A Path–Based Technique for Estimating Hardware Runtime in HW/SW-Cosynthesis

Jörg Henkel, Rolf Ernst

Institut für Datenverarbeitungsanlagen
Technische Universität Braunschweig
Hans–Sommer–Str. 66, D–38106 Braunschweig, Germany
Henkel@ida.ing.tu–bs.de

Abstract

One of the key issues in hardware/software–cosynthesis is precise estimation. The usual local estimation techniques are inadequate for globally optimising compilers and synthesis tools. We present a path based estimation technique which allows a computation time/quality tradeoff. The results show acceptable computation times while revealing much more potential parallelism than local list scheduling.

1 Introduction

System level design becomes more important since the time to market (e.g. about 18 months for complex HW/SW systems [Keu94]) decreases at a continuously increasing complexity of mixed hardware/software systems. As a consequence, uniform specification of HW/SW systems, HW/SW partitioning, HW synthesis, SW synthesis, cosimulation etc. become important research areas.

In the ambitious area of cosynthesis, hardware software partitioning plays a key role. There are some constraint–driven approaches which focus on this problem: The VULCAN system [GuMi92] is a hardware–oriented approach to HW/SW–cosynthesis. In [VaGaGo94] a software–oriented approach (starting with an all–software–solution) decides about the HW/SW tradeoff using a binary search algorithm. The approach described in [KaLe94] uses an algorithm called GCLP that takes into consideration a global time critical measure and a set of local criteria in order to determine the HW/SW tradeoff. COSYMA [ErHeBe93] also belongs to the class of software–oriented approaches and uses the simulated annealing algorithm for automating the HW/SW partitioning process.

Other approaches [BaRoXi94], [JaElOb+94] also focus on HW/SW partitioning but still need some user interaction.

All these approaches have in common that they need sophisticated techniques for estimating parameters (time, area,...) that will decide on the hardware/software partitioning.

We present a precise hardware runtime estimation technique that can even evaluate global optimization potential and that has acceptable computation time.

The next section introduces to the estimation problem in the context of the COSYMA system. In section 3 the use of the path–based technique for estimating hardware runtime is presented. The results are discussed in section 4 while section 5 gives a conclusion.

2 Estimation in COSYMA

A very simplified design flow of COSYMA is shown in figure 1. Input is a real-time system description in a superset of C that is translated to an internal representation, the extended syntax graph (ESG).

HW/SW–partitioning in COSYMA is solved with simulated annealing [OG89]. The iterative annealing process is called *Inner partitioning Loop* (IPL). Simulated annealing generates several thousands of designs that must be evaluated by the cost function. The cost function is based on estimation of hardware and software runtimes, HW/SW–communication time and on trace data. The path–based estimation technique presented in this paper provides the hardware runtime estimation. After partitioning, hardware and software synthesis are executed and the actual values for hardware runtime, chip area, ... are fed back. For more details on COSYMA see [ErHeBe93]. If the constraints (*CS*) have not been met, the whole procedure is repeated. Reason for a deviation from the given constraints is that estimation cannot predict all local and global optimization effects in the computation intensive hardware and software synthesis processes.

As shown in figure 2, COSYMA takes those effects into account by correcting the set of estimation values E_j when the real values C_j have been determined. The result is a set of rules from which a set of new estimation values E_{j+1} is derived. Then, again a hardware/software partition P_j is generated and so on. We call this process the *Outer Partitioning Loop OPL*. Iteration through the OPL should be done until all constraints are met.

As shown in [HeHeEr94] a convergence – i.e. meeting all constraints – can be achieved after a few iterations. *Prerequisite are sophisticated estimation algorithms*: the closer the result of an estimation to the real value the faster the convergence and the final re-

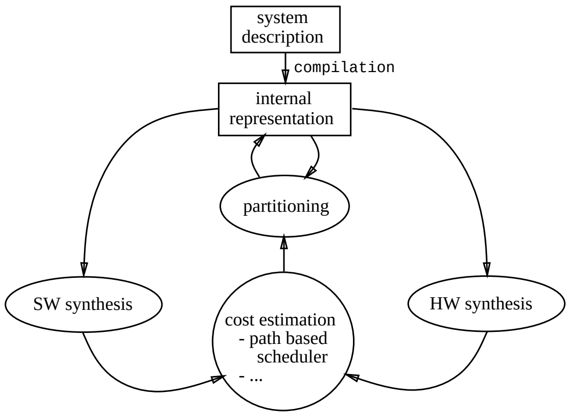

Figure 1: Simplified design flow of COSYMA.

sult (see [HeHeEr94]). This estimation is only executed *once* in the beginning of inner loop computation to derive all estimation values including high order values as E_{ij} [HeHeEr94].

