1

CODE GENERATION FOR TRANSPORT TRIGGERED ARCHITECTURES

Henk Corporaal, Jan Hoogerbrugge

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
Mekelweg 4, 2628 CD Delft, The Netherlands

ABSTRACT

Transport triggered architectures (TTAs) form a class of architectures which are programmed by specifying data transports between function units. As side effect of these data transports these function units perform operations. Making these data transports visible at the architectural level contributes to the flexibility and scalability of processors. Furthermore it enables several extra code scheduling optimizations. These properties make TTAs very suitable for being applied for embedded processors.

In this article we discuss TTAs, and explain how to generate efficient code for these architectures. The compiler must exploit the available instruction level parallelism inside applications by scheduling as many useful data transports per cycle as the architecture permits. The flexibility and scalability of TTAs will be demonstrated by focusing on a particular algorithm, the minimum cost contour detection algorithm. This algorithm is e.g. used in medicine to perform real-time contour detection of the heart in an echocardiogram. Different processor options are researched for this algorithm. Results of generated code will be discussed with respect to costs, performance and code size.

1 INTRODUCTION

The performance of instruction set processors can be improved by using higher clock frequencies, exploitation of instruction level parallelism (ILP), and improvements in compiler technology. Higher clock frequencies are a result of better VLSI technology, better VLSI design techniques, and superpipelining which reduces the number of logic levels between pipeline stages. ILP is made possible due to higher transistor densities which allows for duplication of function units and data paths. The exploitation of ILP consists of mapping the ILP of the application onto the ILP of the target architecture as efficiently as possible. Compiler technology also contributes to performance. By designing an architecture such that the compiler has more control and information over the target hardware it is possible to develop new compiler optimizations. An example of this are the exact specifications of function units. When the number of stages and the pipelining degree are known to the compiler, the compiler can generate code such that pipeline stalls and bubbles are minimized. The class of transport-triggered architectures (TTAs) that we advocate fit nicely into these developments. It allows for superpipelining, exploitation of ILP, and more compiler control about the hardware.

For embedded systems it is extremely important to reduce system design costs. Therefore their processors should be tuned to the application for which the system will be designed and the design time should be short. Tuning requires flexibility and scalability. These requirements are incorporated within the MOVE framework. This framework is designed at the Delft University of Technology; it contains a set of tools for the automatic generation of application specific processor systems by means of TTAs. Central within the framework is a precisely specified architecture template and a set of architecture parameters.

Embedded system design based on a templated architecture has some similarities with high level synthesis (HLS) [1, 2]. Both methods try to find optimal hardware for a particular application. Their main differences are their flexibility and the complexity of the applications they can handle. HLS has more flexibility in generating hardware, but is usually limited to relatively small code fragments. Templated architectures are slightly limited by their template but can be designed for nearly all applications no matter how complex.

Generating code for TTAs requires a parameterized compiler which exploits both ILP scheduling techniques and TTA specific optimizations. This paper presents the code generation method we use, and its application to a particular algorithm, the minimum cost contour detection (MCCD) algorithm. This algorithm is used in the real-time processing of echocardiagram images. The next section introduces TTAs and their characteristics. Section 3 discusses the developed compiler and the scheduling techniques it uses. Section 4 describes the characteristics of the MCCD algorithm, and the code generation results for this algorithm. Section 5 shows that substantial improvements are possible by using an application specific function unit. Finally, section 6 concludes this paper.

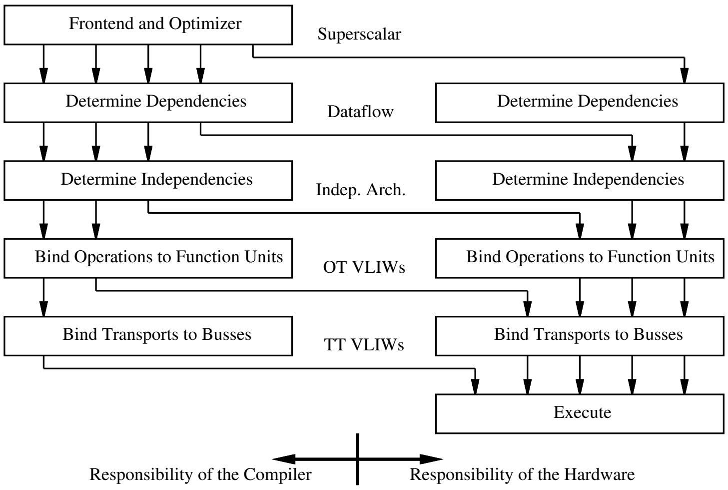

Figure 1 The division of responsibilities between hardware and compiler.

