High Level Software Synthesis for Signal Processing Systems
Sebastian Ritz, Matthias Pardert and Heinrich Meyr
Aachen University of Technology
Templergraben 55, ERT 5240, 5100 Aachen, Germany

Abstract
For the design of complex digital signal processing systems, block diagram oriented simulation has become a widely accepted standard. Current research is concerned with the coupling of heterogeneous simulation environments and the transition from simulation to the implementation of digital signal processing systems. Due to the difficulty in mastering complex design spaces high level hardware and software synthesis is becoming increasingly important. In this presentation we concentrate on the block diagram oriented software synthesis of digital signal processing systems for programmable processors, such as digital signal processors (DSP). We present the synthesis environment DESCARTES illustrating novel optimization strategies. Furthermore we discuss goal directed software synthesis, by which code is interactively or automatically generated, which can be adapted to the application specific needs imposed by constraints on memory space, sampling rate or latency.

1 Introduction
Analysis and design of complex signal processing systems increasingly rely on sophisticated simulation [1,2,3,4,5] and synthesis techniques [6,7,8]. In the area of digital signal processing for the design of communication links, speech processing or measurement systems, several simulation environments have evolved which take into account the specific design requirements in these application areas [1]. Most of these environments are based on data flow oriented modelling and simulation techniques, i.e. the systems are represented by hierarchical block diagrams where each block represents an algorithmic model (e.g. a filter, coder or channel) and the interconnections between the blocks represent the data flow of the system.

Current research is concerned with the seamless transition from data flow oriented simulation to the implementation of a digital signal processing system. In order to obtain this transition it is necessary to couple different simulation engines such that the most efficient simulation technique on every abstraction level is applied [4,2] and to integrate a synthesis environment for interactively or automatically mapping the block diagram oriented representation of signal processing systems onto different target hardware architectures [8,6].

In this paper we discuss a synthesis approach for software programmable architectures. The work presented here concentrates on digital signal processors (DSPs) as target architectures. Due to architectural features (e.g. dedicated multiplier and adder, Harvard bus structure) tailored to the needs of typical signal processing tasks, they offer impressive computational power [9]. Nowadays commercially available 32 bit floating point DSP chips such as AT&T’s DSP32C [10], TI’s TMS320C30/40 [11], Analog Devices’ ADSP 21020 or Motorola’s 96002 [12] range from 25 Mflops to 100 Mflops peak performance per single chip. Together with other fixed point DSP chips such as TI’s TMS320C25/50, Motorola’s...
DSP56002 or Analog Devices' ADSP 210x these processors are attractive target architectures for real time implementations of signal processing systems with sampling rates in the range between 1 kHz and 1 MHz depending on the signal processing complexity. Due to the parallel processing capabilities of the latest generations of DSP chips the achievable sampling rate will be scaled up further. All results presented are as well applicable for processors with DSP like cores such as transputers (T800/T9000) or specialized RISCs (e.g. Swordfish, RS/6000 [13]).

With these target architectures in mind, we present the software synthesis environment DESCARTES [8] focussing on optimization strategies required for an application and target specific mapping of block diagrams onto DSP chips. Particular attention has been paid to supporting interactive trade off between different implementation goals such as optimization of throughput or memory consumption. This allows code adapted to the needs of the user to be generated and thus supports rapid prototyping on the implementation level. DESCARTES is tightly coupled to the design environment COSSAP [3] by means of multiple paradigm block diagrams. This means that the block diagram may be used for both simulation and synthesis by automatically applying appropriate scheduling techniques [8]. In contrast to [2], specialized interfaces are employed between the different scheduling paradigms leading to less flexible but more efficient implementations.

In section 2 we will discuss data flow paradigms which are suitable as the base for software synthesis and introduce a novel data flow paradigm for DESCARTES. Section 3 presents different optimization strategies, which may be applied independently from each other in order to achieve application specific software synthesis. Section 4 introduces concepts of interactive and automatic goal directed synthesis.

2 Data Flow Paradigms for the Computation of Signal Processing Systems

For simulation and synthesis of signal processing systems, data flow languages in form of graphical block diagrams have attracted considerable interest. Block diagrams are data flow graphs where each node performs atomic (e.g. arithmetic) operations or more complex signal processing functions such as FFT or trellis decoding. The directed arcs of the data flow graph specify a possibly infinite stream of data samples propagating from node to node. Though most design packages do not make any assumption on the internal algorithmic complexity of blocks, their underlying data flow paradigm determines the external input/output behavior and thus the flexibility and efficiency for simulation and synthesis of signal processing systems. Below we briefly review two main data flow paradigms which have been successfully employed for modeling signal processing systems and then introduce a new, extended synchronous data flow paradigm for DESCARTES.