The requirements for developing an adequate algorithm that estimates the hardware runtime, are (implied by the arguments above):

- *high precision*
- *low computational effort*
- *completely automatic*

The first requirement excludes all estimation approaches that are limited to the scope of basic blocks because they are not able to identify the global optimization potential in hardware synthesis. Path–based scheduling could be used for estimation instead as it reveals global optimization potential of basic block sequences.
The scheduling algorithm described in [Cam91] furthermore promises a high flexibility since the handling of resource constraints is integrated into the global concept. Nevertheless the major disadvantage is that some of the steps in path–based scheduling are *NP–complete* thereby preventing its use for larger programs.
In [ObRaJe93] an approach is presented that uses an algorithm in order to avoid a path explosion. There paths end if a *wait* statement is encountered or a node is found that has already been inserted in a path. This approach is not able to decide between different alternatives to limit a path since the conditions are fixed (i. e. always end a path if the conditions are true).
We present a path–based estimation technique that reduces the number of paths drastically with a minimum loss of quality[1] (schedule) since our algorithm can decide where to end a path (given a set of possibilities from which only that subset is taken that leads to a small loss of quality) A further advantage is that the user can decide on the *quality/computation time trade-off.*

[1]The original path–based scheduling by [Cam91] does not consider a re–ordering of operations in a path during scheduling. We have overcome this problem by executing a pre–scheduling phase. We do not focus on this problem here due to lack of space.

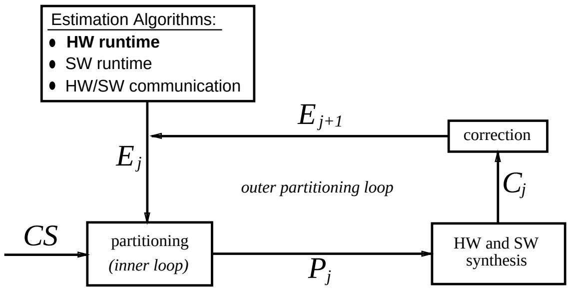

Figure 2: The role of estimation in the Outer Partitioning Loop.

3 A Path–Based Estimation Technique

Path–based scheduling consists of the following passes:

I. Transforming a CDFG into a directed acyclic graph.
II. **Collecting ALL paths.**
III. Scheduling all paths *As–Fast–As–Possible* (AFAP see [Cam91]).
IV. Overlapping all paths.

Computation time intensive steps are III and IV as a consequence of step II.
The AFAP schedule in step III makes use of a clique partitioning algorithm [TsSi86] that has been proved to be *NP–complete.* The computation time depends on the number of operations per path N_O.
Assuming the worst case i.e. each path has at least one basic block with each other path in common the computation effort for step IV results to $O({N_P}^2)$ with N_P the total number of paths found.

So the goal is to minimize the total numbers of paths as well as the number N_O of operations (or basic blocks) per path. Therefore the whole graph representation of an application is split and each part is scheduled by itself. This section deals with determining the so called *cut points* in a sophisticated way.

Let $G = \{V, E\}$ be a directed acyclic graph DAG where each node $v_i \in V$ represents a basic block that contains at least one single operation and where a directed edge $e_i \in E$ specifies the direction of the control flow. Fork nodes correspond to an *if-else* statement in the origin program code. Feedback edges denoting *loop* statements are already removed. Each node v_i has two attributes: *it* (i. e. iterations) is the number of times this node has been visited at simulation time. The nesting level *nl* of a node increases by one if the predecessor is a fork node and it decreases accordingly if the predecessor is a join node.

