimistically taken to be 1/20 of the improvement of the parallel code fraction, i.e. $S_{serial} = 1 + (S_{par} - 1)/20$. It shows that, given the assumptions, a SFU easily outperforms a coprocessor, even when we neglect the coprocessor communication overhead.

From the above discussion it becomes clear that a new class of architectures is needed from which ASIPs can be automatically synthesized. This class should inherit the favorable characteristics of VLIWs without having their disadvantages. This paper proposes such a class, the class of TTAs. The next section describes these architectures.

4 Transport Triggered Architectures

TTAs are a lot like VLIW architectures in that they can perform multiple operations per cycle. The principle difference is the way in which operations are programmed and executed. Whereas in VLIWs instructions specify RISC type operations, in TTAs they specify data transports. Operations are *triggered* as a side effect of these data transports: the destination of a transport implicitly specifies the kind of operation that will be performed on the data.

TTAs are organized as a set of FUs and register files (RFs), which are connected by some kind of interconnection network. RFs contain general purpose registers. E.g., figure 4 shows a TTA, containing 7 FUs, 2 RFs and 6 transport buses. FUs and register files are connected to the interconnection network with one or more *input and output sockets*. Input sockets contain multiplexers, output sockets de-multiplexers. The *network controller* (NCTL) controls the pipelining of the network.

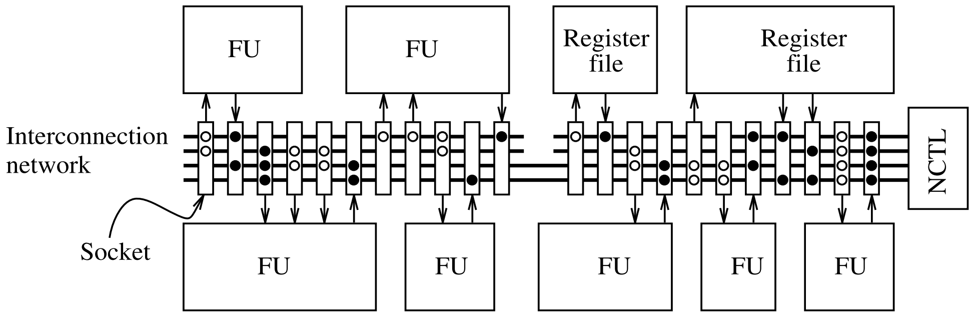

Figure 4: General structure of a TTA.

How is this architecture programmed? A TTA program only specifies transports; so it supports only one type of operation: the *move* operation from source to destination. Sources and destinations are any register, and any FU input and output. An instruction for a TTA with M buses may have the following format:

Bus−1 field			*Bus−2 field*				*Bus−M field*		
i	src	dst	i	src	dst		i	src	dst

Each *bus-field* specifies one *move* operation from source to destination. The i-bit indicates whether the source-id has to be interpreted as an immediate or as a register specification[3]. In general a three address operation translates into three *moves*, e.g.:

```
add r3,r2,r1    ⇒   r1 -> O1add; r2 -> O2add
                    Radd -> r3
```

[3]Note that the given instruction format is far from unique; there are other ways to specify transports and immediates [Cor95].

The notation used is: `O1add` and `O2add` are the operand registers of the adder, and `Radd` denotes the result (output) of this adder. TTAs can be easily made fully programmable. This requires support for control flow operations and conditional execution. Control flow operations can be implemented by making the program counter a visible source and destination location of the instruction fetch unit. Conditional execution can be supported by guarding move operations; all data transports then become conditional on a boolean expression.

The idea of programming only transports is not new; already one of the first commercial computers, the DEUCE, which became operational in 1955, operated that way [Hol59]. The DEUCE was a commercial version of the ACE, a prototype computer designed in the early fifties [Wil48, ACE51]; this prototype can still be viewed in the British National Science Museum. After that time it seems that this idea has been reinvented several times, see for example: [Lip76, Bla89, Jon88]. However, these approaches were restricted in several ways. They all fix the interconnection network to either one external memory bus (which is of course relatively slow[4]), or to two internal buses. In TTAs the transport capacity has been made an independent scalable parameter. Further, some of these approaches do not allow direct communication between FUs; they miss therefore one of the key properties of TTAs, namely the ability to *bypass* values directly from FU to FU without consulting the RFs.

The TTA programming method may seem clumsy when seen the first time: instead of one FU operation we now have to specify three move operations. However, as will be shown in section 4.2, this approach has many advantages. You may already guess that, because all transports can now be controlled by the programmer or compiler, unnecessary transports, which occur quite frequently in current architectures, can be avoided. This is the key to solving the scalability problem of VLIWs.

As shown, the operating principle of TTAs is very simple. In this respect the development of TTAs can be viewed as the next step in the evolution which went from CISCs (Complex instruction set computers) to RISCs; RISCs streamlined their instruction set such that it could be pipelined. As a result a compiler for RISCs needs to put more effort into producing efficient code. TTAs are even more simpler than RISCs, and their compilers correspondingly more complex. A whole set of TTA specific compiler optimizations emerges, and has to be exploited. For a thorough discussion of TTA compilation techniques the reader is referred to [Hoo95]. Following subsections elaborate on the main hardware and software aspects of TTAs.

4.1 Hardware aspects of TTAs

TTAs are constructed using a restricted number of building blocks [Cor94]. Figure 5 illustrates the hierarchical structure of the required building blocks. Essentially TTAs are built by a proper connection of FUs and bus-connections (input and output sockets); register files can be considered as special FUs (implementing the identity operation). FUs are completely independent of each other and of the interconnection network; they only have to apply to the socket interface specifications. FUs can therefore be designed separately, and pipelined independently. Each FU pipeline stage is controlled by a FU-stage controller; a chain of them implements the FU pipelining mechanism.

Sockets primarily consist of decoding logic (comparators), input multiplexers, and output demultiplexers (bus drivers). The interconnection network contains besides the wiring (for source and destination-ids, data and control lines) some global control logic, implemented within the network controller. This controller is responsible for generating locks (e.g., when a function can not yet deliver a result), squashes (e.g., in case of conditional transports) and exception control.

[4] It means that each transport requires a load of the value of a source location into an internal register, and a store of that register value into the desired destination location.