The synchronous data flow (SDF) [14] paradigm requires that every block consumes and produces a fixed a priori known number of input/output samples at each of its ports each time the block is invoked. Since all rates are known prior to execution, the computing the order of the invocation of the blocks (i.e. the scheduling) can be performed at compile time, i.e. statically [14]. Furthermore the size and amount of signal buffers required for intermediate storage of data samples may also be determined at compile time.

The asynchronous data flow (ADF) paradigm [5] offers more modeling flexibility. Blocks are allowed which neither have to consume a fixed number of input samples nor have to produce a fixed number of output samples. Instead, ADF-blocks only require a minimum
number of input samples to be specified. ADF is very important for communication system simulation because it greatly facilitates modeling and simulation of rate-sensitive important subsystems such as timing synchronization. ADF requires the scheduling to be performed during runtime, i.e. dynamically, since only then the scheduler is able to take the actual input and output rates into account. Since SDF imposes little buffering management and scheduling overhead at run time, it is suited for the implementation of signal processing systems, whereas ADF is more suited to communication system simulation. Although SDF has been successfully applied as a graphical programming language for signal processing systems [6], extensions have to be introduced when applying it to highly efficient DSP software synthesis.

SDF blocks behave inefficiently when only few operations have to be performed on single data samples. For example an adder block requires a lot of runtime overhead for memory operations when two single data samples have to be fetched and stored for a single accumulation. Since a common architectural feature of current DSP architectures is an arithmetic pipeline consisting of a multiplier and an accumulator combined with a multi bus system, much better utilization of hardware can be gained, when blocks operate on tuples of data samples. This is because a DSP can perform several memory operations, such as data sample fetches, multiplication, accumulation and address generation in parallel. With these architectural features in mind we have proposed the scalable synchronous data flow (SSDF):

- Fixed data rates are specified for each port of a block.
- A block may consume or produce any integer multiple \( N_b \) (the local blocking factor) of these predefined data rates during each invocation.
- The decision on the actual value of \( N_b \) at each invocation of a block may be taken by the scheduler or by a separate optimizer.

In order to support this kind of vector processing the code of an SSDF block provides a loop where the number of iterations is dependent on the blocking factor \( N_b \). Note that the blocking factor is local to each single block and can vary between different invocations of the same block.

The code of a block supporting this kind of vector processing is shown in figure 1b. (Figure 1a. displays the same block's code without vector processing).

void add() {  
    *AddOut = *AddIn1 + *AddIn2;
}  

void add() {  
    register int i = Nb - 1;
    do {  
        *AddOut++ = *AddIn1++ + *AddIn2++;
    } while ( i-- \times 0 )
}  

a. without  
b. with

Figure 1: Code of an "Add" block without and with vector processing. \( N \) is the blocking factor in b.

This extended definition of an SDF block allows for a wide variety of optimization strategies, presented in section 3.1.
A fundamental limitation of the data flow representation of signal processing algorithms is its lack of control flow, which is important for the implementation of e.g. timing synchronization. A common workaround is to introduce blocks performing control while adhering to one of the data flow paradigms. In ADF, for example, it is straightforward to specify data dependent switch and select blocks. There are also approaches to extend SDF towards limited control functionality [15]. However, when considering highly efficient software synthesis, this introduces undesired overhead. In DESCARTES, no attempts are made to incorporate the control semantics into the data flow block diagram. Instead, strict separation of data path and control path synthesis is preferred [16].

3 Block Diagram oriented software synthesis

When synthesizing real time DSP code, implementation constraints imposed by the available memory space (on- or off chip RAMs), sampling rates, latency, or precision play a decisive role. Different synthesis goals can be derived from these constraints, such as minimizing memory consumption, optimizing throughput or enhancing the precision of the system. Below we define the throughput \( \Theta \) of a system as the processing time per input sample excluding processing time due to external I/O (e.g. AD, DA). The latency \( \Delta \) is defined as the number of sampling periods required after a certain input sample is given for the corresponding instance of the output sample to appear. Memory consumption \( M \) is divided into program memory \( M_p \) (text segment) and data memory \( M_d \) requirements.