Example 1:

Calculating all paths in the graph representation given by figure 3 leads to a number of $N_P = 10$ paths. Each path P contains a number of $N_O = \sum_{v_i \in V} N_{O_i}$ operations.

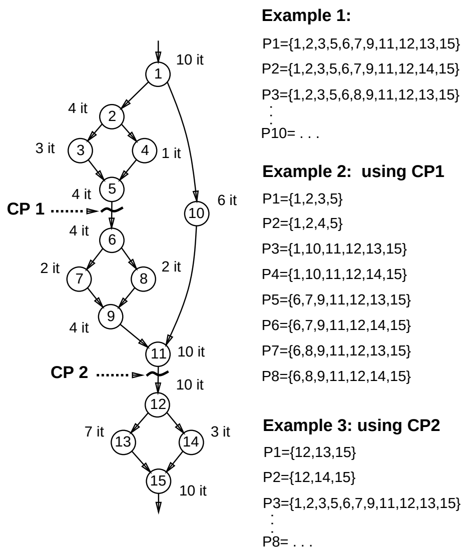

Figure 3: Calculating the number of paths with different cut points set.

Example 2:

Now assume the graph has been split into two parts split by a *cut point CP1*. Determining all paths for each subgraph[2] leads to a total number of $N_P = 8$ possible paths.

Example 3:

Instead of *CP1* cut point *CP2* is set and all possible paths are calculated again. Hereby there is $N_P = 8$ also.

Compared to example 1, example 2 and 3 are expected to gain a near optimum scheduling result because a *cstep* (control step) ends at each cut point and a new cstep starts behind the cut point.
The loss in quality measured in terms of an additional number of csteps depends on the *data dependencies* of operations before and behind a cut point. Assuming that the operations have already been optimally ordered, there is no way to influence this effect.
Another aspect are the hardware constraints (number of available hardware resources). The larger the number of resources the larger is the additional number of control steps since a potential high parallelism is prevented (see example in figure 4). Calculating the number of additional csteps as a result of cut points would require a data flow analysis *through* the according cut point. The computation time saved by reducing the number of paths would have to be re–invested and nothing is gained.
A better measure for the loss of quality is the increase of execution time (measured in clock cycles) implied by the schedule. Let it_j be the number of times an

[2] A *subgraph* is part of the origin graph G with a cut point before the first node and after the last node set but without any cut point inside.

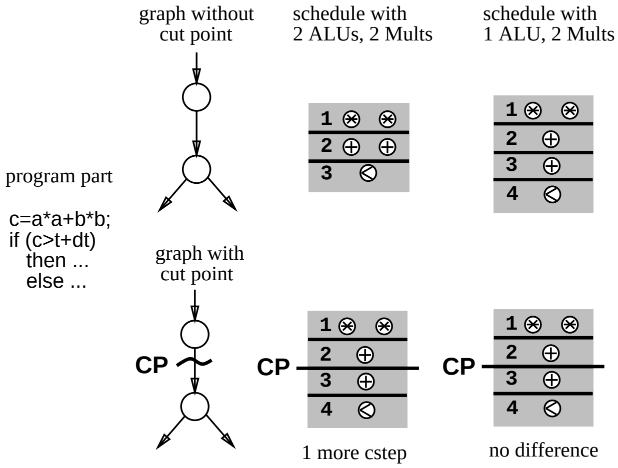

Figure 4: Comparison of schedules without (top) and with (bottom) cut points for different constraints.

operation scheduled in control step c_j (C is the set of all control steps) is executed. Then

$$t_{ex\_cut} = \sum_{g \subset G} \sum_{c_j \in C} it_j \qquad (1)$$

gives the total execution time of an program whose graph representation G has been cut into subgraphs g. In spite of the fact that example 2 and 3 lead to the same (reduced) number of paths, example 2 is expected to imply a smaller execution time (only 4 **it**erations at cut point *CP1* against 10 iterations at *CP2*). That leads to the formulation of rule 1.

Rule 1:

Locate the cut points at positions in the graph representation where a minimum number of additional clock cycles Δt_{ex} compared to a non–cut graph is expected.

$$\Delta t_{ex} = min(t_{ex\_cut} - t_{ex\_org}) \qquad (2)$$

There are t_{ex_cut} and t_{ex_org} the execution times with and without cut points set. Preferred candidates are those with the lowest iterations (executions).