2 TRANSPORT TRIGGERED ARCHITECTURES

The major difference between TTAs and traditional operation-triggered architectures (OTAs) lies in the way they are programmed. OTAs are programmed by specifying operations; data transports are scheduled by hardware. TTAs trust the task of translating operations into data transports to the compiler [3]. This can be depicted as an extension of a scheme presented in [4]. Figure 1 shows a division between the responsibilities of the compiler and the hardware. It shows five classes of architectures: superscalar, data flow machines, independence architectures [5], operation-triggered (OT) VLIWs, and transport-triggered (TT) VLIWs. Superscalars shift most responsibilities to the hardware, while VLIWs shift most responsibilities to the compiler. The figure clearly shows that TT VLIWs depend even more on compilers than OT VLIWs.

TTAs have much in common with microcoded processors. They both present much detail of the micro architecture to the compiler/programmer in order to get better performance and hardware utilization. The major differences are that TTAs are, in our opinion, better programmable, and fully interruptible. We included only those elements in the template TTA which can be efficiently exploited by current compiler technology.

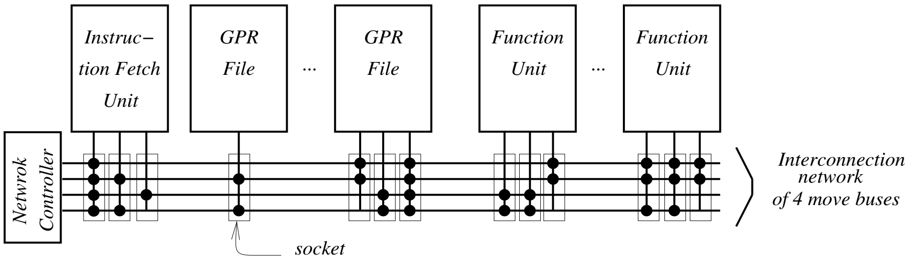

Figure 2 The general structure of a TTA.

2.1 Structure and operation

A TTA consists of a set of function units (FUs) connected by an interconnection network of one or more *move busses* (see figure 2). Each FU input and output is connected to the network by *input* and *output sockets* respectively. These sockets act as data (de-)multiplexors. The interconnection network of a TTA is not limited in any sense as long as point-to-point transports can be specified. E.g., the network is not required to have full connectivity. It is the task of the compiler to schedule the required transports, given a certain connectivity, such that execution time is optimized. Limiting the connectivity is an important tool in reducing bus loads, yielding faster transport times.