Since the synthesis goals often mutually exclude each other, there is no single optimal strategy for synthesizing real time code. Instead, it is very important to allow interactive trade off between different synthesis goals, which is a central issue in rapid prototyping on the implementation level. Thus for the synthesis of DSP code from the block diagram oriented specification we have incorporated independent mapping and optimization strategies into the DESCARTES environment. This gives the user the opportunity of interactively combining different strategies in order to obtain code being adapted to his or her application specific needs. This approach is extended further by goal directed synthesis, which allows the best combinations of optimization strategies to be found automatically for a given goal [17]. This approach is discussed in section 4.

Most of the optimization techniques we present are not dependent on the kind of code synthesized. Due to the rapid evolution in the development of new programmable DSPs or DSP like processors, primarily C code is synthesized. This allows fast migration to new DSPs and saves investments made in the development of block libraries. The disadvantage of synthesizing C code lies in the difficulty of exploiting the special architectural features of DSPs. Furthermore, the efficiency of the synthesized code relies on the built-in optimizations of the C compiler. Experience has shown that efficient code can be synthesized if one follows the design rules of the specific C compiler, that lead to compiler specific instead of processor specific synthesis [18,19]. Since the computation kernel of a block may consist of a call to an assembly routine (or inline assembly), the runtime libraries of DSP vendors or user written assembly code may be utilized for synthesis. Direct assembly code synthesis is supported by DESCARTES for fine grain data flow synthesis (i.e. for systems consisting of atomic blocks) without the need of external assembly libraries.
3.1 Block diagram level optimizations

The basic task on the block diagram level consists in computing the schedule. We propose a new hierarchical scheduling strategy where the degree of vector processing is scalable by a user supplied global blocking factor $N_g$. This scheduling approach leads to improved throughput and latency of the synthesized code. Since it requires an increased amount of memory space for the buffering of data samples, we also address efficient mapping strategies of signal buffers onto memory.

3.1.1 Scheduling of SSDF Systems

First, due to the SSDF paradigm subsequent calls of the same block can be avoided just by calling this block once with an appropriate blocking factor $N_g$. This situation occurs in many typical schedulings in conjunction with multi rate systems. This does not hurt in terms of memory consumption but improves throughput.

A straightforward technique for achieving further enhanced throughput is to increase the minimal scheduling period by an integer factor $N_g$. This implies, that each block $B$ of the graph will be invoked $N_g \cdot q(B)$ times within one scheduling period, where $q(B)$ is the minimum required number of invocations of block $B$ within one scheduling period. Depending on the topology and sample rates of the block diagram, some or all of these invocations of a block can be joined using $N_i = N_g \cdot q(B)$. Increases in throughput up to an order of magnitude can be achieved by consequently applying this technique due to the reduced overhead and to the utilization of DSP pipelining [8].

However, $N_g > 1$ influences the size of the signal buffers. If, for example, the adder in Fig. 1 consumes 10 samples from each input port producing 10 output samples, this implies, that all three buffers have the size 10 (rather than 1).

A major shortcoming inherent to the increase of the scheduling period by a factor $N_g$ is related to feedback loops. The rate with which samples are rippled through a feedback loop depends on the number of delays (initialized buffers) in that loop. Vector processing is restricted to that number. If it is smaller than $N_g$, the blocking capability of the blocks in the loop is partially disabled, resulting in large program code for the repeated calls of those blocks (s. Fig. 2).

![Diagram of a graph with feedback loop](image)

$N_g = 1 : \phi = \{G,A,B,C,D,E,F,H\}$


Figure 2: Scheduling $\phi$ of a graph with feedback loop using blocking

Obviously, this is an undesirable effect, especially if $N_g$ is chosen large (10 to 100).

Taking a closer look at the scheduling for $N_g = 5$ in Fig. 2, it can be seen that the call-sequence of the blocks G,D,H,G is repeated 4 times. In that case, a solution such as

$\phi = \{G,A,B,A,C,D,H,G,C,D,E,F,H\}$

is found.
will be equivalent, but consumes much less program code because \(4(C,D,H,G)\) can be mapped onto a single loop construct.