When rule 1 is executed it is assumed that a selection of cut points has already taken place. In order to simplify the optimization procedure the total number of possible cut points should be minimized by limitation to those cut points that will really reduce the complexity:

Rule 2:

A possible cut point CP_i is located behind a node $v_i \in V$ if v_i is a join point *and* if there exists a path P where v_i is the starting node *and* where

$$\bigvee_{v_j \neq v_i, v_j \in P} (v_j is\ join\ point) \ \wedge \bigwedge_{v_n \in P} nl(v_n) \geq nl(v_i).$$

Here $nl(v)$ is the nesting level of node v. The nesting level increases by one after a fork node and decreases

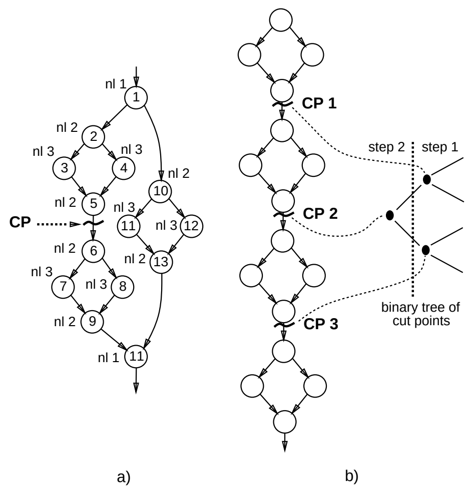

Figure 5: a) Searching for possible cut points. b) Hierarchical reducing of cut points.

when a join node is reached (see figure 5 a) attribute "nl"). The path P ends if a node v_k is encountered with $nl(v_k) \leq nl(v_j)$.

Figure 5 a) shows the result of rule 2 for a small example. Only node 5 fulfills all conditions and therefore a cut point is set accordingly. It is obvious that cut points behind nodes 9 and 13 would not reduce the number of paths.

For the case rule 2, rule 1 and an AFAP schedule have been applied and the user wants to improve the result[3] a reduction of cut points is necessary.

Rule 3:
Search for subgraphs $g \subset G$ that could possibly contain (according to rule 1) more than one cut point at the same nesting level. Find that cut point that would split such a subgraph best, i.e. the resulting 2 pieces are of same length (a measure is the number of operations). Handle each piece in the same way etc. A binary tree is built up where each node represents a cut point and each edge represents a piece of the subgraph of the original program graph G. Then step by step, beginning at the leafs of the binary tree, the cut points are removed. The user determines how many cut points are removed since reduction of cut points implies an increase of computation time (*quality/computation time tradeoff*).

Figure 5 b) shows this procedure for a small subgraph with only 3 cut points. Step 1 would remove cut points 1 and 3 and step 2 would remove cut point 2.
The hierarchical reduction of cut points takes into account that a minimum quality loss at a maximum reduction of complexity (number of paths) shall be guaranteed.

Figure 6 shows how rules 1 to 3 are applied within the path–based estimation technique. The set of cut

[3]The user does not interact with the individual scheduling process but defines the maximum number of cut points as a control parameter.

```
( 1 )  path_based_estim_tech()
( - )  {
( 2 )      CP := {};
( - )
( 3 )      collect_profiling_data(CDFG);
( 4 )      DAG := convert_to_DAG(CDFG);
( 5 )      #paths := compute_num_of_paths(DAG);
( - )
( 6 )      if (#paths < max_paths ) {
( 7 )          CP := compute_cutpoints(DAG);
( 8 )          𝒟𝒜𝒢 := split_DAG(DAG, CP);
( - )      }
( - )
( 9 )      for each dag_i ∈ 𝒟𝒜𝒢 {
( 10 )         P := calculate_all_paths(dag_i);
( 11 )         CS := {};
( 12 )         for each p ∈ P {
( 13 )             CS := CS ∪  do_AFAP_schedule(p);
( - )          }
( 14 )         superpose_all_schedules(P, CS);
( - )      }
( - )  }
( - )
( 15 ) compute_cutpoints(DAG)
( - )  {
( 16 )     CP_list.initialize();
( - )
( 17 )     for each v_i ∈ V { /* apply rule 2 */
( 18 )         if ( fulfills_rule2(v_i)) {
( 19 )             CP_list.insert(v_i);
( - )          }
( - )      }
( - )
( 20 )     sort_by_profiling(CP_list); /* apply rule 1 */
( - )
( 21 )     #cut_pts := input();
( 22 )     if (len(CP_list) < #cut_pts) {
( 23 )         if ( ambiguous_cut(CP_list, #cut_pts) )
( 24 )             CP_list := apply_rule3(CP_list, #cut_pts);
( - )          }
( - )      }
( - )
( 25 )     return(CP_list);
( - )  }
```