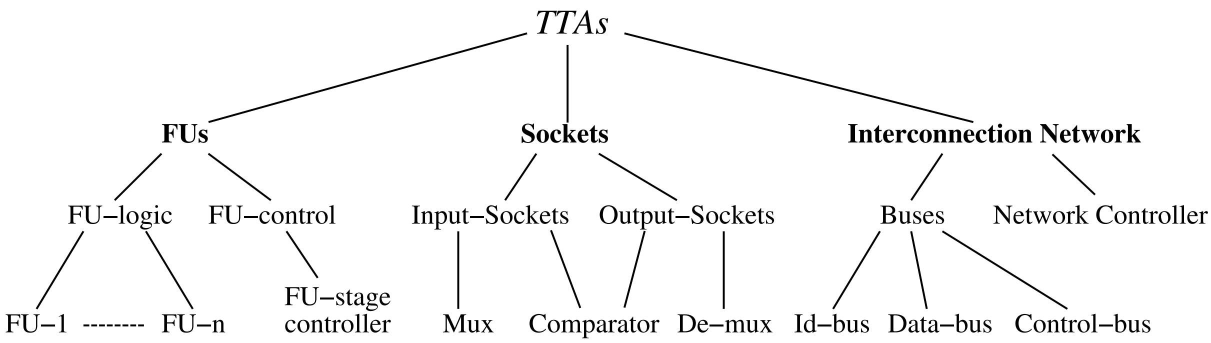

Figure 5: Building block hierarchy of TTAs.

In general, the interconnection network does not have to be fully connected, although full connectivity would ease the code generation process. It is the task of the compiler to optimize the required transports, given a certain connectivity, such that *cycle count* (the number of executed cycles) is minimized. Limiting the connectivity is an important tool in reducing bus loads, yielding faster transport times. Note, that there are no prior constraints on the number of inputs and outputs of FUs and the number of ports on the RFs, and on the capacity of the transport network; e.g., the latter may contain only one bus, or be a very complex network. Thus the modularity and flexibility requirements seem to be fulfilled.

Performance scales by adding FUs to an existing architecture, pipelining existing FUs, or by increasing the data transport and storage capacity. In contrast to traditional architectures, the number of FUs is not coupled to the data transport capacity (e.g. the number of register ports), e.g., FUs can be added without having to change the transport capacity, and therefore the instruction format[5]. Of course, the application should determine the right match between number of FUs and data transport capacity.

The modularity of TTAs allows the hardware design process to be automated. Different TTAs can easily be configured by assembling different combinations of these blocks.

If needed, the processor can be optimized for operation throughput, instead of latency. This requires: 1) result registers at each FU output, and 2) superpipelining those FUs which constrain the achievable cycle time. These measures result in an architecture where the cycle time is limited by the time needed for inter FU data transports only. The implementation can be tuned to optimize this time by using advanced bus implementation techniques.

Hardware efficiency

There are several aspects contributing to an economic usage of hardware resources:

1. *Trivial decoding*. One of the RISC design goals was to reduce decoding logic. In this respect, TTAs take the ultimate approach; they support only one type of operation.

2. *Transport efficiency*. As will be shown in the next subsection, many transports turn out to be superfluous and can be removed by applying TTA specific scheduling optimizations. As a result less connectivity and fewer ports on the register file(s) are required.

3. *Register efficiency*. Because many values produced during the course of a program do not need to be allocated in the RF, not only the RF traffic rate reduces, but also the number of registers required.

[5] As long as we do not run out of source- and/or destination-ids required for specifying the move operations.

4. *FU splitting*. FU logic can be split into independent units, used for different functionality, without a large impact on the interconnection network. E.g., an ALU may be split into an arithmetic and a logic unit. When split, these units may be used concurrently which results in a more efficient hardware usage.

The above economy of hardware usage results from putting complexity into software (the compiler) instead of hardware. Together, the former hardware characteristics make TTAs a challenging architecture for the design of ASIPs.

4.2 Software aspects of TTAs

Here we demonstrate some specific TTA code generation optimizations. Programming an operation on a TTA consists of moving operands to the input registers of a FU that is capable to perform the operation, and moving the result from the output of the FU to a RF or a FU after the operation has been performed. To illustrate TTA specific optimizations, 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,r2,r6       // r4 = r2 - r6
st   r4,r1          // store r4 at address r1
```

First we translate each n-operand m-result operation into $n + m$ moves (n so-called *operand moves* and m *result moves*):

```
r2->O1add; r3->O2add; Radd->r1
r2->O1sub; r6->O2sub; Rsub->r4
r1->O1st ; r4->O2st
```

Note that the store does not require a result move. So far, all moves are to be executed sequentially. Scheduling consists of assigning FUs to operations and assigning cycles, sockets, and move buses 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->O1add.alu1; r3->O2add.alu1; r2->O1sub.alu2; r6->O2sub.alu2
Radd.alu1->r1 ; Rsub.alu2->r4
r1->O1st.ls   ; r4->O2st.ls
```

The suffixes '`alu1`', '`alu2`', and '`ls`' indicate the FU on which the operation is executed. The code that we have produced so far is correct, but not optimal. The following optimizations can be applied: bypassing, dead result move elimination, operand sharing, socket sharing, and scheduling freedom. They are discussed next.

Bypassing
The outputs of the add and subtract operations can be directly moved to the load-store unit. This reduces the schedule with one cycle; the number of moves does not change.

```
r2->O1add.alu1; r3->O2add.alu1; r2->O1sub.alu2    ;r6->O2sub.alu2
Radd.alu1->r1 ; Rsub.alu2->r4 ; Radd.alu1->O1st.ls;Rsub.alu2->O2st.ls
```

Dead result move elimination

Next it may occur that the values in `r1` and `r2` are not life anymore because they are only used once. In that case corresponding moves can be eliminated. This gives the following schedule:

```
r2->O1add.alu1    ; r3->O2add.alu1; r2->O1sub.alu2; r6->O2sub.alu2
Radd.alu1->O1st.ls; Rsub.alu2->O2st
```

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

Operand sharing

This occurs when succeeding operations on the same FU use the same operand. For example, in the following code fragment the second operand move is redundant and can be eliminated since the value of `r1` is already in `O1add.alu`. We call this optimization *common operand elimination*.

```
r1->O1add.alu; r2->O2add.alu
r1->O1add.alu; r3->O2add.alu; Radd.alu -> r2
Radd.alu -> r3
```

It turns out that most of the shared operands are small immediates (like 1, 0, -1, 4, etc.) or the stack pointer. An interesting case of operand sharing may occur within loops, where the shared operand is the offset value used for the address or counter increment operation; the transport of this offset to the FU may turn out to be loop-invariant; in that case it can be put in the loop-prologue instead of the loop-body.

