A different approach to high performance computing

Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Mekelweg 4, 2628 CD Delft, The Netherlands
e-mail: heco@cs.et.tudelft.nl

Abstract

A common approach to enhance the performance of processors is to increase the number of function units which operate concurrently. We observe this development in all recent superscalar and VLIW (very long instruction word) processors. VLIWs are easier extensible to high performance ranges because they lack much of the superscalar hardware required for dependence checking and hardware resource allocation; instead they rely on a compiler to perform these tasks.

In this paper we propose to proceed along this line and go one step further in replacing hardware by software complexity: a new architecture is proposed which requires the scheduling and allocation of transports at compile-time, instead of performing this at run-time. This reduces hardware complexity and creates several new compile-time optimizations. The paper illustrates the compilation steps required, explains the concept and characteristics of the proposed architecture, and shows several measurements which confirm our belief that, especially for high performance embedded applications, this architecture is very attractive.

Keywords: *ILP, compilation, transport triggering, TTA, VLIW, superscalar, architecture complexity*

1 Introduction

Looking into history we see an ever increasing demand for more computing power, both for general and for special purpose applications. Having experienced the radical change from CISC to RISC in the eighties, most processor manufacturers upgraded their processor architectures with superscalar capabilities in the nineties. This allows them to stay binary compatible with what is already on the market; manufacturers are very anxious to change again to a binary incompatible architecture. However, as is well known, the exploitation of instruction-level parallelism using superscalar techniques is rather limited. Primarily this is caused by the limited instruction window of the hardware instruction dispatch mechanism. Enlarging this window results in high hardware costs for dependence checking and resource allocation, extra pipeline stages (which increase branch penalties), and a possible increase of the achievable cycle time.

A different, but binary incompatible, approach to exploiting instruction level parallelism (ILP) is taken by VLIW architectures. Currently, several interesting VLIWs hit the market, like the Trimedia of Philips, the Mpact of Chromatic and the TMS320C6x of Texas Instruments. They deliver high performance at reduced cost. The superscalar hardware has been replaced by compile-time dependence checking and resource allocation.

However, the data path of VLIWs is still too complex when scaled to very high performance levels as required by many future image processing and multi-media type of applications. In particular the register file and the hardware required for bypassing values becomes too complex when adding many function units (FUs) [1]. Therefore, we propose to go one step further in replacing hardware (run-time) complexity by software (compile-time) complexity. The result is a so-called *transport triggered architecture*. In this paper we describe this architecture and illustrate its hardware and software characteristics.

The remainder of this paper is structured as follows. Section 2 discusses the steps required in order to get a program written in a high level language running on an ILP processor; its purpose is to clarify the hardware software division. Section 3 outlines the transport triggering concept, while section 4 describes the hardware and software characteristics of transport triggered architectures (TTAs). A quantitative evaluation of two main TTA advantages is presented in section 5. Finally, section 6 gives a summary and draws major conclusions.

2 ILP architectures and the compiler's job

Exploiting parallelism has its price: the parallelism has to be detected and exploited efficiently by the architecture under consideration. Let us look at which translation and interpretation steps have to be taken in order to execute a sequential program written in a HLL on a single ILP processor:

1. Frontend compilation. Lexical analysis and parsing of the program, performing optimizations, and compila-

tion to basic operations. During this step alias analysis is also performed.
2. Determine dependences and produce data and control dependence graphs.
3. Bind operands to locations. E.g., non aliased variables and temporaries are bound to local registers for a certain life-time in order to reduce off-chip data traffic; this is called *register allocation*.
4. Bind operations to time slots, that is, decide when to execute operations and data transports. This is usually called *scheduling*.
5. Bind operations to FUs. When there are multiple function units of the same type, a binding choice has to be made.
6. Bind transports to buses. Performing operations requires the transport of many data values; buses must be allocated to these transports.
7. Execute the operations and perform the transports.

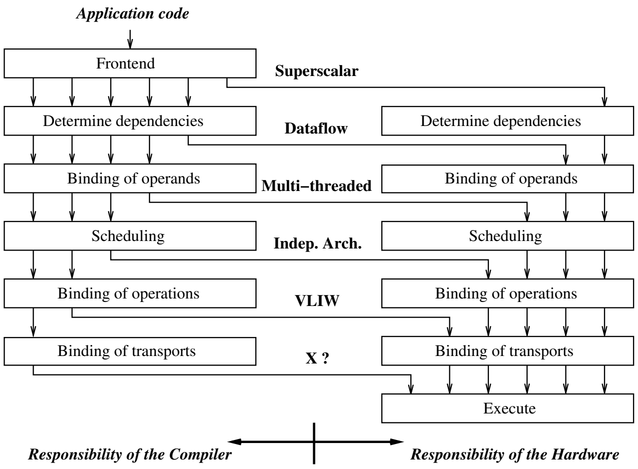

Figure 1: The division of responsibilities between hardware and compiler.

