communication cost optimization for fsm net design. In *Proc. of the Twentieth EUROMICRO conference*, Liverpool, UK, September 1994. IEEE, IEE.

[Ma93] H.-G. Martin. Retiming by combination of relocation and clock delay adjustment. Technical Report SFB - 358 - B1 - 1/93, FhG IIS/EAS Dresden, Germany, July 1993.

[Cath91] S. Note, W. Geurts, F. Catthoor, and H. De Man. Cathedral-iii: Architecture-driven high-level synthesis for high throughput dsp appli cations. In *Proc. of the 28th Design Automation Conference*, pages 597–602. IEEE, 1991.

[Putk91] A. Puttkammer. Entwicklung, implementierung und bewertung von methoden zur modularisierung von steuerwerken. Technical Report Diplomarbeit, Universit"at Passau, Fachbereich Mathematik und Informatik, 1991.

[RaKu92] D.S. Rao and F.J. Kurdahi. Partitioning by regularity extraction. In *Proc. of the 29th Design Automation Conference*, pages 235–238. IEEE, 1992.

[RuGa90] E.A. Rundensteiner and D.D. Gajski. A design representation model for high-level-synthesis. Technical report, University of California, Irvine, 1990.

[SeSiLaMoMu92] E.M. Sentovich, K.J. Singh, L. Lavagno, et al. Sis: A system for sequential circuit synthesis. Technical Report Memorandum No. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California at Berkeley, May 1992.

[GnuCC] R.M. Stallmann. *Using and Porting GNU CC*. Free Software Foundation, inc., December 1992.

[SynDC92] Synopsys, inc. *Design Compiler (tm) Reference Manual*, 3.0 edition, December 1992.

[Tho88] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn. The system architect's workbench. In *Proc. of the 25th Design Automation Conference*, pages 337 – 343. ACM/IEEE, June 1988.

[Trick87] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, 6(2):259–269, March 1987.

[IV91] Stanford University. Interviews reference manual version 3.0. Technical report, University of California, Palo Alto, USA, 1991.

[Van93] J. Vanhoff, K.V. Rompaey, I. Bolsens, G. Goosens, and H. DeMan. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer Academic Publishers, 1993.

[Uni89] J.M. Vlissides and M.A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proc. of the User Interface Software and Technologies*. ACM SIGGRAPH/SIGCHI, 1989.

[WaGo85] W.M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, 1985.

[WaTh89] R.A. Walker and D.E. Thomas. Behavioural transformations for algorithmic level ic design. In *IEEE Transactions on Computer Aided Design*, 1989.

und autonomen automaten. Technical Report SFB - 358 - B1 - 5/93, FhG, IIS/EAS Dresden, November 1993.

[FiMa82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristics for improving network partitions. In *Proc. of the 19th Design Automation Conference*, pages 175–181, Miami, FL, 1982. ACM/IEEE.

[GaDuLiWu92] D.D. Gaijski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston/Dordrecht/London,, 1992.

[GhDe92] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Proc. of the 29th ACM/IEEE Design Automation Conference*, pages 153–159. ACM/IEEE, 1992.

[GrMu89] G. Grass and M. Mutz. Modular implementation of finite state machines observing topological constraints. In *Proc. of the IFIP Congress, Computer Hardware Description Languages and their Applications*, pages 183–196, 1989.

[GuMi92] R.K. Gupta and G. DeMicheli. System synthesis via hardware - software co - design. In *CSL TR-93-548*, 1992.

[HaCa93] W. Hardt and R. Camposano. Trade-offs in hw/sw codesign. In *Proc. of the ACM Workshop on Hardware/Software Codesign*, Cambridge, Massachusetts, October 7 - 8 1993. also available as technical report no. SFB 358 - B2 - 3/93 at TU Dresden.

[HaCa94] W. Hardt and R. Camposano. Specification analysis for hw/sw-codesign. submitted to CODES 94, April 1994.

[Ho93] A. Hoffmann, H.-J. Eikerling, R. Genevriere, W. Hardt, and R. Camposano. Isf - eine entwurfsdarstellung f"ur die high-level-synthese. In *Workshop des Sonderforschungsbereiches 358 der TU Dresden und ITG/GI*, Dresden, September 30, October 1 1993. also available as technical report SFB - 358 - B2 - 1/93.

[JoKo91] L. Jozwiak and J.C. Kolsteren. An efficient method for the sequential general decomposition of sequential machines. *Microprocessing and Microprogramming, North Holland*, 32:657–664, 1991.

[KMR72] R.M. Karp, R.E. Miller, and A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees, and arrays. In *Proc. of the 4th Annual Symp. on Theory of Computing*, pages 125–136, 1972.

[KeLi70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[KoFeFr93] M. Koegst, K. Feske, and G. Franke. Iterative fsm structuring by partial state assignment. In *Proc. of Workshop: Design Methodologies for Microelectronics and Signal Processing*, pages 127–134, Cracow, Poland, Oct. 1993. Selesian Technical University Gliwice, Poland.

[KoFeF93] M. Koegst, K. Feske, and G Franke. Zustandskodierung in einem netz linear partitionierter automaten. Technical Report SFB - 358 - B1 - 6/93, FhG, IIS/EAS Dresden, December 1993.

[KoGFF94] M. Koegst, W. Grass, G. Franke, and K. Feske. Simultaneous state encoding and

10 Bibliography

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.*. Addison Wesley, 1986.

[AmBa89] R. Amann and U.G. Baitinger. Optimal state chains and state codes in finite state machines. *IEEE Transactions on Computer-Aided Design*, 8(2):153–170, Feb. 1989.

[AsDeNe92] P. Ashar, S. Devadas, and A.R. Newton. *Sequential Logic Synthesis*. Kluwer Academic Publishers, Boston/Dordrecht/London,, 1992.

[BaRu93] G. Balster and St. R"ulke. Retiming durch gezielte modifikation des fanout der logik. Technical Report SFB - 358 - B1 - 3/93, FhG IIS/EAS Dresden, Germany, September 1993.

[BaBr93] S. Baranov and L. Bregman. Automata decomposition and synthesis with plam. *Microprocessing and Microprogramming, North Holland*, 38:759–766, 1993.

[CaBr87] R. Camposano and R. Brayton. Partitioning before logic synthesis. In *Proc. of the ICCAD*, pages 324–326, Santa Clara, CA, 1987. ACM/IEEE.

[CaTa89] R. Camposano and R.M. Tabet. Design representation for the synthesis of behavioral vhdl models. In J.A. Darringer and F.J. Rammig, editors, *CHDL'89*. Elsevier Science Publishers, 1989.

[CaVE87] R. Camposano and J.T.J. van Eijndhoven. Combined synthesis of control logic and datapath. In *Proc. of the ICCAD*, pages 327–329, Santa Clara, CA, 1987. ACM/IEEE.

[Cha92] A.C. Chandrakasan, M. Potkonjak, J. Rabaey, and R.W. Broderson. Hyper-lp: A system for power minimization using architectural transformations. In *Proc. of ICCAD 1992*, 1992.

[DeMi88] G. DeMicheli and D. Ku. Hercules - system for high-level-synthesis. In *Proc. of 25th DAC 1988*, 1988.

[DeMi90] G. DeMicheli, D. Ku, F. Mailhot, and T. Truong. The olympus system for digital design. *IEEE Design and Test Magazine*, 1990.

[LaTh89] E. Dirkes Lagnese and D.E. Thomas. Architectural partitioning for system level design. In *Proc. of the 26th Design Automation Conference*, pages 62–67. IEEE, 1989.

[Du93] G. Dueck. New optimization heuristics: The great deluge algorithm and the record-to-record-travel. *Journal of Computational Physics*, 104(1):86–92, 1993. also available as technical report TR 89.06.11, IBM Germany, Heidelberg Scientific Center.

[EiCa93] H.J. Eikerling and R. Camposano. Cp/dp-partitionierung und resynthese. In *Handouts 6. EIS Workshop Entwurf Integrierter Schaltungen*, 1993.

[ErHe92] R. Ernst and J. Henkel. Hardware-software co-design of embeded controllers based on hardware extraction. In *Proceedings of the ACM Workshop on Hardware-Software Co-design*, October 1992.

[FeFr93] K. Feske, G. Franke, M. Koegst, and J. Lattig. Iterative fsm structuring by partial state assignment. Technical report, Fraunhofer-Institut fuer Integrierte Schaltungen, Erlangen, Aussenstelle Dresden, Germany, 1993.

[FeMu93] K. Feske and S. Mulka. Fsm-partitionierung und -synthese auf der grundlage von plas

9 Conclusion and Future Work

We have presented an environment for the automatic synthesis of digital systems. The synthesis process is broken into several interacting tasks which have different impact on the optimization of the entire design regarding different criteria.

On system level we have presented an approach to hardware/software codesign which is based on partitioning guided by the generation and inspection of profiling data. Reasons for implementing a part of the system in hardware can be found in needs for security and increasing system performance. Once a part of the design has determined which is suited for hardware implementation, the selected part of the specification is compiled into an intermediate behavioral format. Depending on the final target architecture, one can choose among different methods to optimize the design by methods partly known from compiler theory. First, the amount of required digital hardware can be diminished by extracting similar or nearly identical parts in the behavioral description. This can achieved by merging them into one block which is elaborated by the subsequent behavioral synthesis. Second, we have presented a bunch of transformations which can be applied at behavioral level. We have shown how these methods influence the outcome of subsequent synthesis task. After the preprocessing is accomplished, behavioral synthesis generates a RT level hardware structure of the behavioral description. Our behavioral synthesis system PMOSS is modular in the sense that it is capable to provide alternatives during the behavioral synthesis process (e.g.,different kinds of scheduling) out of which one can select. This increases the opportunity for interaction and enlarges the design space. The result of the synthesis process up to now, the controller and the datapath, is passed to controller and datapath synthesis, respectively. For the control part we have shown a variety of target architectures and methods which have been tailored to these out which the user can choose. Our intention is that resynthesis by means of retiming has to accompany datapath synthesis. Therefore, in this section we did not address the library matching problem, i.e., the instantiation of already optimized components. After all steps of synthesis have been completed, the result of the previous processes might be critically reviewed by the designer. If the final design does not fulfill the constraints given by the designer or by the foundry, normally one would start to change something manually. We have shown, how certain criteria can be fulfilled by modifying the hierarchical description on the upper most level in which the controller and datapath can be roughly distinguished automatically.

By means of a working example we have shown the applicability of our methods and the need for delivering design alternatives to achieve satisfactory results.

Future work includes developing iterative techniques to improve the structure being generated, extracting area and delay information from logic synthesis and applying them to structural synthesis and to involve geometrical information which is generated by physical design tools. In order to achieve independence from other synthesis steps one has to model the impact of methods on other (normally lower) levels of abstraction. Furthermore, correctness of the results needs to be verified by means of a formal verification or simulation tool.

15% according to the cost measures. For the Knuth, Morris & Pratt algorithm we have achieved the following results:

	Area	Max. Slack	Sum Slacks	Communic.	Power
non-optimized	37181 / 617	-243.40 / -30.40	-134 937 / -388	76,4 / 5,76	1491 / 1.23
optimized	36148 / 749	-240.80 / -30.10.	-134 818 / -517	76,4 / 5,76	1337 / 23.11

Table 7. Comparison of optimized and non-optimized KMP circuit for controller and datapath. Power is measured in mW assuming a 20 Mhz clock frequency.

The data has been generated using SIS and the power estimation tool contained in the synthesis package FLAMES from MIT. The logic for the controller and datapath was mapped to mcnc.genlib. As the initial partition indicates, the datapath dominates the overall cost when compared to the controller. If the gain is related to the cost of the entire circuit, the impact of the proposed procedure is rather low; but if it is related to the effectively manipulated region, i.e. the new controller portion, which has been completely rebuilt by means sequential logic synthesis, the effect becomes fairly considerable.

8.4 Limitations

In order to optimize the boundary of controller and datapath, one has to adjust the hierarchical levels of both partitions. Mostly, this leads to a flattening of the structures in the datapath. Due to this high granularity, the repartitioning process becomes rather slow. In order to decrease the granularity, an initial clustering at the border has to be done which results in a faster repartitioning. The optimization of performance can be heavily done by a local optimization procedure the sum of slacks which was the main criteria to optimize the timing properties of the design is only loosely related to overall performance of the circuit; instead, a global analysis step has to fix the region in which by an aimed resynthesis the slack on the critical path can be diminished. Future work will extend our framework in both directions.

estimation of the global gain the local gain of a move is determined by a fast synthesis[1] of the region around the candidate. Further constraints are implied by the size of the extracted state transition graphs which grows exponential in the number i of inputs, o of outputs and l of latches. So we will have to find a subcircuit $N'(V', E') \subseteq N(V, E)$ of maximal size (number of internal components) with at most l latches and i inputs, for which $n \in N'(V', E')$ holds. Because this is a NP-hard optimization problem, we propose an *breadth-first search* (BFS) clustering process starting from a node $v \in V$ (adjacent to the cutline) that obeys the parameters i, l. On each level of the BFS tree built during the clustering the size and the number of inputs of the cluster is analyzed. If the size of the actual cluster reaches the penalty implied by these restrictions the clustering process terminates; if not, the traced instances are added to the cluster.

Now, the cost of the previously determined region is computed. This is done by applying boolean optimization and mapping to the region separately for all 4 members of the induced partitioning (N'^{local}_{cp}, N'^{local}_{dp}, N^{local}_{cp}, N^{local}_{dp}), so that the local (estimated) gain is

$$gain^{local}(v) = c_{\langle\alpha\rangle, P_{ref}(N)}(N^{local}_{cp}) + c_{\langle\alpha\rangle, P_{ref}(N)}(N^{local}_{dp}) + c_{\langle\alpha\rangle, P_{ref}(N)}(N'^{local}_{cp}) + c_{\langle\alpha\rangle, P_{ref}(N)}(N'^{local}_{dp})$$

However, as we only want an estimation of the lower bound on the overall gain, this is sufficient. If the considered component is accepted for swapping, the STG for the subcircuit is extracted, state-minimization and state-assignment, boolean optimization and mapping is done. Finally, since we are doing partitioning and resynthesis, the former subcircuit is replaced by the optimized design.

8.2.3 Ordering the Set of Candidates

Although the algorithm for the local gain estimation contains a lot of inherent parallelism, it is possible to further speed it up. The algorithm can be accelerated by influencing the order in which the candidates are examined by ranking the most prospective ones first. In this context we only consider topological aspects which can be analyzed rather fast. We give two heuristics for doing this:

- If *Area*, *Power* and *Comm*(unication) is the main objective we find it suitable to order the set of candidates for their expected hypergain, i.e., the reduction in connection cost between the hierarchical entities.
- For speeding up the circuit (regarding *Time*) and only gates lying at the boundary are considered it is useful to choose that gate that balances the *sequential depth* (i.e., minimal distance over both of the partition to a latch) because this increases the optimization potential concerning *Time*

8.3 Applications and Extensions

The previously described methods have been applied to various digital circuits coming from high-level synthesis. Due to the above mentioned cost criteria an average improvement ranging from 5% to

1. Not all options of sequential logic synthesis are exploited.

As pointed out before, the region $N^{local} \subseteq N$ around $v \in V$ is analyzed for the decision whether to accept or dismiss a candidate $v \in V$ for moving. This is justified by the fact that if by a swap the cost of the partition is affected this also counts for the region around the candidate.

By the actual partitioning $P(N)$, a local partitioning $P(N^{local}) = (V_{cp}^{local}, V_{dp}^{local})$ on the set of nodes in this part of the netlist is induced with

$$\varnothing \subset V_{cp}^{local} \subseteq V_{cp}^{local}, \varnothing \subset V_{dp}^{local} \subseteq V_{dp}^{local}$$

If $P'(N)$ is the result of swapping $v \in V$, instead of the computation of the *absolute gain* a *local gain analysis* on the global gain is performed:

$$gain^{local}(v) = c_{\langle\alpha\rangle, P_{ref}(N)}(P(N^{local})) - c_{\langle\alpha\rangle, P_{ref}(N)}(P'(N^{local}))$$

This is a lower bound on the overall gain since we have $gain(v) \geq gain^{local}(v)$. Δ describes the distance of the cost of the actual partition from the optimal cost.Illustration of Great Deluge Algorithm; a configuration corresponds to a partition $P(N)$ which is compared to the actual optimal partition $P_{opt}(N)$. The shaded area marks the forbidden zone.

For the local gain analysis the following rules are established:

- A node $v \in V$ is moved (i.e. it is appended to `nodelist`) from one to another partition if

  $$gain^{local}(v) > \Delta - gain_{\varepsilon}$$

  holds. So, a negative gain denotes a deterioration of the cost (which is allowed within certain bounds given by $gain_{\varepsilon}$).
- If $gain(v) > \Delta$ then a new optimum configuration is obtained which is derived out of $P(N)$ by swapping v into the other partition, resetting $gain_{\varepsilon}$

  $$gain_{\varepsilon} = gain_{\varepsilon} + \varepsilon \cdot (\Delta - gain^{local}(v))$$

  and setting $\Delta = 0$.
- If no new optimum is reached then $\Delta = \Delta - gain^{local}(v)$ is set. The process is aborted if during the last k trials no improvement of the cost function has been achieved, i.e., no change to Δ occurred. In this application, k depends on the cutsize as we do not want to consider members lying at the cutline twice.

As can be seen from this the algorithm can be carried out without evaluating the cost of the entire design successively if$gain_{\varepsilon} = \varepsilon \cdot c_{\langle\alpha\rangle, P_{ref}(N)}(P(N))$ is initialized with the cost of the initial partition $P_{init}(N)$. Notice, that if $P_{init}(N) = P_{ref}(N)$ $gain_{\varepsilon} = \varepsilon$ holds. The decision whether for a certain candidate a swap should be done or not is made upon the result of the local gain analysis.

8.2.2 Resynthesis

Two policies are viable for resynthesis. First, it is possible to *resynthesize after the partitioning* algorithm has terminated. Since each move depends heavily on all other previous and influences all successive moves we favor *resynthesis during the partitioning* process. In order to get an accurate

over the cutline and the cost of the new design is calculated. This step is repeated until a stopping criterion is fulfilled. It might be that the final cost after all swaps have been executed is worse than an intermediate configuration; so the final configuration can be improved by analyzing the log via the function *bestPrefix(P(N),* Log*))* and recovering the best configuration reached, so far.