Socket sharing

The usage of an output socket may be shared by multiple reads. This socket sharing occurs in two situations:

1. Reading the same register on different buses.
2. Reading different registers which are connected to the same output socket, in case these reads are mutually exclusive (i.e., having opposite guard conditions).

In both cases the number of moves is not reduced, but a schedule may be generated which was otherwise inhibited by hardware resource constraints. Socket sharing has the most effect on the register file, where it results into sharing of register ports.

Scheduling freedom

A final optimization concerns the extra scheduling freedom when scheduling moves instead of operations. In TTAs the scheduling of the data transports are decoupled; as a result a compiler has much more freedom *when* to schedule these transports. E.g., the transport of source operands can be scheduled in different cycles. Also, depending on the implementation of the FUs, there can be more or less freedom in scheduling the transport of the FU result.

This scheduling freedom has two effects: 1) it enhances the application of described optimizations, and 2) values can stay longer in the FUs; this lowers the demands on the register file.

5 The MOVE framework

The most important advantages of TTAs (when compared to traditional architectures) are their inherent flexibility, scalability and simplicity (resulting in a short design cycle). To exploit these advantages, an automated design framework, called the *MOVE framework* (figure 6), has been developed. It consists of three components: a *hardware subsystem* and a *software subsystem*, which are used by an *explorer* to explore the architecture design space as defined by the used TTA template. The main parameters of this template and their supported range of values are described in table 3.

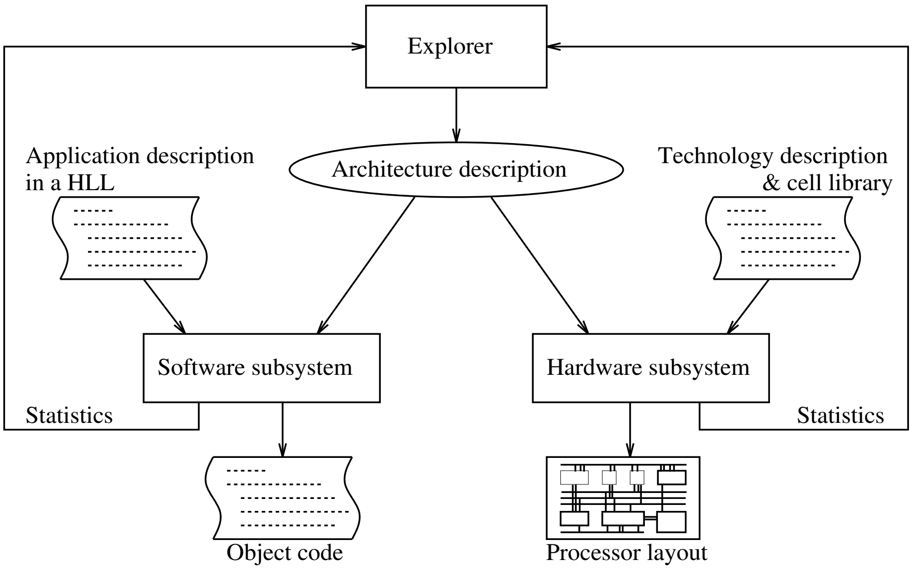

Figure 6: The MOVE framework.

TTA parameter	Supported range or types
#Buses, M	≥ 1
#FUs, K	≥ 1
FU type	any combination of basic and special operations
FU latencies	≥ 1
FU pipelining	non-pipelined, fully pipelined
FU pipelining method	one of two different pipelining methods
Branch latency	≥ 1
Short immediate size	≥ 0 bit
Sign extension of short immediates	no, yes
#IRF (#Integer reg. files)	≥ 1
• N_{IRF}: #Registers in int. reg. file	≥ 1
• RP_{IRF}: #Read ports on int. reg. file	≥ 1
• WP_{IRF}: #Write ports on int. reg. file	≥ 1
#FRF (#Floating point reg. files)	≥ 1
• N_{FRF}: #Registers in fp. reg. file	≥ 1
• RP_{FRF}: #Read ports on fp. reg. file	≥ 1
• WP_{FRF}: #Write ports on fp. reg. file	≥ 1
#BRF (#Boolean reg. files)	≥ 1
• N_{BRF}: #Registers in boolean reg. file	≥ 1
• WP_{BRF}:#Write ports on boolean reg. file	≥ 1
#additional immediates per instruction	≥ 1
Bus socket connection	all connections are individually specified

Table 3: The main parameters of the template TTA used within the MOVE framework.

By varying several architecture parameters from this template, such as number of transport buses, number and type of function units, etc, the explorer tries to find the processor configuration that yields a good cost/performance ratio for the given application. The MOVE framework components and its user interface are detailed in following subsections.

5.1 Explorer

Two main design evaluation criteria are cost and performance, where performance is defined as the inverse of execution time. Cost may include the amount of chip area, the number of pins, the power dissipation, and the code size[6]. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time.

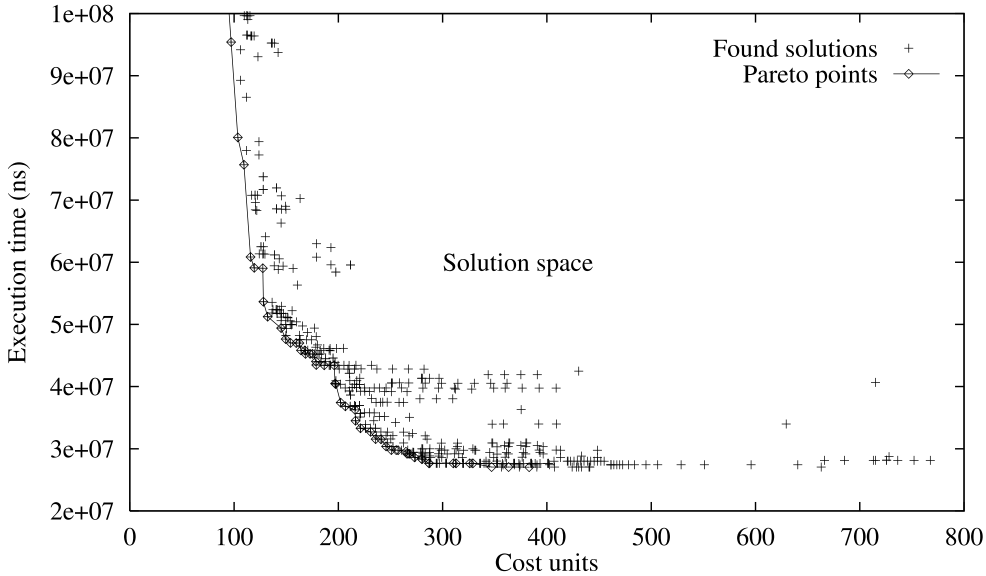

Figure 7: Possible solutions and Pareto points.