The above observations suggests that a scheduling algorithm should not just consider the flat net list of a given block diagram, but take topological features of the graph into account. Therefore an analysis of the topology has to be performed prior to the actual scheduling. Then, scheduling techniques must be developed that take advantage of all useful topological information while keeping the program code small even in case of an arbitrarily large global blocking factor \(N_B\).

Graph partitioning When searching for feedback loops (circuits) in a directed graph, it is useful to partition the graph into its strong components. A subgraph \(F \subseteq G\) is a strong component of the graph \(G\), if for all pairs of nodes \(u, v \in F\) there exist paths \(p_{uv}\) and \(p_{vu}\) and this property holds for no other subgraph \(F'\), \(F \subseteq F' \subseteq G\) [20].

Circuits can only be found within such strong components. After partitioning, the resulting top level graph is purely feedforward and thus well suited to being scaled with the global blocking factor \(N_B\). All strong components contain either a primitive block or at least one circuit. However, nested or overlapped circuits can occur. Therefore, the strong components are further analyzed in order to detect all elementary circuits of the graph. Algorithms for these graph operations are provided by [20,21].

Let us define a subgraph's \(q\)-value as follows: given a subgraph \(F \subseteq G\), \(q(F) = \gcd\{q(B_i) \mid B_i \in F\}\). For internal scheduling of this subgraph, \(q'(B) = q(B)/q(F)\) is taken into account.

The read data rate from arc \(A\) of the block succeeding to arc \(A\) will be referred to as \(r(A)\). The predecessor of arc \(A\) writes with the rate \(w(A)\). The number of samples available in an arc \(A\) at time step \(k\) is \(b_A(k)\). The number of initialized samples (delays) in arc \(A\) is called \(b_A(0)\). A block reading from arc \(A\) is \(\&_A\), a block writing to \(A\) is \(\&_A\).

Information about the delays, contained in each feedback loop \(L\) is gathered during graph analysis.

\[
N_l(L) = \max_{A \in L} \left\{ \frac{b_A(0)}{r(A) \cdot q'(B_A(A))} \right\}, \text{ with } \lfloor X \rfloor : \text{integer part of } X \quad (1)
\]

gives the maximum bound of the factor \(N_k\), by which \(q(L)\) can be increased while maintaining the shortest possible schedule for that loop (s. Fig. 3). Each block needs to be activated only once during execution of the loop when scaling \(N_k \leq N_l(L)\).

![Figure 3: \(N_l(L)\) depending on the delays in a circuit](image-url)
Hierarchical scheduling  Remember that our goal is to support the vector processing facility as efficiently as possible for high throughput while maintaining the program code size small.

After graph partitioning, a hierarchical representation of the block diagram is available. The graph is split up into its strong components. These are further subdivided into elementary circuits, which in turn consist of primitive blocks. Note that nested circuits can share some blocks.

If a strong component internally consists of multiple circuits, all with homogeneous delay and data rates, no further advantage can be taken of the internal topology of this component. If, in contrast, the delays in the circuits are different from each other, significant gains can be obtained by scheduling them hierarchically.

The first step for building the hierarchical scheduling is to determine the order in which the strong components are processed. This is a straightforward task, because they are interconnected in a pure feedforward network. The strategy is to process as many samples as possible upon each component's activation. If the component is a primitive block, this is performed by applying the appropriate blocking factor $N_b$. The algorithms starts with scheduling the signal sources of the block diagram. Then all direct successor blocks of the already scheduled blocks are considered. A block will only be scheduled if all of its predecessors in the block diagram have run $N_q(b)$ times (As often as required for the schedule to become periodic).

No assumptions have been made concerning the particular value of $N_q$ up to now. It can be either user supplied or otherwise computed by the scheduler and adapted to the block diagram's loops (s. below).

Next, the strong components are considered, each separately. Components consisting of only one primitive block have already been worked off at the top level. Nested circuits are hierarchized in such a way that an inner circuit appears as a single block with respect to an outer circuit. The hierarchization is performed depending on the values of $N_i(L)$ of the circuits (the outermost circuit having the largest $N_i$). Scheduling of the circuits is started with the block being runnable with the largest $N_b$ due to delays at the input arcs.

