The Interplay of Run-Time Estimation and Granularity in HW/SW Partitioning

Jörg Henkel, Rolf Ernst

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

Abstract

An important presupposition for HW/SW partitioning are sophisticated estimation algorithms at a high level of abstraction that obtain high quality results. Therefore the granularities of estimation and partitioning have to be adapted adequately. In this paper we discuss the effects that arise when the granularities of partitioning and estimation are not adapted in a necessary way. Furthermore we present our solution that allows to choose different levels of granularities adapted to the estimation and partitioning phase. The experiments show that this refinement in estimation at a high level of abstraction leads to an inprovement (in terms of run-time and chip area) of the whole mixed HW/SW system.

1 Introduction

In recent years HW/SW cosynthesis has become a very interesting field of investigation. This is due to the promising advantages of this design method.

In HW/SW cosynthesis a system is specified by its behavior — independent from later implementation in software or in hardware — instead of its structure. So the designer can focus on the really challenging problems as, for example, the correct specification of a system. In the case of HW/SW cosynthesis the term *system* is often a synonym for *real–time system* because of the large economical potential (areas of application are telecommunications, car electronics, office automation, consumer electronics,...). The correctness of a reactive system is not only determined by specifying an appropriate algorithm for a given problem (e. g. which compression algorithm to select given a limited bandwidth and a constrained quality) but also by the timing behavior: if a message of a multiple-process reactive system is not transmitted within a given time slot, other processes may have to wait and cannot serve the actuators in the right time. In this case the functionality would not be maintained. The designer's task is to prevent such cases by analyzing the demands and specifying accordingly. This is the more urgent the complexer the system becomes. Otherwise the time-to-market factor would increase to such an extend that the resulting system will become too expensive. A typical time-to-market is about 18 months for complex HW/SW systems [15].

Presupposition for this increasing abstraction level in system design is the existence of an almost automatic synthesis process for both hardware and software parts. This is called HW/SW cosynthesis.

The well known design steps of a generic cosynthesis flow are:

I. High-level specification (C, C++,...)
II. **Estimation** (HW run-time, SW run-time, chip area,...)
III. HW/SW partitioning
IV. Synthesis (HW, SW, interface)

Steps II. and III. are strongly correlated, so only if estimation values are close enough to the real values (obtained when really implemented) partitioning makes sense.

The approaches used in the estimation phase — which is that relevant that it determines the quality (in terms of monetary costs) and the feasibility (given the structure of a target architecture) of a mixed HW/SW solution — cover a wide diversity of algorithms: it spans from simple profiling to sophisticated approaches that also take into account the micro-architectural features (pipeline interlocks, register sets,...) of a given processor core that is provided to execute, for example, the SW part. The same holds for the hardware part: the question is, how well the HW runtime estimation is suited to that algorithm used when the HW part is really synthesized (see also [10]) and what will happen if there is a great deviation between estimation and real data. The same questions could similarly be applied to the communication mechanism between the hardware and software part and the implied costs of an application specific hardware (chip area).

In any case, the answer is that it is nearly impossible or at least economically of no interest to perform an automatic HW/SW partitioning unless the underlying analyzing/estimation algorithms are of an appropriate accuracy.

There are a lot of approaches to HW/SW cosynthesis that differ mainly in two points: the partitioning algorithm itself (automatic by means of a combinatorial optimization algorithm or manually driven) and the accuracy of the analyzing/estimation algorithms. Automatic partitioning is proposed by [7, 8, 13, 14, 19]. They differ in the way the partitioning process is performed: hardware-oriented/software-oriented approaches. Others [2, 3, 4, 5, 6, 12, 17, 18, 20] focus on input specification, interface synthesis, rapid prototyping ,....

An investigation of how the quality/accuracy of an analyzing/estimation algorithm influences the quality (i. e. costs) of a HW/SW system that have automatically partitioned has not been performed so far. This is mainly due to the fact that those algorithms were imported from the high-level synthesis domain and have not been adapted in an adequate way to the cosynthesis problem. So it becomes necessary to perform a re-partitioning or limit the partitioning process to a coarse-grain granularity[1] even though a limitation to a coarse-grain granularity only, might lead — at least potentially — to a sub-optimum result.

This paper focusses on one part of this problem, namely the estimation of the hardware run-time. The dominating question to consider is: in which way is the additional amount of run-time and/or application specific hardware (i. e. chip area that implies larger costs) in an automatic HW/SW partitioning procedure due to imprecise/inappropriate estimation. Introduced are some (of the large amount) of possible effects. Presented is a solution that covers (partly) the discussed effects.