The **solution space**, $S_{solutions}$, is given by all possible, in the sense that they can be generated by the MOVE framework, design points in the 2-dimensional cost-performance space. Figure 7 shows many generated solutions for a particular application. Each marked solution corresponds to a particular TTA; it is specified by a 2-tuple (t_{exec}, c_{tta}) where t_{exec} is the execution time (in ns) required to run the application on this TTA, while c_{tta} represents the realization costs for this TTA (in units to be specified later on). 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 [Mic94]:

A point $(t_{exec}, c_{tta}) \in S_{solutions}$ *is a Pareto point* $\Longleftrightarrow \forall (t,c) \in S_{solutions}, (t \geq t_{exec} \vee c \geq c_{tta})$.

The explorer finds its way through this solution space by iteratively trying different architecture solutions, and letting the software and the hardware subsystems produce relevant information about these solutions, such as cycle time, cost and the number of cycles needed to run the application. Based on this information a next design point is chosen by modifying the parameters (table 3). The user has to specify an initial (large) TTA configuration from which the explorer starts the exploration process by removing resources. Different local search options exists: *first fit*, *best fit*, and *backtracking*.

With first fit a neighbor configuration (which differs in only one resource with the current configuration) is immediately selected if it has a better quality; with best fit all neighbor configurations are

[6]Currently we included area and pins only.

evaluated and the best one taken. The quality of a solution is defined by:

$$Quality = \frac{1}{t_{exec}^{a} . c_{ttp}^{b}} \tag{4}$$

where and a and b are constants which express the relative importance of performance and cost. If required, this function can easily been changed.

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.

The backtracking option allows for searching for better Pareto points by searching the local neighborhood of points found by first or best fit, up to a certain backtrack limit.

The explorer does not add resources which were not available in this initial configuration. E.g., if one wants to exploit SFUs, one or more of them should be part of this configuration. In section 6 the explore process is illustrated by a working example. For further details we refer to [CH95, Hoo96, CH96].

5.2 Hardware subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of an application specific TTA in silicon. Figure 8 shows its organization. It is based on three components: a processor generator, a silicon compiler, and a hardware modeller. The processor generator accepts an architecture description and generates a synthesizable description in a HDL (hardware description language). Two HDLs are supported: the SID language used by the ASA silicon compiler[7], and VHDL; a VHDL description can be used as input for many available silicon compilers and other synthesis tools. The silicon compiler takes the HDL description, the technology information and a cell library as input, and produces a VLSI layout (e.g. in CIF format) of the generated processor. The shown layout in figure 8 is taken from the 16-bit Phoenix TTA processor, which has been designed using the MOVE framework, and is realized in a 1.0 μm CMOS technology [CL95]; first silicon has been demonstrated to operate correctly at 45 *MHz*.

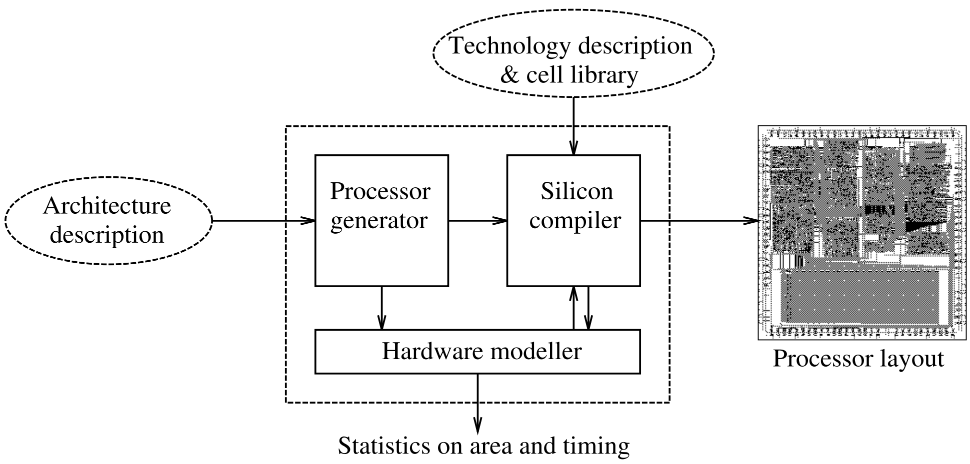

Figure 8: The hardware subsystem.

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

The hardware modeller is a tool which can quickly make area and timing estimations. This is required in order to be able to research many different architecture descriptions during the synthesis process. The hardware modeller exploits the fact that during the synthesis process many TTAs are researched which only slightly differ from each other; timing and area information of unchanged parts is kept in a database for quick access. Further details of the hardware subsystem can be found in [CL95, Lam93].

Since the hardware modeller is not yet fully operational and too slow for a exploring many potential solutions, the designer has the option to specify a *hardware model* himself; this model consists of a set of algebraic equations *predicting* area and timing of the generated processors. For the experiments presented in this paper we will use a simple hardware model, based on realizations within a CMOS 1.0 μm technology. The main values and expressions used for cost (area, including number of pins), within this model, are shown in table 4. The cost of FUs are based on a 16-bit data path width, and relative to a 16-bit integer FU.

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
Buses[b]	$3.5 + N_{bits}/16$
Registers[c]	$0.0125 \times N_{bits} + 0.0031 \times (\#RP + \#WP)$
Sockets[d]	$0.02 + 0.0125 \times N_{bits} \times N_{conn}$

[a] Including load-store data pads.

[b] Including pads for instruction fetch and on-chip wiring; each data bus corresponds to an instruction field. N_{bits} has been included mainly to distinguish 16-bit buses from single bit boolean buses.

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

[d] N_{conn} is the number of bus connections of a particular socket

Table 4: Main values and expressions of the hardware cost model as used in this paper; the user can freely change this model.

The minimum clock cycle time for TTAs is largely determined by the time needed to perform (and control) the data transport. This depends on the number of connections to a bus. For the experiments reported in this paper we used the following cycle time (in nanoseconds):

$$t_{cycle} = \max\{14.0, \max_{\forall \text{ buses } b}\{10.0 + 0.4 \times \#\text{connections at bus } b\}\} \tag{5}$$

Its minimum value is taken 14.0 ns, the time to do a 16-bit integer operation.