Some classical approaches appear as special cases in the generic algorithm. Depending on the function *getPromisingNodes* (*P(N*)) and *abort*(*N*, Log, cost) that comprises the stopping criterion several resolutions can be considered. For instance, if only the cost measure *comm(P(N))* in *getPromisingNodes (P(N))* is considered and |`nodelist`| = 1, the algorithm is identical with the *Fiduccia-Mattheyses* mincut heuristics [FiMa82] which is an improvement of the *Kernighan-Lin algorithm* [KeLi70] due to the minor number of iterations.

We implemented an algorithm which terminates when a certain critical cost *THRESHOLD* is reached for the variable `cost`. This enables us to deal with complex cost functions (area, power and speed); for instance, if partition $P'(N)$ results from $P(N)$ by swapping the instances stored in `nodelist`, it is possible to apply the following strategy for the selection of the nodes `nodelist`: In a *Greedy* like fashion the `nodelist` is composed out of the very most promising nodes which are supposed to maximize the gain of a move and minimize the cost $c_{\langle\alpha\rangle, P_{ref}(N)}(P'(N))$ of the new partition. In order to allow for deteriorations of the cost function the list is pertubated which reduces the probability to end up in fairly high costs.

The candidate selection is done by applying a optimization procedure called the G*reat Deluge Algorithm*[1] (GDA) [Du93] which has been appeared to be superior to Simulated Annealing for several intractable problems; it overcomes the disadvantage of the Greedy approach by permitting deteriorations up to a certain limit $(1+\varepsilon)$ with respect to the cost of the actual configuration:

$$c_{\langle\alpha\rangle, P_{ref}(N)}(P'(N)) \le (1+\varepsilon) \cdot c_{\langle\alpha\rangle, P_{ref}(N)}(P_{opt}(N))$$

GDA chooses in *getPromisingNodes* (*P(N*)) an arbitrary hierarchical component for which

$$(1+\varepsilon) \cdot c_{\langle\alpha\rangle, P_{ref}(N)}(P_{opt}(N)) = THRESHOLD$$

holds. The *THRESHOLD* variable is updated and refers to the new found optimum now. For practical instances it is recommended to choose in the range $\varepsilon \approx 0.05\ldots0.10$ (constant). It should be remarked that this algorithm can be modified so that it is not needed to consider the whole design; instead of a global evaluation of the design a local gain analysis is performed.

For the performance of the algorithm it is necessary to estimate the success of a transformation for the given cost function. Because each move of a set of nodes can be simulated by (a) a sequence of swaps of single nodes or (b) by a swap of a hierarchical components that contains all nodes of the set, the task is reduced to the selection of a single candidate at a time.

---

1. This designation refers to an illustration for maximization problems where a lower bound (like a water level) for the cost of the actual configuration is increased steadily. The speed of this rise is determined by ε.

8.2.1 Partitioning

After boolean synthesis frequently it is not known what parts of the circuit are datapath and which ones belong to the controller. Therefore, we have 3 possible choices where to start from:

Definition 30: A *0-Partitioning* denotes a entire design to be regarded as the datapath, i.e. $V_{cp} = \emptyset, V_{dp} = V$. On the other hand, starting from a *1-Partition* means the whole design to be treated as a controller, i.e. $V_{cp} = V, V_{dp} = \emptyset$.

More generally speaking, a so called *p-Partitioning* can be defined which can be the result of the preceding synthesis process initiated by the designer. p describes the participation of the controller in the overall cost regarding a particular weighting $\langle\alpha\rangle$ and N_{ref}; more precisely:

$$p = \frac{c_{\langle\alpha\rangle, P_{ref}(N)}(N_{cp})}{c_{\langle\alpha\rangle, P_{ref}(N)}(P(N))}$$

The goal of the optimization is: find a partitioning $P_{opt}(N) = (V_{cp}^{opt}, V_{dp}^{opt})$ for the weighted cost function $c_{\langle\alpha\rangle, P(N_{ref})}$ with optimal cost $c_{\langle\alpha\rangle, P_{ref}(N)}(P_{opt}(N))$.

We will consider a class of partitioning algorithms for which during one pass cell instances are swapped over the initial or prescribed cutline. A log is kept which protocols each change of the configuration (i.e., swaps of cell instances over the cutline).

<u>CP-/DP-Partitioning:</u>

```
P_init(N)=(N_cp,N_dp) = makeInitialPartition (N);
Log.init(); cost = calculateCost(P(N)); ]
P(N) = P_init(N); optcost = cost;
while (!abort(N, Log, cost)) {
    nodelist = getPromisingNodes (P(N)) ;
    forall (v ∈ nodelist) {
        if (v ∈ N_cp) P(N) = (N_cp \ {v}, N_dp ∪ {v};
        else P(N) = (N_cp \ {v}, N_dp ∪ {v};
        Log.append();
    }
    cost = calculateCost(P(N));
    if (cost < optcost) {
        optcost = cost;
    }
}
P_opt(N) = bestPrefix(P(N), Log));
```

Figure 27. The Generic Partitioning Algorithm.

Figure 27 shows the approach: Starting from an initial partition $P_{opt}(N) = (V_{cp}, V_{dp})$ which is determined by *makeInitialPartition* (*N*), iteratively a number of candidates (cell instances) to be considered for swapping is determined by *getPromisingNodes* (*P(N)*). The components are swapped

Definition 28: The *cost* of a circuit under a partition are described by a vector:

$$c(P(N)) = \langle Area(P(N)), Time(P(N)), Power(P(N)), Comm(P(N)) \rangle$$

- $Area(P(N))$ characterizes the area consumption of the design.
- $Time(P(N))$ is the *slack* on the critical path i.e. the maximal difference between the expected and the actual arrival of a signal at a sequential element. The overall performance which has to improved of the system is reciproportional to this value.
- $Power(P(N))$ denotes the (average) power dissipation of the circuit.
- The communication between the controller and datapath is measured via $Comm(P(N))$.

In the subsequent explanations it is assumed, that the above mentioned measures are additive; this means that if $Area(N_{cp})$ and $Area(N_{dp})$ denote the area of the controller and datapath, respectively, $Area(P(N)) = Area(N_{cp}) + Area(N_{dp})$ is the calculated area of the entire circuit which is the result of combining both subcircuits[1].

Now, by introducing a *scalar cost function* of the circuit with respect to a given cost vector and a partition $P_{ref}(N)$ it is possible to distinguish good designs from bad ones by considering multiple cost metrics simultaneously. This allows the problem to be treated by different optimization techniques where preferences regarding the metrics can be easily expressed.

This assessment of a partition is done by applying a real weighting vector $\langle\alpha\rangle = \langle\alpha_1, \alpha_2, \alpha_3, \alpha_4\rangle$ to the components of the cost vector:

$$c_{\langle\alpha\rangle, P_{ref}(N)}(P(N)) = \alpha_1 \cdot \frac{Area(P(N))}{Area(P_{ref}(N))} + \alpha_2 \cdot \frac{Time(P(N))}{Time(P_{ref}(N))} + \alpha_3 \cdot \frac{Power(P(N))}{Power(P_{ref}(N))} + \alpha_4 \cdot \frac{Comm(P(N))}{Comm(P_{ref}(N))}$$

, where for the weights $\alpha_i \in [0, 1], \sum_i \alpha_i = 1$ counts.

Definition 29: We also define the *weighted cost* with respect to an initial partition $P_{ref}(N)$ for one part $N^{part} \subseteq N$ of the bipartition by $c_{\langle\alpha\rangle, P(N_{ref})}(N^{part})$.

8.2 Methods

The procedure can be divided into two rather independent substages. The first is to find an adequate repartitioning of the design; second, the obtained partitioning has to be turned into a modified structure. The last item we will refer to as resynthesis. Note that the results of the synthesis are also used for establishing sets of candidates which are determined to change their actual partition. The ordering of this set strongly influences the result of the repartitioning and will be explained later on.

1. The area for the wiring of the interface between both partitions is neglected. If the system is pipelined the additional circuitry (i.e. latches) is regarded to be accumulated in one partition.

8 Restructuring and Resynthesis of Designs

If a design coming from RT level synthesis does not fit the design rules implied by the designer further elaboration is required. As with other problems arising in synthesis where high amounts of data are being produced, a full automation of optimizing a design for different goals (such as performance, area or power consumption) is needed. A technique frequently applied to flat descriptions is *Retiming*. However, this technique can be applied to medium sized datapaths in order to optimize them for speed.

We consider the design on the top level, at which it can be roughly divided into two portions; the *controller* which is the result of scheduling operations to time-steps during behavioral synthesis and the *datapath* which is the outcome of allocating hardware to operations. Both parts are connected by status and command lines which is why this target architecture is called FSMD [GaDuLiWu92]. For the further processing both parts are considered separately: for the controller sequential logic synthesis [AsDeNe92] applies and for the datapath module generators are frequently used. An optimization of the whole design by means of sequential logic synthesis seems to be far away from reality.

On the logic or gate level this partitioning seems to be arbitrary since it could be possible to obtain a superior design by incrementally swapping components from one portion to the other one and by applying sequential optimization techniques to the new partitions. By considering the controller/ datapath interface this method takes into account the entire design, so that a higher optimization potential can be gained.

Regarding a hierarchical design on logic level, an optimization regarding the design measures can be gained by the so called *controller / datapath repartitioning*. This is done by combining *partitioning* with *local sequential resynthesis*. Basically, no additional information about the previous synthesis process is required.

We will discuss the methodology by introducing a generic repartitioning framework first. Afterwards, we will discuss cost measures. Finally, the evaluation of the cost function and the resynthesis aspects are examined.

8.1 Input and Output Specification

In the following, only the problem of optimizing bipartitions is considered since the extension to the general case in which the design consists out of *m* hierarchical blocks (multiway partitioning) is straight-forward by decomposition into a set of bipartitioning problems.

Definition 26: The circuit is represented by a hierarchical *netlist* $N(V, E)$. On the top level, V denotes the set components (gates, latches etc.) and E the set of connections running between these components. These components are referred to as *instances* of *cells* having a certain behavior. $N_{cp} = (V_{cp}, E_{cp})$ and $N_{dp} = (V_{dp}, E_{dp})$ depict the controller and datapath, respectively.

Definition 27: $P(N) = (V_{cp}, V_{dp})$ denotes a *partitioning* of the circuit, i.e. a decomposition of the set of nodes N in the flat netlist into disjoint sets such that $\forall v \in V(v \in V_{cp} \vee v \in V_{dp})$ and $V_{cp} \cap V_{dp} = \emptyset$ holds.

The results of the optimization of the whole KMP-circuit and its datapath are shown in Table 5 and Table 6, respectively. The clock cycle is slightly decreased. The expectation that the combined method will deliver the best results, which are confirmed by other examples, e.g., the fibonacci-circuit in this design environment, differs from the results of the overall example. For interpreting the results we presume that loops predominate the timing in the circuit. So further investigations are necessary.

In this example with a bitwidth of 4, the critical path is connected to the control part. In case of a bitwidth of 32, the longest paths will be in the datapath.

7.4 Limitations

A library with variable timing models has not been included yet. Gates with different timing increased by fanout load or different fanin load cannot be handled exactly. Currently only a simple timing model is used.

The logic of the circuit is only slightly modified.

In case of loops forming the critical path in the circuit the tool cannot reduce the clock cycle.

The minimal delays of the gates in the circuit are not available in libraries. As a worst case assumption and as a safe lower bound they are expected to be an arbitrary but fixed percentage of the maximal delays.

7.3 Application and Extensions

7.3.1 Expected Effects of the Procedure

For summarizing the impact of the procedure from a global point of view we can state:

- some redundant logic is eliminated,
- the clock cycle is reduced,
- some delay elements must be added into the clock path of some registers for purposive postponing of some clock signals - so the area can increase slightly,
- the disadvantage is that the required delay elements for the clock lines may not be part of the library. In most cases, their delay can be inside a period of time, but the problem of creating suitable delay elements e.g. by transistor resizing is cut out here,
- some additional registers may be necessary compared with the original circuit, even though a nearly optimal register position is chosen for reducing the number of registers - so the area can increase,
- the combined relocation and clock delay adjustment (Table 6) will deliver the best results.

For avoiding such delay elements in the clock paths, the clock delay adjustment must be switched off. The remaining relocation procedure is an advanced one because of handling the substitute enable register.

7.3.2 Results for the Overall Example

The bitwidth of the example is 4. For the gate library a subset of the *synch.genlib* was used (the simple gates with 1-4 inputs). The minimal delays of the gates are assumed to be 50% of there maximal delay. Setup- and Hold-times are ignored.

method	clock cycle	# of registers	max. clock delay
original circuit		234	
only mapping	28.8	170	0
relocation	26.4	177	0
clock delay adjustment	26.4	170	3.4
combined relocation and clock delay adjustment	26.4	179	3.2

Table 5. Results for the whole KMP-circuit, bitwidth = 4.

method	clock cycle	# of registers	max. clock delay
original circuit		228	
only mapping	21.0	164	0
relocation	20.8	176	0
clock delay adjustment	20.4	164	1.6
combined relocation and clock delay adjustment	20.4	165	1.6

Table 6. Results for the datapath of the KMP-circuit, bitwidth = 4.

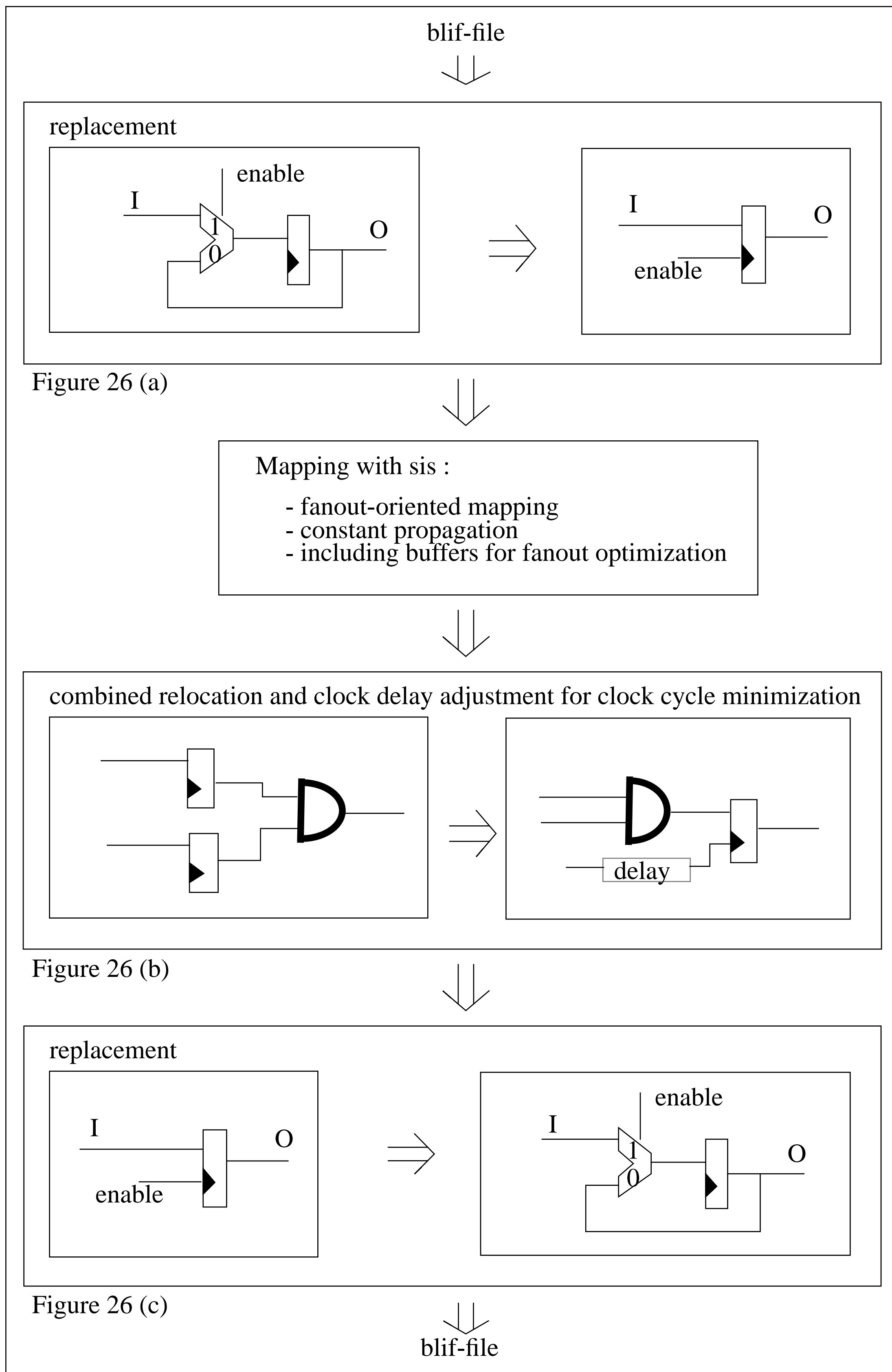


Figure 26. structural optimization after high level synthesis - principal steps.

2. propagation of constants,
3. relocation of register,
4. clock delay adjustment,
5. especially the combination of relocation and clock delay adjustment is a successful part of the procedure.

The different optimization methods are shortly characterized in the following subsections.

7.2.1 Mapping

The fanout and timing oriented mapping and the buffer optimization of SIS is used (see [BaRu93]), which

1. substitutes subcircuits without an equivalent in the library,
2. eliminates redundant logic, especially constant values and unused pins of gates,
3. handles signals having a high fanout.

7.2.2 Register Replacement

The above mentioned replacement of each flip-flop-multiplexor-loop by a single block creates movable registers and preserves them from modification during the mapping process. But their movability is more limited than that of a normal D-flip-flop. This is a real handicap of the usage of enable-registers and it is considered in the advanced procedure. (see Figure 26 and a further SFB-Report). This replacement allows the usage of the following retiming methods.