The paper is structured as follows: The next section gives a definition of the estimation problem when applied to the domain of HW/SW partitioning. In section 3 we discuss in detail which effects should be avoided in order to prevent a cost-intensive mixed HW/SW system. Also, a solution is presented. Experiments in section 4 underline the relevancy of applying a sophisticated estimation technique. The quality metric is given by the hardware area and hardware run-time. Finally section 5 gives a conclusion.

2 Problem definition

2.1 Meeting real-time constraints

An essential goal in HW/SW cosynthesis is the high-level i.e. behavioral specification of the system that is to be implemented. The designer should not care about the question whether to implement dedicated parts in hardware or in software.

Hardware/Software partitioning is the most challenging step in HW/SW cosynthesis since the best hardware/software tradeoff promises the cheapest design while meeting the specified constraints.

Independent of the fact whether HW/SW partitioning is performed automatically or manually, the quality of the estimation techniques that control the partitioning process will implicitly determine the costs of a mixed HW/SW system in terms of the overall *chip area*. So estimating plays a key role in this context.

Let $T^{CS}_{HW/SW\ sys,c_i}$ be a time-constraint assigned to a code segment c_i in the behavioral description of the system. The goal is to find an implementation for c_i such that

$$T^{I}_{HW/SW\ sys,c_i} \leq T^{CS}_{HW/SW\ sys,c_i},$$

where $T^{I}_{HW/SW\ sys,c_i}$ denotes the execution time of c_i after implementation (i.e. assigned to the software part or to the application specific hardware part of the mixed HW/SW system). This is one of the presuppositions to guarantee the correct functionality of a real-time system.

In this context the **task of estimating** is to find a $T^{E}_{HW/SW\ sys,c_i}$ that represents a lower bound for the execution time of each c_i and that is as close as possible to the according constraint $T^{I}_{HW/SW\ sys,c_i}$ in order to obtain a cheap design.

[1] We will show later that the choice of coarse-grain granularity hides the inherent problems since the relative deviations are smaller.

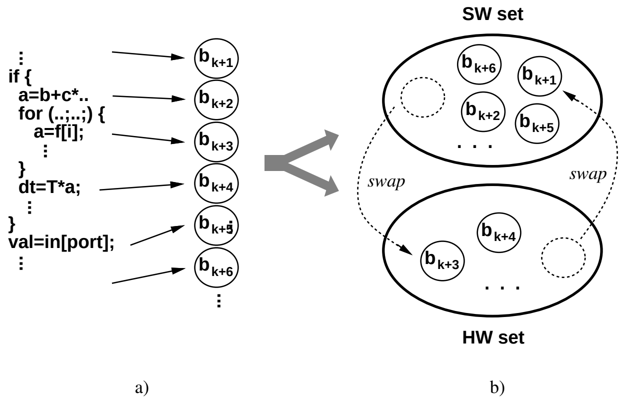

Figure 1: Defining a granularity from the behavioral description a) for the purpose of HW/SW partitioning b)

2.2 Estimating and granularity

Estimating in HW/SW cosynthesis does not only mean *estimating from behavioral specification* but also *estimating at a specific level of* **granularity**. Thereby granularity is defined as follows:

> **Definition 1** *Given the behavioral description of an algorithm the* **granularity** *determines how to cut this algorithm into a set B of n pieces b_i*
> $$B := \{b_0, b_1, \ldots, b_{n-1}\}$$
> *for the purpose of distributing these pieces among the hardware part and the software part of a mixed hardware/software system (see fig. 1.)*

Examples for obvious granularities, directly derived from the structure of the behavioral descriptions, are: *operator-level*, *statement-level*, *base-block-level*, *control-block-level*[2] and *task/process-level*.

Note that the choice of the granularity *does not* necessarily mean that the largest code-segment to be implemented either in hardware or in software is limited to the extension of the according granularity[3]. But it *does mean* that the smallest code-segment can not be smaller than the smallest element $b_i \in B$. So the granularity determines a *lower bound* of a code-segment.

Once the granularity has been determined, for each piece b_i the execution time T^{E}_{HW,b_i} has to be estimated in order to guide the HW/SW partitioning. Thereby T^{E}_{HW,b_i} is the estimated execution time of piece b_i for the assumption that b_i would be implemented in hardware. Principally there are two different ways how to obtain T^{E}_{HW,b_i}:

[2] We use the term *control-block* for a set of adjacent base blocks containing also control statements (**IF, FOR,**...) and clustered by a *closeness-metric*.