The order of these steps is not necessarily the same as presented above. It is clear that the first step is performed by the compiler, and the final step by the hardware. Intermediate steps may either be performed by hardware or the compiler. This is shown in figure 1. It illustrates how this division is made for different architectures.

Superscalar architectures shift most responsibilities to the hardware. Although they are able to run unmodified object code of former generation sequential architectures, they certainly need compiler assistance to execute programs efficiently. Hardware complexity limits superscalars to a very small run-time scheduling window (see e.g. [2]); therefore the compiler has to schedule code in such a way that operations within the scheduling window are independent from each other as much as possible.

Dataflow architectures let the compiler perform the frontend compilation and the data dependence analysis; other steps are performed by hardware. Dependences are made explicit in the object code, that is, each operation specifies its successor operations to which the result value(s) of the operation must be send. These values are sent by so-called *tokens*; they contain both the value and the successor specification. When tokens are received they have to be matched at run-time with instructions and other tokens (most instructions require multiple tokens). This matching turns out to be very expensive and time consuming [3].

Multi-threaded architectures can be considered as architectures which do the scheduling at run-time, but, in contrast to dataflow architectures, rely on the compiler for binding of operands to locations (this avoids token matching). A further difference concerns the scheduling granularity; it is not one instruction but a thread, containing multiple instructions. Although multi-threaded architectures support *fine grain* execution in the form of small threads, the compiler should try to enlarge threads (and therefore the scheduling granularity) in order to avoid the still existing thread switching overhead. A number of multi-threaded architectures have been proposed [4, 5, 6, 7]. These architectures support multiple threads by offering hardware support for efficient thread synchronization and thread switching. Just like superscalar and dataflow architectures, they can tolerate long, possibly non-deterministic, latency operations, like data loads, by switching quickly to another runnable thread of execution.

More recently several research groups have proposed usage of multi-processor architectures running multiple threads extracted from a single (sequential) program, with the aim to exceed the amount of instruction level parallelism exploitable from a single thread [8, 9, 10, 11, 12]. Each thread is run on a separate processor, presumably a superscalar processor; this way they combine both the exploitation of fine grain (at the instruction level) and more coarse grain (thread level) parallelism.

Independence architectures require that the compiler has to specify independent operations whose operands are available in a given cycle; in other words, the compiler performs the scheduling. The hardware performs the binding of operations and transports to hardware resources. An example is the Horizon architecture as presented in [13]; each operation encodes a number H which specifies the next H concurrent operations. The hardware can execute these H (or less) operations concurrently, without having to test if they are dependent or whether their operands are ready. The hardware still has to allocate (bind) these operations to FUs.

VLIW architectures take the most radical approach; apart from the binding of transports to buses they perform all steps in software. This results in a simple and cheap architecture which can be scaled to high performance levels. One may question if it makes sense to perform the final step also in software as well: what kind of architecture results and what are its characteristics? The next sections will answer these questions and demonstrate that, especially for high performance embedded systems, attractive solutions result.

3 Transport triggering concept

All architectures discussed in the former section are programmed by specifying the required operations. During execution these operations trigger several data transports (for source and destination operands) as side effect. In this section we propose an architecture which is programmed the opposite way: data transports are specified which, during execution, trigger operations as side effect. We call this: *mirroring the programming paradigm*. The result is a *transport triggered architecture* or TTA.

TTAs are organized as a set of FUs and register units (RUs), which are connected by some kind of interconnection network. RUs contain general purpose registers. Figure 2 displays a picture of this organization. It shows a TTA containing three FUs, of which one communicates with the data memory, and two RUs. Note, as becomes clear from the figure, that there are no prior constraints on the number of inputs and outputs of FUs and the number of ports on the RUs, and on the capacity of the transport network; e.g., the latter may contain only one bus, or be a very complex network.

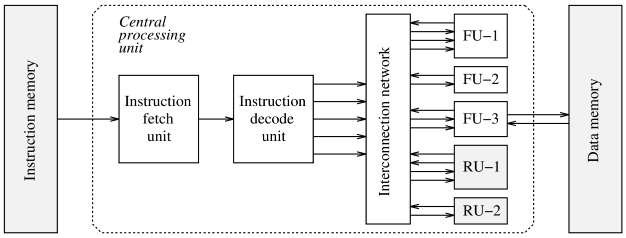

Figure 2: Organization of a TTA.