7.2.3 Relocation and Clock Delay Adjustment

The combination of relocation and clock delay adjustment is a successful approach to reduce the circuit's clock cycle (see [Ma93]). Both the pure relocation without any delay elements and the pure clock delay adjustment with a fixed register position are also implemented. But the combination is the best way to balance the paths in a circuit. The idea was to enlarge optimization space by combining two different approaches in one method. The tool considers long and short paths in the circuit simultaneously and it selects the best register position and suitable delay elements at the same time. The clock cycle is given or the tool searches the minimal clock cycle. In spite of subsection 7.2.1, which works timing oriented but without real constraints, subsection 7.2.3 tries to fit timing constraints.

7.2.4 Reverse Register Replacement

At last, the register replacement must be undone for getting a correct circuit.

7 Datapath Synthesis

The timing improvement starts with the structural description of a circuit or network resulting from the high-level synthesis. The preceding performance optimization at the higher level of abstraction is now supplemented by improvements in the structural domain.

The optimization in this chapter focuses on register relocation and clock delay adjustment and further techniques. The first two methods and their combination described in [Ma93] is conceived for nonhierarchical circuits with edge-triggered D-flip-flops.

In order to make use of these methods in the design environment the original procedures had to be modified and adapted to the netlist generated by the high-level synthesis.

7.1 Input and Output Specification

The input of the optimization on the structural domain this section is a *blif* file containing parts of a circuit at the netlist level. It could contain the whole circuit - both datapath and control part together - or only the datapath is optimized. The given file may be a hierarchical description of mapped or unmapped logic. The registers can be either D-flip-flops or enable registers. An enable register consists of a multiplexor and a D-flip-flop, whose output is feed-backed to the multiplexor input (see left side of Figure 26 (a)). The enable register cannot be handled by the original algorithm because the D-flipflop is caught in the loop. To optimize the circuit by means of the original algorithm we replace the enable flipflops by D-flip-flops and use the enable signal as a modified clock. The replacement of the loop with the D-flip-flop and the multiplexor is shown in Figure 26 (a). After optimization the replacement will be undone (see Figure 26 (c)).

Therefore the input file must be modified and the procedure had to be upgraded to manage such substituted registers which does not read the incomming data at every clock cycle. This seemingly small difference requires a larger adaptation of the algorithm, which will be described in a future SFB-report.

The output of the task of this section is a *blif* file as well (no hierarchy, flat circuit, mapped or unmapped logic blocks).

Additionally the tool delivers a list of registers whose clock signals must be postponed. For most of these registers, the list contains not a fixed value, but a lower and an upper bound for the delay elements which must be included into the clock line. These delays are given in the form of a text file.

The principal sequence of transformations of this chapter is depicted in Figure 26.

7.2 Optimization Methods

The first aim (cost function) of the optimization on the structural domain is speeding up the circuit by minimizing the clock cycle. A second goal is reducing the number of registers.

The following transformations and optimization methods are applied to the circuit:

1. fanout optimization, especially a fanout and timing oriented mapping,

Concerning the linear partitioning (program NET) the controller synthesis is done for different number of partition blocks targeting PLA implementation and multi-level implementation.
We used the PLA cost functions

$area_{PLA} = (2*(\#I) + (\#O))*(\#PT)$ and

$delay_{PLA} = max(i,j)$ (fanout input i + fanout productterm j)

without consideration of communication costs.
In the case of multi-level circuits the values for area and delay include communication network and additional flip-flops (computed by SIS using the library synch.genlib). Table 3 shows the experimental results. Concerning PLA implementations the area is essentially reduced with a increasing number of partitions but the demand for the communication costs for the OR-elements of the communication and for the additional flip-flops have to take into account. In both cases the cycle time can be important diminished by partitioning, especially for a increasing number of partition blocks.

Concerning the method of subsection 6.2.3 a considerable design improvement can also be achieved for our example by splitting off an autonomous automaton from initial state transition graph (see Table 4).

$area_{PLA}$ for	**single FSM**	**partitioning heuristic A**	**partitioning heuristic B**
Command Block		720 / 800*)	800 / 720*)
Autonomous Block		630 / 522*)	594 / 504*)
Sequencing Block		261 / 290*)	290 / 290*)
Total	2632 / 2576*)	1611 / 1612*)	1684 / 1514*)

Table 4. Extraction of an autonomous automaton (see subsection 6.2.3).

The results are compared for two partitioning heuristics without, resp. with, previous state minimization*) in relation to $area_{PLA}$ of original FSM.

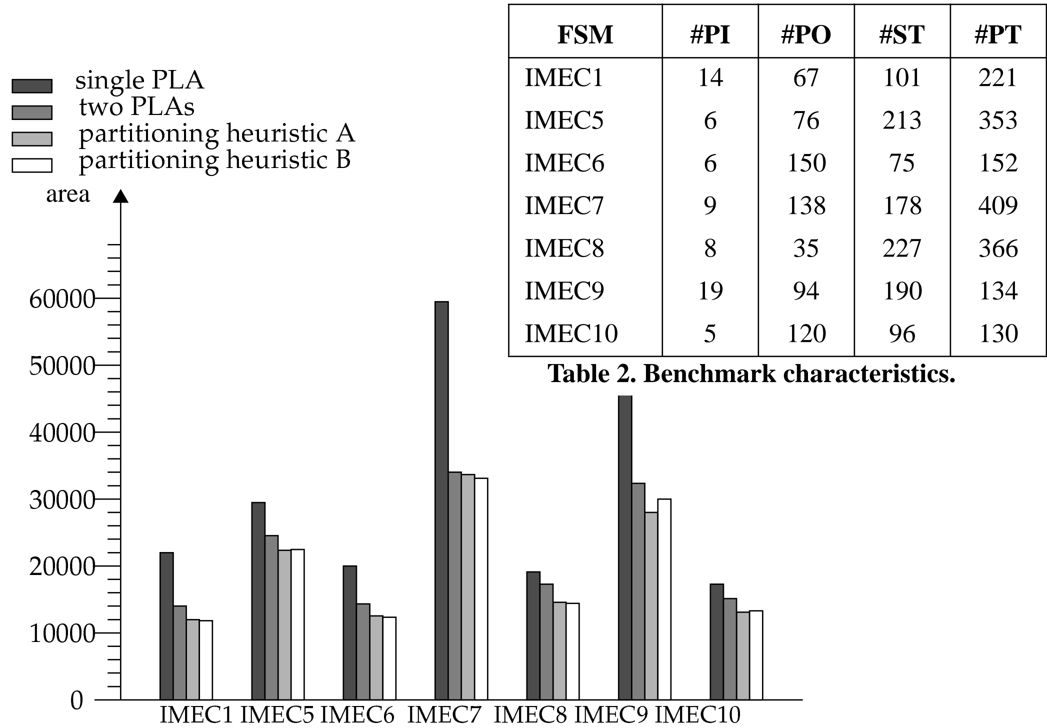
6.4 Limitations

For a given partition $\pi = \{B_1,...,B_p\}$ of the set of states the aim of the tool NET (subsection 6.2.2) is finding a linear FSM net minimized with regard to the subFSM cost and the communication between them. This method is made (and restricted) for a two-level circuit therefore both optimizations of the communication and of the subFSMs could be interpreted as a common encoding task.
In NET a special communication structure is proposed, which results for a lot of applications in an implementation with reduced overall costs. On special conditions it seems to be possible to extend this approach to the general FSM partitioning problem.

The method implemented in the program CNT (subsection 6.2.3) is especially suitable for large Moore sequencers whose graph representation contains a relatively small number of forks.

One extension of these partitioning approaches is to operate with programmable logic. In this case the subFSM have to fulfil strong limitations concerning the number of inputs and outputs and the communication structure. An other aspect is to diminish the power consumption of the linear partitioned FSM net because in each time only one subFSM is active while all other FSMs are in the wait state.

This partitioning and optimization approach is implemented in the program **CNT**. Figure 25 shows experimental results obtained by CNT for FSM Benchmarks (MCNC) and application specific Moore Sequencers (IMEC). Design improvements are obtained by the proposed design modification method using two heuristics for constraint driven partitioning of state transition graph as shown in [FeMu93].



FSM	#PI	#PO	#ST	#PT
IMEC1	14	67	101	221
IMEC5	6	76	213	353
IMEC6	6	150	75	152
IMEC7	9	138	178	409
IMEC8	8	35	227	366
IMEC9	19	94	190	134
IMEC10	5	120	96	130

Table 2. Benchmark characteristics.

Figure 25. Experimental results.

6.3 Applications and Extensions

The application of the methods to benchmarks are reported in subsection 6.2.2 and subsection 6.2.3. For the overall design example (Knuth, Morris & Pratt algorithm) the PMOSS High Level Synthesis System outputs a controller description characterized by 5 primary inputs (#PI), 28 nonredundant primary outputs (#PO), 44 states (#ST) and 54 transitions (#PT).

	PLA implementation	**multi level implementation**
number of partition blocks	**$area_{PLA}$ / $delay_{PLA}$**	**$area_{SIS}$ / $delay_{SIS}$**
1	2408 / 30	215 / 10.89
2	1828 / 21	285/ 11.87
3	1567 / 17	305/ 9.14
4	1247 / 14	286 / 8.06
5	1244 / 12	301 / 7.83
6	1135 / 11	320 / 7.46
7	1129 / 11	342 / 7.42
8	1047 / 10	334 / 9.26

Table 3. Linear FSM partitioning (see subsection 6.2.2).

A method to refine structural approach by partitioning is the synthesis of FSM targeting an autonomous FSM and PLAs. This partitioning approach is proposed for the synthesis of complex finite state machines (FSM) and is characterized by splitting off an autonomous automaton from the initial description. The target architecture can be viewed as a generalization of FSM implementations with embedded loadable counters [AmBa89].

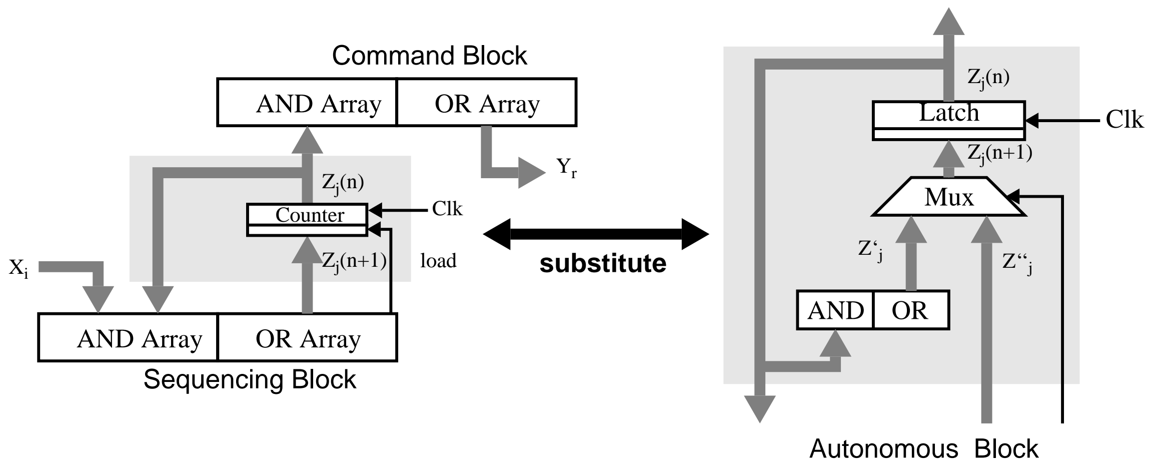


Figure 23. Structural approach using autonomous FSM.

The method starts with a subgraph extraction algorithm using a modified state transition graph generated from the given state transition table. Subsequently, constraint-driven partitioning separates an autonomous automaton and a remaining FSM. This results in generation of three symbolical descriptions corresponding to a Sequencing-, Command- and Autonomous Logic Block. For a better exploitation of its inherent optimization potential the functional models are transformed to a special common state encoding representation. This facilitates us to solve the encoding problem for all logic blocks simultaneously. Consequently logic optimization leads to minimize the total costs of partitioned circuits. Figure 24 illustrates the main steps of the proposed approach for a small example created in [AmBa89]. More details about partitioning, simultaneous encoding and optimization are explained in the report [FeMu93].

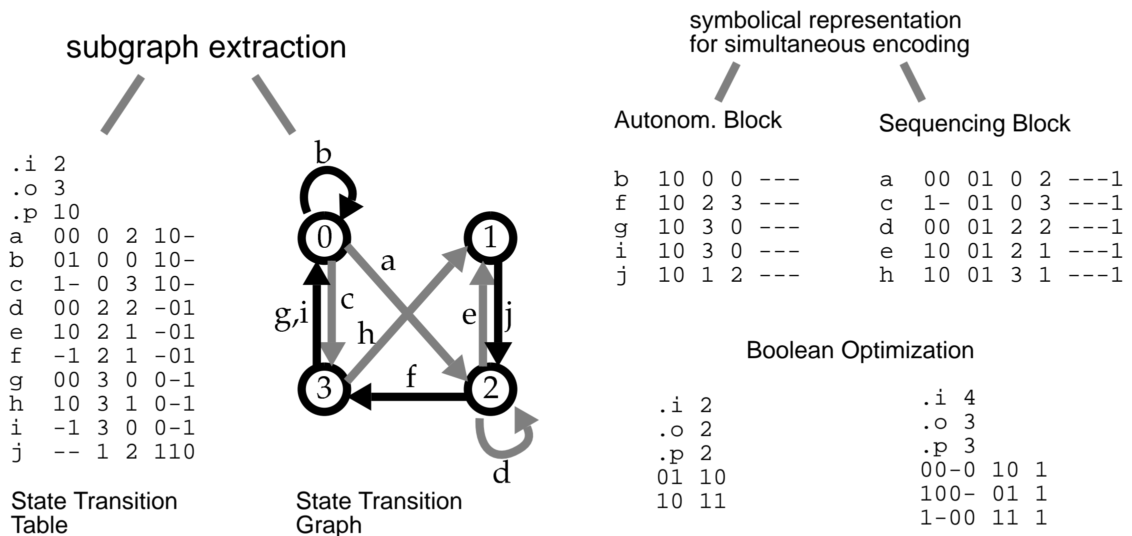


Figure 24. Illustration of main steps of the proposed approach.

- dominance constraints with respect to our communication structure and
- constraints for reducing the number of signal lines for cooperation between the subFSMs.

In Figure 21 a FSM net is depicted consisting of two subFSMs M_1 and M_2. SubFSM M_1 controls sub-FSM M_2 by means of an intermediate output IO_1 and subFSM M_1 is controlled by two intermediate outputs IO_{21} and IO_{22} of M_2.

In Figure 22 we show results of this approach using different MCNC FSM benchmarks. A/A_0 and T/T_0 are the ratio of area and delay, respectively, between the partitioned and non partitioned case. After partitioning the cost of the synthesized net depends on the selected encoding constraints. In the table PI and PO denote the number of primary inputs and outputs, ST and PT the state and product term number, respectively. It demonstrates clearly that in many applications the partitioning approach is advantageous not only in respect to the critical path delay but also in respect to the total costs. For some examples this is even true if we consider the cost for the additional flip-flops and communication logic.

FSM	#PI	#PO	#ST	#PT
bbsse	7	7	16	30
cse	7	7	16	91
dk14	3	5	7	56
dk16	2	3	27	108
ex1	9	19	20	198
ex4	6	9	14	21
jo	8	10	17	56
planet1	7	19	48	115
s1	8	6	20	107
train11	2	1	11	11

Table 1. Number of inputs, outputs, states and product terms of Figure 23.

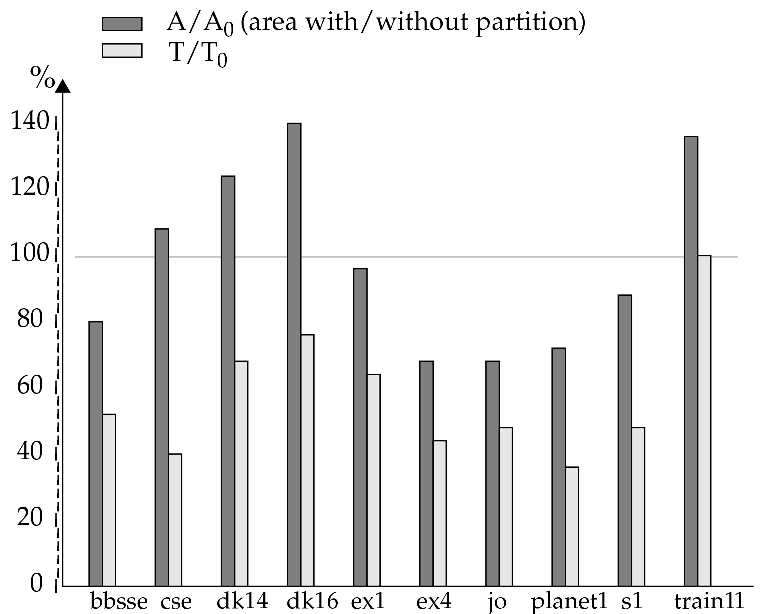


Figure 22. Benchmarks results.

The approach of simultaneous state encoding and communication cost optimization starting from a STT and a given partition of the set of states is implemented in the program **NET**. Detailed information about this approach are given in the report [KoFeF93] and in [KoGFF94]. At present a program is under development generating suitable state partitions for the general partitioning of automaton. The program contains the case of linear partitioning.

6.2.3 Generalization of Counter Based Controllers

Qualitative and quantitative characteristics of the state transition graph influence size and performance of commonly used controller architectures. Design improvement can be achieved by behavioral adapted FSM partitioning.

The different approaches for the linear decomposition of FSMs use different ways to organize the communication between subFSMs. Figure 20 shows our new proposal the following design procedure is adapted to.

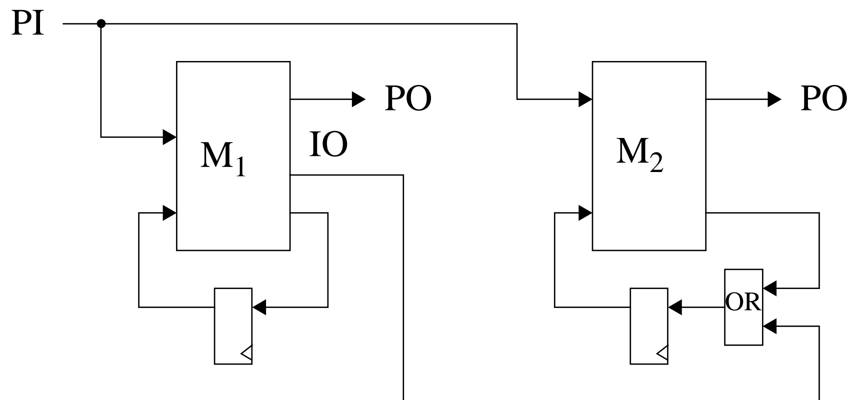


Figure 20. Communication structure for linear FSM partitioning.

