Acknowledgments

This work was supported by the DFG under grant SFB 358, project *Automated System Design*.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. AT&T Bell Telephone Laboratories Inc., 1986.

[2] R. Camposano and R. Brayton. Partitioning before Logic Synthesis. In *Proc. of the International Conference on Computer-Aided Design*, pages 324–326, Santa Clara, CA, November 1987. IEEE.

[3] T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, London, 1990.

[4] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[5] H.-J. Eikerling. Repartitionierung regulärer, hierarchischer Datenpfade. In *40th Intern. Wiss. Kolloquium Ilmenau*, pages 67–72, Ilmenau, September 18-21 1995.

[6] H.-J. Eikerling, R. Hunstock, and R. Camposano. Optimization of Hierarchical Designs using Partitioning and Resynthesis. In *Proc. of the International Conference on Computer-Aided Design*, San Jose, CA, November 6-10 1994.

[7] H.-J. Eikerling, M. Schmidt, and W. Rosenstiel. A Fully Symbolic Framework for Area-minimizing Hardware Resynthesis. In *IFIP TC10, WG 10.5 Workshop on Logic and Architecture Synthesis*, pages 163–171, INPG, Grenoble, France, December 19-20 1995.

[8] Ch. Ewering. Automatic High-Level Synthesis of Partitioned Busses. In *Proc. of the International Conference on Computer-Aided Design*, pages 304–307, Santa Clara, CA, November 1990. IEEE.

[9] R. Genevriere. Flexible Synthesis of Hierarchy with PMOSS. In *40th Intern. Wiss. Kolloquium Ilmenau*, pages 61–66, Ilmenau, September 18-21 1995.

[10] R. K. Gupta and G. De Micheli. Constrained software generation for hardware/software systems. In *Third International Workshop on Hardware /Software Codesign*, pages 56–64, Grenoble, September 1994.

[11] R.K. Gupta and G. De Micheli. Partitioning of Functional Models of Synchronous Digital Systems. In *Proc. of the International Conference on Computer-Aided Design*, pages 216–219, Santa Clara, CA, 1990. IEEE.

[12] W. Hardt and R. Camposano. Specification Analysis for HW/SW Partitioning. Technical Report SFB - 358 - B2 - 5/94, University of Paderborn, Technical University of Dresden, 1994.

[13] W. Hardt, A. Günther, and R. Camposano. Pipelined Interface for HW/SW Codesign. Technical Report SFB - 358 - B2 - 3/94, University of Paderborn, Technical University of Dresden, 1994.

[14] W. Hardt and W. Rosenstiel. Speed-Up Estimation for HW/SW-Systems. In *CODES/CASHE*, Pittsburgh, PA, March 1996. To appear.

[15] A. Hoffmann, H.-J. Eikerling, W. Hardt, and R. Genevriere. PSF - Paderborn Synthesis Format. Technical Report SFB - 358 - B2 - 6/94, University of Paderborn, Technical University of Dresden, 1994.

[16] W. Rosenstiel. Optimizations in High-Level Synthesis. *Microprogramming and Microprocessing*, 18:543–549, 1986.

[17] Synopsys, Inc., Mountain View, CA. *VHDL Design Analyzer (tm) Manual*, 3.3a edition, 1995.

[18] Vlissides, J.M. and Linton, M.A. Unidraw: A Framework for Building Domain-Specific Graphical Editors. In *Proc. of the ACM SIGGRAPH/ SIGCHI User Interface Software and Technologies*, 1989.

[19] R.A. Walker and D.E. Thomas. Behavioral Transformation for Algorithmic Level IC Design. *IEEE Transactions on Computer-Aided Design*, 8(10):1115–1127, October 1989.

[20] M. Wendling and W. Rosenstiel. A Hardware Environment for Prototyping and Partitioning Based on Multiple FPGAs. In *Proc. of the European Design Automation Conference*, pages 77–82, Grenoble, France, September 1994. IEEE.

[21] W. Wolf. Hardware-Software Codesign of Embedded Systems. *Proceedings of the IEEE*, 83(7):967–989, 1994.