[3] In fact in many cases it is advantageous to put *several adjacent* b_i's to hardware or software.

Case1
The hardware execution time is estimated taking the *whole code* without dividing it into a set of pieces B. So the whole behavioral description would be assumed to get executed by the hardware part. Let $T^E_{HW,sys}$ be the according execution time. If we intend to design a mixed HW/SW system we now have to choose a granularity i.e. we have to find a set B. The HW run-time T^E_{HW,b_i} for each b_i can be derived from $T^E_{HW,sys}$ by simply dividing the whole description according to the granularity and assigning all of the run-time attributes that are found within each piece to b_i.
This method is advantageous for the case that there is already a run-time estimator available (e.g. a scheduler from the domain of High-Level Synthesis). Nevertheless, this method lacks accuracy: if the granularity is *fine-grain* (a large n i.e. a lot of b_i's in B) with only a few operators within a b_i, we should be aware of the fact that a schedule strongly depends on what has been scheduled some control steps before. Say, the schedule of b_i depends on data/control dependencies outside of b_i[4]. Hence, the schedule of b_i is only 100% correct if all adjacent b_i's are also implemented on the hardware side (see above).

Case2
This is the reverse case. T^E_{HW,b_i} is obtained by an estimation technique that is limited to the boundaries of a b_i (e.g. a simple List-Scheduler). It works great if we do not have any code segment to put to hardware that is larger than a single code segment b_i. But remember that the choice of a granularity is only a basis for partitioning (see beginning of this subsection). So, a larger code segment (consisting of more single-estimated b_i's) has also an inherent inaccuracy coming from the inaccuracy implied by each b_i. This is because none of the b_i's does take advantage of *synergetic effects*.

Definition 2 *A synergetic effect arises if in the case of an interplay of two or more b_i's an effect is obtained that is different from the effect obtained by the sum of effects of the single b_i's.*

In the context of scheduling we could observe that synergetic effects will lead to a better result. So, without taking these effects into account we will get sub-optimum results.

We can summarize the two cases by formulating an inequality expressing the all-over relative error ($Err_H^{case\{1,2\}}$) assuming that some of the $b_i \in B$ (denoted as H with $H \subseteq B$) are intended to get implemented in hardware:

$$\begin{aligned} Err_H^{case1} &= \sum_{b_i \in H} \left(\frac{T^{case1}_{HW,b_i} - T^{Opt}_{HW,b_i}}{T^{Opt}_{HW,b_i}}\right) \\ \leq & \;\; 1 \;\; \leq \\ Err_H^{case2} &= \sum_{b_i \in H} \left(\frac{T^{case2}_{HW,b_i} - T^{Opt}_{HW,b_i}}{T^{Opt}_{HW,b_i}}\right) \end{aligned} \tag{1}$$

Consequently $case1$ is a lower bound and $case2$ is an upper bound according to the *optimum solution*.
Obviously the estimation error can be reduced if we simply try to reduce the number n of blocks b_i. This is equivalent to choosing a *coarse-grain* granularity. *Task/process*-level granularity belongs to this group. During the partitioning the question will arise whether to move the *whole* task/process to hardware or to software — independent from the extension of the task (i.e. lines of code). But note that the area of application are small embedded systems which — in most cases — are very cost sensitive: even a few more gate equivalents for the part of the application specific hardware lead to an increase in price that makes this product uninteresting in the market of e.g. consumer products (audio, video,...) where competition becomes harder and harder. So, if the designer describes an embedded system only by aspects like *modularity*, *portability* etc. — i.e. programming style[5] — this might not be a good choice as a basis for the partitioning process. Therefore it is very probable that a coarse-grain granularity hides inherent optimization potential.
Or, on the other side, a fine-grain granularity will keep all the potential open.
In the next section we will start to discuss effects implying the inaccuracies in more detail and present our approach.

3 Effects in estimating and how to solve

3.1 Effects

As described in the last section HW/SW partitioning swaps code-segments b_i of a previous chosen granularity between hardware and software in order to find the best hardware/software tradeoff. This procedure is driven by estimation of relevant aspects. As for this paper we investigate the HW run-time estimation although the principles can also be applied to SW run-time estimation for example.
Thereby an error is invoked that is mainly due to effects at the borders of a code-segment at which control is transmitted from software to hardware, from hardware to software or — in case of a parallel execution of hardware and software parts — a *fork* or a *join* is performed.
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 $e_{i,j} \in E$ specifies the direction of the control flow, meaning that there is an edge from v_i to v_j. A node $v_i \in pred(v_j)$ is a node in G that is called a predecessor of v_j since there is an edge $e_{i,j}$. Accordingly a node $v_k \in succ(v_j)$ is called a successor of v_j. A granularity determines code-segments $b_i \in B$, where

$$b_i = \{v_j \in G : \bigwedge_{v_j \in b_i} \bigvee_{v_k \in b_i} v_k \in pred(v_j) \vee v_k \in succ(v_j)\}$$

So, this recursive definition says that a b_i is a *connected* part of the origin code.