An intermediate output IO of FSM M_1 controls FSM M_2 by setting its register in the moment t with the aim that in the moment t+1 M_2 can start with the actual initial state.

In order to optimize the FSMs net cost we have to encode the internal states of all subFSMs and to minimize the communication overhead for controlling the cooperation between the subFSMs simultaneously. We solve this task by interpreting it as a state assignment problem under different encoding constraints.

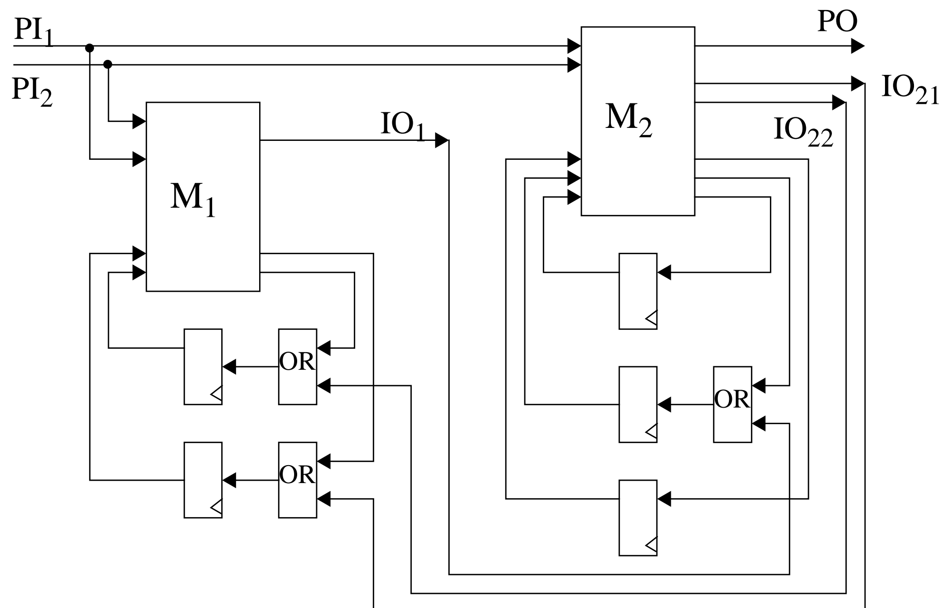


Figure 21. Example of a FSM net.

We will take into consideration:

- face embedding constraints for minimizing the subFSM costs,

ments including MCNC benchmarks the effectiveness of this approach was proved. In most cases the optimal implementation was generated by using selected but not all sharing possibilities.

The possibilities to improve a design by this encoding method cannot be predicted. The possibilities depend on the topology of the state transition graph and of similarities between its Boolean expressions and output function.

6.2.2 Linear FSM-Partitioning

A special FSM-partitioning approach is proposed in [Putk91], [JoKo91], [BaBr93] and is called linear partitioning in [GrMu89]. The specifications of the subFSMs are constructed by

1. cutting edges in the state transition graph of the FSM such that the graph is decomposed into unconnected subgraphs,
2. connecting the outgoing edges of a subgraph to an additional node assigned to a wait (idle) state of the corresponding subFSM and
3. introducing additional edges that define the conditions for remaining in a wait state or leaving it.

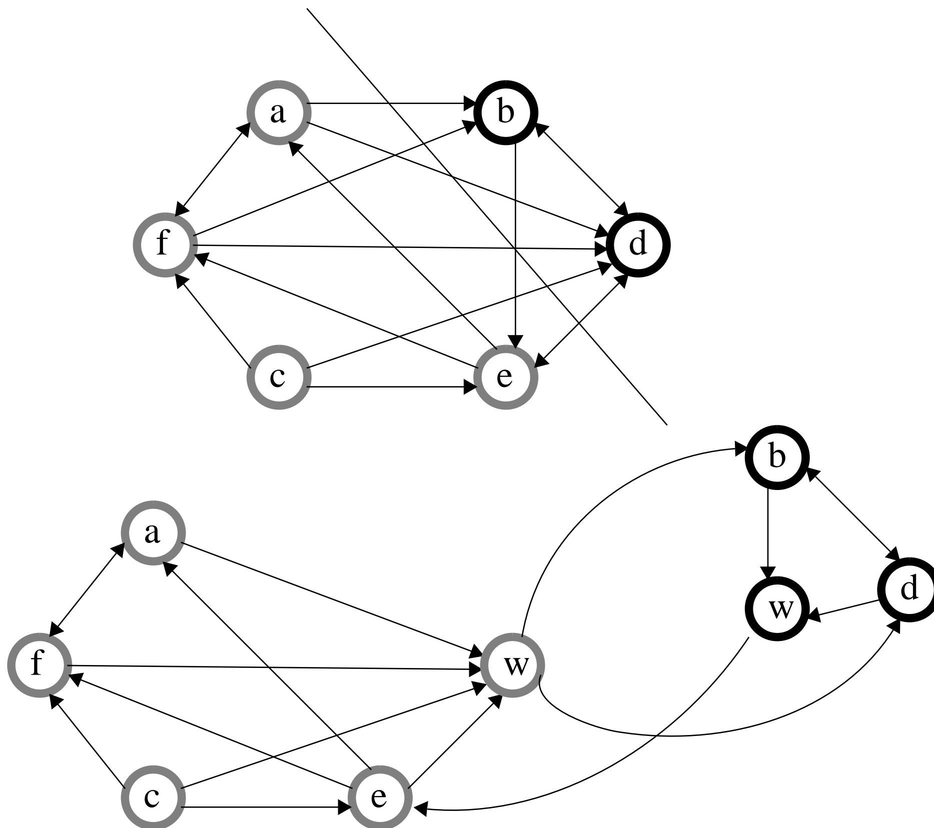


Figure 19. Example for a linear STG partitioning.

In a FSM net derived from such a set of STGs at any moment only one of the subFSMs is active, i.e., not staying in a wait state. The price one has to pay is an enlarged number of flip-flops compared with an implementation of the original FSM. On the other hand, the combinational circuits for the submachines are often smaller resulting at least in a reduced critical path and mostly in a reduction of the overall costs.

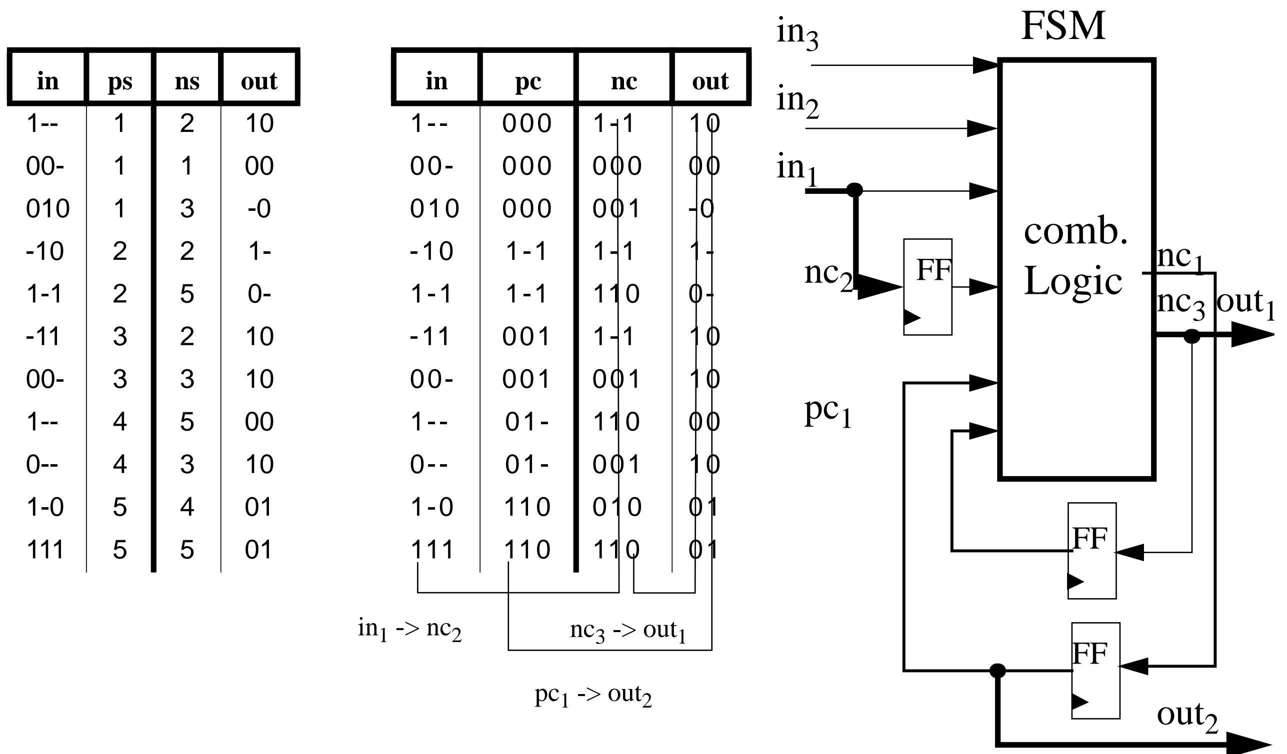
the encoding procedure. The procedure encodes symbolic states so that code variables share some inputs and/or output functions and also logic combinations of these functions. Thus the number of next state and output functions which have to be implemented can be reduced.

The aim of the procedure is to find and to exploit a maximum of possibilities for sharing

- next state (code) and output functions [case (nc,out)]
- inputs and next state (code) functions [case (in,nc)]
- code components of the present states (code) and output functions [case (pc,out)].

To illustrate the intended effects we use the state transition table (STT) of Figure 18 (a) with symbolic states. By means of the iterative encoding procedure the encoded STT of Figure 18 (b) is computed. Three columns may be deleted. Output function 1 (out_1) and 2 (out_2) are formed by the third next code (nc_3) and first present code variable (pc_1), respectively. The first input variable (in_1) will be used to generate the second next code variable (nc_2). Figure 18 (c) shows the resulting circuit. Three sharing pairs are found in three iterations. All states of the remaining FSM are in this special example compatible in pairs. Therefore the remaining FSM becomes combinational logic.

To benefit from the growing number of states of the remaining FSM which are compatible in pairs we compute an implementation using synthesis tools (SIS, MIPRE), which minimizes and encodes states together. As a result the final code is a concatenation of the partial code, performed in this way, and the code computed by SIS or MIPRE.



in	ps	ns	out
1--	1	2	10
00-	1	1	00
010	1	3	-0
-10	2	2	1-
1-1	2	5	0-
-11	3	2	10
00-	3	3	10
1--	4	5	00
0--	4	3	10
1-0	5	4	01
111	5	5	01

in	pc	nc	out
1--	000	1-1	10
00-	000	000	00
010	000	001	-0
-10	1-1	1-1	1-
1-1	1-1	110	0-
-11	001	1-1	10
00-	001	001	10
1--	01-	110	00
0--	01-	001	10
1-0	110	010	01
111	110	110	01

Figure 18. (a) STT with symb. states. (b) Encoded STT. (c) FSM with shared variables.

The main steps of the iterative procedure are outlined in [KoFeFr93]. This iterative partial state encoding procedure can be used for an iterative structuring of the automaton. By means of experi-

6 Controller Synthesis

The high-level synthesis by means of PMOSS outputs controller descriptions given as state transition tables (STT). A primary controller design can now be done by SIS, Synopsys, MIPRE or other tools. To meet special demands (area, time, power, adaptation to FPGAs etc.) the initial design has to be improved by resynthesis or partitioning.

The synthesis process of controllers consists of several subtasks: state count reduction, encoding of symbolic inputs, outputs and symbolic states, among which the assignment of binary codes to the symbolic states is suitable to influence speed and cost (i.e., area) at a very high extend. Therefore this problem is regarded as a vital task and algorithms which tackle this problem are presented in the following subsections.

The aim of controller synthesis and resynthesis is to find an improved and optimal realization targeting a special architecture (e.g., multiple-level or programmable logic) out of the behavioral controller description. Our approach to this problem is (a) to detect special properties in the symbolic description, (b) to select a suitable target architecture and (c) to select an appropriate method to achieve the goal by obeying the implied restrictions.

In the area of controller synthesis we investigate FSM partitioning and reencoding approaches. The focus is on flexibility of the target structures, increasing the optimization potential, exploitation of propagated constraints and propose design alternatives.

6.1 Input and Output Specification

The input specification of the controller design and redesign is given in the form of state transition tables (STT). The STT is a tabular representation of the FSM. Each row i can be described as a 4-tuple

$$(in_i, pc_i, nc_i, out_i),$$

where in_i, and out_i are assignments of the input and output variables respectively, pc_i and nc_i are codes of the symbolic present state (ps) and next state (ns) respectively. For a given FSM the task consists in finding a minimal implementation. The results of controller design are generated in the form of PLA-format and/or of hierarchical netlists including the communication network of sub-FSMs.

To demonstrate the design flow by means of the overall example we concentrate on linear partitioning and partitioning concerning autonomous automaton as examples for the controller design.

6.2 Methods

6.2.1 Iterative Partial State Assignment of FSM

The aim of state assignment of finite state machines (FSMs) is to compute binary or ternary codes for the states so that the implementation is minimal with respect to a target structure. To improve the primary design of an FSM an iterative state assignment method can be used. The exploitation of the optimization range and in this way the quality of the solution is controllable by the number of iterations of

5.4 Limitations

If the module chosen for the HW implementation is smaller than the area available for these special function unit (SFU), it is possible to map another - in our terms called similar - structure onto the SFU. For this application we need a fast estimation for the trade-off between the performance deterioration for the original task, and the improvement by implementing an additional task in hardware.

To merge the basic blocks we have to determine, in which order the tasks of the different matches are to be executed. This is important, because for every *pair* of matches in BB_i and BB_j, the pattern structure should implemented only *once*.

To determine the execution ordering, the matches are handled like operator nodes in a DFG: for every basic block, which should be included in the new function, a match graph G_{match} (V,E) is build.The nodes in G_{match} represent the matches, an edge from V_1 to V_2 denotes a data dependency between a node $v_1 \in V_1$ and $v_2 \in V_2$ (see example in figure 16).

Figure 16. Nodes covered by matches of distinct patterns with match graph, denoting their dependency

Afterwards, a *common schedule* for the match graphs of all corresponding basic blocks is computed. Common schedule means, that we try to arrange the matches in an order, that the structure can be used by *all* corresponding basic blocks. This is comparable to resource sharing for conditional branches.

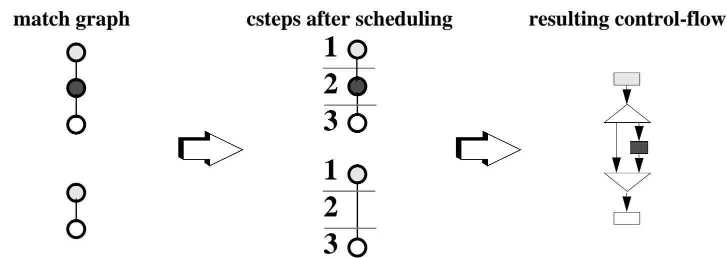


Figure 17. Common scheduling of the match graphs of corresponding basic blocks for determining the merging strategy

5.3.2 Guiding the High Level Synthesis

Up to now, our approach achieves all its effects - as guiding the sharing, speedup of the following tasks of the high-level synthesis - implicitly, by restructuring the CDFG. It has to be considered, whether the informations obtained through the clustering process are also useful to guide the synthesis process (e.g., the allocation) explicitly by annotating the original CDFG. Guiding the high-level synthesis by annotations in the CDFG would allow us to use more hardware related information for computing the similarity functions. E.g., the DataPathSimilarity could accept an addition as similar to a subtraction if an ALU for Add/Sub is included in the library. This relaxation of the definition of similarity would lead to an increase of the number of matches.

As long as we use the result of the partitioning for a restructuring we can not consider knowledge about the library, because we are not able to model an ALU at the behavioral layer of the PSF.

5.3.3 Mapping to Predefined Structures under Area and Timing Constraints

Another application of the approach presented in this chapter, which seems to be very important in the future is the application of our approach for HW/SW Codesign. The results of different researches show, that the modules which are to be implemented in HW contain mostly a high amount of control.

5.2.3 Pruning the Search Space

For an ‘exhaustive-search’ approach to obtain the best overall cluster structure, we would have to compare *all* nodes with each other. But we can prune the search space by recalling the rules for the similarity of DFG and CFG: We only have to compute the DataPathSimilarity and ControlPathSimilarity for basic blocks whose corresponding nodes in the CFG are of the same type.

A second methodology to reduce the number of CFGs to be compared, is based on the PathVector.

Definition 25: A *PathVector* is a vector of tuples $nodetype\ branchinfo$ denoting the path from the start of a thread to a certain node in the CFG. The straightforward approach to build the PathVector for a certain node is like counting brackets: For every passed control node, we add a tuple $nodetype\ branchinfo$ to the vector. If we pass a join node (end-if, end-case), or if we reach an already passed while node, the corresponding entry is deleted An example for PathVectors is given in figure 15. Note, that the length of a PathVector denotes the number of nested control structures encapsulating a node.

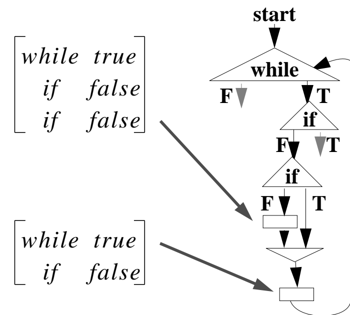


Figure 15. Examples for the PathVector.

Our selection algorithm assumes a pair of nodes, N_1, N_2, to be the ‘innermost’ nodes of the CFGs $C_1$and C_2 respectively. That means, we assume N_1, N_2 to be the nodes with the longest PathVector in C_1, C_2. As a precondition for the similarity of C_1 and C_2, the PathVectors of N_1 and N_2 must have an equal suffix. To ensure, that C_1 is no subgraph of C_2 or vice versa, there must not exist a ‘substring’ (or subvector) relationship between the PathVectors of N_1 and N_2, starting at the first entry. This can easily be seen in Figure 15. The two basicblocks which PathVectors are shown, are not encapsulated by disjunct control structures despite their equal suffix $if\ false$.