Programming a TTA consists of moving operands to the input registers of a FU that is able to perform the operation, and moving the result from the output of the FU to a general purpose register (GPR) or a FU after the operation has been performed. To make TTA programming more clear 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
```

`O1` and `O2` denote the input operand latches of the FUs, and `R` indicates the output (result) of a FU. So far, all moves are to be executed sequentially. A move instruction consists of multiple *move slots*; each slot specifies a separate transport, or *move*. Scheduling consists of assigning FUs to operations and assigning cycles, sockets, and move busses to moves. Let us assume we have two FUs named `alu1` and `alu2` for ALU operations, and one FU named `ls` for load-store operations. The suffixes '`alu1`', '`alu2`', and '`ls`' indicate the FU on which the operation is executed. The distance between the final operand and a corresponding result move should be at least the latency of the FU[1].

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

Instruction set processors need further support for immediates, control flow operations, and conditional execution. This can be easily incorporated within TTAs; the reader is refered to [3].

2.2 Software characteristics

TTAs offer the compiler control at the data transport level. This results in a number of new optimizations. They are discussed next, using the code example from subsection 2.1.

Bypassing The outputs of the add and subtract operations can be directly moved to the load-store unit. This reduces the schedule by 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 live anymore because they are used only once. In that case corresponding moves can be skipped. This gives the following schedule:

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

[1] The notation used in this paper has some redundancy. In the actual code the operation type is only specified with the last scheduled operand move.

From this small example we can make an important observation. We did not need 3 moves and 3 register file (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 useless and can be eliminated since the value of `r1` is already in `O1_add.alu`. We call this optimization *common operand elimination*.

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

Socket sharing This means that the usage of an input socket can be shared by multiple reads. This occurs when 1) reading the same register on different busses, or 2) when reading different registers which are connected to the same input socket, in the case that these reads are mutually exclusive. In both cases the number of moves is not reduced, but a schedule might be generated which was otherwise inhibited by hardware resource constraints. Socket sharing results in fewer ports on the register file [6].

Scheduling freedom A final optimization concerns the extra scheduling freedom when scheduling moves instead of operations. In OTAs transports are scheduled by hardware, usually at fixed time slots. TTAs decouple operand and result moves. As a result a compiler has much more freedom *when* to schedule these transports. E.g. the operand moves can be scheduled in different cycles. Also, depending on the implementation of the FUs, there can be more or less freedom in scheduling the result move. This scheduling freedom has two effects: 1) it enhances the applicability of former optimizations, and 2) values can stay longer in the FUs; this lowers register file usage.

2.3 Hardware characteristics

TTAs have several hardware characteristics which make them interesting for embedded systems. We will look at the following characteristics: regularity, flexibility, scalability, processor cycle time, and hardware efficiency.

Regularity TTAs are constructed using a restricted number of building blocks. Figure 3 illustrates the hierarchical structure of the required building blocks. TTAs are built up by a proper connection of FUs, and input and output sockets (register files

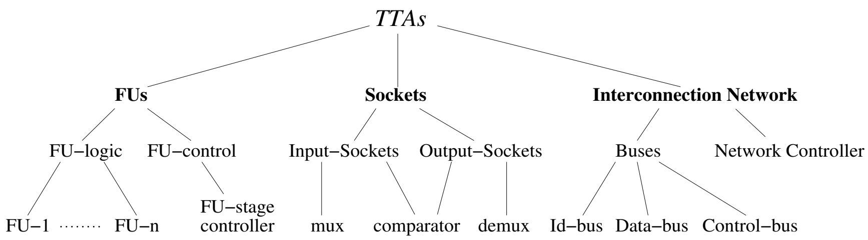

Figure 3 Building block hierarchy of TTAs.

can be viewed as FUs implementing the identity operation). FUs can be designed separately, and pipelined independently. Latching of pipeline stages is controlled by FU-stage controllers. The regularity of TTAs allows to automate the hardware design process. Different TTAs can easily be configured by assembling different combinations of these blocks [7].

Flexibility and scalability TTAs have a large flexibility; because the interconnection network is separated from the FUs via standardized sockets, both can be designed independently. This means, that there are 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. Further FUs can implement any functionality, even if this requires more than two operands and multiple results.

Performance scales by adding FUs to an existing architecture, pipelining existing FUs, or by increasing the data transport and storage capacity. In contrast to OTAs, the number of function units is not coupled to the data transport capacity (e.g. the number of register ports). E.g. special FUs can be added without having to change the transport capacity, and therefore the instruction format[2]. Section 5 shows an example of improving the performance by adding a special FU without increasing the transport capacity.

Processor cycle time If needed, the processor can be optimized for operation throughput, instead of latency. This requires: 1) result registers at each FU output, and 2) pipelining those FUs which constrain the achievable cycle time. These measures result in a architecture where the cycle time is limited by the time needed for inter FU data transport only.

Hardware efficiency There are several aspects contributing to an economic usage of hardware resources: 1) decoding is trivial because there is only one instruction

[2] As long as source or destination identifier space is available.

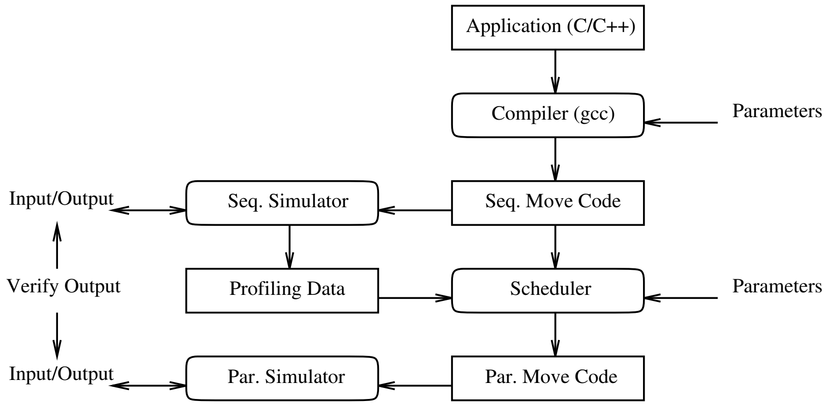

Figure 4 The compiler trajectory.

format; 2) the interconnection network can be designed to support average required transport capacity only, instead of worst case, as needed for OT VLIWs; 3) because many values produced during the course of a program do not need to be allocated in the RF, not only the GPR traffic rate reduces, but also the number of GPRs required; finally, 4) FU logic can be split into independent parts, used for different functionality, without a large impact on the interconnection network. When split, these parts may be used concurrently which results in a larger efficiency.

3 CODE GENERATION

The compiler trajectory consists of the GNU gcc compiler, a scheduler, and a simulator (see figure 4). The gcc compiler is ported to a generic sequential TTA[3]. The produced sequential code contains besides move code also annotations to transfer information from gcc to the scheduler. Gcc has parameters that allow the user to strip the target operation set whenever that target embedded processor does not support certain operations (e.g., byte and half word loads and stores). The scheduler is controlled by a machine description file that describes the target processor. This description includes available function units, the operations that each function unit supports, latencies, RF sizes, and a description of the interconnection network. A simulator for both sequential and parallel code is provided 1) to obtain profiling information, 2) to verify the compiler, 3) to verify the scheduler, and 4) to obtain application characteristics.

[3] A different frontend for functional languages producing generic sequential TTA code is also available [8].

3.1 Scheduling an executable

The input to our scheduler is a complete executable of generic MOVE code. The scheduler starts with reading the executable and its profiling data when present. After that the following steps are carried out to convert a sequential executable into a parallel executable:

```
schedule an executable {
    perform inlining, unrolling, and loop peeling
    for each procedure do {
        perform data and control flow analysis
        perform memory reference disambiguation
        make a data dependence graph and determine regions
        perform register allocation
        schedule each region
    }
}
```

Function inlining, loop unrolling, and loop peeling are done in order to expose more parallelism for the scheduler. These code expansions are controlled by profiling data, the amount of code duplication, and a user specified parameter. The data and control flow analysis step is pretty standard. It partitions a procedure into natural loops [9] which correspond to *regions*. It performs life-analysis to detect off-liveness during inter-basic block code motions and to build an interference graph during register allocation. The data flow analysis step also assigns each life-range to a unique GPR. This should eliminate most of the false data-dependencies. The memory disambiguator makes address derivations by means of reaching definitions produced by the data flow analysis phase. These derivations are used for the gcd-test in order to try to prove independence between two memory references. Register allocation is performed using Pinter's algorithm [10]. The register allocator attempts to assign registers without creating false dependencies for the scheduler. After building a data-dependency graph for the whole procedure, the scheduler schedules each region of the procedure individually. Scheduling a region will be discussed in the next subsection.

3.2 Scheduling a Region

In order to find sufficient ILP to justify the cost of multiple function units and data paths, a scheduler should have a larger scope than a single basic block at a time.

Various scheduling scopes that are larger than a single basic block are known from the literature, like a *trace* [11], *superblock* [12], *hyperblock* [13], *decision tree* [14], and a *region* [15]. Our scheduler operates on regions which is the most general (largest) scheduling scope of the five mentioned above. A region corresponds to the body of a natural loop. Since loops can be nested, regions can also be nested in each other. Like natural loops, regions have a single entry point (the loop header) and may have multiple exits.

The scheduler schedules a region using the next algorithm:

```
schedule a region {
    for each basic block b of the region in topological order do {
        1: schedule b's operations by means of list scheduling
        2: try to import operations from reachable basic blocks into b
    }
}
```

The list scheduler repeatedly takes an operation from the ready list and schedules it at the first cycle where data-dependencies and resource constraints are satisfied. The ready list contains those operations for which all predecessor operations have been scheduled. The selection from the ready list is based on the critical path heuristic used by almost every other list scheduler.

Next all operations from basic blocks in the current region reachable from the current basic block are examined. The priority function that is used to select a ready operation is computed as follows:

$$global\_priority = probability * local\_priority - duplication\_cost$$

where $probability$ is the probability (estimated or based on profiling data) that the operation will be executed after the current basic block is executed, $local\_priority$ is the critical path priority[4] function scaled between 0 and 100, and $duplication\_cost$ is a parameter that is subtracted whenever the inter-basic block code motion requires code duplication. The scheduler tries to schedule the selected operation in the current basic block. This is repeated until all ready operations from reachable basic blocks have been tried.

Code motion from one basic block *source* to another basic block *destination* requires code duplication whenever *destination* does not dominate [9] *source*, and requires

[4] The local priority is based on data-dependencies between moves within the same basic block.

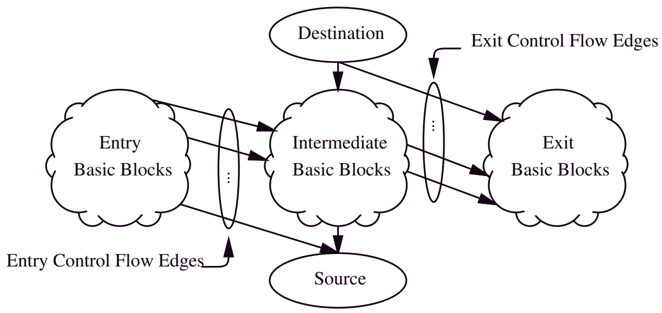

Figure 5 A fragment of a region.

off-liveness checks whenever *source* does not post-dominate *destination*. Figure 5 shows what needs to be done when an operation is taken from *source* and placed in *destination*. First the scheduler determines all intermediate basic blocks in the current region between *source* and *destination*. All immediate predecessors of *source* and of the intermediate basic blocks that are not reachable from *destination* require a duplication. This is the set of *entry basic blocks* in figure 5. Whenever it is not possible to place one of these duplications the whole code motion from *source* to *destination* is aborted.

The off-liveness checks consist of testing the immediate successors of *destination* and the immediate successors of the intermediate basic blocks from which *source* is not reachable. These blocks are shown in figure 5 as *exit basic blocks*. Whenever the value that an operation defines is live at the beginning of one of the exit basic blocks the operation needs to be predicated by the predicate register that controls the corresponding exit control flow edge. If predicated execution is needed an extra (*flow*) dependence is introduced between the operation and the (compare) operation that defines the predicate. A memory store operation can be viewed as a normal operation that defines *memory*, which is always live.

3.3 Scheduling an Operation

Scheduling an operation[5] consists of scheduling the moves that perform the operation. Each operation consists of one or more operand moves, and zero or one result moves (only a store has no result). The scheduler starts with scheduling the latest available operand move (of the selected operation). It has to find the first cycle where 1) a suitable

[5]For example, an addition, multiply, load, or store. Jumps, function calls, OS calls, and copy operations are handled differently.

function unit is available, 2) a suitable move bus is available, and 3) an immediate can be provided when needed. The scheduler proceeds with scheduling the other operand moves and the result move. This consists of finding a cycle where a suitable move bus is available and, when needed, an immediate for an operand move. During these steps the scheduler has to make the following checks in order to use the FU pipelines correctly:

1. Operand and result moves should be scheduled in first-in first-out order.
2. The number of outstanding operations on a function unit may not exceed the capacity of the function unit.
3. An operand register is in use until the operation proceeds to the second pipeline stage.

When one of the checks fails the (operand or result) move has to be placed in an earlier (in case of an operand move) or later cycle (in case of a result move). There are cases where this does not help. In these situations the scheduler has to unschedule the first scheduled operand move and try it one cycle later.