How is this architecture programmed? A VLIW may specify multiple FU operations, like `add`, `sub` and `mul`, within each instruction. In TTAs these FU operations occur as a side effect of data transports. A TTA program only specifies the data transports to be performed by the interconnection network; so it supports only one type of operation: the *move* operation from source to destination. A TTA instruction consists of one or more move operations; the number of move operations per instruction is equal to the number of transports which can be handled in parallel by the interconnection network. Sources and destinations of move operations are any register contained in the RUs, and any FU input and output. E.g., to perform an addition, just specify two moves to transport the source operands to the FU which supports the `add` operation, and, within one of the following instructions, specify a move which reads the result of the addition, by reading from the output of the same FU, and transports this result to another desired destination (e.g., the input of another FU). Let us explain the operation of a small TTA program and illustrate several TTA specific code optimizations.

Consider the addition of a constant to a vector: $\vec{a} := \vec{a} + 25$. This vector addition can be translated to the following loop for a RISC like architecture (assuming that vector $\vec{a}$ contains 100 4-byte integer elements starting at address zero):

```
  ld   r2,396     // load &a[99] into r2
L:ld   r1,(r2)    // load a[i] into r1
  add  r1,r1,25   // add constant
  st   r1,(r2)    // store a[i]
  sub  r2,r2,4    // decrement address register
  bgez r2,L       // repeat if r2 >= 0
```

For a particular TTA, supporting at least three data transports (moves) per cycle, the compiler may produce the following code:

```
  396 -> r2                           // load address
L:r2  -> O1d                          // start load op.
  Rld -> O1add ; 25 -> O2add          // add constant
  Radd-> O1st  ; r2 -> O2st           // perform store
  r2  -> O1sub ; 4  -> O2sub          // decr. address
  Rsub-> O2bgez; L  -> O1bgez; Rsub-> r2
                                      // repeat if >= 0
```

Destination locations start with an `O` if they are FU operands, or with `O1` and `O2` when the FU requires two operands. Source locations start with an `R` if they contain FU results. We observe that RISC operations are coded as one or more moves. The example code shows 6 instructions with 11 moves; on average we need about two moves per RISC operation. Further, register `r1` is not used anymore; the result of the load operation is directly forwarded to the adder, without being put in one of the register units. The same holds for the result of the addition. If all operations are performed on separate FUs a smart compiler may also recognize that some moves are 'loop invariant' and be put in the loop prologue; this gives the following code:

```
  25  -> O2add ; 4 -> O2sub  // prepare operands
  396 -> r2    ; L -> O2bgez // prepare operands
L:r2  -> O1d                 // start load op.
  Rld -> O1add               // add constant
  Radd-> O1st  ; r2-> O2st   // perform store
  r2  -> O1sub               // decr. address
  Rsub-> O1bgez; Rsub-> r2   // repeat if >= 0
```

In this case the loop body only requires seven transports. It is even possible (for a very smart compiler) to get rid of the use of register `r2`; try yourself to see what the code should look like (the loop body then contains six transports).

As demonstrated, 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.

4 Transport triggering characteristics

In this section we discuss hardware and software characteristics.

Hardware characteristics TTAs have several characteristics which make them very interesting from a hardware design point:

Modularity TTAs are constructed using a restricted number of building blocks. Essentially they are built by a proper connection of FUs and bus-connections (so-called *input and output sockets*). 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 an FU-stage controller; a chain of them implements the pipelining mechanism.

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.

Flexibility and scalability TTAs have a large flexibility; because the interconnection network is separated from the FUs, both can be designed independently; this in contrast to VLIWs where each additional FU implies a change of the interconnection network and adds extra ports to the register file. TTAs have virtually no constraints on how to design and pipeline the network, and the different FUs, as long as both obey the interface specification prescribed by the used sockets. FUs can implement any functionality, even if this requires more than two operands and multiple results.

Processor cycle time If needed, the processor can be optimized for operation throughput, instead of latency. This requires superpipelining those FUs which constrain the achievable cycle time. This result is a cycle time which 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:

- *Trivial decoding*. One of the RISC design goals was to reduce decoding logic. In this respect, TTAs take the ultimate approach; they basically support only one operation format.
- *Transport efficiency*. Since the compiler is in control of every data transport, many transports can be optimized away.
- *Register efficiency*. Because many values produced during the course of a program do not need to be allocated in general purpose registers, the register file traffic rate reduces.
- *FU splitting*. FU logic can be split into independent parts, used for different functionality, without a large impact on the interconnection network and register file. When split, these parts may be used concurrently which results in a larger efficiency.