For every pair of nodes N_1, N_2, for which the preconditions are fulfilled, we put the pairs of control nodes into the list of nodes to be tested for ControlPathSimilarity. The entries of the lists are sorted by the ‘distance’ between N_2 (N_1) and the control node, i.e. by the depth of the nested structure.

5.3 Applications and Extensions

5.3.1 Restructuring

In the restructuring phase of the algorithm, the subgraphs (or ‘macro functions’) of a cluster are combined to a new function (‘multi functional unit’) able to compute the tasks of the subgraphs.

To be able to calculate the Data Path Similarity, we have to determine a ‘good’ covering of the DFG with matches of certain patterns. To obtain such a good cover (i.e., only few different patterns), we first have to determine the patterns that occur in the DFG repeatedly. For this purpose, we chose the algorithm of Karp, Miller and Rosenberg which was presented in [KMR72]; it computes all patterns that occur more than once by building equivalence classes. The original algorithm has slightly been modified according to our definition of similarity. One of the advantages of the algorithm is his applicability to the original DFG. There is no need to convert it e.g. into a forest of trees. Also, the algorithm computes the pattern itself and the matches at the same time.

Because every match for patterns occurring repeatedly is computed, the matches sometimes overlap. To determine a - if possible complete - non overlapping cover of the DFG, we first delete the matches, that are not valid. Afterwards we chose from the remaining matches by ordering them according to the relation $P_i > P_j \Leftrightarrow (|P_i| > |P_j|) \vee (|P_i| = |P_j| \wedge matches(P_i) > matches(P_j))$.

5.2.2 Control Path Similarity

Our partitioning approach provides information for the further synthesis of a design. The main objective is to force architectural synthesis to share the register transfer (RT) level representation of whole functions. That encompasses not only the sharing of functional units and registers, but also sharing the controller part. To achieve the latter, we have to compare subgraphs of the CFG. As a precondition for clustering two sub-CDFGs, not only the DFG part has to be similar but also the CFGs.

As in our data format the only valid control statements are loops (i.e. while, for, repeat...until) or branches (if, case) and according to the definition of a basic block, every path through the CFG can be described by a sequence of the following three ‘CFG-patterns’:

- *Start* [BB]
- Ctrl [BB]
- EndCtrl [BB]

The term [BB] denotes an optional occurrence of a basic block, *Start* denotes the start of the thread. Using these three patterns, we define the *ControlPathSimilarity.*

Definition 24: Two (sub-) CFGs are similar, if all paths through the CFGs can be described by the same sequence of the three basic CFG-patterns described above.

Due to this definition, the similarity value $ControlPathSimilarity(CFG_i, CFG_j) \in \{0, 1\}$. Figure 14 shows five example CFGs. CFG_1 is similar to CFG_2 and CFG_3 is similar to CFG_4.

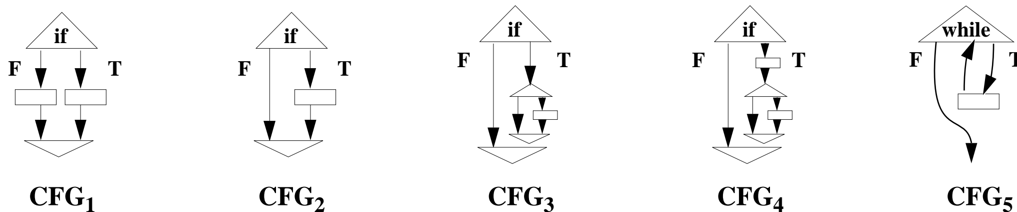


Figure 14. Examples for ControlPathSimilarity.

Definition 20: The smallest of all possible search arguments consists only of operator nodes and is called a *pattern*.

Definition 21: A subgraph of a DFG similar to a pattern P is a *match* for P.

Definition 22: The subset of matchnodes, which have an input from outside the match is defined as the *match boundary.*

Due to the use of graphs rather than trees for the internal representation, we need a test to ensure that a match of a pattern represents a sequential part of the original code.

Definition 23: A match M is *valid*, if there exist no dependency between two nodes of M, which crosses the match boundary. An example of an invalid match is shown in Figure 12.

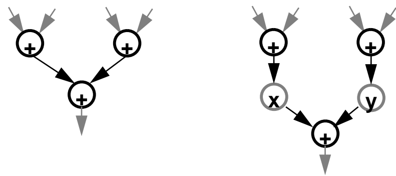


Figure 9. Example for similar graphs.

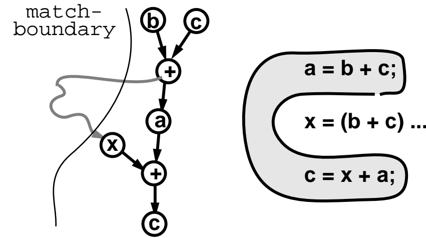


Figure 12. Invalid match.

Given the definitions from above, we are able to define the DataPathSimilarity (DFS).
The similarity value for two basic blocks in the DFG, $DataPathSimilarity(BB_i, BB_j)$, is defined as

$$min(WeightedCover(BB_i, BB_j), WeightedCover(BB_j, BB_i))$$

with

$$WeightedCover(BB_i, BB_j) \qquad \frac{\sum_{p \in P_{ij}} size_p \times |matches_p(BB_i)|}{|BB_i|}$$

P	denoting a pattern
$matches_P(BB_i)$	the number of matches of pattern P covering BB_i
P_{ij}	$\{P \mid matches_p(BB_i) \neq \emptyset \wedge matches_p(BB_j) \neq \emptyset\}$
$\|BB_i\|$	number of operator nodes in BB_i.

The minimum selection of the two *WeightedCover* values is needed to force the clustering of large basic blocks, as Figure 13 shows.

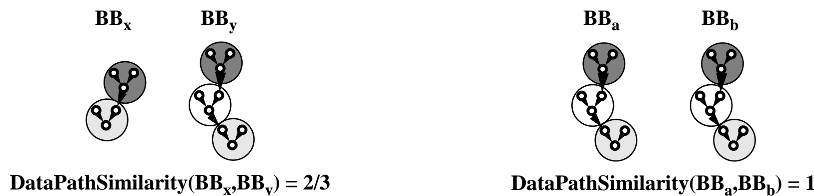


Figure 13. Example for DataPathSimilarity.

For an easier computation, the nodes in a DFG representing a basic block are encapsulated by basic block-begin and -end nodes.

Definition 17: A control data flow graph, CDFG, is the representation of a behavioral specification by a CFG and a DFG. The nodes of the CFG and the DFG are linked together with pointers, to denote their correspondence (e.g. a branch node is linked to the basic block-begin node in the DFG, which encapsulates the computation of the corresponding condition).

We have to add, that the CDFGs of the PSF are hierarchical in the terms of Software. A hierarchical node in the CFG and in the DFG respectively, represents a call to another function, also represented by a CDFG.

5.2 Behavioral Partitioning

As stated before, we want to base our partitioning not only on physical considerations of single nodes of the CDFG representing the design. For this reason we have to define control flow similarity and data flow similarity as well.

5.2.1 Data Flow Similarity

As the goal of the clustering is the building of multi functional units for macro operations. We do not only compare the occurrence of the different operator types in the basic specification. We also consider dependencies between the operations. Therefore we compare subgraphs of the DFG, which represent basic blocks.

If we would restrict the applicability of our approach to data driven applications, we could apply a straightforward scheme to compare the subgraphs of the design's representation as a signal flow graph. But to preserve applicability to specifications in general, we have to cope with the following problem: There is no single assignment. Every variable can be written several times. It is even possible, that the specification includes operations, with source and target being the same variable.

This leads us to the following definitions:

Definition 18: The operator nodes O_1 and O_2 in a DFG are called *connected*, if there exist a path between O_1 and O_2 with no other operator in between. That means O_1 is either a direct input for O_2, or O_1 writes its result to a register R, which is an input for O_2, as shown in Figure 11.

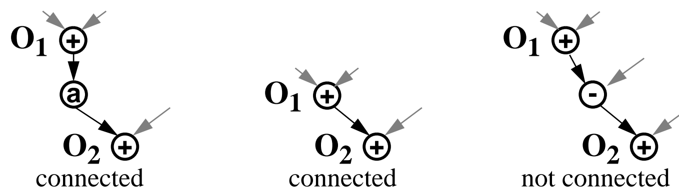


Figure 11. Connected and not connected operator-nodes.

Definition 19: Two subgraphs are *similar*, if they have the same operator connectivity, no matter, if the operands differ or sub-results are used. Figure 9 shows an example for similar graphs.

The definition of similarity allows to unify more expressions than by exact comparison. To identify similar subgraphs in the DFG, a 'reference' structure is needed as search key. Due to our definition of similarity, there always exist several keys.

5 Partitioning and Restructuring a Design on Behavioral Level

Partitioning has always been an important aspect in the synthesis of digital systems. It has been used for many different purposes.

Camposano and Brayton [CaBr87] used partitioning mainly to speed up logic synthesis without deteriorating the resulting design. To accomplish these objective they used a similarity function considering physical information (e.g., the number of common in- and outputs of partitions) as well as semantic information (e.g. the amount of common logic of different operations). Dirkes Lagnese and Thomas [LaTh89] introduced the multi-stage clustering, a methodology to handle partitioning for multiple criteria. Besides the similarity functions of Camposano and Brayton, they also used informations extracted from the control flow of the specification (e.g., the probability, that two operations are activated sequentially). Lagnese and Thomas used the informations gathered in the clustering process for guiding the following high level synthesis. Both approaches mentioned above have in common, that their similarity functions are only based on single operations or groups of single operations. They do not examine subgraphs of the specification.

Recently, the efforts to take advantage of similarity of subgraphs of a specification have been increased. Rao and Kurdahi [RaKu92] presented a partitioning approach based on the inherent regularity of DSP applications. They use it to concentrate the synthesis on only a few, but repeatedly occurring subgraphs in the signal flow graph, called patterns. By doing the design space exploration only for these patterns, they reached a significant speed-up of the whole synthesis process. In the *Cathedral-III Compiler* [Cath91], regularity is used to generate so called application specific units (ASU). These are macro function units, i.e., groups of operators connected by a minimum number of multiplexers, allowing a chained computation of repeatedly occurring computations on the critical path. Due to this chaining, the Cathedral-III tool is able to meet the requirements of high throughput applications. As the approach of Rao and Kurdahi and the Cathedral-III tool are designed towards data driven applications like digital signal processing (DSP), they do not consider control flow in their definition of regularity.

The partitioning approach presented in this chapter is based on a metric for *regularity in data and control flow*. The code segments (represented by graphs as we explain later), the objects of the clustering, we can regard as macro operations. By clustering the macro operations we obtain a ‘multifunctional unit’, which can be implemented more efficiently than the implementation of the macro operations would be.

5.1 Input and Output Specification

The interface to the partitioning and restructuring tool is the Paderborn Synthesis Format (PSF), a graph based internal data format of the Paderborn Modular Synthesis System (PMOSS). In this section, we describe the basic definitions for the behavioral layer of the PSF.

Definition 15: A data flow graph, DFG, is a graph $G_{data\ flow}(V,E)$, where the nodes represent operations or storage and the edges represent data transport.

Definition 16: A control flow graph, CFG, represents the control flow of a specification. There exist two types of nodes in the CFG: control nodes and basic block nodes. Control nodes represent control statements (while, if) as they appear in the specification. The term basic block, BB, also used in compiler theory [ASU86], is defined as a set of statements always executed sequentially. There are no jumps in or out of a basic block.

4.4 Limitations

The approach which is taken does not contain all known transformations of compiler theory. Other will be applied soon. Considering the example in Figure 7 in this state of implementation no transformation can be applied. This does not mean that transformational design change is not applicable at all. The reason for this is the size of the taken benchmark, in this report. The designs need to be much bigger than the Knuth, Morris & Pratt algorithm to be transformed using compiler technics.

- Δ**SIZE**$_{\mathbf{CP}}$ (*t*): change of the controller size
- Δ**SPEED** (*t*): change of speed of the overall design
- Δ**POWER** (*t*): change of power consumption

The classification procedure **CLASS**$_{\mathbf{trans}}$ **(*t*)** returns the results of the four functions above:

$$\mathbf{CLASS_{TRANS}}\,(t) = (\Delta\mathrm{SIZE_{DP}}(t), \Delta\mathrm{SIZE_{CP}}(t), \Delta\mathrm{SPEED}(t), \Delta\mathrm{POWER}(t))$$

The results for calculating Δ**SIZE**$_{\mathbf{DP}}$ and Δ**SIZE**$_{\mathbf{CP}}$ are done using SIS [SeSiLaMoMu92].

The results to estimate Δ**POWER** is evaluated using the power estimation tool by Ghosh et.al [GhDe92].

The results to evaluate Δ**SPEED** after applying a transformation is done by counting the states of the FSM, which is created by PMOSS synthesis. The number of states which are visited during execution gives the significant results. This state count needs to be multiplied with the Δ**DELAY** of the data path to get the final speed estimation value.

Regarding again the example. No transformation can be applied to the design. Hence, this section does not present any quantitative values concerning the transformations.

4.3 Applications and Extensions

The approach is an experimental analysis over transformations onto different designs. Applying the original behavioral design representation and the transformed to the synthesis process allows first all design space exploration and experimental comparison of a transformations impact to a design.

The final result of the work will be a data base and an heuristic estimation about the impact of an applied transformation to pattern of constructs in a specification. Having done lots of experimentation a data base will be created which contains all information.

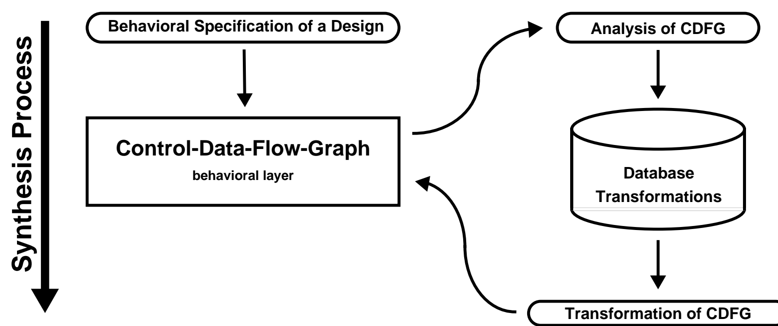


Figure 10. Behavioral transformations for incremental design modification.

Using this stored experimental information a design can be optimized due to the cost function. A mechanism will be integrated in the later process.

The conditional branches are to be analyzed in a later stage of the project.

4.2.3 Block-Level Analysis

This stage analysis covers the investigation of a basic block. A basic block is a code sequence: once entered it is sequentially executed [WaGo85]. A function **getBB(*spec,mod$_i$*)** creates $\mathbf{SET_{BB}}$ which represents all basic blocks (bb_0,...,bb_i) of a module ***mod$_i$*** in the design ***spec***.

On block level the data dependencies have to be analyzed. If there are assignments in the basic block containing the same memory location **M**, the following dependencies are possible concerning **M**:

- $\mathbf{BB_{RAW}}$: a memory location **M** will be first written, then it will be read (read after write),
- $\mathbf{BB_{WAR}}$: write after read,
- $\mathbf{BB_{RAR}}$: read after read,
- $\mathbf{BB_{WAW}}$: write after write.

The basic block will be analyzed according to the domination of one class of dependency. The function $\mathbf{DEP_{BB}}$(*bb*) returns a vector with four items, each counting the occurrence of a dependency presented above. For the classification of basic blocks this is the necessary information. This dependency analysis is required because it is necessary to decide about the possibility to apply a transformation. If a basic block is in a loop body, the dependency analysis provides the information about possible pipelining or parallelism after an unrolling transformation.

Therefore a basic block $bb \in \mathbf{SET_{BB}}$ is classified by the classification function $\mathbf{CLASS_{BB}}$() which returns the results of the described function above:

$$\mathbf{CLASS_{BB}}(bb) = ((\mathbf{DEP_{BB}}(bb), \mathbf{ALG_{BB}}(bb)).$$

Concerning the basic blocks in the Knuth, Morris & Pratt algorithm it can be seen, that the basic blocks in most cases consists out of an assignment. It makes no sense to give all vectors for all basic block, because all transformations which work on this level, constant-propagation and common subexpression elimination, are not applicable in any basic block of the analyzed algorithm.

4.2.4 Behavioral Transformations Impact on HW Implementation

Any hardware implementation is rated concerning three cost functions: size, speed and power consumption. These criteria are sufficient to characterize a behavioral transformation's impact to the synthesis process. A behavioral transformation might speed up the overall speed of the final hardware implementation but increases its size. Behavioral transformations do have a massive impact to the final synthesis result. The reason is the change of the design's structure on a high abstraction level. Using an initial behavioral specification and synthesizing it, gives the initial version of a hardware implementation. Taken the initial behavioral specification applying any transformation will result in an altered final result. The comparison between the altered result with the initial hardware implementation gives the information about the transformation's impact.

This will be reflected by using the Δ-symbol in the following items:

- $\mathbf{\Delta SIZE_{DP}}$ **(*t*)**: change of the data path size

This classification function gives the information to decide later whether a transformation should be applied to the module or not. The transformation which works on this level is function-inlining.

Concerning the system in Figure 7 the classification function results in the following vector.

$$\mathbf{CLASS_{MOD}(kmp) = (1,0,\text{-},\text{-})}.$$

This results in a non applicable inlining transformation, because there is function call, its just a flat design.

4.2.2 Module-Level Analysis

A module in a specification is a function or procedure in the specification.The module contains control flow, i.e., loops, conditional branches, or function calls. Potential degree of optimization is contained in loop structures. In this stage of implementation the loop analysis is done.