[22] Zycad Corporation, Inc., USA. *Concept Silicon Software (tm) Manual*, 6.0 edition, 1994.

	#reg	#mux	#inp. (mux)	#states (FSM)	#trans. (FSM)
SLS	7	10	22	19	45
FDS	7	6	14	14	35

Table 5. PMOSS structural synthesis results for concerning KMP algorithm. The number of outputs of the FSM can also be reduced from 27 to 22 by the FDS. The number of FSM inputs is 8 in both cases.

	+	-	==	<	<=	>	>=	||	&&
SLS	1	1	1	1	1	1	1	2	1
FDS	2	2	1	1	2	1	1	2	1

Table 6. Comparison of allocated resources.

After structural synthesis, an appropriate gate-level structure has to be found. While the controller can be synthesized quickly the datapath might become too large so that a partitioning step has to be carried out. As pointed out above, our approach tries to minimize the number of tasks for logic synthesis (i.e. the number of blocks) while achieving a close to optimum realization. This is done by a 2-phase algorithm. In the first phase global partitioning of the description is done; the second phase tries to minimize the number k of tasks by balancing the blocks. The results for an optimization concerning cost (area) and power consumption while achieving minimum synthesis times are shown in table 8.

	k	T_{syn} not part. vs. part.		T_{part} glob. vs. impr.	
KMP4	2	174.3	125.4	0.28	1.0
KMP8	3	274.1	170.1	0.29	4.2
KMP16	6	1605.4	379.3	0.29	13.6
KMP32	11	-	588.7	0.29	45.4

Table 7. Results of logic partitioning. The synthesis time can be significantly reduced by partitioning the initial design.

The final results of the partitioning are shown in table 8 and table 8. The different rows refer to implementations which use different bitwidths for the registers in the HW implementation. These bitwidths can be manipulated during the behavioral synthesis via attributes.

	Power not part. vs part.		*Area* not part. vs part.	
KMP4	5985.0	5998.3	2056.0	2059.0
KMP8	11712.5	11367.2	3933.0	3925.0
KMP16	21977.1	22012.3	7554.0	7584.0
KMP32	-	38976.4	-	15643.0

Table 8. Results of logic synthesis. The synthesis achieves close to optimal synthesis results. The data is given with respect to the mcnc.genlib.

The area for the controller which does not depend on the bitwidth in the datapath is 266.0. The synthesized HW can be mapped to different target architectures. For instance, an ASIC or a programmable device (FPGA) may be used. The synthesized result is passed to commercial backends [17] for technology dependent optimization. Moreover, verification by emulation is facilitated [14, 22].

5 Summary and Conclusions

We have presented a methodology for rapid analysis and implementation of hardware/software co-systems. Basically, the synthesis and optimization process is understood as a partitioning process which is applied at different levels of abstraction and aims at different optimization criteria (area, power consumption, throughput, overall performance). Using the environment the HW part including the interface can be gained automatically from a system specification rather rapidly. Because a central unique database controls all design tasks on all levels of abstraction the design flow becomes modular, i.e. algorithms for designs tasks are exchangeable. Thus, a variety of design goals can be explored in combination.

In our approach, the special function hardware is entirely implemented as an ASIC/FPGA which is a limitation since for some parts domain specific hardware (e.g. micro-controllers) could be used. Therefore, future work will focus on embedding these commonly used structures such that they can be optimally utilized. Moreover, the compilation of standard C library components requires the handling of memory management constructs. At present, only static memory can be handled by the synthesis system. Dynamic memory access and update requires capabilities to treat generic dynamic memory management units which is a vivid need.