During scheduling the optimizations listed in section 2.2 are exploited. Whenever the scheduler schedules an operand move the scheduler checks whether the value used by the move needs to be bypassed from the function unit that produced it instead of getting the value from a GPR. If bypassing is required the source field of the move is changed. Furthermore, a check is made to find out whether the result move of the operation that produced the value has become dead and can be eliminated. This happens when all usages of a value are in the same cycle as the result move that produced the value. A dead result move is marked as being dead and its move bus is released.

Operand sharing is handled while scheduling an operand move. A few instructions above the instruction in which the operand move is placed are examined to find out whether an operand move with the same source is taking place to the same operand register on the same FU.

Socket sharing is handled when a socket is required. When a suitable socket is in use by another move it can still be used if the register that is accessed via the socket is the same for both moves.

When an operation is imported from a reachable basic block into the current basic block it is not allowed to increase the length of the current basic block. This guarantees that extended basic block scheduling is always better than basic block scheduling. However, when a result move reaches the "bottom" of the current basic block it is allowed to move to the immediate successor(s) of the current basic block. Similarly, operand moves are allowed to move to immediate predecessor(s) of the current basic block when they hit the "top" of the current basic block. Due to these two code motions it is possible that operations are in progress during the transition from one basic block to another.

4 COMPILING THE MCCD ALGORITHM

Figure 6 MCCD applied to echocardiogram images.

In this section we demonstrate the capabilities of the compiler by exploring different TTAs which are suitable for the Minimum Cost Contour Detection (MCCD) algorithm [16]. The MCCD algorithm is able to generate a contour in an image. E.g., as shown in figure 6, it can be used to generate the contour of the heart in an echocardiogram of the heart. This is a useful application in medicine, because this way the volume of the heart can be estimated. In practice, images are processed real-time, i.e. 25-30 images have to be processed per second. For each image a new contour has to be calculated and drawn. The MCCD algorithm uses the contour of the previous image to reduce the search area for the next image. First the characteristics of this algorithm are described. Next we look into the results of compiling this algorithm for different TTAs.

4.1 MCCD characteristics

Table 1 lists several characteristics of the MCCD algorithm. The parallelism upper bound is determined by trace analysis; it is an indication of the amount of available parallelism. Its value is determined by considering run-time data dependencies only. A real compiler will never be able to exploit all the available parallelism, because it has to

Characteristic	Value
Lines of C code	2851
Number of executed operations/image[a]	747,888
Parallelism upper bound	36.4 ops/cycle
Most frequently executed operations	
• add	39%
• and	14%
• load byte	10%

[a]This number depends on the contour sampling accuracy; the given value is for a sampling area of 128×25, where each sample is taken from an interpolation of 25 pixel values.

Table 1 Characteristics of the MCCD algorithm.

Routine	Description	Operation count
gvi5	Gray value interpol.	329,741
path	Find minimum cost path	144,482
cost	Cost function calc.	108,765
sptr	Spatial transformation	60,176
n-ct	Normalize costs	46,244
next	Next contour prep.	37,009
ipol	Interpolation	8,738
sqrt	Integer square root	7,168
smth	Contour smoothing	5,565
misc	Miscellaneous	6,075
all		747.888

Table 2 The operation profile of the MCCD algorithm.

consider also control dependencies and resource constraints. Further, at compile-time not all data dependencies are precisely known.

The first step to design an embedded processor is compiling the application to sequential move code and profiling it in order to identify the most critical routines. Table 2 shows the operation profile of the MCCD algorithm resulting from processing one 512×512 pixel image. The gvi5 routine is clearly the most time consuming one; it is responsible for 44% of all executed operations. This function computes the sum of 25 pixel (8-bit) values of a 5×5 matrix somewhere in the image. Figure 7 shows the C source code