Loops consist out of a loop condition and a loop body; they can be divided into data dependent and data independent loops. The decision if and how often the loop body is passed is predictable in data independent loops. The decision in data dependent loops is computed at runtime. The body might contain just loop (nested) or additional control flow (complex) or just a basic block (simple). Hence, whenever a loop exists in a design one can decide among seven classes of loops:

- $\mathbf{LOOP_{CdBs}}$: data dependent condition, simple body
- $\mathbf{LOOP_{CdBn}}$: data dependent condition, nested body
- $\mathbf{LOOP_{CdBc}}$: data dependent condition, complex body
- $\mathbf{LOOP_{CiBs}}$: data independent condition, simple body
- $\mathbf{LOOP_{CiBn}}$: data independent condition, nested body
- $\mathbf{LOOP_{CiBc}}$: data dependent condition, complex body
- $\mathbf{LOOP_{end}}$: endless loop

As stated before, in a data independent loop the number of body passes can be defined a priori. This leads to a classification function for loops which two vector components:

$$\mathbf{CLASS_{LOOP}}(\mathit{loop}) = \mathbf{(class\ of\ loop}, \mathbf{PASS_{BODY}}(\mathit{loop}))$$

The transformation which works on this level is the unrolling of loops. Another transformation on this level is the common subexpression elimination and constant propagation. Both transformation start at a basic block and then cross their boundaries. A block level analysis proceeds the application over the basic block boundaries. This will be stated in the next section.

Regarding now the loops of the overall example in Figure 7. There are two loops. both are depending on the length of the pattern to be searched. Therefore both loop conditions are data dependent, the number of loop body passes is defined at runtime. The first and the second loop contain additional control flow. This leads two the following classification.

$$\mathbf{CLASS_{LOOP}}(\mathit{loop_1}) = (\mathbf{LOOP_{CdBs}}, \mathbf{PASS_{BODY}}(\mathit{loop_1})) = (\mathbf{LOOP_{CiBs}}, \mathbf{undefined})$$
$$\mathbf{CLASS_{LOOP}}(\mathit{loop_2}) = (\mathbf{LOOP_{CdBn}}, \mathbf{PASS_{BODY}}(\mathit{loop_2})) = (\mathbf{LOOP_{CiBn}}, \mathbf{undefined})$$

The impact of the mentioned transformations can be propagated to the lower levels of the design process. This allows to investigate the impact of a transformation an algorithmic level on the results of register-transfer-level.

4.2 Method

Due to structural aspects a behavioral specification is usually written hierarchically. This allows better detections of occurring faults and the reuse of specifications. This hierarchy needs to be investigated. Control which is invented in each of the hierarchy modules is significant for the opportunity to apply any transformation on this level. Finally a basic block has to be researched for further optimization. This shows the importance of classifying different levels of a specification. This leads to three levels which are to be analyzed: System-, Module- and Block-Level.

The current state of implementation contains as a set of transformations the common compiler transformations, function inlining, common subexpression elimination, constant propagation and loop unrolling.

4.2.1 System-Level Analysis

The system level analysis starts with a static analysis of a call hierarchy. The analysis of the global module inter dependence is known from HW/SW-Codesign of [HaCa94].

A hierarchical design is specified with functions or procedures which are called *modules* in further discussion. A function **getMOD(*spec*)** creates the set $\mathbf{SET_{MOD}}$ which represents all modules (mod_0,...,mod_i) of the design ***spec***. The activation of a module mod_j by a module mod_i is presented by the relation *access*(mod_i, mod_j). The set $\mathbf{E_{MG}}$ represents all *access*-relations in ***spec*** and it is created by the function **getACCESS(*spec*)**. Using these two sets one is in the position to create a directed module graph representing a hierarchical design:

$\mathbf{MG}(V_{MG}, E_{MG})$:

- $\mathbf{V_{MG}}$, set of a nodes representing $\mathbf{SET_{MOD}}$ all modules of the design(mod_0,...,mod_i)
- $\mathbf{E_{MG}}$, set of edges representing represents all *access*-relations in ***spec***. The edges will be weighted to express the number of activation of the module mod_j by the module mod_i.

A module at system level is classified after analyzing the created module graph.

- $\mathbf{ACT_{MOD}}$(*mod*): denotes the number of static activations of a module.
- $\mathbf{LOC_{MOD}}$(*mod*): counts the number of locations from which a module is called.
- $\mathbf{CNT_{REG}}$(*mod*), $\mathbf{CNT_{FU}}$(*mod*)): the analysis of the module's size is necessary. The number of registers $\mathbf{CNT_{REG}}$ and functional units $\mathbf{CNT_{FU}}$ for a potential hardware implementation of the module is calculated by the PMOSS synthesis system.

Therefore a module $mod \in \mathbf{SET_{MOD}}$ is classified with the function $\mathbf{CLASS_{MOD}}$() which returns a result vector defined by the functions above:

$$\mathbf{CLASS_{MOD}}(mod) = (\mathbf{ACT_{MOD}}(mod), \mathbf{LOC_{MOD}}(mod), \mathbf{CNT_{REG}}(mod), \mathbf{CNT_{FU}}(mod)).$$

4 Behavioral Transformations

The approach to transform a given design on the behavioral level is common and used in the synthesis community. Transformational design modification in this process is to be understood as a change of the control and data flow in a design by preserving its overall behavior. These behavioral transformations are starting point of the High-Level-Synthesis. Behavioral transformations alter the control or data flow but preserve the specification's behavioral.

Current synthesis tools usually apply there transformations on the behavioral level in a predefined order to the synthesis process. The technique of a prefixed order of transformations is being used in code optimizing compilers as well. For example during a compilation the public domain *GNUCC* [GnuCC] applies optimization procedures in a static order. Synthesis tools usually use simple compiler optimization technics, i.e., constant propagation, common subexpression elimination and loop unrolling. The *OLYMPUS* Synthesis System [DeMi90] shows a static ordering of common compiler optimizations in its behavioral synthesis part *HERCULES* [DeMi88]. The *System Architect's Workbench* [Tho88], [WaTh89] allows interaction with the designer to apply special transformations in order to explore the design space. The *Hyper-LP* system [Cha92] concentrates the applied transformations due to one aspect of optimization, power consumption. The *FLAMEL* approach is to introduce a higher degree of parallelism into the design [Trick87]. DSP applications have certain characteristics like single assignment or nested loops. The synthesis systems of the *Cathedral* family [Van93] have been developed to deal with DSP applications, using their characteristics. The transformations which are applied in these systems deal with the significant structure of DSP behavior. For example loop folding introduces parallelism and pipelining.

These synthesis systems do have several drawbacks: some are restricted in their applications (Cathedral), others do not allow flexible ordering of the transformations (HERCULES) or the transformations are included in the synthesis process to improve just one certain aspect (Hyper-LP).

The impact of a transformation on this high level might cause a massive impact to the further synthesis process and the final chip. If a transformation is not applicable it does not have any impact at all. Very often transformations do cause trade-offs. The impact of a transformation result in an improvement concerning one aspect causes disadvantages on others (speed / size). One difficulty in using any behavioral transformation is that the estimation of its impact to the final result of compilation concerning lower levels is very difficult. Therefore, a detailed analysis of behavioral transformations is necessary.

This work deals with an experimental analysis of the correlation between a set of transformation at three different levels of a given specification presented as a CDFG. The experimental results are used to determine an order of the transformations for a particular design.

4.1 Input and Output Specification

As stated in the introduction and as it can be seen in Figure 10 the input for a behavioral transformation is a the representation of the design on the CDFG, the behavioral level of PSF. Any transformation which gets as input a CDFG, and return a possibly altered CDFG. This altered CDFG is data base to the further synthesis process.

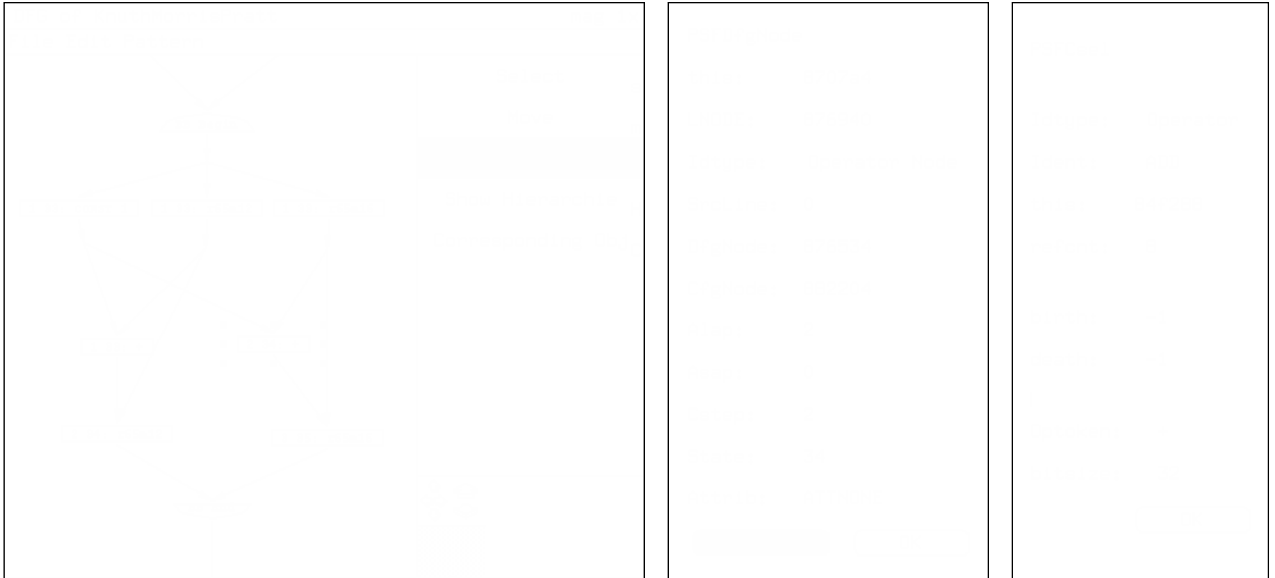


Figure 9. Inspection of scheduling of a part of the algorithm in Figure 7.

In the left most picture you can see the data flow graph. The editor option “show data” makes it possible to prove the annotations of the selected node. In Figure 9 you can see the value for the asap and alap values (to calculate the mobility for other scheduling algorithms), and the value for the “resulting” control step. In addition the symbolic state of the add operation is displayed. This is used by the forthcoming creation of the FSM. In the right most display, there are more detailed informations of the add operation, i.e. bitwidth.

Further extensions to be applied in the later PMOSS version, are different structural synthesis algorithms. This will provide a comparison between different algorithms and their impact to the final synthesis result.

3.4 Limitations

The front ends which have the task to transfer a behavioral specification into the CDFG are to be extended. Their first versions do not cover for example the whole definition of the C language. Therefore there exists in the current state a limitation concerning the specification.

Forthcoming research activities will apply a wide range of applications onto PSF from HW/SW-Codesign [HaCa93] to sequential logic synthesis [FeFr93].

caller remains in a wait state until the submodule executing sends a ready signal after finishing its function.

3.3 Applications and Extensions

The overall examples, which will be referenced in this report is the string matching algorithm by Knuth, Morris & Pratt presented in the following Figure 7.

This algorithm is used in the UNIX command “grep” to find strings in files.

The structural synthesis application which is done by PMOSS results concerning the Knuth, Morris & Pratt Algorithm in the following Figure 8.

Benchmark				**Time consumption^{x}**					**Synthesis Results**			
	lines of C-Code	**DfgNodes**	**CfgNodes**	**Static List Scheduler**	**Greedy FU Bindung**	**Register Binding**	**Interconnection**	**Conversion Netlist/FSM**	**#Register**	**#Mux**	**area**	**states in the FSM**
Knuth,Morris & Pratt Algorithm	**27**	**160**	**29**	**69**	**9**	**79**	**19**	**89**	**58**	**222**	**35214**	**49**

Figure 8. PMOSS structural synthesis results concerning Knuth, Morris & Pratt String Matching Algorithm. (^{X} runtime on a SUN SPARC 10 in milliseconds measured with rusage)

The PSF supports a visual inspection of the structural synthesis tasks. There exists a powerful editor which supplies a control of the CDFG’s interdependence. To analyze the synthesized finite state machine and its state transition graph there exist another editor.

The editor’s realization is based on the class libraries InterViews [IV91] and Unidraw [Uni89] developed at Stanford University. InterViews is a library of C++-classes to develop X-Window applications. Unidraw is some kind of toolbox for building domain specific editors, based on InterViews. It provides complex methods, i.e. for 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.

Due to the sophisticated displaying facilities of the editor concept the user gets in addition to visualize the graphs of the PSF the opportunity to partially control his applications. As an example Figure 9 shows the informations of the ‘add’-operation in our example program after scheduling.

The default synthesis flow calculates the final schedule of each operation in the current state of implementation by list scheduling. This static list scheduling integrates the functional unit allocation. This is done simultaneously. The functional unit allocation approach provides resource sharing. The task of allocation is done straightforward. The allocator looks in the library and picks the first module type which could 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.

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. The usage of an algorithm is just restricted concerning its precondition.

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) into CDP. The annotation scheme is one of the key ideas for getting a bunch of algorithms which are exchangeable. The exchange of design data is done only via the CDFG.

```
void KnuthMorrisPratt() {

    const max_str_length = 30;
    int kmpsearch = patlngth = strnlngth = 0;
    int i = j = 1;
    char pattern [max_str_length];
    char strn [max_str_length];
    char next [max_str_length];

    cout << "Enter length of string: ";
    cin >> strnlngth;
    cout << "Enter length of pattern: ";
    cin >> patlngth;
    cout << "Enter string: ";
    cin >> strn;
    cout << "Enter pattern: ";
    cin >> pattern;

    if (patlngth >= 1)
        next[0] = 0;

    while (i < patlngth) {

        if ((j == 0) ||
            (pattern[i-1] == pattern[j-1])) {
            i++; j++; next[i-1] = j;
        }
        else {
            j = next[j-1];
        }
    }
```

```
    i = 1; j = 1;
    while ((j <= patlngth) && (i <= strnlngth)) {

        if ((j == 0) || (strn[i-1] == attern[j-1])) {
            i++;
            j++;
        }
        else {
            j = next[j-1];
        }
    }

    if (j > patlngth)
        kmpsearch = i - patlngth;
    else
        kmpsearch = i;

    cout   << "Index of pattern is: "
           << kmpsearch << "\n";
}
```

Figure 7. Knuth, Morris & Pratt String Matching Algorithm.

Our approach is based on synthesis of functions as called variable delay macro as introduced by Camposano [CaVE87]. The called submodule is activated by a start signal of the calling module. The

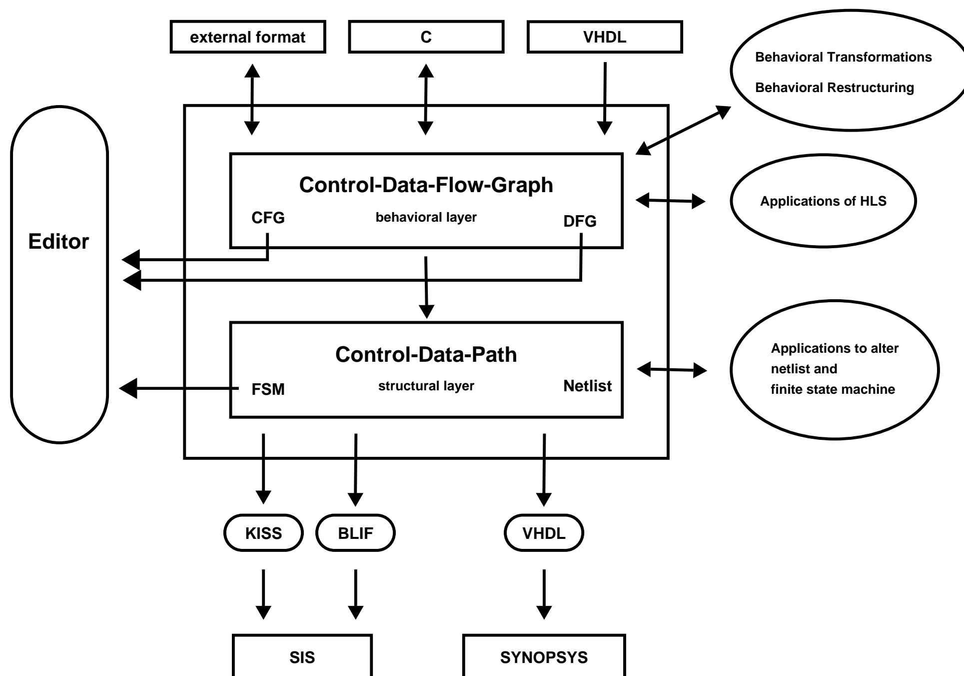


Figure 6. The PMOSS synthesis system.

To increase the PSF's usability, there have been back ends implemented to interface external tools like the logic synthesis system SIS [SeSiLaMoMu92] or SYNOPSYS [SynDC92]. These back ends produce data exchange formats which can be interpreted by external tools. The aspect of HW/SW-Codesign is supported by two back ends which produce external code. C is sufficient to give a description on software level. VHDL is well suited to give a description on hardware level. Figure 6 shows the overall structure of the PSF. PMOSS allows to apply HW/SW-Codesign applications on both layers.

Besides the various front- and backends there exist an external format for the behavioral layer of the PSF. Due to this opportunity for backup and restore, applications running on the PSF can be combined by the UNIX script-mechanism. This results in a lesser storage requirement and a higher degree of efficiency.

3.2 Method

One major objective in developing the modular fashion of PMOSS was to separate the synthesis tasks. This allows flexibility to combine and to compare different algorithms. Using the modular fashion of the PSF synthesis the user picks the appropriate algorithms to satisfy his need, constraints and so on. Another advantage: suppose the impact of two behavioral transformations onto the data path need to be analyzed. Hence, the creation of the controller in not necessary. This reduces the time consumption for analysis enormously. Beside higher flexibility in synthesis this approach leads to a higher degree of code reusability.

3 Structural Synthesis

State-of-the-Art synthesis systems try to allow IC design independent of the specification language. Often it is necessary to implement a system's specification in one language to use commercial tools. In "classical" task of design automation like processor design the usage of one specification language is no disadvantage because the processor designers do usually have high skills in hardware design languages (e.g., VHDL, Verilog). Current trends are to synthesize parts of a design which were former implemented in software into hardware. This requires that programming languages which are used for software design are to be taken in consideration in hardware design approaches. Under the catchword HW/SW-Codesign there have been several publications [HaCa93], [ErHe92], [GuMi92].