This will enable enlarged interactive exploration facilities and give an emphasis to the transformational approach which we believe is the major strength of our system.

design space exploration is unfeasible. The application of our partitioning heuristic results in a very small set of partitions and generates a good partitioning as the final system implementation proofs. At system level, the codesign task finds one suitable module for HW implementation and the design space was reduced reasonably. As mentioned above static analysis ($Stat$) provides the weakest partitioning criteria. By compilation and execution dynamic analysis (Dyn) data is obtained. Parameter or interface analysis ($Intf$) determines the interface costs which was formerly pointed out as a main problem for a HW implementation.

characteristic	
lines of C source code	12.902
number of modules	173
lines of assembler source code	35.081
size of executable in byte	204.800

Table 1. Characteristics of "grep".

Table 2 gives the number of modules of the grep command and the number of modules qualified for HW implementation by each analysis phase ($Stat$, Dyn, $Intf$). The HW/SW partitioning phase evaluates a cost vector Ψ and qualifies exactly one module for HW implementation. The entire partitioning process takes about 19 cpu seconds (static analysis 13.44s, dynamic analysis 2.87s, parameter analysis 0.40s, module generation 0.50s) on a Sun SS20 workstation.

UUC	$Stat$	Dyn	$Intf$	Ψ
grep	122	29	5	1

Table 2. Specification analysis results.

The selected module realizes the central algorithm of this application, i.e. the pattern matching algorithm. Several algorithms solving the same problem are known from literature [3]. We chose the variant by *Knuth*, *Morris* & *Pratt* (KMP) for HW implementation. Some of the other algorithms for implementing the pattern matching step work faster in practice (Boyer-Moore, Karp-Rabin). However, these algorithms backup information on the string which has already been examined in a hash table. The KMP algorithm has the advantage to achieve worst-case runtime $O(|Pattern| \cdot |String|)$ for all instances and keeps storage requirements low. Therefore, the KMP is an ideal candidate for a low cost hardware implementation. On the other hand the algorithm prefers repetitive patterns which are rather seldom to occur.

Table 3 gives some characteristics of the pattern searching algorithm which has been selected for HW implementation by the system-level synthesis task. It turns out, that the description of the algorithm is comparable small, so that a compact implementation can be expected. Now, this description is fed into structural synthesis. The behavioral synthesis results are depicted in table 3 which are given by the number of registers and the number of muxes and their input count in the datapath on one hand and by the number of states and transitions of the controller on the hand. In the algorithm an array *next* is used. If the string at position *k* does not match the pattern at position *i* the pattern can be shifted by *next(i)* positions to the right. The entries of this array can be computed in a pre-processing step. Besides the registers for storing the values of the variables which are needed to implement the algorithm a register is assigned to the computation of the *next* array.

Since the scheduling algorithm has the highest influence on the final HW design characteristics, we have compared the results obtained from applying the static list scheduler (SLS) and the force-directed scheduler (FDS) to the basic-blocks. In this particular example, fairly better results with respect to the schedule length can be obtained by using the FDS. However, as table 6 indicates, more functional units are being allocated by the FDS. If the cost of implementation is the main objective we may decide to use the result from the static list scheduling procedure for further treatment.

	#lines (C-code)	#nodes (DFG)	#nodes (CFG)
KMP	53	122	22

Table 3. Characteristics of Knuth, Morris & Pratt string matching (KMP) algorithm. In terms of the size of the internal representation.

As can be seen from table 3 the expense for behavioral synthesis (scheduling, register, functional unit and interconnection binding) keep in very reasonable bounds, so that architectural trade-offs can be studied intensively.

	$T_{Schedule}$	T_{Bind}	$T_{Intercon}$
SLS	0.23s	0.03s	0.05s
FDS	0.36s	0.03s	0.04s

Table 4. PMOSS structural synthesis results concerning KMP algorithm. The table shows the runtimes of behavioral synthesis (measured with rusage).

account similarities of the basic building blocks in the RT-level description (e.g. adder/subtracters) and the presence of hierarchical operators (e.g., an n-bit adder is built out of a set 1-bit adders at the next lower hierarchical level). The result of the partitioning process [5] is a set of blocks which are balanced concerning size. This enables us to apply more powerful scripts to the extracted blocks and to get a circuit which is close to optimum with respect to various cost measures (area, power consumption, pin count of blocks).

