High-Level Estimation Techniques for Usage in Hardware/Software Co-Design

Jörg Henkel

C&C Research Laboratories
NEC USA, Princeton, NJ 08540
henkel@ccrl.nj.nec.com

Rolf Ernst

Institut für DV –Anlagen
Technische Universität Braunschweig
D–38106 Braunschweig, Germany

Abstract--- High-level estimation techniques are of paramount importance for design decisions like hardware/software partitioning or design space explorations. In both cases an appropriate compromise between accuracy and computation time determines about the feasibility of those estimation techniques. In this paper we present high--level estimation techniques for hardware effort and hardware/software communication time. Our techniques deliver fast results at sufficient accuracy. Furthermore, it is shown in which way these techniques are applied in order to cope with contradictory design goals like performance constraints and hardware effort constraints. As a solution, we present a cost function for the purpose of hardware/software partitioning that offers a dynamic weighting of its components. The conducted experiments show that the usage of our estimation techniques in conjunction with their efficient combination leads to reasonable hardware/software implementations as opposed to approaches that consider single constraints only.

I. Introduction

Though complex systems–on–a–chip are already reality ([1]), accurate and fast high–level estimation techniques are still rare. Systems–on–a–chip comprise software parts (e.g. processor cores running a software program) and hardware parts (dedicated hardware implementing system parts with high performance constraints). The tradeoff between these hardware and software parts is controlled by design constraints like limitation of the die size, performance considerations, design time, design flexibility, design re–use etc. This large variety of constraints makes it almost impossible or at least very hard for the system designer to find a sophisticated compromise that best matches these constraints in a particular case. So, the *current* procedure of providing a customer with such a design is as follows: the vendor delivers a couple of designs with slightly different focuses to the costumer who decides for that one that meets best his constraints. Afterwards, one or more refinements on the favorite design are conducted. The whole process is iterative and takes up to half a year. This time–consuming and therefore expensive process could be drastically reduced if high–level estimation methodologies would be performed *before* mayor design decisions are made. The intention is to explore the design space with respect to different design constraints. As a result, fast and sufficiently accurate statements concerning a possible implementation can be obtained. At that point the designer can be guided towards his next design steps while preventing misleading design decisions.

In this paper we introduce some high–level estimation techniques that aim at exploring the design space as well as guiding the process of hardware/software partitioning. Due to limited space in this paper we concentrate on two high–level estimation techniques: hardware effort and hardware/software communication estimation. While the first technique is useful for estimating the implementation cost, the other techniques is part of estimating the performance in a hardware/software system. As mentioned above, design constraints are often manifold. As an example, for a special class of systems–on–a–chip — hard real–time systems — there is a hard timing constraint as well as the constraint to keep the hardware effort as small as possible. We show in which way this contradictory constraints can be combined into one single cost function. Furthermore, we demonstrate that this technique — a dynamically weighted cost function — can save hardware effort as opposed to techniques that take into consideration one constraint at a time only.
Not less important are techniques for software run–time estimation and hardware run–time estimation. As for our approach, those techniques are described in [17, 14].

This paper is structured as follows: section II gives a short overview of research activities in co–synthesis as far as these approaches focus on hardware/software partitioning *and* provide a cost function that considers at least two different constraints — such as run–time and hardware effort, for example. Other approaches to co–synthesis and especially to hardware/software partitioning are not introduced here since partitioning and co–synthesis are not the primary goals of that paper. For a summary of those approaches, the interested reader is referred to [2]. The following two sections are dedicated to the estimation of hardware effort and hardware/software communication, respectively. Each of these sections also contains an overview of related work in that specific area rather than to include that in section II. Afterwards, section V presents our approach of combining these different estimation techniques into one single cost function with the goal of meeting a real–time constraint and at the same time minimizing the hardware effort. For this purpose, a dynamically weighted cost function is presented. Results of this combination are presented in section VI. However, individual results i.e. those referring to estimation techniques solely, are presented in the according sections. Finally, section VII gives a conclusion.

II. Related work

Rather than giving an overview of all approaches to system–level design techniques and estimation, we introduce two approaches that explicitly describe their method of reducing the hardware effort while meeting real–time constraints during hardware/software partitioning. More approaches to hardware/software co–design are described in [2]. Furthermore, related work to specific high–level estimation techniques is introduced within this paper in the according sections.