Different number of resources
Fig.2 shows the example of a graph G with b_1 consisting of 3 elementary nodes v. b_1 is now assumed to get implemented in hardware, the rest of G should remain in software.
The number of hardware resources (ALUs, multipliers, shifters,...) available on the software side $HWRes^{SW}$ and on the hardware side $HWRes^{HW}$ will be different. In order

[4] Detailed effects will be the topic of the next section.

[5] In fact we state that the designer has in most cases no chance to decide whether a code-segment is very time-consuming and/or cost intensive when implemented in hardware since the chain from behavioral description to real implementation is long with a lot of optimization potential in it.

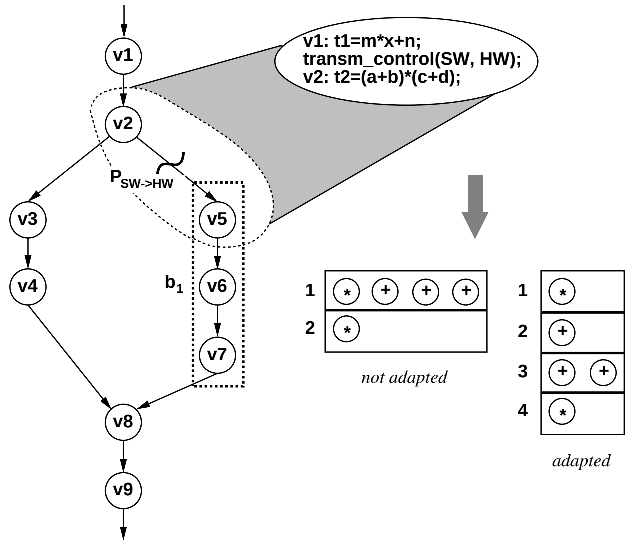

Figure 2: Estimating the run-time of the statements adjacent to a control transfer point

to obtain a speedup by using an application specific hardware, it is $HWRes^{HW} \geq HWRes^{SW}$. That implies that scheduling across point $P_{SW \to HW}$ where control is transmitted from SW to HW (or vice versa) causes an inaccuracy unless $P_{SW \to HW}$ is identical to the point where a new b_i starts.

Let us assume that there is $HWRes^{SW} = 1$[6] with no parallel invoking of the resources and $HWRes^{HW} = 2$ with possible parallel execution of the resources in the example. For the adjacent area (statement before and behind) of $P_{SW \to HW}$ we obtain a scheduling result of 2 control steps for the case that scheduling has no information about the existence of $P_{SW \to HW}$. In the other case a more precise estimation obtains 4 control steps. The difference of 2 control steps has to be put in relation to the schedule of the following code-segment that will be put to hardware (see inequality 1). The same is for the point $P_{HW \to SW}$ where control is transmitted back from HW to SW.

The effect is the larger the smaller a b_i is *and* run-time estimation *does not* use the same granularity as partitioning does.

Data transfer

At a control transfer point there is also transmitted data for the general case that hardware side and software side are data dependent. So, HW run-time consists not only of the scheduling result of the pure code-segment that is intended to be put to hardware, but it also consists of the data transfer time $T^{transfer}$. The choice of the granularity i.e. the extension of the $b_i \in B$ should be adapted to the amount of $T^{transfer}$. Otherwise a performance increase by using an application specific hardware with more resources than the software has, will be decreased drastically.

The extend of influence depends on the specific application (data- or control-dominated) and therefore it is indispensable to *dynamically* determine the granularity at which estimation

[6]meaning 1 ALU, 1 multiplier, ...