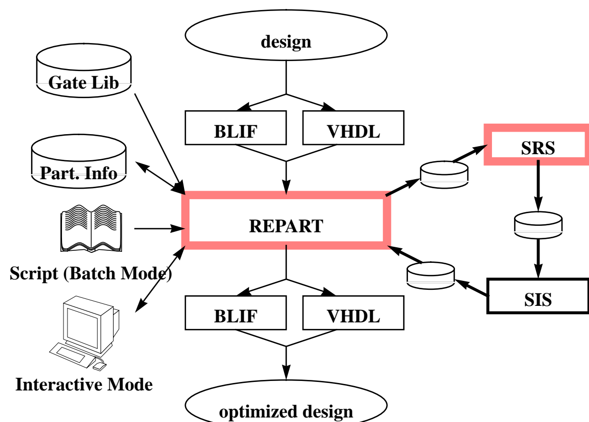

Figure 6. Redesign couples partitioning task with sequential resynthesis.

Our approach exploits the hierarchy explicitly by building a hierarchical graph reflecting the similarity between connected entities. A *connectivity graph* is being build in which all weights represent the savings for each particular cost measure under consideration. The weights normally have to be pre-computed for all cost measures, for all possible connection types and all bitwidths. This information is stored in a data base as shown in figure 6. At the first glance, it seems to be that this data base may become very large. However, one has to recall that during high-level synthesis only very regular structures are being produced. For instance, a mux is always inserted at the input of a register and a mux is also placed at the inputs of each functional unit (mux-reg-mux-fu structure). This leads to a significant sparsification of the partitioning data base. Based on the edge weights, the area estimates for the size of the entities and an upper bound for the block size a partitioning of the datapath can be gained. This is done by a two phase approach.

(1) In the first stage a partitioning at the top-level hierarchy is being computed. This step is called *global prepartitioning*.

(2) Since at the top-level the sizes of instantiated instances can vary significantly which may result in rather unbalanced designs, some further optimization has to take place. In the next stage the optimization of these parts is done by using an iterative improvement approach. This step is called *local improvement step.*

The number of partitions is equivalent to the number of blocks which have to be synthesized. We provide information about the desired block size to the partitioner. The estimated block count can be computed by dividing the estimated size of the datapath by this value. Now, the global partitioning is the result of computing the solution of a multi-way partitioning problem without considering the block size constraint. In the second step iterative improvement is performed to satisfy the block size constraint. As experimental results show, this allows for the rapid organization of datapaths while achieving nearly optimal results.

3 Implementation

An overview of the system is given in figure 4. The entire system has been implemented in C++ on top of a class library [15]. Besides various file formats and interfaces, PMOSS supports a visual inspection of the structural synthesis task. There exists a powerful editor which supplies a control of the CDFG's interdependence. Additionally, there exist another editor to analyze the synthesized finite state machine and its state transition graph. The editor's realization is based on the class libraries *Interviews* and *Unidraw* [18] developed at Stanford University. Interviews is a library of C++-classes to support for creating X-Window applications. Unidraw is some kind of toolbox for building domain specific editors based on Interviews. It provides complex methods, e.g. for creating and moving of objects. One problem in using the Unidraw classes is the implicit data handling concept. Therefore we had to add some concepts to attach the extended data structures of the CDFG.

In order to uncouple front-end, synthesis and back-end activities, an external format for data exchange between the toolset has been defined. For instance, the CDFG can be written to a file and then being restored by the editor for manual intervention. This can be done at all stages of the synthesis process.

4 Case Study

Our approach was applied to a set of benchmarks of reasonable complexity [11, 12]. For example, the HW/SW partitioning of the well known and frequently used UNIX command *grep* was examined in more detail. In table 1 some characteristics of the grep command which searches for patterns in text strings are listed. In general, the HW/SW partitioning has to consider each possible bipartitioning of the design. This is restricted by the partitioning granularity, e.g. a module is an atomar unit. Thus, the number of possible partitions is very large and exhaustive

PMOSS allows the user to control the structural synthesis process. A status vector contains the information about the stage of synthesis. Each vector component reflects a group of algorithms, i.e. all scheduling algorithms. After performing any available scheduling algorithm onto the CDFG, the status vector component for scheduling is set automatically. This approach gains in the flexible use of different structural synthesis algorithms of one group.