The presented Paderborn Modular Synthesis System, PMOSS for HW/SW-Codesign and Structural-Synthesis is based upon the Paderborn Synthesis Format PSF. PSF which is a further development of the Internal Synthesis Format, ISF [Ho93]. The PSF satisfies the needs of a uniformed data format for Structural-Synthesis, HLS. Rundensteiner & Gajski [RuGa90] and Camposano & Tabet [CaTa89] describe the necessity of such a data base.

The PSF presents two representations in different levels of abstraction to provide these synthesis algorithms an abstraction level which satisfies their needs. The behavioral description of a system will be transferred into a Control-Data-Flow-Graph, CDFG. The Control-Data-Flow-Graph of the PSF consists out of two hierarchical graphs, the data flow graph, DFG and the control flow graph, CFG. Both together present a behavioral description of the system. A hierarchical component is to be understood as a function or procedure in the specification. The opportunity to cover hierarchy in the CDFG's representation is necessary to reflect the software orientated requirements. This supports exposition of information which is not done by inlining. Inlining is the flattening of function calls into a non hierarchical form. The behavioral layer is data base for applications which alter the design by e.g. transformations and partitioning. This will be presented in Section 4 and Section 5. The structural description is presented by a Control-Data-Path, CDP. This style of representation is graph based as well and consists out of hierarchical netlist and a finite state machine. This data structures are very efficient to perform applications on register transfer level. It has been used successfully already in retiming and CP/DP partitioning [EiCa93].

PMOSS synthesis approach annotates the CDFG with several information. The PSF synthesis process' major concept is modularity. This approach has been taken to provide optimal succession of each task and an optimal choice of synthesis. The CDFG is to be assigned with synthesis informations like scheduling steps, functional units to perform an operation and its instance etc. Each algorithm can be started after the required data has been assigned to the CDFG.

3.1 Input and Output Specification

To present the overall structure and to explain the input/output specification Figure 6 gives an overview of the used structural synthesis system PMOSS.

The front ends which transfer a system specification into the behavioral layer of the PSF takes the HW/SW-Codesign into account by using a software language as well as a hardware description language as input. Systems which are specified in C can be handled with the Control-Data-Flow-Graph. An additional front end exists to transfer VHDL descriptions into the CDFG of the PSF. This second input path provides the combination of software and hardware descriptions.

2.4.1 Overall limitations

The codesign task starts form $Spec_C$. This restriction results from the initial goal of the codesign task. Approaches focused on other aspects, e.g., checking time constrains, may start from other specifications. Here general computing hardware was combined with special extensions and $Spec_C$ has been found a suitable specification base.

The presented implementation is restricted to one target architecture. But the concepts can be applied to other architectures as well. Only the static specification pass and the HW/SW interface protocol are to change.

Other HW/SW- interface solutions can be considered also. This will change the constants used by the partitioning cost function. Nevertheless, the presented concepts are applicable in principle.

2.4.2 Static analysis limitations

Until now, static specification analysis evaluates heuristic criteria. No comparison of HW and SW implementation is performed. Also further aspects, e.g., regularity, similarity, operation mix etc. could be examined during this codesign pass.

The approximated runtime of SW implementation is considered as a lower bound. This is based on the summarizing algorithm and no formal proof is presented.

Static specification analysis can only be applied to assembler code. Design parts which are compiled to libraries cannot be examined. This seems to be reasonable because library functions depend very often on the operating system.

2.4.3 Dynamic analysis limitations

A limitations of this codesign pass can be seen in the profiling procedure based on “C”-functions. However, this is for technical reasons only. A more fine granulated profiler which examines each line of code explicitly can be also applied in principle. Only implementation versions of the compiler and the profiler must be suitable.

Perhaps there are only inputs for dynamic analysis used, which will not cover all specification code. Our method performs no code coverage check, because this is a different subject. But a simple control mechanism could be helpful.

Dynamic analysis is based on elements of statistic theory, but no accuracy check of the computed expectation values is performed. This would be helpful, to control the data input set given to this codesign pass.

2.3 Applications and Extensions

Our approach was applied to a set of benchmarks of reasonable complexity [Hardt94]. For example, the HW/SW partitioning of the well known and frequently used UNIX command “grep” was examined in more detail. In Table 12 are some characteristics of this tool listed:

characteristic of UUC “grep”	
lines of $Spec_c$	12.902
lines of pec_{ASI}	35.081
size of executable in byte	204800

Table 12. Characteristics of “grep”.

UUC	#mod.	$Stat$	Dyn	$Stat \times Dyn$	$Intf$	Ψ
grep	173	122	29	16	5	1

Table 13. Specification analysis results

This design is given to the codesign task which applies static and dynamic specification analysis. The result is given in Table 13. First, the number of regarded modules is listed. For reasons of simplicity a module was understood as a hierarchical element of the description language (function). The number of modules found suitable for HW implementation by static ($Stat(mod) > 0$) and by dynamic ($Dyn(mod) > 0$) specification analysis is listed explicitly. In addition the correlations of both passes is presented ($Stat(mod) \times Dyn(mod) > 0$). Then the number of modules suitable for HW implementation by interface analysis $Inf(mod)$ and the final result $\Psi(mod)$ are displayed.

Only one module was found suitable for HW implementation and the design spaces was reduced reasonable. As mentioned above static analysis provides the weakest partitioning criteria. Interface costs are pointed out as a main problem for a HW implementation. 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 (see e.g. [CoLeRi90]). We chose the variant by Knuth, Morris & Pratt for HW implementation which is presented in Figure 7. Table 14 lists the module’s characteristics found by passing it through the previously described analysis.

For all other modules the partitioning cost function Φ results to zero. A quantitative interpretation of

HW partition of grep	lines of $Spec_C(KMP)$	lines of $Spec_{ASM}(KMP)$	$Stat(KMP)$	$Dyn(KMP)$	$\Phi(KMP)$
module KMPexec	145	332	21	55	615

Table 14. Characteristics of identified HW- partition.

these numbers is subject of current work. However, the codesign task results in a correct HW/SW partitioning, the central algorithm is automatically detected by specification analysis and reasonable overall speed-ups are expected to be found during synthesis of the HW partition.

2.4 Limitations

This subsection points out the most severe limitations of our codesign approach.

Definition 11: The function $Intf_{limit}: UUC \rightarrow IR$ is defined as:

$$Intf_{limit}(mod) = \text{INT_MAX_COST} - Trans(mod)$$

While $Intf_{limit}(mod)$ is not negative, the regarded module is acceptable for HW implementation, otherwise $Intf_{limit}(mod)$ is set to zero. The overall transportation costs depend on the number of module accesses $Acc^{SW}_{dyn}(mod)$ which can be determined by dynamic analysis.

Definition 12: If $L_{Acc^{SW}_{dyn}}(mod)$ is the list of data items $Acc^{SW}_{dyn}(mod)$ of module $mod \in UUC$, the expected value is $EAcc^{SW}_{dyn}(mod) = E(L_{Acc^{SW}_{dyn}}(mod))$.

The overall interface costs of a given module are defined by function $Intf_{total}(mod)$.

Definition 13: The function $Intf_{total}: UUC \rightarrow IR$ summarizes the interface costs for a given module $mod \in UUC$:

$$Intf_{total}(mod) = EAcc^{SW}_{dyn}(mod) \times Trans(mod)$$

The algorithmic determination of this interface cost function is under development.

2.2.4 Partitioning Cost Function

Now, the results of static and runtime specification analysis and the interface costs are correlated. The partitioning cost function $\Phi(mod)$ decides for a given module $mod \in UUC$, if a HW implementation should be considered.

Definition 14: The **partitioning function** $\Phi: UUC \rightarrow \{SW, HW\}$ is defined as:

$$\Phi(mod) = \begin{cases} HW \Leftrightarrow \dfrac{Stat(mod) \times Dyn(mod) \times Intf_{limit}(mod)}{Intf_{total}(mod)} > 0 \\ SW \Leftrightarrow \dfrac{Stat(mod) \times Dyn(mod) \times Intf_{limit}(mod)}{Intf_{total}(mod)} \leq 0 \end{cases}$$

The suitability of a module for HW implementation found by static analysis is related with the result of dynamic analysis and divided by the interface costs. If the overall interface costs are too high $Intf_{limit}(mod)$ results to zero. This partitioning function splits the UUC into a HW part P_{HW} and a SW part P_{SW} so that $UUC = P_{HW} \cup P_{SW}$ and $P_{HW} \cap P_{SW} = \emptyset$ holds. Φ is evaluated for all modules in the module-graph and the evaluation result is stored in this data structure.

This HW/SW- partitioning process selects a comparable small set of modules which are good candidates for HW implementation. Obviously, specification analysis can be done much faster than design synthesis. So candidate selection is important.

These correlation results measure the importance of the given module for the overall runtime behavior and they are combined by the function $Dyn(mod)$. Here normalization factors are also used, in order to direct the partitioning subtask.

Definition 9: The function $Dyn: UUC \to IR$ summarizes the suitability of module $mod \in UUC$ for HW implementation found by dynamic analysis:

$$Dyn(mod) = \frac{Dyn_{abs}(mod)}{\text{SW_ABS_RT}} + \frac{Dyn_{average}(mod)}{\text{SW_AVERAGE_RT}} + \frac{Dyn_{rel}(mod)}{\text{SW_REL_RT}}$$

The algorithm performed during dynamic specification analysis, given in Figure 4, performs first profiling data generation. Then $Dyn(mod)$ based on the above mentioned statistic definitions is computed for each module in the module-graph.

```
DynAnalysis(Spec_c(UUC)) {
    for all val_i ∈ input_data_set {
        execute_sw_application(val_i, data_file);     // generate profiling data
        add_data_to_module_graph(data_file);          // update module-graph
    }
    for all mod_i ∈ module_graph {
        compute_dyn_abs(mod_i);          // determine absolute dynamic runtime Dyn_abs(mod_i)
        compute_dyn_average(mod_i);      // determine average dynamic runtime Dyn_average(mod_i)
        compute_dyn_rel(mod_i);          // determine relative dynamic runtime Dyn_rel(mod_i)
        compute_dyn(mod_i);              // determine dynamic runtime cost criteria Dyn(mod_i)
    }
}
```

Figure 5. Dynamic specification analysis algorithm.

2.2.3 HW/SW- Interface Costs

Specification analysis must also take the interface costs into account. For quantification several reasons causing interface costs have to be distinguished. Global data and parameter passed to and returned from a module $Trans^{HW}_{direct}(mod)$, accesses to main memory $Trans^{HW}_{memory}(mod)$ and calls of further submodules $Trans^{HW}_{indirect}(mod)$ are time consuming actions. Such costs are not only known from HW/SW- codesign. Data transports are expensive for SW implementations as well as for HW implementations. The transport costs for SW implementations are named analogously $Trans^{SW}_{direct}(mod)$, $Trans^{SW}_{memory}(mod)$ and $Trans^{SW}_{indirect}(mod)$.

Definition 10: The function $Trans: UUC \to IR$ expresses HW transportation cost in relation to the SW transportation costs:

$$Trans(mod) = \frac{Trans^{HW}_{direct}(mod) + Trans^{HW}_{memory}(mod) + Trans^{HW}_{indirect}(mod)}{Trans^{SW}_{direct}(mod) + Trans^{SW}_{memory}(mod) + Trans^{SW}_{indirect}(mod)}$$

The maximal acceptable interface costs are limited by a constant (INT_MAX_COST) (Table 11).

can be handled by normalizing module characteristics. The normalization base defined by the designer (constant) is input to the partitioning process. The values listed in Table 11 lead to results listed in Table 13 but the designer may change them also for further design space exploration.

constant	value	note
SW_APPROX_RT	1000	This constant defines the intended size of the approximated software runtime in cpu cycles.
SW_ABS_RT	10	The intended absolute runtime of a given module is set to 10 ms.
SW_AVERAGE_RT	5	Normalization factor for the average module runtime in ms.
SW_REL_RT	5	Definition of the intended relative module runtime in percent of the design runtime.
DYN_CONFIDENCE	3	Maximal value of σ/E indicating the confidence of dynamic analysis.
INT_MAX_COST	3	Interface costs of a HW implementations may not be higher than 3 times the costs of a SW implementation.

Table 11. Constant definitions.

During runtime analysis profile data is classified by statistical aspects. The terms probability P, the expected value E and standard deviation σ are well known from statistical theory e.g. [Li62]. To apply these UUCs behavior have to be examined with a set of input data which is large enough to indicate confidence. This set of input data is understood as random variable, the examination of a UUC as experiment. An experiments confidence is heuristically measured by the quotient σ/E. The maximal acceptable value DYN_CONFIDENCE depends on the characteristics (e.g., size) of the input data and may be defined by the designer (Table 11). Note, it is not intended to generate statistical independent test patterns, but to avoid design decisions caused by invalid or hardly relevant data.

Definition 6: If $L_{RT^{SW}_{abs}}(mod)$ is the list of data items $RT^{SW}_{abs}(mod)$ of module mod, the expected value is $ERT^{SW}_{abs}(mod) = E(L_{RT^{SW}_{abs}}(mod))$.

In the same way $ERT^{SW}_{average}(mod)$ and $ERT^{SW}_{rel}(mod)$ are defined.

Definition 7: If L is a list of data values and $E(L) \neq 0$, the **confidence** $Conf(L)$ is defined as:

$$Conf(L) = \text{DYN_CONFIDENCE} - \frac{\sigma(L)}{E(L)}$$

If the regarded profiling data is not acceptable this value becomes negative and it is set to zero. This avoids the influence of irrelevant profiling data if the expected values of the absolute, average and relative runtime are correlated with their confidence.

Definition 8: $Dyn_{abs}(mod) = ERT^{SW}_{abs}(mod) \times Conf(L_{RT^{SW}_{abs}}(mod))$ [1]

$Dyn_{average}(mod)$ and $Dyn_{rel}(mod)$ are defined analogously.

1. $Dyn_{abs}(mod)$: result of **dyn**amic specification analysis concerning **abs**olute module runtime

Definition 4: The function $Stat: UUC \rightarrow IR$ summarizes the suitability of module $mod \in UUC$ for HW implementation found by static analysis with respect to the normalization factor:

$$Stat(mod) = \frac{RT^{SW}_{approx}(mod) \times Ctrldom(mod)}{\text{SW_APPROX_RT}}$$

with

- RT^{SW}_{approx} **approx**imated **runtime** of **SW** implementation
- $Ctrldom$ **con**t**r**o**l** **dom**inance, defined by the percentage of jump instructions per module

The algorithm performed during static specification analysis, given in Figure 4, transforms first $Spec_c(UUC)$ into $Spec_{ASM}(UUC)$. Then each occurrence of all modules is examined.

```
StatAnalysis ( Spec_c (UUC) ) {
    Spec_ASM (UUC) = GNU_gcc-S(Spec_c (UUC) ;           // compilation using GNU compiler
    for all module_occurencies m_i ∈ Spec_ASM (UUC) {
        if (m_i ∉ module_graph )                         // module definition found
            create_module(m_i);
        if (occurence = module_definition ) {
            approx_sw_runtime(m_i);                      // perform runtime approximation
            compute_ctrldom(m_i);                        // compute control dominance
        }
        else
            create_edge(calling_mod, m_i);               // call of a module found
    }
}
```

Figure 4. Static specification analysis algorithm.

2.2.2 Runtime Analysis

Further specification analysis examines the runtime behavior of a given UUC during SW execution. Profiling data generated dynamically is analyzed. Considering a single module, three aspects, the absolute, average and relative runtime of a module are taken into account.

Definition 5: Considering SW implementation,

- the time consumed by a module mod during the whole design execution is called **absolute runtime**, formally: $RT^{SW}_{abs}(mod)$.
- the average of all execution times of module mod found during design execution is called **average runtime**, formally: $RT^{SW}_{average}(mod)$.
- the percentage of a modules absolute runtime to the design execution time is named **relative runtime** of module mod, formally: $RT^{SW}_{rel}(mod)$.

Often the designer will guide the partitioning subtask by predefined restrictions. Typical examples are maximal or minimal module size. This can be done by parameters given to the subtask indicating the optimization priority, e.g., time vs. size [DeMi90] or many partitions vs. few. On the other hand relative directives are not always sufficient especially if there are hard constraints for each partition. This

module is labeled for implementation either in HW or in SW and the module-graph can be given to further design processes. The next subsection describes each codesign subtask and the partitioning cost function in detail.

2.2 Methods

The analysis subtasks consist of two main phases: static- and runtime- specification analysis. Both start from the system specification and their results are collected in the module-graph representing the design. In Figure 3 the analysis scenario is given.

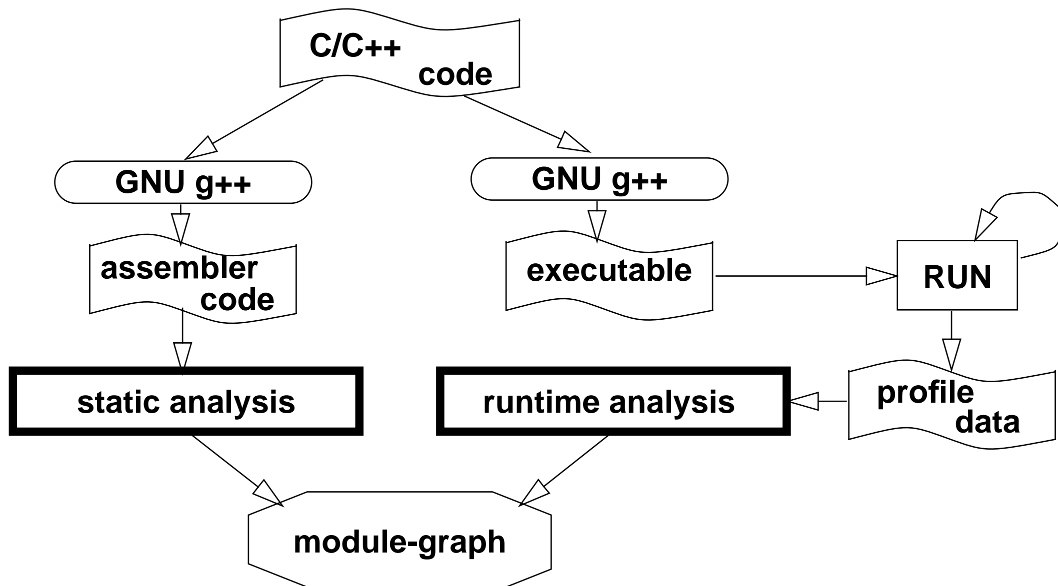


Figure 3. Codesign analysis scenario.