Software characteristics As became clear in section 3 when discussing the transport triggering concept, TTA compilers may exploit new code generation optimizations. They are:

Software bypassing Results from FUs may (under program control) directly be forwarded (bypassed) to FU inputs, without first being put in the register file. This is the equivalent of hardware bypassing available in all state-of-the-art ILP processors. TTAs do not need this complex hardware.

Dead result move elimination A large fraction of results produced by FUs does not have to be written in the register file at all, because these results are directly consumed, and not needed later on anymore.

Operand sharing This occurs when succeeding operations on the same FU use the same operand. This operand has only to be moved once to the FU. This is particularly interesting for loops, where often operand moves can be removed from the loop body (they become loop-invariant code).

Port sharing Sometimes the usage of a register port can be shared by multiple reads. It occurs in when reading the same register on different buses, or when accessing different registers which are connected to the same port, in case these accesses are mutually exclusive. In both cases the number of moves is not reduced, but a schedule may be generated which was otherwise inhibited by hardware resource constraints.

Scheduling freedom A final optimization concerns the extra scheduling freedom when scheduling moves instead of operations. E.g. source operands do not have to be moved in the same cycle. Also, depending on the implementation of the FUs, there can be more or less freedom in scheduling the transports of the FU result values.

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 Experimental evaluation

In the former section it was argued that making data transports visible at the architectural level results in a number of advantages. Two main advantages will be measured in this section: reduction of the number of required register ports, and the extra concurrency available when splitting FUs. For these measurements we use a highly ILP optimizing TTA compiler developed by Hoogerbrugge [14] to schedule a large set of benchmarks:

> *A68, Bison, Cjpeg, Cmp, Compress, Cpp, Dhrystone, Diff, Djpeg, Expand, Flex, Gawk, Grep Gs, Gzip, Linpack, Livermore, Mpeg_play, Od, Queen, Sed, Sieve, Sort, Stanford, Sum, Tr Unexpand, Uniq, Virtex*, and, *Wc*.

These benchmarks are well known within the 'UNIX world'. They mainly contain integer type of code, except for Linpack and Livermore, which contain substantial amounts of floating point code as well.

5.1 Register file traffic reduction

One of the main claims we have made about TTAs is that they reduce the number of register ports. This section presents measurements which support this claim. The architecture that we shall use is rather large: it contains twelve move buses, eight FUs, and four long immediates; each FU supports all operations. The number of read and write ports is varied between one and six. The motivation for this architecture is that we do not want to limit the performance by other resources than register ports.

In the experiment we vary the number of read and write ports between 1 and 6, giving 36 different RU configurations. Table 1 shows how many operations are executed per cycle for each of the 36 configurations. This experiment shows extremely good results. For example, even with the one read and one write port RU two operations per cycle can be executed. A VLIW or superscalar requires at least three FUs to reach this performance; it needs therefore minimally six read and three write ports on its RU(s). A TTA saves in this case 78% on the number of RU ports.

5.2 FU splitting

One of the advantages of TTAs is that different functionality (like arithmetic and logical operations) does not have to be combined into one FU (e.g. an ALU); it is easier to have separate FUs for non-related operations. This results both into design and performance advantages. The design becomes more modular; e.g., different numbers of adders and comparators can be used in a design. The performance is enhanced because multiple operations can be started in a single cycle, resulting in a better FU utilization.

		Number of Read Ports					
		1	2	3	4	5	6
Number of Write Ports	1	2.03	2.33	2.39	2.41	2.39	2.40
	2	2.38	3.01	3.15	3.18	3.17	3.17
	3	2.42	3.01	3.33	3.35	3.36	3.37
	4	2.41	3.01	3.34	3.38	3.38	3.39
	5	2.39	3.02	3.35	3.39	3.40	3.40
	6	2.39	3.02	3.36	3.40	3.41	3.41

Table 1: Operations per cycle as function of the number of RU ports.