During the structural synthesis the CDFG is annotated with the structural informations required for later conversion (control-step, allocated module type, instance of the module type) to the CDP. The annotation scheme is one of the key ideas for getting a bunch of algorithms which are exchangeable. The usage of an algorithm is just restricted concerning its precondition.

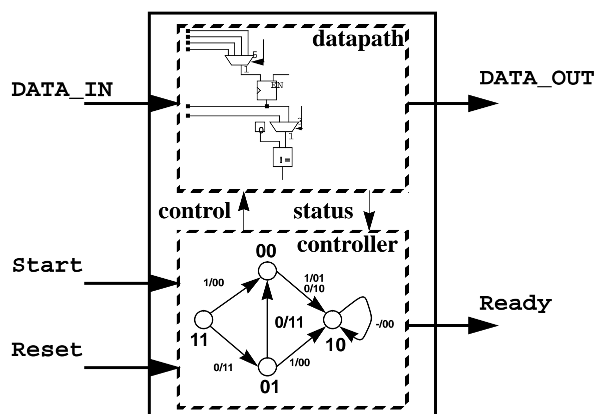

Figure 5. The target microarchitecture of PMOSS consists of a controller and a mux-based datapath.

2.3.3 Function Calls and Memory Access

The synthesis out of high-level description requires a methodology to handle function calls. In figure 5 the handshake mechanism to handle functions calls is shown [9]. Since hardware cannot be shared among concurrent processes, so that the range for further optimizations is reduced, we have examined the sharing of resources at the behavioral level. That is, functions share all the resources in the datapath and the function hierarchy (function calls) is entirely mapped to the controller, i.e. if a function call appears the controller implementing the calling function is suspended and the controller implementing the called is being activated[1]. This protocol can be implemented fairly easily using the handshake mechanism.

One additional thing which seems to be marginal at first glance needs to be addressed. Since the synthesis starts from a high-level description (for instance a piece of C code) the interface of the top-level function does not directly correspond to the entity which has to be synthesized. This is mainly due to the fact that at the interface of the high-level function specification a certain protocol for passing arguments to that particular function may be defined. For instance, one can pass a pointer describing a character string to the function. This kind of description enforces the data to reside in an external memory (belonging to a connected entity).

Therefore, additional control and status lines have to be added to the HW entity which defines a certain protocol for external memory access and update. For convenience and ease of use this is done automatically.

2.4 Structure Optimization

The output of the behavioral synthesis is a structural description at the register-transfer-level which describes a *deterministic, synchronous system* and needs to be transformed into a gate-level representation. The result of the behavioral synthesis is shown in figure 5. The structure has to be passed to logic synthesis in order to obtain an optimized implementation.

Mainly three tasks can be distinguished.

(1) The symbolic state-table describing the controller has to be synthesized. This can be done by using state-of-the-art sequential optimization techniques.

(2) The datapath has to be optimized. This is normally done by applying Boolean logic synthesis to the combinational logic in the datapath.

(3) Moreover, combining the controller and datapath portion may gain additional don‘t care conditions which can be used for optimization along the controller/datapath boundary. We have proposed a method which optimizes the interface by iteratively swapping nodes over the boundary and doing a partial resynthesis of the considered regions [6].

(3) is a typical redesign activity. The idea is to treat datapath portions by means of sequential logic synthesis which is generally more powerful than Boolean logic synthesis since we have the addition and removal of sequential redundancy at our disposal. However, the optimization of the controller is limited to state machines with a few hundred states only. By casting the methods known from controller synthesis into a BDD-based framework [7], this extends (1) to classes of circuits which also involve datapath components. Therefore (1) can also be regarded to be a redesign task.