If $N_q < N_i(L)$, $N_b = N_q$ is applied for all blocks. If $N_q$ is an integer multiple of $N_i(L)$, the circuit's scheduling order is repeated $N_q/N_i(L)$ times using $N_b = N_i(L)$ for the internal blocks. If $N_q > N_i(L)$ but not an integer multiple, two strategies can be applied: (1) the circuit schedule is repeated $N_q$ times using $N_b = 1$ internally which produces the smallest possible program code at the cost of throughput or (2) the circuit's schedule is repeated as often as possible using $N_b = N_i(L)$ and then once more for the remainder to $N_q$. This improves throughput but also enlarges code size.

Applying the above scheduling strategy allows the synthesis of highly efficient real-time code without unnecessarily wasting program memory space.

3.1.2 Memory allocation for signal buffers

In this section we will consider memory allocation strategies for the signal buffers connecting the blocks of a block diagram. We distinguish between memory allocation of signal buffers interconnecting strong components (so called top level buffers) and those connecting blocks within a strong component. The total number of samples written by a block $B$ into arc $A$ within one scheduling period is $W_B(A) = N_B w(B) q(B, A)$ A straightforward way of allocating memory space is to map each buffer onto a distinct memory segment of length $W_B(A) + 2b_A(0)$. Considering top level buffers, this leads to a total amount of
\[ M_{D_{\text{max}}} = \sum_A (W_B(A) + b_A(0)) \]

memory space for all top level arcs A. Remember that every strong component C incident at an arc A is invoked once with the blocking factor \( N_b = q(C)N_s = N_q(B_w(A)) \). Due to the relationship between \( M_{D_{\text{max}}} \) and the blocking factor \( N_b \), increasing vector processing may result in increased memory consumption.

However, since not all signal buffers contain samples during the complete schedule period, one can reuse identical memory segments for different signal buffers.

---

**Figure 4: Memory allocation for an m-tap FIR system**

Considering the example of Fig. 4, an FIR filter consisting of m taps (each modeled as a primitive block), one can easily recognize that only four memory segments are needed to store data samples, since every signal buffer of size \( k = N_g \) is occupied during two block invocations.

In the sequel, we assume that all arcs are of compatible data type (such that they can
share memory) and that every memory segment \( W_B(A) \) consists of consecutive memory space.

Furthermore, every block has direct access to the input and output buffer memory. Then the task is to find the minimum amount of memory \( M_{D_{\text{min}}} \) given a schedule of the strong components. This entails finding a mapping \( \text{mem}(A) \) which allocates arc \( A \) to a distinct memory segment of minimum length \( W_B(A) + b_A(0) \) at a certain place in memory. For the mapping \( \text{mem()} \) the following condition has to be considered: all input arcs \( A_i \) and output arcs \( A_j \) of a block must be mapped onto distinct memory segments.

In general, finding a memory allocation \( \text{mem()} \) for multi rate systems with arbitrary number of delays on the top level such that all \( W_B(A) \) are consecutive is a complex optimization problem.

Next, we will consider heuristics applied in DESCARTES for single and multi rate systems. In case of single rate systems (systems in which the data rates at all ports of each individual block are identical, but possibly different from one block to another) lengths of all memory segments are equal to \( W = W_B(A) \) for any block \( B \) and any arc \( A \). Thus the minimum amount of memory space required is

\[
M_{D_{\text{min}}} = kW \quad \text{with } k \text{ an integer } \geq 1
\]

under the assumption that \( b_A(0) = 0 \) for all arcs \( A \). In this case the memory consists of at most \( k \) consecutively ordered segments of equal size. \( M_{D_{\text{min}}} \) may be found during the scheduling by keeping free segments in a list and allocating them to the output buffer of a scheduled component and afterwards inserting the memory segments of the input buffers in the list. In case of the m-tap example of Fig. 4 this approach saves \( 2(m - 1) \) buffers of size \( N_p \). In case of a multi rate system, \( W_B(A) \) may be different for each arc \( A \). This implies that it is not possible to find the minimum \( M_{D_{\text{min}}} \) during the scheduling. Since only already allocated memory segments can be taken into account, the memory space may result being fragmented at a certain scheduling step such that later realisation becomes impractical, even though enough deallocated memory may be available.

The approach we have taken in DESCARTES for single as well as multi rate systems is the simple heuristic that

- All delayed arcs are mapped onto distinct memory segments.
- During scheduling, a list of deallocated memory segments is maintained. Only the length of the memory segments is stored, not their positions in a linear memory space. The allocation of output buffer is performed in such a way that the best fitting deallocated segment is selected (the one with minimum difference of length). This memory segment is removed from the free list. If the output buffer does not fit into any deallocated memory segment, the largest deallocated memory segment is selected and enlarged such that the output buffer fits. Again, this memory segment is deleted from the free list. In case there are no deallocated memory segments, a new memory segment is created. After a block has been scheduled, the memory segments of the input buffers deallocated.