5.3 Software subsystem

The software subsystem contains a compiler, simulators for sequential and parallel code, profiling and trace analysis tools, and code viewers. The main component is the compiler; it is responsible for generating instruction level parallel code for any given TTA which fits in the template, and produce statistics about cycle counts and usage of hardware resources. The compilation trajectory is shown in figure 9.

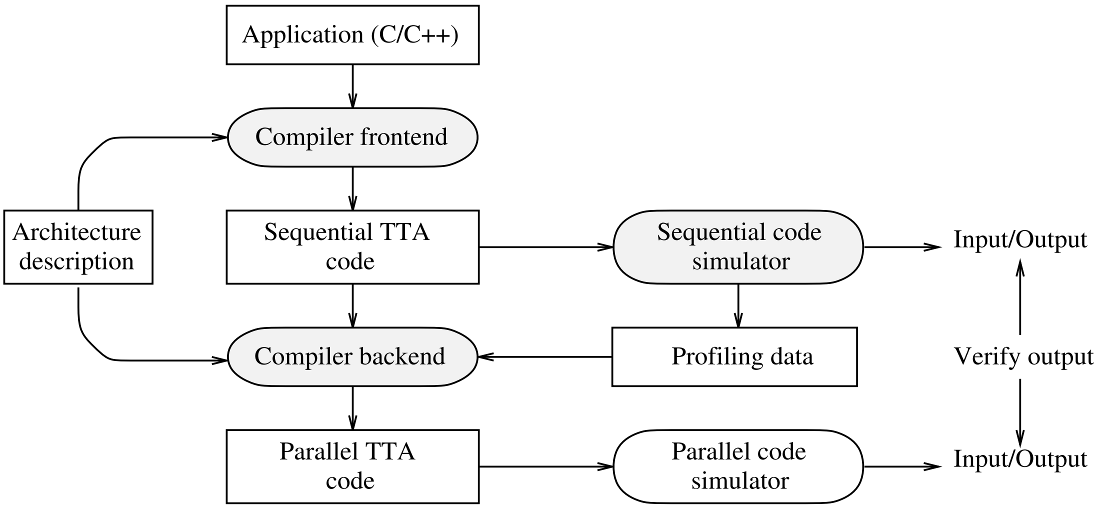

Figure 9: The TTA compilation trajectory of the MOVE framework.

The compiler frontend is based on GNU gcc[8]. The output of the frontend compiler is sequential TTA code. This code can be simulated by the *sequential code simulator*. This simulator provides profiling data which is used by the backend. The backend reads the sequential code and (if available) the profiling data and produces parallel (scheduled) TTA code. This code can be verified by the *parallel code simulator*. The simulators also serve to verify the correct operation of frontend and backend.

The backend is completely parameterized; it can generate code for any TTA which fits into the described template, under the assumption that the TTA provides sufficient functionality such that the application can be mapped onto it; e.g., the support for a `shift` operation can only be left out if the frontend does not generate `shifts`. The backend exploits advanced inter basic block scheduling and software pipelining techniques in order to compact the code as much as possible and to efficiently exploit all the available hardware [Hoo96].

5.4 User interface

The explorer, the software subsystem and the hardware subsystem are integrated via a user interface tool. Using this tool the designer can control all options during the search process and inspect the results. To give an impression of this tool, figure 10 displays a screen dump containing a few of the windows through which the designer communicates with the synthesis tools. It shows the main synthesis window (top left) from which all individual tools can be started, part of the 'Explorer' window (top right), which is used to guide the search process, the 'Mach Viewer' (bottom right) for viewing and editing architecture descriptions, and a 'Time versus Cost' graph (bottom left) showing the Pareto points for a specific application (the screen dump shows them for the Crypt application). When clicking on one of the Pareto points the corresponding architecture description will be shown and can be displayed by the Mach Viewer (in this case Pareto point 'mach-0116' is displayed; this is a fully connected TTA with four buses). The viewer can also be used to edit a new or existing architecture description.

[8]A different frontend for functional languages producing generic sequential TTA code is also available [MB94]. Further an experimental GNU Fortran frontend can be used.