For the rapid synthesis of digital designs, the datapath needs to be synthesized quickly. Since the datapath can become too large to be synthesized as one block we provided partitioning algorithms [2, 11] which take into

1. We do not allow for recursive calls.

The foundation for the efficient application of the transformation is the modular organization of the behavioral synthesis task.

2.3 Behavioral Synthesis

Behavioral or structural synthesis is the task of compiling a behavioral hardware specification into a register-transfer level (RTL) structure given a set of resources (adders, multipliers, ALUs etc.).

2.3.1 Target Architecture

The interconnections between the RT components can be either realized by busses or by multiplexers [4]. The choice of the target architecture may have a significant impact on the subsequent optimization steps. The difference stems from two sources:

(1) In a bus architecture the RT components are formed by standard cells, i.e. a set of pre-optimized components. The advantage here is that geometrical aspects (layout) can be involved at early stages of the synthesis process [8] and tight area estimates can be gained quickly. However, it turned out that this type of architecture is somewhat difficult to test.

(2) In contrast to this, mux based implementations are easy to test and test concepts can be integrated either on the behavioral or on the logic level. Since during the structural synthesis abstract measures are used for cost and performance estimates, one has to use module generators to build the actual instances in the netlist which is then passed to logic synthesis. The need for a subsequent optimization step may serve as disadvantage (since the synthesis time increases) or advantage (since we have the powerful methodology of logic synthesis at our disposal), as well. Another advantage is the gain in performance when compared to bus based architectural style.

Because of the reasons pointed out above, PMOSS supports the mux style architecture.

2.3.2 Modular Synthesis

One major objective in developing the modular fashion of the synthesis engine was to separate the synthesis tasks in a transformational synthesis environment. This allows flexibility to combine and to compare different algorithms. Using this modular fashion the user is able to pick the appropriate algorithms to satisfy his needs or constraints. Furthermore, if the impact of two behavioral transformations on the datapath needs to be analyzed, no creation of the controller is necessary. This reduces the time consumption for the analysis of the expected results enormously.

In the central data structure (PSF), there exist two different levels of abstraction, the behavioral level and the structural level. The description of the behavior results from an extended control-data-flow-graph (CDFG) [1], the structural information is given by a controller and a datapath (CDP), see figure 4. The data-flow is represented as a set of basic blocks. The control-flow gives the connections among these blocks. This enables a description of the overall behavior of the specification as a CDFG.

In the current state of the implementation, the default synthesis flow calculates the final schedule of each operation by using either the static list-scheduling or the force-directed scheduling algorithm which have been extended to handle control constructs and hierarchy as well as pure data-flow. Both algorithms encompass the functional unit allocation which is done simultaneously. The task of allocation is done straightforward by selecting the first module type within a library which is able to perform the operation. The binder picks the first free instance of this allocated type of functional unit. Register allocation and binding is performed by the left-edge algorithm after life-time analysis. Interconnection binding is also done straight-forward by a simple traversal of the DFG.

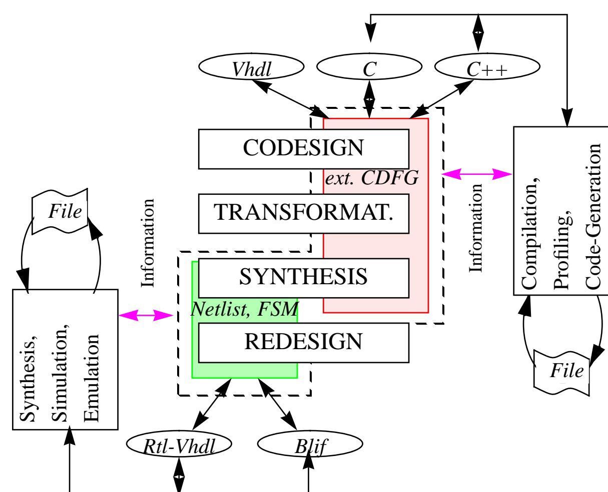

Figure 4. PMOSS - modular synthesis and codesign environment.