```
void gvi5(...)
{
   for(...)  {
      sum = 0;
      for(i = -2; i <= 2; i++)
         for(j = -2; j <= 2; j++)
            sum += image[...  + i][...  + j];
      *scanmatrix++ = sum;
   }
}
```

Figure 7 The source code of gvi5. Unrelevant details are replaced by dots.

of gvi5. In order to expose more parallelism and to reduce loop overhead it is wise to unroll the two inner loops completely.

4.2 Compiling MCCD

In order to find a satisfying TTA configuration for a certain application we have developed a tool that explores the design space [17]. Each point in the design space corresponds to a particular TTA configuration. Each point has a certain performance and cost. The goal of the design space exploration tool is to find the so-called *Pareto* points. These are the points for which there are no other points in the design space that are both cheaper and faster. The design space exploration tool presents the found Pareto points to the designer and lets him choose the preferred configuration for his particular situation. Figure 8 shows these points (generation of this figure takes about half an hour on a HP730). The listed execution times[6] and costs are based on a CMOS $1.0\mu m$ processor realized at our laboratory [18].

We selected the processor with a cost of 150. It contains 6 move busses, an integer register file containing 32 registers with 3 read and 3 write ports, 2 integer adders, 1 load-store unit, and 2 booleans for conditional execution. The run-time cycle count profile for MCCD scheduled on this architecture is shown in table 3. The shape of this cycle profile is quite similar to the operation profile; this roughly means that the amount of exploitable parallelism is distributed homogeneously. The gvi5 routine still

[6]Integer, logic and compare units have a latency of one, the load-store unit has a latency of two.

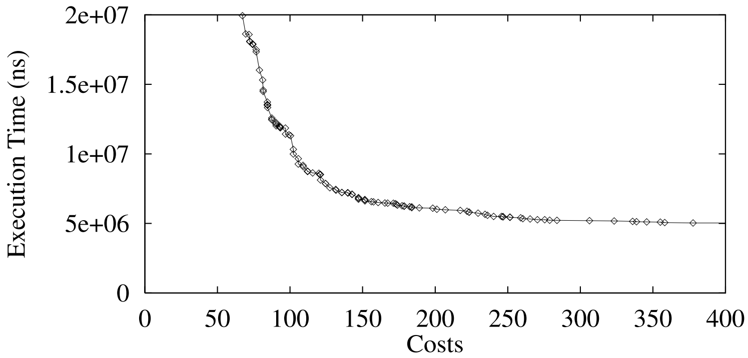

Figure 8 Pareto points for the MCCD application.

Routine	Cycle count
gvi5	132,498
path	73,161
cost	48,040
sptr+sqrt	38,173
n-ct	26,002
next	15,743
ipol	5,030
smth	2,288
misc	4,448
all	345,383

Table 3 The MCCD cycle count profile for a processor with cost = 150.

consumes about 37% of all the executed cycles; in the next section we research this routine in more detail.

5 IMPROVEMENTS

If the designer is satisfied with one of the processors configurations corresponding to the Pareto points in figure 8, he can generate silicon processor layout, compile the code for this particular processor, and realize the embedded system. However, there are many more solutions. It is recommended to look more closely at the most time consuming routines. As an example we will do that for the gvi5 routine (see figure 7).

The outer loop of this routine contains (after unrolling inner loops, and constant propagation) 24 additions which are sequentially dependent on each other. A first approach might be to perform tree height reduction on this chain of additions. This approach certainly may reduce the cycle count of the gvi5 routine. However, a search with our search tool shows that it requires a processor with multiple load units, many adders, and many move busses. This means that 1) the data memory should be multiported, and 2) instructions become very large, and code density will be low.

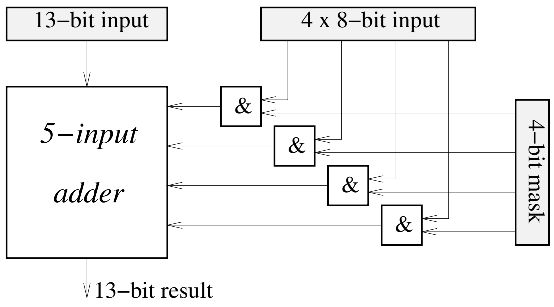

Figure 9 A special function unit for selective, 5-input, addition.

A more interesting solution is 1) to use a 32-bit load unit capable of loading up to four pixel values per cycle, and 2) to add a special function unit (SFU), as shown in figure 9. This SFU is capable of adding up to four 1-byte values to a running sum (of 13-bit). A 4-bit mask is used to select the bytes to be added. A multiple input adder is not as expensive as one might expect at first sight. Each extra adder input only costs one additional layer of full adders and selectors (and-gates).

The source code of gvi5 is slightly rewritten to use this SFU. Note that in order to load a square of 5×5 pixel values, always ten 32-bit loads are needed. The results of searching the design space, after changing the TTA configuration to include one SFU, is shown in figure 10. For comparison, the former results, without SFU, are also drawn. Again we selected a processor with costs equal to 150. It contains 6 move busses, an integer register file containing 32 registers with 3 read and 2 write ports, 2 integer adders, 1 load-store unit, 2 booleans, and the SFU. The cycle count profile for this processor is given in table 4. We observe that 1) the total cycle count is reduced by 16% (for an equal cost processor), and 2) the gvi5 routine is not anymore the major cycle consumer. It will be clear that even better cost-performance ratios can be obtained when studying the other major cycle consumer routines as well.