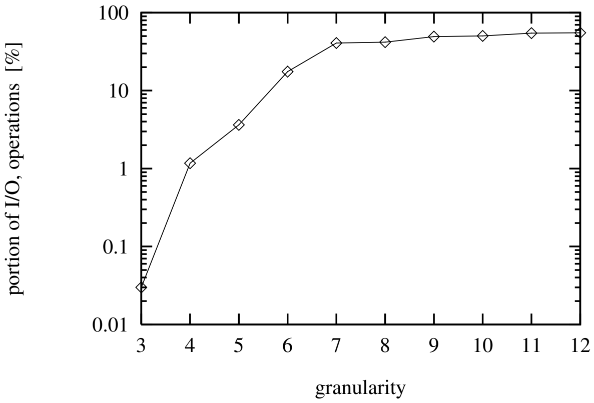

Figure 3: Data transfer overhead for different granularities

should take place. *Dynamically* means that it is adapted to the application and not fixed to structural peculiarity of the behavioral description like the composition of a program in its tasks/processes as one extreme and its statements as the other extreme. Fig. 3 shows the data transfer overhead of different granularities relative to the hardware execution time. For different granularities (the horizontal axis points in direction of decreasing granularities)

$$\sum_{b_i \in B} \frac{T_{b_i}^{transfer}}{T_{HW,b_i}^{exec}}$$

is shown (vertical axis), where T_{HW,b_i}^{exec} is the pure hardware execution time. This application[7] comes from the domain of digital signal processing and therefore is strongly data dominated. Concerning a selection of an appropriate granularity it can be seen that an index of 7[8] really makes sense since the data transfer overhead is beyond 50%. The HW/SW partitioning will prefer moving code-segments to hardware that are expected to obtain a large speedup[9]. So the granularity for estimation should be chosen accordingly (see above).

The following subsection describes the method we use to find a granularity adapted to estimation.

3.2 Clustering for estimation

In order to prevent low quality results for run-time estimation due to the fact that the granularity it takes place on is either too small or too large (section 2.2) we introduce in short terms our approach. Note that this clustering *does not* fix the extension of code-segments to get implemented in hardware but it does find a granularity that is suited for run-time estimation and that is acting as a *basis* for the following HW/SW partitioning process. In our cosynthesis design flow clustering is located as follows:

I. High-level specification

II. **Clustering for estimation** (HW run-time, SW run-time, chip area,...)

[7]It is a chromakey algorithm for HDTV studio equipment.

[8]The granularity index is introduced in the next section.

[9]This is the performance increase of a mixed HW/SW system compared to a simple SW system. It is illustrated in more detail in the section 4.

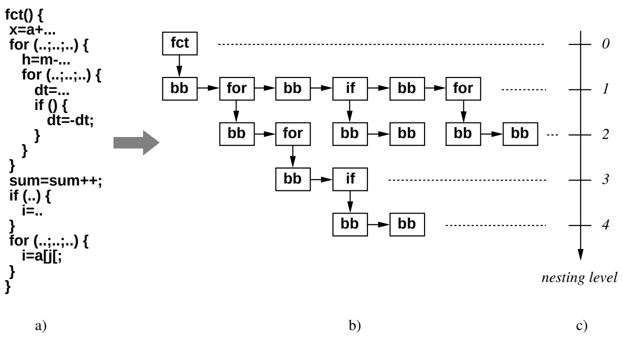

Figure 4: Small example program a) converted to a graph and according nesting level c)

III. Estimation (HW run-time,...)
IV. HW/SW partitioning
V. ...

For clustering we define a graph G with the following characteristics: $G = \{V, E^H, E^V\}$ is a directed acyclic graph where each node either represents a basic block[10] or a control statement like **if**, **for**, ... There are two kinds of directed edges, both pointing in direction of the control flow. The edges $e^H \in E^H$ do not change the hierarchy level whereas the edges $e^V \in E^V$ do. This implies the definition of a *hierarchy level*: the hierarchy level is an attribute of a node v_i that is incremented by one compared to $pred(v_i)$ if $pred(v_i)$ represents a control node *and* v_i is nested in the hierarchy of $pred(v_i)$. As an example assume there is an **if**-statement that encloses a base block executed when the condition is true. Then this base block is called to be *nested* in the **if**-statement and consequently has *hierarchy* incremented by 1. A statement that follows the whole **if**-construct would have the same hierarchy level as the **if**-statement.

The edges E^H are drawn from left to right and the edges E^V are drawn from top to bottom. Fig.4 gives an example of a code and its representation as graph G. All nodes at a specific line own the same hierachy level. Counting starts with 0 that is reserved for the head of the function/task.

Furthermore the graph has the characteristic that the feed back edges are implicit: if we arrive at a leaf node we have to skip to node $succ(v_i)$ where v_i is the node we previously descended to the hierarchy we are currently in.

Clustering always starts at the leaf nodes of the graph. The cost function for controlling contains mainly two components:

Number n of pieces b_i

The number of pieces determines the computation effort for both estimating and following HW/SW partitioning. As for the run-time estimation it means that we have to execute n schedules[11]. But the complexity for a based path-based scheduling depends on the number of possible paths within each b_i. The number of paths grows exponentially with the number of operations within a b_i in the worst case.