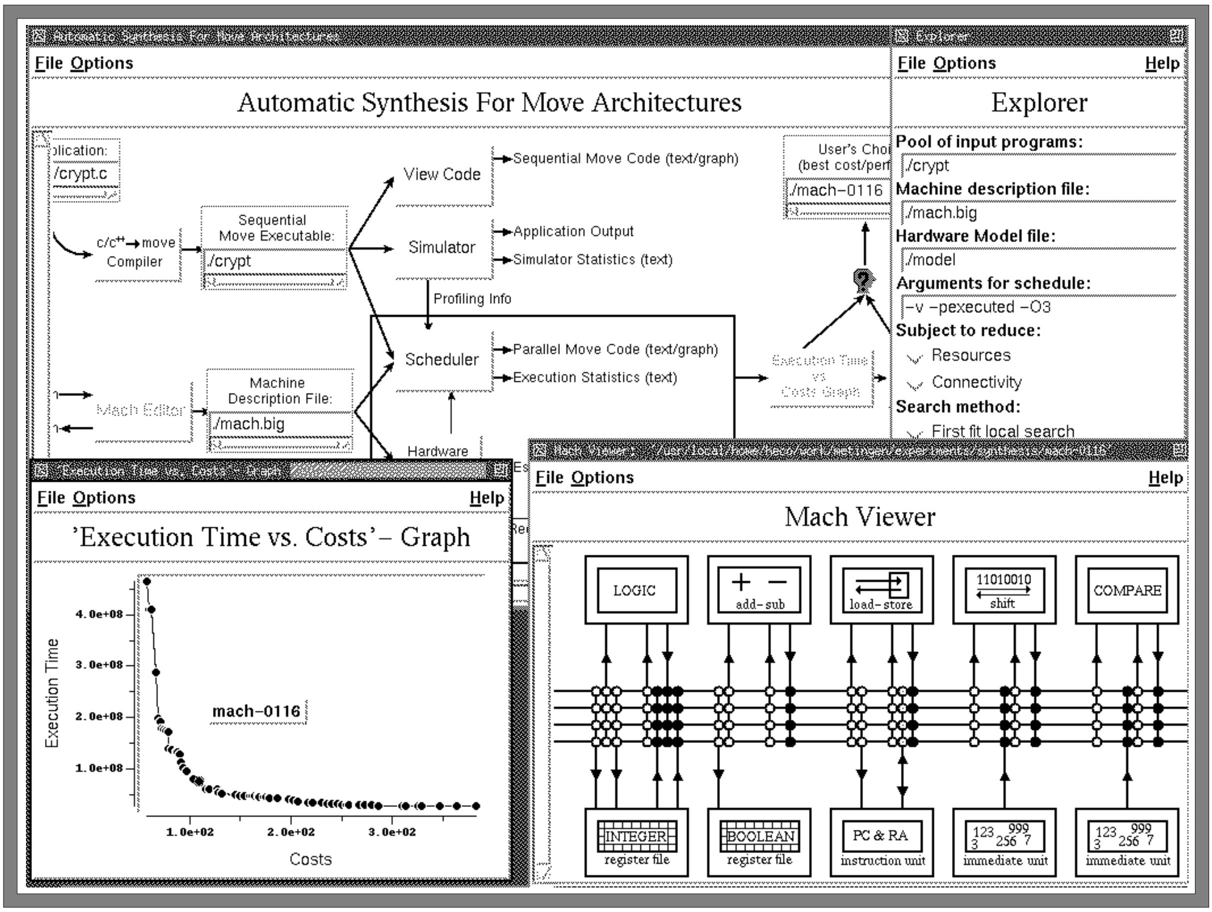

Figure 10: Screen dump of the MOVE framework user interface as used during the synthesis process.

6 Image processing case study

In order to demonstrate the usage of the MOVE framework, we concentrate on implementing two examples of gray-scale neighborhood operations [Jai89]: convolution and edge detection on a 3x3 area (figure 11). These operations are part of a larger image processing application [ALC96].

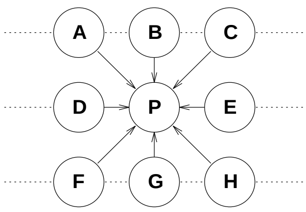

Figure 11: The 3x3 pixel neighborhood.

The *convolution* operation is a *linear* gray-scale operation. For each pixel *P*, a new value P_{out} is calculated from its old value and the values of its neighbors. For the neighborhood shown in figure 11, the operation can be written as:

$$P_{out} = c_0 \cdot P + c_1 \cdot (A + C + F + H) + c_2 \cdot (B + G) + c_3 \cdot (D + E) \tag{6}$$

The values of the coefficients $c_{1..3}$ determine the kind of transformation that is performed (e.g. positive values smoothen the image, negative values sharpen it). In principle, all pixels can be processed in parallel since their calculations are independent of each other's new values. Control flow is simple, no branches need to be evaluated during the processing of a pixel. The actual level of parallelism that can be attained in a TTA processor implementation is determined by the maximum number of concurrent operation slots in the processor (upper bound on parallelism), as well as the ability of the operation scheduler (part of the compiler) to fill these slots with actual operations (or moves).

The *edge detection* algorithm based on the min/max operation is *non-linear*. Each output pixel P_{out} is assigned the difference between the maximum and the minimum value in a neighborhood around input pixel *P*, including *P* itself. For the neighborhood shown in figure 11, the operation can be written as:

$$P_{out} = \max(A \ldots H, P) - \min(A \ldots H, P) \tag{7}$$

While the potential for parallelism is the same as for the convolution operation, the minimum/maximum calculations (requiring lots of branches) make it more difficult to parallelize by a compiler. It will be shown in subsection 6.2 that adding special functionality to the TTA processor template increases the compiler-detected parallelism significantly. Before that, subsection 6.1 describes the design process and results for the two categories of image processing algorithms using only the default, RISC-like functionality. The architecture configuration that resulted from the automatic design process is presented in subsection 6.3.

6.1 Implementation with default operations

The first step in mapping an application onto a TTA processor is to write a C or C++ version of the algorithm. This algorithm is compiled to sequential TTA code, which includes only RISC-like operations, comparable to those found in most general purpose processors[9]. Critical procedures are identified using profiling tools. The explorer will concentrate on these procedures while searching the design space.

In our case, the critical procedure is the part that calculates the output value for each pixel from its own input value and those of its neighbors. The operation count of the critical procedures of the convolution and edge detections algorithms is given in table 5, columns two and three.

Using the RISC-like, default set of operations, the MOVE framework is used to find the optimal TTA configuration for both types of gray-scale neighborhood operations (convolution and min/max edge detection). The exploration is performed in two steps: in the first step, called *resource optimization*, large resources, like FUs and buses, are considered; the second step, called *connectivity reduction*, reduces the number of bus-socket connections. The results of the first step are shown in figure 12; the second step is considered in section 6.3.

It turns out that the framework is able to find a much more efficient implementation for the convolution operation than for the edge detection algorithm[10]. This is mainly caused by the large number of branches needed when calculating the greatest or smallest of two numbers. In VLIW architectures and TTAs, such branches can usually be eliminated by means of a technique called *if-conversion* [R$^+$92]. This is also the case for this application. However, our current compiler is unable to detect in advance register delay-line problems that occur when an attempt is made to software-pipeline the if-converted code. The exact nature of these problems falls outside the scope of this report but is discussed in some depth in [Hoo96].

[9]Of course, the TTA code contains only moves which trigger these RISC-like operations.

[10]When scheduling on an 'ideal' (i.e. very large) processor configuration. This is done to find an upper bound of the compiler-detected parallelism, without being constrained by hardware resources.

Operation	#ops/pixel convolution (no SFU)	#ops/pixel edge-detect (no SFU)	#ops/pixel convolution (Addercmp)	#ops/pixel edge-detect (Addercmp)
add/sub	12	6	12	7
greatest	n/a	n/a	1	4
smallest	n/a	n/a	1	4
mul	4	-	4	-
gt	3	8	1	1
ld	3	3	3	3
st	1	1	1	1
shr	1	-	1	-
Total	24	18	24	20

Table 5: Operation counts without and with the Addercmp SFU.