Thus after scheduling the number of required memory segments together with their length is known. For these memory segments code may be synthesized which define the real memory on the target DSP. Though this is not optimal, since unused memory space may remain whenever a bigger memory segment is reused than needed, this approach generates
very efficient implementations. In the case of single rate systems (or blocks with equal input and output rates) this approach leads to the minimum required memory space $M_{\text{min}}$.

Memory allocation at component level is performed following a different approach, since delays are always present at least in some arcs and blocks may be invoked several times depending on $N_p$ and $N_r(L)$. If $N_r(L) \geq 1$ and $N_p$ is an integer multiply of $N_r(L)$ then every block is invoked once thus enabling to apply the same allocation scheme as for top level arcs. Otherwise the high water (i.e. the maximum amount of data samples present at a certain scheduling step) of all signal buffers is monitored during scheduling, and the required buffer length is rounded up to the next common multiple of the arc’s input/output sample rate (remember that we do not assume circular buffers).

The same mapping heuristic is applied as for top level signal buffers but the resulting length of the shared memory segments being determined via the maximum of all high waters of the signal buffers sharing a segment makes the algorithm potentially suboptimal.

### 3.1.3 Memory allocation for computational kernels

Besides memory allocation for signal buffering, it is also necessary to efficiently map the computational kernels of all blocks onto memory. For this purpose, either the kernels are mapped onto a C-function, or expanded inline. In the first case the schedule consists of calls to the functions, whereas in the second case the computational kernels are replicated as often as they are activated in the schedule. The first solution saves memory space, since identical blocks of a block diagram can share the program code. Inline expansion increases throughput in general due to less function call overhead. Inline expansion is the default strategy if a block $B$ occurs unly once in the schedule list. Furthermore the user may individually configure the inline attributes of a block. The optimization of inline expansion may be automated by partially expanding small blocks (such as an adder) more often than bigger blocks. The degree of expansion depends on the goal of the synthesis. Minimizing memory consumption leads to a definition of the computational kernels as functions, whereas throughput optimization is achieved by an increased degree of inline expansion.

### 3.1.4 Signal forks

Memory space optimization may also be performed for fork blocks invoked once during a scheduling period. Forks copy input samples onto several output buffers. All input and output arcs of a fork block may be mapped to one memory segment, as long as there are no delayed input or output arcs. If there are delayed output/input arcs with identical initial data samples, the allocated memory segment is enlarged by the maximum number of delays, and the start position of every arc is initialized according to the number of delays. This saves memory space for buffers as well as program memory space and also increases throughput due to less copying operations.

### 3.2 Optimization strategies on the block level

In this section we will concentrate on one optimization strategy concerning loops, which play a decisive role for the SSDP paradigm. Loops are supported by means of a generic non-C loop construct. The generic loop consists of...
a. Fork on the block diagram  
b. Two realizations

Figure 5: Memory mapping for forks

```
LOOP (ezpr)
    loopbody
ENDLOOP
```

where ezpr may be a functional expression of block parameters, the blocking factor or any constant. The loopbody is any valid C expression, which may be replicated ezpr times. In figure 6 the generic loop is unfolded partially and the mapping on a C-loop construct best supported by the target DSP is done automatically. In the case of the DSP32C this

```
register int i = 3;
    do {
        accu += *in1++ *k;
        accu += *in1++ *k;
        accu += *in1++ *k;
        accu += *in1++ *k;
    } while (i-- > 0);
```

Figure 6: Partial loop unfolding, blocking factor \( N_b = 16 \)

is the do-while loop, whereas e.g. for the DSP96002 a `for()` construct is generated. Any generic LOOP may be totally or partially unfolded or just mapped onto an ordinary loop without replicating code depending on the quality of the C compiler. In case \( \text{ezpr} = 1 \) the LOOP construct is removed thus avoiding any overhead. Replicating the code of loopbody utilizes the DSP-pipeline much better than ordinary loops in case the target DSP (or the C-compiler) does not support zero overhead looping. However DSPs with zero overhead looping such as the TI and Motorola DSPs are well supported by this LOOP construct.
since for these loops C-constructs are generated which the C-compiler is able to map onto zero overhead loops. The replication of loopbody leads to increased throughput and also increased memory space for program code. This tradeoff also has to be kept in mind when zero overhead loops are used, since these loops require a constant iteration counter which in case of different blocking factors or parameters for blocks of the same model implies the replication of the model code or inline expansion.