Since the complexity of the combinatorial optimization problem for partitioning grows exponentially with n (O(2^n)) the requirements concerning n seem to be contradictory. We have overcome this problem by using a *path-based estimation technique* [11].

So the user can determine the number n of pieces b_i as one constraint for clustering.

The "quality" of pieces b_i

For each possible piece b_i a quality metric is defined as follows:

$$quality(b_i) = \frac{T^E_{SW,b_i}}{T^{pre-exec}_{HW,b_i} + T^{transfer}_{b_i} + T^{misc}_{b_i}}$$

The term T^E_{SW,b_i} expresses the run-time estimation for the case b_i would be executed at the software part and T^{exec}_{HW,b_i} expresses the run-time estimation for the assumption of executing b_i. Since we only want to find a possible granularity where a sophisticated run-time estimation can take place afterwards, $T^{pre-exec}_{HW,b_i}$ is determined by a simple list schedule. The transfer time for data from software to hardware and vice versa is obtained by a data flow analysis that determines the $IN\_Set(b_i)$ and $OUT\_Set(b_i)$ similar to the method described in [1]. A possible overlap of b_i can get expressed by $T^{misc}_{b_i}$ (then there is $T^{misc}_{b_i} < 0$).

So the *quality* metric gives a hint whether b_i could be a good candidate for the HW/SW partitioning with the difference that in HW/SW partitioning itself a more sophisticated cost function is used ([9, 7]).

The task is now to find the n best clusters within G. Since a detailed description of the used algorithm would fill a separate paper, and we mainly concentrate on the interplay of run-time estimation and granularity in this paper, a shortened textual description is given here:

Phase 1

I. Get all leaf nodes in G.
II. Define each leaf node as a primary cluster.
III. Extend each of these clusters by merging a new v_i. This v_i is obtained by walking against the direction of edges e^H_i.
IV. If there is no more edge e^H_i found then climb up one hierarchy by walking against the direction of an edge e^V.
V. If this proceeding causes that two or more clusters will touch then merge them to a new cluster.
VI. For each merge operation note the number of all clusters n obtained so far. If the current number of clusters is greater than n then proceed by repeating steps III-V.

Phase 2

VII. Now determine for all n clusters the quality.
VIII. By a random function two clusters a picked out. One of the clusters is enlarged by applying the steps III-V while the other one is reduced applying the according reverse operations. Thereby the total number has to be kept constant (contraints).
IX. For each cluster gained in the previous step the quality is calculated. The acceptance of a cluster depends on more aspects than the pure qual-

[10] The term *basic block* is used in the same manner as in [1].

[11] n is the number of all b_i's in B.

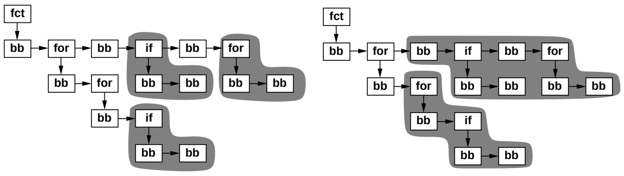

Figure 5: Two different scenarios of clustering during phase 1

ity: an important aspect for run-time estimation phase is that all of the clusters should be of almost the same extension and should not be too small in order to prevent the inaccuracy for the later estimation (see ineq.1).

X. The algorithm ends either by a user-specified time or by a threshold that defines the necessary quality improvement as a average of the last x merge operations.

Fig.5 shows two scenarios during clustering in the first phase.

4 Experiments and Results

Aim of the experiments is to show that a refinement in run-time estimation and an adaptation of granularity for estimation and partitioning leads to better results. In this context **better** could possibly mean a lot of things like simply the improvement of a run-time estimation technique compared to the real schedule. We will start to discuss this point. Much more interesting is the question in which way the quality of the *whole mixed HW/SW systems* will be improved by only refining one (namely the HW run-time estimation) estimation technique. Of course we cannot expect that the improvements to the whole system will be as large as the comparison of two single algorithms would imply. The main question is whether it is worth to refine an estimation at a high abstraction level. Does it have any influence on the implementation of the whole system ?