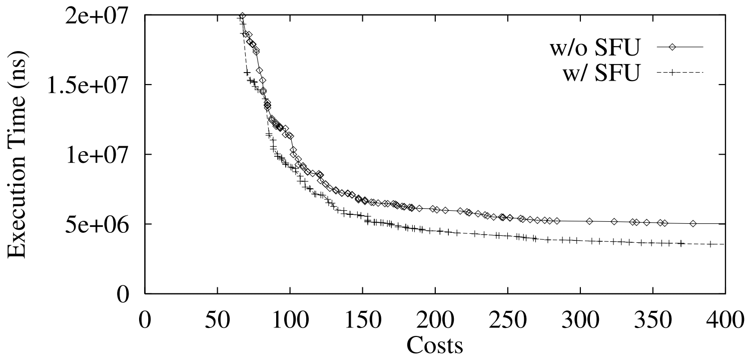

Figure 10 Pareto points, with and without a special function unit.

Routine	Cycle count
gvi5	61,837
path	73,186
cost	51,700
sptr+sqrt	47,133
n-ct	28,012
next	15,743
ipol	5,026
smth	2,329
misc	4,428
all	289,394

Table 4 The cycle count profile for the MCCD algorithm.

It is further interesting to look at the code size of the gvi5 routine. Table 5 compares the two schedules of gvi5 for the indicated processors. The listed number of move slots includes move operations and NOPs (unused move slots). We conclude that adding a SFU results in a substantial enhancement of the code density. This is particularly important for embedded processors, where the code might be located in on-chip memory.

Characteristic	w/o SFU	with SFU	Reduction
Operations	112	51	54%
Cycles	45	21	53%
Move slots	270	106	61%

Table 5 Comparing the schedules of gvi5; w/o SFU, and with SFU.

6 CONCLUSIONS

In this paper we have shown that TTAs have many properties that make them perfectly suitable for embedded processors. They are flexible, scalable, and allow for a short cycle time and high hardware efficiency. Furthermore they make new compiler optimizations possible. The code generation trajectory for TTAs has been outlined. It included a parameterized compiler and scheduler with exploits instruction level parallelism, and supports the TTA specific optimizations.

The flexibility of TTAs and its compiler trajectory has been demonstrated by means of the MCCD algorithm. The most critical routine of the algorithm has been identified and a special operation has been designed for this routine. After informing the scheduler and simulator about this operation it is possible to generate code that uses this operation. Due to this operation, which can be implemented in a FU which costs about the same as a 32 bits adder, the cycle count and the code size of the most critical routine is reduced by more than 50%. The whole application runs 16% faster on an equal cost processor.

The use of a framework containing automatic design and code generation tools, which are based on a templated transport triggered architecture, allows us to do a quantitative search within a large processor design space in a short time. This makes this framework particularly suitable for the design of embedded systems.

REFERENCES

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

[2] Christian Ewering. Automatic High Level Synthesis of Partitioned Busses. In *Int. Conf on Computer-Aided Design (ICCAD)*, pages 304–307, Santa Clara, November 1990.

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

[4] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1/2):9–50, May 1993.

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

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

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

[8] 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.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Reading, Massachusetts, 1985.

[10] Shlomit S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 248–257, June 1993.

[11] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[12] Scott A. Mahlke et al. Sentinel scheduling for vliw and superscalar processors. In *Proceedings of ASPLOS-V*, pages 238–247, Boston, September 1992.

[13] Wen mei W. Hwu et al. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1/2):229–248, May 1993.

[14] Peter Y.T. Hsu and Edwards S. Davidson. Higly Concurrent Scalar Processing. In *Proceedings of ISCA 13*, pages 386–395, June 1986.

[15] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[16] J.G. Bosch, G. van Burken, S.S. Schukking, R. Wolff, A.J. van de Goor, and J.H.C. Reiber. Real-Time Frame-to-Frame Automatic Contour Detection On Echocardiograms. In *Proceedings van Computers in Cardiology*, 1994.

[17] Jan Hoogerbrugge and Henk Corporaal. Automatic Synthesis of Transport Triggered Processors. In *First Annual Conf. of ASCI*, May 1995.

[18] Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, May 1995.