To measure the effect of FU splitting we scheduled the benchmarks for two TTAs: a *small* one and a *medium* sized TTA. For each TTA two versions are used: an *unsplit* and a *spit* one. The small, unsplit TTA contains one (single cycle) ALU executing all integer type of operations including all types of load and stores, and one FPU (floating point unit), which supports all floating point operations. The small, split TTA has the same functionality except that its FUs are split; the ALU is split into five separate FUs for arithmetic, logic, shift, compare, and one for load and store operations. Similar, the FPU is split into two FUs; one for multiply and divide, and the other supporting all remaining floating point operations (except the load and stores of floating point numbers which are handled by the integer units). The medium sized, unsplit and split TTAs are similar except that their integer functionality is doubled (2 ALUs, etc.). During the measurements the number of buses is varied from one to ten.

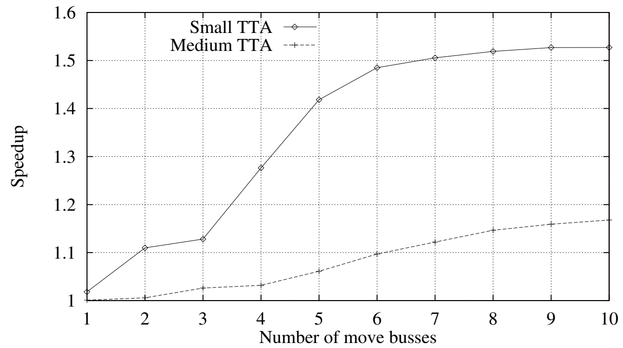

Figure 3: Speedups achieved when splitting FUs.

Figure 3 plots the speedup achieved when splitting FU functionality. Clearly for a small number of buses the performance gain is not large, because in that case even the unsplit TTA contains sufficient FUs to utilize all available transport capacity. However, for a larger number of buses the speedup can easily be more than 15% for the medium sized TTA, and more than 50% for the small one. For more parallel applications the gains will certainly be larger. Since the costs of FU splitting are marginal, it requires only extra sockets[1], we may conclude that indeed FU splitting is a key TTA feature which should certainly be exploited.

6 Summary and conclusions

A common road taken to achieve high performance processors is the exploitation of instruction level parallelism (ILP); two extreme variants of this approach are superscalar and VLIW processors. VLIWs are much cheaper to design and easier scalable to high performance levels; however they are not binary compatible with earlier architectural generations. In the embedded system world binary compatibility is of less concern, but price and performance are; therefore VLIWs are mainly targetted at that area.

VLIWs replace hardware complexity by software (compile-time) complexity. This approach can be taken one step further by making all data transports visible at the architectural level. A new type of architecture, called transport triggered architecture (TTA), results, which has shown to posses many interesting hardware and software features. Two important features have been demonstrated experimentally. It has been shown that one important bottleneck of VLIWs, the large number of required ports on the register file, can be substantially reduced. Furthermore, 15-50% speedups have been demonstrated by splitting FUs into multiple independent components. For traditional VLIWs and superscalars this splitting is difficult, because it directly results into more complexity of the register file and the FU interconnection network.

TTAs turn out to be very modular, flexible and scalable, and allow for several new code optimizations. Together, these properties make TTAs particularly interesting for very high performance embedded system design. Currently, several TTAs have been designed, and a set of tools has been developed for the automatic generation of TTAs and corresponding code for arbitrary applications written in C/C++ [15, 16].

References

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

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

[3] Arvind and D.E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[1] Sockets act as (de-)multiplexers. In fact, when FUs are not split we need also (de-)multiplexers in order being able to distribute the operands to the different logic units (contained within one FU) and to extract the results. So the hardware advantage of unsplit TTAs is really marginal.

[4] David E. Culler et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of ASPLOS-IV*, pages 164–175, Santa Clara, California, April 1991.

[5] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: a Killer Micro for a Brave New World. Technical Report Computation Structures Group Memo 325, MIT, January 1991.

[6] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *ISCA'20 proceedings*, pages 224–235, May 1993.

[7] Anant Agarwal et al. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[8] Gurinda S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *ISCA'22*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[9] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[10] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. Technical Report TR 96-037, Univ. of Minnesota, Department of Computer Science, 1996.

[11] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conference of ASCI*, Lommel, Belgium, June 1996.

[12] Z. Li, J.Y. Tsai, X. Wang, P.C. Yew, and B. Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[13] M. R. Thistle and B. J. Smith. A processor architecture for horizon. *Proceedings of Supercomputing*, pages 35–41, November 1988.

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

[15] Herman Adriani, Françoise Harmsze, and Henk Corporaal. The utilization of a fully configurable microprocessor development environment for rapid VHDL prototyping and implementation of 'C'-based algorithms. In *Proceedings of ProRisc workshop*, Mierlo, The Netherlands, November 1996.

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