We first start to compare the scheduling results of using our path-based estimation technique [11], that operates on the graph described in the last section, and a simple list scheduler that has been used before in our system. Table 1 shows the results in terms of clock cycles that could be obtained after scheduling and profiling. The last column gives the fraction of time the according test program uses for execution compared to the list schedule. It can be seen that for all test programs (*testp*) the path based estimation technique (PBET) can obtain better results in terms of clock cycles although we picked out the worst case results the path-based estimation method delivered. This is due to the effect that the granularity for applying the list schedule has been too small i.e. limited to the scope of basic blocks only. This reflects the effect discussed in section 2.2 *case2* and which is expressed by the inequality 1.
In [11] it is declared in detail in which way the computation time can be kept small when the quality is improved using the path-based estimation technique.

A quality metric describing the whole HW/SW system is

testp	**loc**	**List-Schedule [cycs]**	**PBET [cycs]**	**frac rtime [%]**
contour	127	360,524	261,441	72.5
distance	695	256,680	184,255	71.8
fuzzy	100	80,353	72,321	90.0
median	302	26,680,323	21,980,162	82.4
table	664	17,623,269	10,108,341	57.4

Table 1: Comparison of schedules obtained by simple List-Schedule and path-based estimation technique (PBET) using an appropriate granularity

the *speedup*:

$$spu_{HW/SW\,sys} = \frac{T_{SW\,sys}}{T_{HW/SW\,sys}}$$

In this equation $T_{SW\,sys}$ is execution time in clock cycles for the case the whole system is implemented in software i.e. running on a standard processor core. The term $T_{HW/SW\,sys}$ denotes the execution time in cycles after partitioning in hardware and software parts. So $spu_{HW/SW\,sys} = 2$ for example means that a mixed HW/SW implementation is twice as fast as a pure software solution of the same behavior. The time $T_{HW/SW\,sys}$ contains all relevant times like the part the software needs for execution of the HW/SW system, the hardware execution time, HW/SW transfer time for data, the time for transmitting control from software to hardware and vice versa and synchronization time. For the different test-programs in table2 we changed two parameters: the first one is the constraint-speedup (*constr*) of the system and the the second is the variation of the run-time estimation algorithm (*LS* for the small granularity of base blocks only and *PBET* for estimating path-based at various levels of granularity) in order to measure its influence according to the performance of the mixed HW/SW system. The HW/SW partitioning has been performed automatically as described in [7, 9]. The column *spu* shows the estimated speedup as described in equation above. If we compare according lines, for example lines 2 and 6 of test program *trs* (both have the same constraint speedups) we can note that in all cases estimation using *PBET* obtains better results (in most cases about 5 %) than *LS* meaning that *PBET* is superior due to the effects described in section 3.2.
Next part of the experiments[12] discusses the question whether the increased quality can also influence the total monetary costs of the mixed HW/SW system. A quality metric is the chip area in terms of gate equivalents. Therefore the test programs have been automatically partitioned as above and in the following steps they have been synthesized using the High-Level synthesis system BSS and SYNOPSYS for implementing the application specific hardware on Xilinx FPGA's. Afterwards we transformed the specific CLB-usage into gate equivalents (*geq*). The results are shown in column *area* of table 2. For two of the benchmarks (signed by asterisks) we could detect that the number of gate equivalents is different at the same constraints (*constr*) due to the two different run-time estimation techniques. In all other

[12] In all experiments we switched off the weighting function for the Simulated Annealing described in [7] in order to obtain more design points.

testp		constr	spu	area [geq]	PB ET	LS
bpic	2	7.0	5.58	8986.5	x	
	⋆3	4.0	3.93	7697.0	x	
	4	2.0	2.14	7893.0	x	
	5	3.0	3.33	7791.0		x
	⋆6	4.0	4.38	9101.0		x
	7	7.0	7.40	–		x
key	1	10.0	11.79	16483.0	x	
	2	7.0	6.52	15057.5	x	
	3	3.0	1.01	1088.5	x	
	4	10.0	11.20	16483.0		x
	5	7.0	6.33	15057.5		x
	6	3.0	1.01	1088.5		x
smth	1	3.0	1.01	7883.0	x	
	⋆3	10.0	10.96	11089.5	x	
	4	12.0	11.00	12706.5	x	
	5	3.0	1.01	7883.0		x
	⋆7	10.0	10.70	13902.5		x
	8	12.0	10.52	12706.5		x
trs	1	5.0	1.04	3449.5	x	
	2	10.0	11.65	12837.5	x	
	4	5.0	6.16	15057.5		x
	6	10.0	11.20	16483.0		x

Table 2: Schedules and according chip area obtained for different design points using a List-Schedule and the path-based estimation technique (PBET)

cases the total hardware area is the same.
The complexity of a part of the test programs has been so large that they consumed a CPU time of > 1000 (more than 90% due to the SYNOPSYS tool). This also confirms the need of estimation tools at a high level of abstraction that can execute much faster at acceptable accuracy.
Fig.6 shows the dependency of gate equivalents and $spu_{HW/SW\,sys}$ for all test programs using PBET.
As a general result we can note that the accuracy in HW run-time estimating from behavioral description does influence the quality of the mixed HW/SW system in spite of the fact that a lot of other aspects (estimation of SW run-time, data transfer time, ...) are superposed.