Figure 6: Path–based estimation technique applying rule 1 to 3.

points is initialized with the empty set, profiling data are collected and written to the CDFG-representation, the CDFG is transformed into a directed acyclic graph DAG and the number of a all paths *#paths* is computed (lines 2–5). If the number of paths would exceed the computation time (table 2 says that a number of ≈ 1000 leads to acceptable computation time of a few minutes only) cut points are calculated (l. 7, l. 15ff, see below). Then the DAG is split at the according locations (l. 8) and for all dag's in $\mathcal{DAG}$ a path based scheduling is executed. First all paths are calculated and for each path a *AFAP*-schedule is performed (l. 9–13). Then the set of all constraints $\mathcal{CS}$ is taken in order to superimpose all constraints of all paths (l. 14).
Cut points are computed in the function *compute_cutpoints*. It starts with scanning all nodes v_i of the DAG (l. 17). If a node fulfills the conditions formulated by rule 1, the node is inserted to the list of potential cut points *CP_list* (l. 18,19). Now the cut points are sorted in such a way that those that are located at less often executed parts of the graph have the highest priority since they lead to the smallest deviation from the optimum schedule (l. 20).
Now the user can determine the quality/computation-

benchm	loC	List-Schedule [cycles]
fuzzy	100	80,353
distance	695	256,680
contour	127	360,524
median	302	26,680,323
table	664	17,623,269

Table 1: Benchmarks scheduled with a simple List-Schedule using 4 ALUs, 4 Multipliers.

–time–tradeoff by choosing the number of cut points to apply (l. 21). If the number of potential cut points found exceeds this number (l. 22) a selection is necessary: for the case there are more than one cut points in the list which would lead to the same loss of quality — assuming only rule 1 and 2 have been applied — rule 3 will decide (l. 23,24) which of them to delete from the list in order to hit exactly the user–defined *#cut_pts*.

4 Experimental Results

For the experiments 5 benchmarks have been selected: *fuzzy* is a fuzzy controller, *distance, contour* and *table* are parts of a complex chromakey-algorithm (digital image processing) and *median* is a median noise reducer. The variety in lines of C–code (*"loC"* in table 1) allows to find the *optimum solution* for the smaller benchmarks (since the computation time is small enough even if all paths are scheduled and superposed) and by means of the larger benchmarks the functionality of heuristics described in chapter 3 can be shown. As a reference, all benchmarks have first been scheduled using a simple List–Schedule[4]. Table 1 shows the result in terms of clock cycles. This is possible since a profiling has already taken place. Remember that one of the ideas is to reduce the total execution time (equation 1). Reducing the number of control steps will possibly not lead to the best result.

First result is that the number of clock cycles for a specified benchmark using the path–based estimation technique (column *"schedule"* in table 2) in all cases is smaller than the according minimum for a List–Scheduling — independent from the number of cut points set.

This is mainly due to the principle of a path–based schedule. In terms of HW/SW–cosynthesis in COSYMA it means that the hardware/software tradeoff can be computed more precisely.

The remaining question is, how close the described estimation method approaches the optimum (path–based) schedule. If the number of cut points (*"cpt"* in table 2) is chosen to 1 the estimation technique is identical to the path–based schedule[5]. So, the line in bold face letters for benchmark *"contour"* represents the optimum solution. The number of total paths detected (2,430) is still acceptable since the computation time using a SPARC20 is about 1 an hour (3,426 seconds). For most of the other benchmarks an optimum solution ($cpts = 1$) could *not* be computed (too many paths).