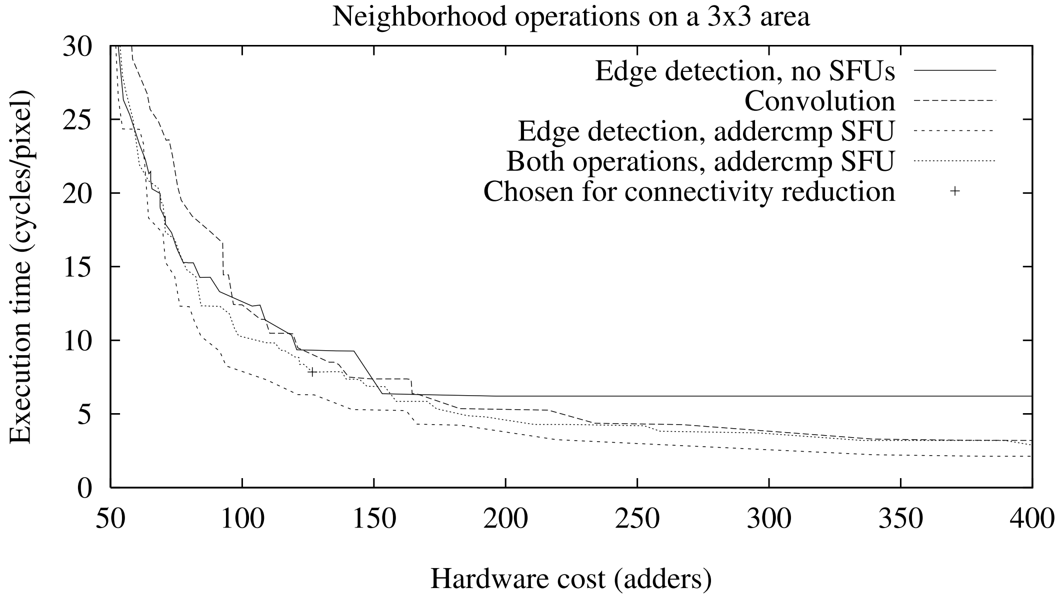

Figure 12: The TTA design space for the 3x3 neighborhood operations, with and without SFU.

The problems' effect is a rather large steady state of the software pipeline: 8 cycles for edge detection[11], as opposed to 3 for the convolution algorithm, given a very large hardware configuration (e.g. one with cost 400, in figure 12). It can be seen from the graph, however, that the cost/performance curve "Edge detection, no SFUs" already flattens out at 8 cycles per pixel at a cost of around 200. Any hardware resources that are added beyond this point can not be used to increase performance.

6.2 Implementation with Special Function Units

An important part of the processor design process is to see if and how the use of *special function units* (SFUs) can improve the quality of solutions produced by the MOVE framework. In this subsection, we describe an SFU that was designed specifically to solve the aforementioned problem with the edge-

[11]Figures are obtained using software-pipelining combined with if-conversion, but without loop unrolling.

detection algorithm.

The performance of the edge detect implementation can be increased dramatically by adding a special function unit, the *addercmp* (adder-comparator) FU. It is a simple extension of an adder which can do conditional assignments, i.e. return the greatest or smallest of two operands as its result. Since this eliminates the branches, it is possible to efficiently schedule (software-pipeline) the critical loop.

This is reflected in figure 12, which shows a significant improvement of the cost-performance ratio. Table 5 shows the operation count of the critical loop when the addercmp FU is used. It turns out that while, for edge detection, the operation count is actually higher than in the initial implementation (20 vs. 18 operations, see table 5, columns four and five), the TTA compiler schedules the new code much more efficiently, i.e. it exploits the parallelism better.

In the convolution algorithm, the addercmp FU is applicable only twice (for clipping). This does not yield any scheduling gains because these branches could easily be eliminated with if-conversion.

The special functions "greatest" and "smallest" are a cheap extension of functionality, since they are implemented using mostly existing hardware (the adder). The unit's latency increases with the delay of one selector, but this is outweighed by the scheduling advantage that the added functionality affords. The addercmp unit's usefulness is actually higher than that of a normal adder, since it can still perform normal additions and subtractions in addition to the "greatest" and "smallest" operations. This is especially noticeable when we combine the convolution and edge detection operations on the same TTA configuration. The convolution operation needs many additions (adder units), whereas the edge-detection operation needs mostly compares (comparator units). When we replace the adders needed by the convolution operation with addercmp units, the comparators are no longer needed.

The design space exploration does not stop here. Other SFUs may be considered as well. Especially for these applications it becomes worthwhile to buffer one or more image lines in local on-chip line buffers and include SFUs which perform load and store operations on these buffers. Furthermore, it pays off to add an auto-increment addressing mode to these SFUs. Details of these SFUs are beyond the scope of this paper, however (see [ALC96]).

6.3 The resulting TTA processor configuration

If we want to develop a processor that is equally suited to both convolution and edge-detection, we let the explorer search the design space for both applications simultaneously. After resource optimization, a hardware configuration is chosen from the graph in figure 12. Based on the cost-performance ratios and what we deemed hardware-feasible, a reasonable configuration might be the one indicated with a '+'. It contains 9 buses and 8 FUs. This configuration is used as the starting point for connectivity reduction, i.e. the explorer attempts to remove unnecessary connections between the FUs and the buses. The resulting configuration is shown in figure 13.

Final performance figures are then obtained by scheduling the applications for the final processor configuration. The convolution operation is executed in 8 cycles per pixel, the edge-detection operation in 7 cycles per pixel (using addercmp FUs).

It is also interesting to see how the chosen configuration performs on the edge detection algorithm for a larger 5x5 pixel area. While essentially the same as the 3x3 version, the workload increases significantly, since now 25 pixels have to be considered each time, instead of 9. Scheduling the application code for the processor configuration of figure 13 yields a performance of 13 cycles per pixel; scheduling this application on a very large processor yields 4 cycles per pixel. The performance loss due to hardware constraints is comparable to that of the 3x3 edge detection operation: about three times as many cycles are needed (13 vs. 4 and 7 vs. 2, respectively).

Being able to use the same TTA configuration for multiple applications and different neighborhoods demonstrates the advantage of having a fully programmable solution.