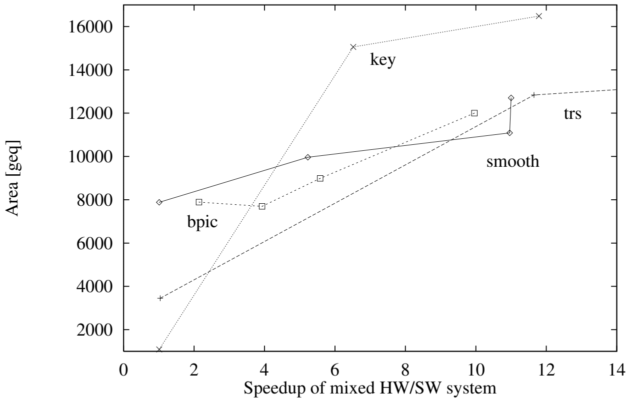

Figure 6: Area-Speedup-Dependency using the PBET

5 Conclusion and future work

We have shown that estimating HW run-time at a high-level of abstraction is a challenging problem. By a sophisticated technique where the granularity of estimation is adapted dynamically and adapted to the following HW/SW partitioning we could demonstrate that the all over quality of the hardware software system can be increased in terms of *speedup* and *area*.
The results seem not to be specific to the problem of HW run-time estimation only. So we are motivated to refine other high-level estimation techniques (e.g. area estimation) also. Our future work will concentrate on this point.

References

[1] A.W. Aho, R. Sethi and J.D. Ullmann, *COMPILERS Principles, Techniques and Tools,* Bell Telephone Laboratories, 1987.

[2] 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.

[3] K. Buchenrieder and C. Veith, *CODES: A Practical Concurrent Design Environment,* Handout from Int'l Workshop on Hardware-Software Co-Design, Estes Park, Colorado, Oct. 1992.

[4] P. H. Chou, R. B. Ortega, G. B. Borriello, *The Chinook Hardware/Software Co–Synthesis System*, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 22–27, 1995.

[5] J. G. D'Ambrosio, X. Hu, *Configuration–Level Hardware/Software Partitioning for Real–Time Embedded Systems*, IEEE/ACM Proc. of 3rd. Int. Workshop on Hardware/Software Codesign, pp. 34–41, 1994.

[6] M. Edwards, J. Forrest, *A Development Environment for the Cosynthesis of Embedded Software/Hardware Systems,* Proc. of EDAC'94, pp. 469–473, 1994.

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

[8] 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.

[9] J. Henkel, Th. Benner, R. Ernst, W. Ye, N. Serafimov and G. Glawe, *COSYMA: A Software–Oriented Approach to Hardware/Software Codesign,* The Journal of Computer and Software Engineering, Vol. 2, No. 3, pp. 293–314, 1994.

[10] 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.

[11] J. Henkel, R. Ernst, *A Path-Based Estimation Technique for Estimating Hardware Runtime in HW/SW-Cosynthesis*, IEEE/ACM Proc. of 8th. Int'l Symp. on System Level Synthesis, pp. 116–121, 1995.

[12] T.B. Ismail, M. Abid, A. Jerraya *COSMOS: A CoDesign Approach for Communicating System,* Proc. of 3rd IEEE Int. Workshop on Hardware/Software Codesign, pp. 17–24, 1994.

[13] 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.

[14] 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.

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

[16] 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.

[17] M. B. Srivastava, R. W. Brodersen, *SIERA: A Unfied Framework for Rapid–Prototyping of System–Level Hardware and Software*, IEEE Trans. on CAD, Vol. 14 No. 6, pp. 676–693, June 1995.

[18] M. Theisinger, P. Stravers, H. Veit, *Castle: An Interactive Environment for Hw–Sw Co–Design,* Proc. of 3rd IEEE Int. Workshop on Hardware/Software Codesign, pp. 203–209, 1994.

[19] 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.

[20] T. Y. Yen, W. Wolf, *Multiple–Process Behavioral Synthesis for Mixed Hardware–Software Systems*, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 4–9, 1995.