2.2.1 Static Specification Analysis

Static specification analysis computes an approximated lower bound on the execution time for the given specification running on the unextended target architecture. This brings the best case of SW implementations into view and shows limits of this kind of implementation. At this point, a HW implementation will lead to a raise of the overall system performance. Additionally, some specification characteristics, e.g., control dominance are examined. Earlier experiments have shown that the reached speed-up by introducing a special function unit is high if the control dominance is high [HaCa93]. For instance, consider an operation comparing two single bit. This can be done in HW very fast by simple gate. A SW implementation for a pipelined architecture needs at least one cycle.

Furthermore, it has been found that reasonable speed-ups can be reached only if the approximated runtime is not to small. The appropriate size is defined as constant (SW_APPROC_RT) (Table 11) by the designer. These effects lead to a result function of static analysis measuring the **suitability** of a given module for HW implementation:

2.1 Input and Output Specification

Now input and output of our codesign task formulated as function Ψ are defined in a formal manner.

Definition 1: The **specification** of a design UUC[1] is called $Spec_x(UUC)$, with $x \in \{\text{C,C++,VHDL,CDFG,ASM}\}$[2], a set of well known specification languages for either HW or SW.

A sequence of constructs of $Spec_C$ is referred to as a **module.** A module may initially be seen as a function of the description language. Such a module definition is random like and can be used as experimentation base. But specification analysis is not limited to restrictions like this.

Definition 2: A unit under codesign $UUC \in Spec_C$ is defined by the set of all modules. The codesign task bipartitions the input specification and is defined as function $\Psi\text{:Spec}_C \rightarrow Spec_C \times Spec_C$:

$$\Psi(UUC) = \{Spec_C(mod) : \Phi(mod) = \text{HW}\} \times \{Spec_C(mod) : \Phi(mod) = \text{SW}\}$$

with $mod \in UUC$ and the partitioning function $\Phi: UUC \rightarrow \{HW, SW\}$ defined later on.

This definition restricts the codesign task's input to $Spec_C$. But $Spec_C$ is a very common programming language which is used for description of a wide variety of applications of different architectures and enables us to examine many different applications. Ψ divides the given specification into two disjunct parts for HW and SW implementation, respectively. The software can be compiled from $Spec_C$ easily and hardware can be synthesized from $Spec_C$, e.g., by PMOSS (subsection 3).

The partitioning function Φ is based on the different phases of specification analysis described separately in subsection 2.2. Each phase's results are stored in a graph based data structure called module-graph. This data structure is defined next.

Definition 3: The activation of module mod_j by module mod_i is called $access = (mod_i, mod_j)$. The graph $MG = (UUC, E_{MG})$ with a set of nodes UUC representing all modules and a set of edges $E_{MG} = \{access_0, access_1, ..., access_{m-1}\}$. MG is called **module-graph**.

For example, consider a simplified version of a design PM performing a pattern matching task.

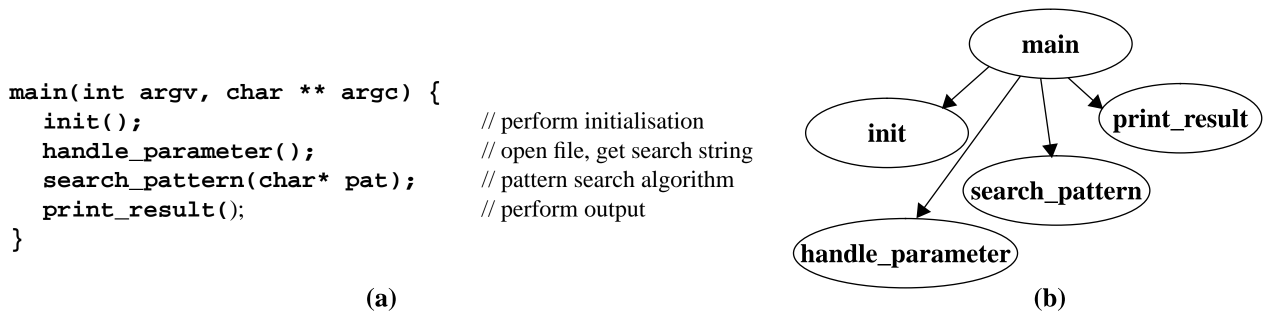


Figure 3. (a) Simplified design specification $Spec_C(PM)$ and (b) the corresponded module-graph

Figure 3 shows $Spec_C(PM)$ and the corresponding module-graph. During the codesign task each

1. <u>u</u>nit <u>u</u>nder <u>c</u>odesign
2. <u>ASM</u> stands for assembler code of the SPARC processor.

2 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 the 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 it is intended 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) as displayed in Figure 2 (a).

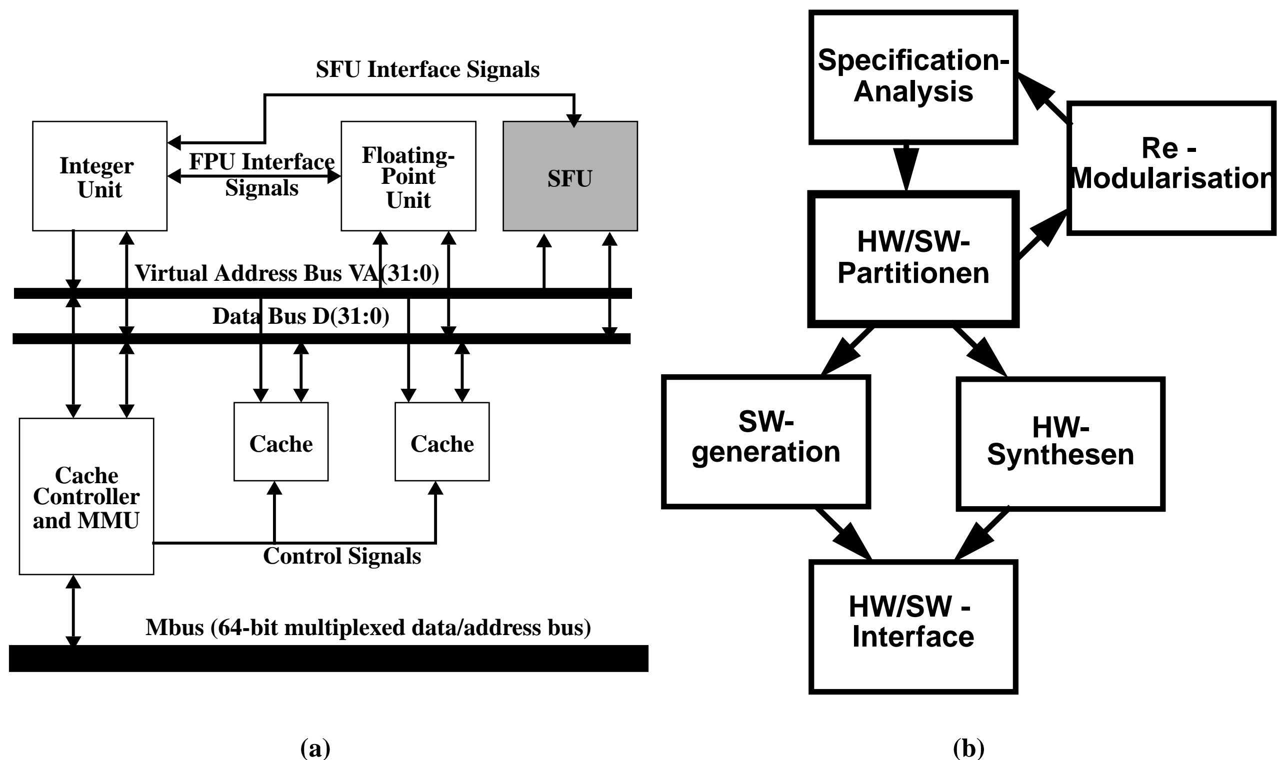


Figure 2. (a) Target architecture and (b) Codesign subtasks.

The codesign task maps one part of a given specification to the general purpose processor and the other to the SFU. The codesign task itself may be divided into several subtasks. Specification analysis should provide all information needed to partition the specification into a HW and a SW part heuristically. The partitioning subtask results into a specification part which is to implement in HW and a second part which is to implement in SW. The HW part is input to the next synthesis pipeline stage, high level HW synthesis. The HW part communicates with the SW part via a pipelined interface. Figure 2 (b) shows these subtasks and their dependencies. Re-modularization widens the design space extremely. But the concepts of codesign presented here are independent from re-modularization which is subject to further research.

that may be processed by subsequent stages. Each chapter is dedicated to a special design task, starting from system-level synthesis going down to adjustments at logic level and resynthesis.

- System-level synthesis extracts parts of the overall behavior to be realized as an ASIC in the hardware target environment (e.g., processor core). Therefore, it can be seen as a partitioning process separating hardware and software parts.
- High-level or behavioral synthesis which has been the object of extensive investigation both in the academic environment [DeMi90] and in industry transforms a behavioral specification of the hardware in terms of a hardware description language (HDL) into a structural interconnection of pieces of modules that may be mapped efficiently onto silicon.
- Behavioral transformations accompany behavioral synthesis and involve optimization techniques similar to those used in optimizing compilers, such as dead code elimination, common subexpression elimination and constant propagation. In addition, it includes hardware-oriented optimizations such as procedure in-line expansion, loop unrolling, and meta-variable evaluation. Additionally, the behavioral representation can be altered in order to simplify or guide the subsequent synthesis process.
- Once behavioral synthesis is completed, datapath and sequential logic synthesis map the optimized intermediate structural representation into a hardware realization.
- Since the result may not be acceptable from a performance and/or area standpoint, partial resynthesis of the structure is employed.

This process can be interpreted as an iterative refinement of the structural model, guided by the estimates on area and timing provided by structural synthesis. Hence, there is a strong need for accurate modeling of hardware behavior at different stages of the synthesis pipeline and for a focus on the transformations that preserve the functionality without changing the expected behavior, and the mapping of behavior into a variety of target architectures.

We will introduce a design-flow model that allows for incremental modifications at each stage of the pipeline. We will also address the hardware description language problem which is caused by the need to represent the model at different stages of the design flow in different domains. We will demonstrate the scope of our methods in terms of an example complex settled at the system-level. By this approach we enlarge the scope of methods which have been presented in [DeMi90] to system level design. At each level of the synthesis process there is a corresponding intermediate representation that serves as the abstraction on which optimizing transformations may be carried out. On system and behavioral level this is captured by a combined control- and dataflow graph. In the structural domain the functionality is represented as state-diagrams and netlists.

We will present our proposed methods in terms of an ongoing example. The input of the system is the source code for the UNIX command “grep”. The algorithmic kernel of the program is a pattern matching algorithm which will be synthesized as a hardware component passing through the above mentioned stages. Besides the procedures, different design alternatives, and fallacies and pitfalls will be discussed, as well.

1 Introduction

Recent trends show that the process of digital system synthesis is more and more embedded in other industrial processes, e.g., in the design of complex mechatronic devices such as engines to control the fuel injection. The interaction of the digital hardware with components from other domains (mechanical, analogous systems, or software) requires the synthesis process to be highly flexible in coping with cost and performance constraints. Therefore the synthesis should enable the designer to choose from a variety of design alternatives. For example, the designer may choose a processor core to implement the software and an ASIC to implement the additional hardware; if a rapid evaluation is desired the user might choose FPGA to realize this additional part of the overall system. The knowledge about the target architecture can be involved in the synthesis process for optimization purposes. In this paper a description of a system for the computer-aided design and synthesis of digital systems is provided, covering a bunch synthesis tasks, i.e., varying from system-level to logic-level synthesis. The proposed methodology is developed as part of the SFB project "*Automated System Design*" and consists of three major components: system-level, high-level and logic (re-)synthesis, as depicted in Figure 1.

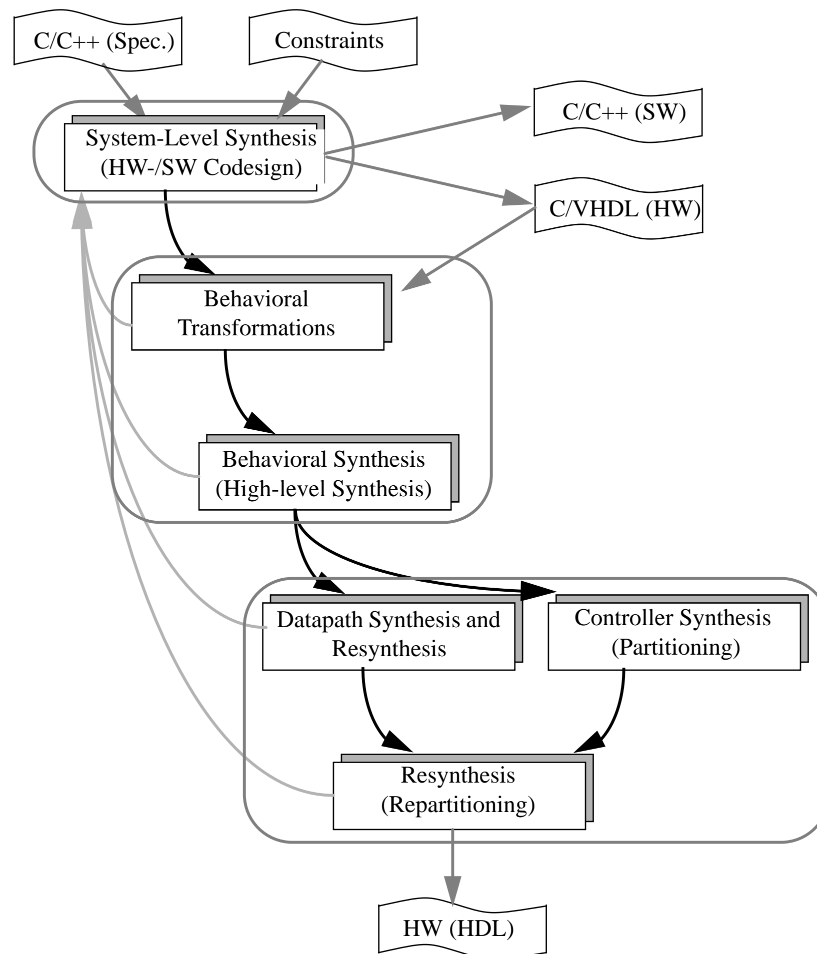


Figure 1. Synthesis environment and design flow.

The whole synthesis can be seen as a pipeline through each process of the design system. Each process performs transformations on a given representation, the result of which is a new representation

5.3.1 Restructuring 30
5.3.2 Guiding the High Level Synthesis 31
5.3.3 Mapping to Predefined Structures under Area and Timing Constraints 31
5.4 Limitations 32
6 Controller Synthesis 33
6.1 Input and Output Specification 33
6.2 Methods 33
6.2.1 Iterative Partial State Assignment of FSM 33
6.2.2 Linear FSM-Partitioning 35
6.2.3 Generalization of Counter Based Controllers 37
6.3 Applications and Extensions 39
6.4 Limitations 40
7 Datapath Synthesis 41
7.1 Input and Output Specification 41
7.2 Optimization Methods 41
7.2.1 Mapping 42
7.2.2 Register Replacement 42
7.2.3 Relocation and Clock Delay Adjustment 42
7.2.4 Reverse Register Replacement 42
7.3 Application and Extensions 44
7.3.1 Expected Effects of the Procedure 44
7.3.2 Results for the Overall Example 44
7.4 Limitations 45
8 Restructuring and Resynthesis of Designs 46
8.1 Input and Output Specification 46
8.2 Methods 47
8.2.1 Partitioning 48
8.2.2 Resynthesis 50
8.2.3 Ordering the Set of Candidates 51
8.3 Applications and Extensions 51
8.4 Limitations 52
9 Conclusion and Future Work 53
10 Bibliography 54

1 Introduction 4
2 System-Level Synthesis 6
2.1 Input and Output Specification 7
2.2 Methods 8
2.2.1 Static Specification Analysis 8
2.2.2 Runtime Analysis 9
2.2.3 HW/SW- Interface Costs 11
2.2.4 Partitioning Cost Function 12
2.3 Applications and Extensions 13
2.4 Limitations 13
2.4.1 Overall limitations 14
2.4.2 Static analysis limitations 14
2.4.3 Dynamic analysis limitations 14
3 Structural Synthesis 15
3.1 Input and Output Specification 15
3.2 Method 16
3.3 Applications and Extensions 18
3.4 Limitations 19
4 Behavioral Transformations 20
4.1 Input and Output Specification 20
4.2 Method 21
4.2.1 System-Level Analysis 21
4.2.2 Module-Level Analysis 22
4.2.3 Block-Level Analysis 23
4.2.4 Behavioral Transformations Impact on HW Implementation 23
4.3 Applications and Extensions 24
4.4 Limitations 25
5 Partitioning and Restructuring a Design on Behavioral Level 26
5.1 Input and Output Specification 26
5.2 Behavioral Partitioning 27
5.2.1 Data Flow Similarity 27
5.2.2 Control Path Similarity 29
5.2.3 Pruning the Search Space 30
5.3 Applications and Extensions 30

Flexible HW Synthesis and Optimization by Incremental Design Modification

Heinz-Josef Eikerling, Reiner Genevriere, Wolfram Hardt, Andreas Hoffmann,
Universität Paderborn,
Warburger Str. 100,
D-33 095 Paderborn, Germany

Klaus Feske, Günther Franke, Manfred Koegst, Hans-Georg Martin,
Fraunhofer-Institut für Integrierte Schaltungen (IIS),
Zeunerstrasse 38,
D-01 069 Dresden, Germany

Abstract: In this paper a design methodology for automatic digital circuit synthesis is presented. The design flow is analyzed in terms of examples showing the optimization steps taken during this process. By dividing the entire activity into different tasks (structural vs. logic synthesis) working on different domains (RT level vs. logic level) the mostly taken compilation (i.e., one way) approach is replaced by coupling these tasks in both directions by propagating constraints and alternatives besides the specification.

Keywords: System-level, High-level and Logic Synthesis, Behavioral Transformations, Retiming, Partitioning, Resynthesis.