The goal is to find a solution that results in a good schedule for a given amount of paths.

Each cut point should reduce the complexity, it should lead to a smaller computation time. Figure 7 reflects the result implied by rules 1 to 3 described above. For each of the benchmarks an increase of cut points (horizontal axis) leads to a reduction of computation time (vertical axis). A drastic reduction is gained when the first cut points are set. A saturation is reached when more cut points do not lead to a profitable reduction of computation time. Such points (marked by arcs) are assumed to lead to a good compromise between precision and computation time. Therefore figure 8 shows the quality of a schedule (deviation of a specified schedule from the according best schedule of this benchmark) as a function of the number of paths (a measure for the complexity). Assuming that e.g. a deviation of 15% for estimation during partitioning phase in HW/SW Cosynthesis is acceptable, 4 of the 5 *saturation points* are within this limit. The figure shows furthermore the non-linear dependency of precision and number of paths: the small improvement when passing the *saturation point* does not justify the large increase in computation time (number of paths).

Even quite large benchmarks could be scheduled with

benchm	#pth	#cpt	schedule [cycles]	ctime [sec]
fuzzy	21	11	72,321	2
	22	8	69,567	3
	27	6	67,049	7
	41	3	64,445	12
	43	**2**	**57,562**	**12**
distance	123	41	184,255	697
	403	10	178,864	957
	448	9	178,876	1,558
	2,463	**8**	**178,876**	**3,728**
contour	25	9	261,441	4
	102	3	231,583	82
	813	2	231,654	1,257
	2,430	**1**	**231,653**	**3,426**
median	44	21	21,980,162	5
	55	16	20,321,282	6
	72	11	19,077,122	16
	74	10	18,662,402	16
	80	9	18,247,682	19
	93	8	17,832,962	20
	437	7	17,832,962	317
	819	6	17,418,242	823
	1,585	5	16,588,801	1,732
	3,119	**4**	**16,174,082**	**3,640**
table	140	45	10,108,341	363
	146	43	10,100,459	399
	183	36	10,076,203	748
	238	27	10,076,236	749
	309	**25**	**10,076,257**	**946**

Table 2: Benchmarks scheduled with path–based estimation technique using 4 ALUs, 4 multipliers.

[4]Hardware constraints have been selected to 4 ALUs and 4 multipliers in order to achieve a fast schedule.

[5]By definition, a single cut point means that the according program graph is *not* split since in every case one cut point is set behind the last node in the graph. This comes from the algorithm of the implementation.

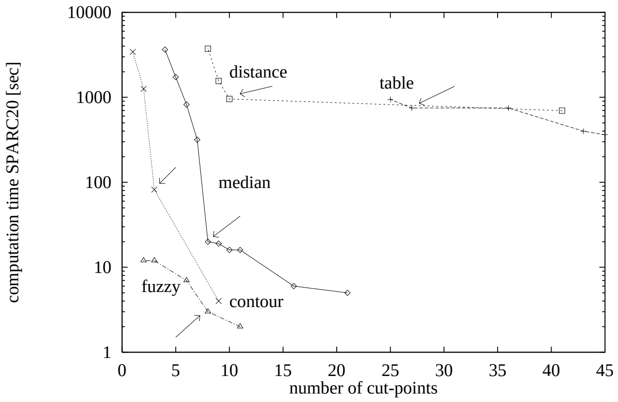

Figure 7: Computation time (using a SPARC20) versa number of cut points.