A design may consist out of multiple processes. However, these processes will be treated separately by behavioral synthesis. Therefore, we concentrate on how to synthesize one single process (thread). The entire synthesis-task is basic-block oriented, i.e., in the central task of the synthesis procedure, scheduling, the basic blocks are being scheduled first. Afterwards, the control-flow is being considered. However, optimizations over basic-block boundaries can be achieved by applying transformations on the particular part of the specification beforehand.

specific units low, the interface to the system bus is being pipelined.

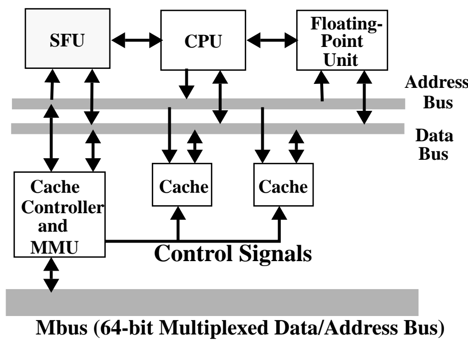

Figure 1. Target system architecture.

The codesign task maps one part of a given specification to the general purpose processor and the other to the SFU. The actual partitioning aims at a *maximum speed-up* of the overall system. Since normally by a dedicated hardware implementation a speed-up can be always expected, we also have to obey cost constraints.

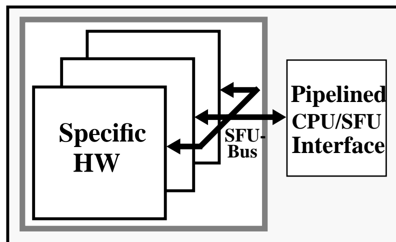

Figure 2. Special function unit (SFU).

The specification is assumed to be given as a set of executable code, e.g. as a C program. The codesign task itself is divided into several subtasks.

(1) *Specification analysis* provides all information needed to partition the specification into a HW and a SW part. First, we have to derive a discrete model, on which the partitioning can be done. Therefore, the input specification is partitioned into a set of modules representing pieces of behavioral code upon which the actual partitioning is being carried out heuristically. This model is called *module graph.* The decision on where to move a particular module is based on *statically analyzable criteria* (control dominance) and *dynamically analyzable criteria* (runtime and memory access patterns on a set of representative test data). The output of this phase is a real-weighted vector which assigns each module information about the static and dynamic criteria.

(2) The *partitioning* subtask results into two parts of the specification, i.e. a set of modules which have to be implemented in HW and a second part which has to be implemented in SW. This is done by evaluating the weighting vector for each module. By combining the components of the weighting vector a *fitness* for hardware implementation can be defined. The hardware modules can now be allocated in a Greedy fashion: hardware modules are allocated unless the cost threshold is reached.

The specification analysis is outlined in figure 3.

The HW part is the input of the next synthesis pipeline stage, high-level and transformational HW synthesis. The HW part communicates with the SW part via a pipelined interface [13].

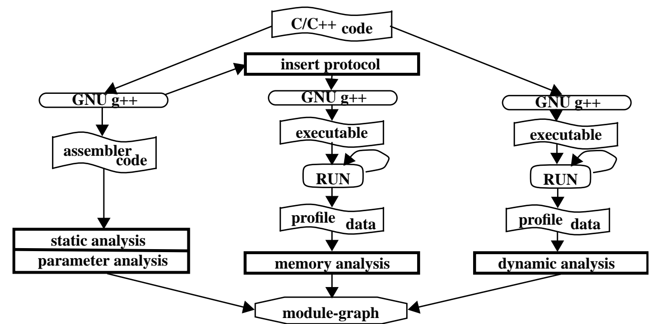

Figure 3. Specification analysis during codesign.

2.2 Transformational Synthesis

Transformational synthesis is concerned with the application of compiler transformations (e.g., loop unrolling, constant propagation, common subexpression elimination) to the intermediate representation of the behavioral description which has to be synthesized. The impact of these transformations might be twofold:

(1) First of all, transformations are known to achieve a significant *acceleration* of the subsequent synthesis steps [16].