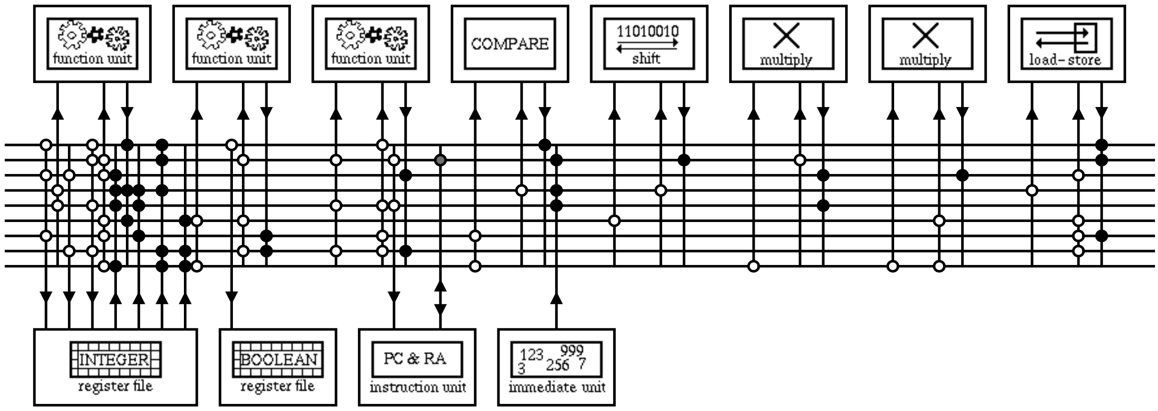

Figure 13: The resulting TTA processor configuration.

7 Evaluation and conclusions

In this paper we discussed the problems of designing application specific processors (ASPs) for embedded systems. It has been shown that these problems can largely be solved by having an automated processor design trajectory based on a templated architecture. The template should be scalable, flexible, modular, and fully programmable. Existing architectures can not fully satisfy these requirements. Therefore a different *Transport Triggered Architecture* (TTA) has been proposed.

The principles of operation and characteristics of TTAs are described. Based on TTAs a set of tools, collectively called the *MOVE framework*, for the automatic design of ASPs has been implemented. Using this framework the designer can explore a large space of possible solutions for a given (set of) application(s). Each solution corresponds to a particular processor instantiation of the TTA template, together with the generated object code for this processor. The code generator exploits the instruction level parallelism available in the application and supported by the hardware.

The MOVE framework has been applied to finding solutions for digital image processing applications. Many alternative solutions can be compared within reasonable time without having to invest a lot of manual design effort.

The framework can also be used to exploit the flexibility and reusability of the architecture. It is possible to find a processor that is optimized for one application, but it is equally possible to find one dedicated to a whole class of applications (in our case, different neighborhood operations with different neighborhood sizes).

A large part of designing the TTA processor is done automatically. However, manual interaction is still needed when Special Function Units (SFUs) are considered. Currently, these need to be called explicitly from the application code if they are to be used. Thus the decision whether to use an SFU has to be made beforehand, by the designer; it is not included in the automatic design space exploration phase. Future research will concentrate on automating this design decision.

Comparison with other approaches The synthesis process used within the MOVE framework has much in common with high level synthesis (HLS) systems which have been developed [Bre91, Cam90, CW91, GDWL92, KGD90, MLD92, WC91]; 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 [V^{+}93], 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 **??** 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 TTAs 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 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.

References

[ACE51] Automatic Computing Equipment at the Nat. Phys. Laboratory. *Engineering,* 171, page 6, 1951.

[ALC96] Marnix Arnold, Reinoud Lamberts, and Henk Corporaal. High Performance Image Processing using TTAs. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[Bla89] Zdenek Blazek. Preliminary design of a parallel SIC architecture. *Microprocessing and Microprogramming*, 28:103–108, 1989.

[Bre91] M. Breternitz Jr. *Architecture Synthesis of High-Performance Application-Specific Processors.* PhD thesis, Carnegie Mellon Univ., April 1991.

[Cam90] R. Camposano. From behavior to structure: High-level synthesis. *IEEE design & test of computers*, pages 8–19, October 1990.

[Cas94] Brian Case. Philips hopes to displace DSPs with VLIWs. *Microprocessor Report*, December 1994.

[CH95] 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.

[CH96] Henk Corporaal and Jan Hoogerbrugge. Cosynthesis with the MOVE framework. In *CESA'96 IMACS Multiconference*, pages 184–189, Lille, France, 1996.

[CL95] Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, pages 18–27, Heijen, The Netherlands, May 1995.

[Cor94] Henk Corporaal. Design of Transport Triggered Architectures. In *Fourth Great Lakes Symposium on VLSI*, pages 130–135, Univ. of Notre Dame, Indiana, March 1994.

[Cor95] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[CW91] Raul Camposano and Wayne Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[GDWL92] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-level Synthesis; introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[Hol59] S. H. Hollingdale. *High speed computing; methods and applications*. The english univ. press, 1959.

[Hoo95] Jan Hoogerbrugge. *Compiler design for transport triggered architectures*. PhD thesis, Delft University of Technology, 1995. in preparation.

[Hoo96] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996. ISBN 90-9009002-9.

[HP95] John L. Hennessy and David A. Patterson. *COMPUTER ARCHITECTURE, a Quantitative Approach, Second Edition*. Morgan Kaufmann publishers, 1995.

[Jai89] Anil K. Jain. *Fundamentals of Image Processing*. Prentice Hall, 1989.

[Jon88] Douglas W. Jones. The Ultimate RISC. *Computer Architecture News*, 16(3):48–55, June 1988.

[KGD90] M. Koster, M. Geiger, and P. Duzy. ASIC design using the high-level synthesis system CALLAS: a case study. In *ICCD 90*, pages 141–146, 1990.

[Koc94] Andreas Koch. SPARXIL: Ein konfigurierbarer FPGA-Coprocessor. In *GI/ITG-workshops on Architecturen für hochintegrierte Schaltungen*, pages 65–66, Schloss Dagstuhl, July 1994.

[Lam93] Reinoud Lamberts. A MOVE Processor Generator. Technical report, Delft Univ. of Technology, November 1993.

[Lip76] G. J. Lipovski. Architecture of a simple, effective control processor. In *Second Symposium on Micro Architecture*, pages 187–194, 1976.

[MB94] J. Mountjoy and M. Beemster. Functional languages and very fine grained parallelism: Initial results. Technical report CS-94-24, Department of Computer Systems, University of Amsterdam, December 1994.

[Mic94] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGRAW-HILL, 1994.

[MLD92] Petra Michel, Ulrich Lauther, and Peter Duzy, editors. *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992.

[R$^+$92] B. R. Rau et al. Register allocation for software pipelined loops. In *Proceedings of conf. on programming language design and implementation*, pages 283–299, San Francisco, June 1992.

[V$^+$93] Jan Vanhoof et al. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer academic publishers, 1993.

[WC91] Robert A. Walker and Raul Camposano, editors. *A survey of high-level synthesis systems*. Kluwer academic publishers, 1991.

[Wil48] J. H. Wilkinson. The Automatic Computing Engine at the Nat. Phys. Laboratory. In *Proceedings of the Royal Society*, volume 195(A), page 265, December 1948.