DESCARTES can optimize every loop independently. This gives the user the opportunity to individually select loops for optimization. Thus again tradeoff between the goals of enhancing throughput and saving memory size is supported by this approach.

4 Towards goal directed software synthesis

DESCARTES offers several features to support an iterative search process which is necessary to find the optimal combination of optimization techniques for a specific application. At the beginning of the search process the designer configures a search routine by choosing the most promising optimization techniques. She or he manually sets optimization flags and parameters, such as vector processing with global blocking factors $N_p = 1, 2, 4, 8$ and 16. Then DESCARTES automatically generates and tests all possible solutions which can be produced from the combination of the chosen optimizations. After the measurement of the implementation parameters, a graphical representation of the solution space serves as user feedback. Additionally, several analysis functions can be applied to the solution data, such as the automatic extraction of the tradeoff points which build the envelope of the search space. The tradeoff points describe implementations with the best possible properties for a given implementation parameter e.g. those with a maximal throughput for a given code size (see figure 7). If the achieved results do not meet the specification, she or he initiates new search processes by choosing different optimization techniques.

![Figure 7: example of a search space (25 taps FIR filter)](image)

As an example, an exhaustive search process on a 25 taps FIR filter composed of single tap blocks was carried on a SUN workstation. The synthesized code ran on an AT&T DSP32C signal processor (VME-bus based DSP board) for the measurement of the throughput. The search space for the implementation parameters throughput and overall...
memory consumption (test plus data code size) is shown in figure 7. From the 168 generated solutions 11% contributed to the envelope of the search space (trade-off points), which shows the relatively poor efficiency of this exhaustive search. The time analysis of this synthesis process reveals that 44% of the search time was spent by DESCARTES to create the C-Code, 49% was needed to compile the DSP-Code and 7% to measure the implementation parameters. This distribution shows that estimating the implementation parameters in advance can save a considerable amount of time within the synthesis process. This approach is discussed in the following section.

Automatic synthesis The previously described interactive synthesis already incorporates simple automatic search routines, such as exhaustive and guided exhaustive search. However, the user still has to guess which optimization techniques suit best to his or her application. Furthermore, these simple search routines waste much time in generating solutions that do not contribute to the tradeoff since they are worse than previously created solutions. Consequently there is a need of making the synthesis more efficient by introducing an implementation model which allows the estimation of the implementation parameters and the automatic selection of optimizations strategies.

By detailed analysis of the effects of the available optimizations, it is possible to model the behavior of the implementation parameters. The implementation model incorporates equations representing time and memory consumption according to the chosen optimizations, such as scalable inlining [22]. The model parameter values used in these equations are gained from previously extracted measurements of single blocks and stored together with the block library. The implementation model superimposes the data of single blocks according to the system configuration given by a block diagram and provides reliable data on the application specific properties of a system. Since the model parameters are determined from empiric measurements, there is no need for an underlying hardware model [23].

The estimation data are made available to the system designer and therefore supports the interactive search process by shortening the iteration cycle.

Therefore automatic selection of optimization techniques with regard to implementation goals becomes practicable. The available estimation data also allows the direct comparison of implementation techniques at different system levels. Automatic selection strategies for several optimization techniques are now being developed, e.g. for scalable inlining and scalable vector processing.

5 Conclusion

Tool support for the design of digital signal processing systems is currently offered either for the simulation and analysis of algorithms (block diagram oriented simulation environments) or for the implementation of these systems (C-compilers, assemblers, source level debugger). DESCARTES offers a link from the one to the other of these two design levels.

Since it is crucial for the designers to steer the synthesis such that the resulting software is adapted to application-specific needs, DESCARTES offers a wide spectrum of optimization strategies. These strategies enable a tradeoff between different implementation goals. Since exploring the design space on the implementation level gets too time consuming when iteratively applying all kinds of optimizations, DESCARTES incorporates goal directed software synthesis which can be freely mixed with interactive optimization selections. Having started with an interactive and guided search process we are currently
investigating automatic search processes together with software synthesis techniques for control flow.

References