(2) Second, the *result* with respect to cost measures (area, performance, power consumption) of the synthesis steps can be influenced [19].

One major difficulty is that these transformations are applied in a off-line fashion before the actual synthesis is being carried out. The problem with this procedure is that the impact of a particular transformation on the final result is hard to predict. Therefore, we propose a tight integration of the transformations and the synthesis activities, in which specific knowledge on the applied method in that particular synthesis step (i.e. basic block oriented scheduling) is used for the selection of transformations to be applied. By doing so, the optimization potential can be increased on one hand and on the other hand the change of the intermediate representation has only local effect and estimates of design characteristics need to be re-evaluated only locally.

A Methodology for Rapid Analysis and Optimization of Embedded Systems

H.-J. Eikerling†, W. Hardt††, J. Gerlach†, W. Rosenstiel†

†Universität Tübingen,
Technische Informatik
Sand 13,
D-72 076 Tübingen, Germany

††Universität Paderborn,
FB 17 Mathematik-Informatik
Fürstenallee 11,
D-33 102 Paderborn, Germany

Abstract

This paper describes an environment for the rapid analysis, synthesis and optimization of embedded systems. Since the implementation of these systems is rather complicated, we propose a methodology which automates the entire design flow. Flexibility is achieved by allowing manual intervention which is realized via a modular implementation of algorithms which are being provided. The applicability of the proposed approach is shown in terms of an example (UNIX command grep).

1 Introduction

As current trends in the engineering of computer based systems indicate, there is a strong need for the rapid evaluation and assembly of such systems mainly caused by economical requirements (time-to-market). To overcome the problem of designing the hardware components and the software separately, the catchword *hardware-software codesign* (e.g. [21]) has been established which includes rapid organization of hardware and software modules. Since the tasks of hardware design, i.e. defining an architecture which is close to optimal in the sense of cost/performance constraints for a specific problem domain, and software design, i.e. writing efficient code for a given architecture, are highly complicated when considered separately, the situation becomes even more difficult if we encounter the problem of optimizing hardware and software simultaneously.

1.1 Previous Work

A straightforward approach which turned out to be successful in hardware design is to divide the entire process into a set of tasks [4, 10, 20] which in turn are divided into several even more dedicated smaller tasks. Since normally, each task heavily interacts with previous and successive tasks, a main focus has to be spent on the transitions between the different tasks. At some points human interaction may be required.

1.2 Approach

In our approach, we address this problem by providing modularity. First of all, one has to appropriately choose languages for information exchange. We tackle this problem by proposing a variety of formats (C, C++, behavioral VHDL, structural VHDL, BLIF[1]) for optimized data exchange between the recognized tasks. Second, we use modularity to select the best among a set of problem-specific algorithms and therefore ensure optimized succession of subsequent tasks. In all steps of the design flow we also provide a set of options which can be set by the user to achieve the desired result.

The paper is organized as follows: in section 2 we give a sketch of the concepts underlying the different phases of the proposed approach. Section 3 explains the implementation of the concepts which have been accumulated in a toolset referred to as *PMOSS*. The application of the methodology is explained in terms of an example which is presented in section 4. Finally, we give a short conclusion and some ideas on how to further use the entire methodology.

2 Concepts

2.1 System-level Synthesis

Considering the first stage of the synthesis pipeline, system synthesis, HW/SW codesign comes into view. Regarding complex systems which are typically implemented in hardware and software are described by abstract (technology independent) system specifications. Classical approaches define a fixed HW/SW interface, e.g. the processor instruction set and develop the HW part and the SW separately. Performing codesign both parts are considered together without a predefined interface. However, every design process will focus on a target architecture. On one hand there are state of the art general purpose computers which are very powerful. On the other hand special function designs which are much more efficient for special applications can be found. In our codesign approach we intend to join both aspects together.

Therefore, we define a target architecture based on a general purpose processor extended by an additional special function unit (SFU, see figure 1 and figure 2). The interface is customized to the SPARC architecture. In order to keep latency which is implied by the data transfer to HW

1. Berkeley Logic Interchange Format