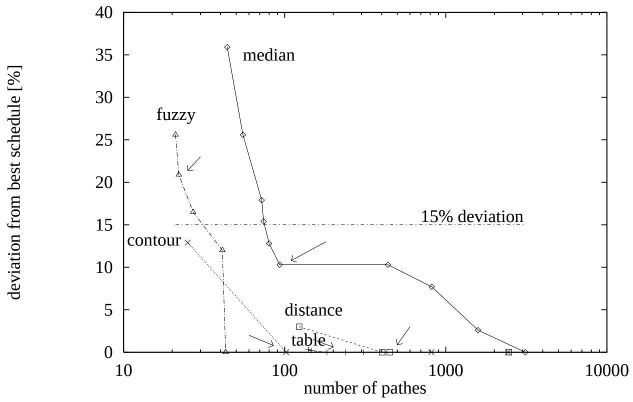

Figure 8: Tradeoff between quality (deviation from according best schedule) and complexity (number of paths).

the path–based estimation technique and lead to better results than a simple list schedule. So, real benchmarks like *distance* and *table* can make use of a path–based approach whereas the computation time of a pure path–based schedule would (in most cases and especially in HW/SW Cosynthesis) not be acceptable ($>$ 1 day since the number of paths $\gg$ 10000).

5 Conclusion

The advantage of a good schedule as well as the flexibility makes path–based scheduling interesting for HW/SW–cosynthesis. The disadvantage of large computation times for real benchmarks could be overcome by the described path–based estimation technique. We demonstrated that with a few heuristic rules a sophisticated placement of cut points can reduce the complexity with only a small loss of quality. As a result the path–based estimation technique provides a reasonable tradeoff between quality and computation time.

References

[BaRoXi94] E. Barros, W. Rosenstiel, X. Xiong, *A Method for Partitioning UNITY Language in Hardware and Software,* Proc. of Euro–DAC'94, pp. 220–225, 1994.

[Cam91] R. Camposano, *Path–Based Scheduling for Synthesis,* IEEE Transactions on Computer–Aided Design, Vol. 10, No.1, pp. 85–93, Jan. 1991.

[ErHeBe93] R. Ernst, J. Henkel and Th. Benner, *Hardware/Software Co-Synthesis for Microcontrollers,* IEEE Design & Test Magazine, Vol. 10, No. 4, Dec. 1993.

[GuMi92] R.K. Gupta and G.D. Micheli, *System-level Synthesis using Re-programmable Components,* Proc. of EDAC'92, IEEE Comp. Soc. Press, pp. 2–7, 1992.

[HeErHo+94] J. Henkel, R. Ernst, U. Holtmann, Th. Benner, *Adaptation of Partitioning and High-Level Synthesis in Hardware/Software Co–Synthesis,* Proc. of ICCAD'94, pp.96–100, 1994.

[HeHeEr94] D. Herrmann, J. Henkel, R. Ernst, *An approach to the adaptation of estimated cost parameters in the COSYMA system*, Proc. of 3rd IEEE International Workshop on Hardware/Software Codesign, pp. 100–107, 1994.

[JaElOb+94] A. Jantsch, P. Ellervee, J. Öberg et. al., *Hardware/Software Partitioning and Minimizing Memory Interface Traffic,* Proc. of Euro–DAC'94, pp. 220–225, 1994.

[KaLe94] A. Kalavade, E. Lee, *A Global Critically/Local Phase Driven Algorithm for the Constraint Hardware/Software Partitioning Problem,* Proc. of 3rd IEEE Int. Workshop on Hardware/Software Codesign, pp. 42–48, 1994.

[Keu94] K. Keutzer, *Hardware–Software Co–Design and ESDA,* Proc. of 31st Design Automation Conference, pp. 435–436, 1994.

[ObRaJe93] K. O'Brien, M. Rahmouni, A. Jerraya, *DLS: A Scheduling Algorithm For High–Level Synthesis in VHDL,* Proc. of EDAC'93, pp. 393–397, 1993.

[OG89] R. Otten, P. van Ginneken, *The Annealing Algorithm,* Kluwer, 1989.

[PeKu93] Z. Peng, K. Kuchcinski, *An Algorithm for Partitioning of Application Specific System,* Proc. of The European Conference on Design Automation 1993, pp. 316–321, 1993.

[TsSi86] C.J. Tseng, D.P. Siewiorek, Automated Synthesis of Data Paths in Digital Systems, IEEE Trans. on CAD, Vol. 5, No. 3, pp. 379–395, 1986.

[VaGaGo94] F. Vahid, D.D. Gajski, J. Gong, *A Binary–Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning,* Proc. of Euro–DAC'94, pp. 214–219, 1994.