Kalavade and Lee [3] describe an algorithm called *GCLP* to minimize hardware effort while meeting timing constraints. The main idea is to select different cost functions during hardware/software partitioning according to a criticality measure that is computed for each possible hardware/software partitioning.
Vahid and Gajski [4] use a binary search algorithm to minimize hardware effort while meeting timing constraints during partitioning. They relax the contradictory goals of low hardware effort and timing constraints by accepting all hardware efforts below a given size rather than trying to optimize hardware effort and timing at the same time. They use an incremental algorithm for estimating hardware effort that is adapted to function–level partitioning [5].

Our approach handles the contradictory goals of minimizing hardware and meeting real–time constraints by a cost function that is dynamically weighted rather than selecting from a set of different cost functions.

III. A high-level Hardware Effort Estimation Technique

The aim of this hardware effort estimation technique is to obtain helpful hints about the possible hardware effort of various system parts *before* the final decision of partitioning into hardware and software is done. This is one mayor difference compared to methods who assume that a particular piece will be implemented as a hardware in any case. As a consequence, our technique is subject to different constraints. It should a) deliver fast estimations, b) be as far as possible independent from synthesis c) easily get adapted to various synthesis tools

A. Related work in high-level hardware effort estimation

The technique presented in [7] estimates hardware effort and delay time for functional units only. They calculate a *minimum* hardware effort for a given delay time or a *minimum* delay time for a given amount of hardware resources. The approach described in [8] is an extension of [7] that additionally takes into consideration hardware effort for connectivity. The approach described in [9] is interval based. Due to not considered data dependencies between intervals the actual demand on resources might be larger. In [10] a lower bound estimation technique for data dominated applications is presented. The approach in [11] is very complete since it takes into consideration data path resources as well as the hardware effort for the controller and the memory. However, statistical assumptions may lead to larger deviations. Furthermore, it is not assumed that a scheduling has already been performed. The method introduced in [5] is one of the few that is tailored to the demands of hardware/software co–design since it features an *incremental* estimation for each possible hardware/software partitioning. The method is accurate for coarse–grain estimation (e.g. estimating whole functions or tasks) but inaccurate for small–grain estimation (base blocks etc.).

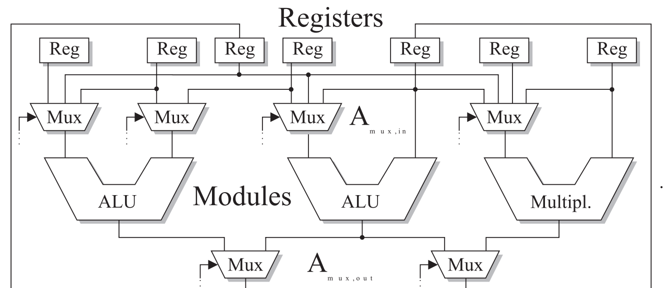


Fig. 1. Components to estimate: Registers, input multiplexer, modules and output multiplexer

To summarize, there are some fast but not sufficiently accurate approaches available. Their inaccuracy results from their principle: they are lower bound methods that, for example, do not assume that a scheduling has already been performed.

Our approach is comparable to those in terms of execution times but due to a previously performed scheduling as well as not relying on statistical data, it promises to be more accurate. An additional feature is the adaptation to different synthesis tools. We made the experience that especially the controller of a hardware design is very specific to the applied synthesis tool i.e. hard to estimate in a general approach. Therefore, this part of the estimation is separated from the estimation of the data path.

B. Our Approach

Our approach takes into consideration controller as well as data path estimation. But only the estimation of the data path is presented here[1], because the controller part needs to be adapted to the synthesis tool we use before a satisfactory result can be obtained. An important feature of our technique is performing scheduling before the estimation is started.

In the following, the algorithms for estimating *modules*, *data registers*, and *multiplexer* are presented (Fig. 1 gives an example of a data path). The estimation takes place at a CDFG (control and data flow graph) representation that is directly derived from a C system–level description.

B.1 Modules

A module m performs an operation $o_{i,l}$ where i denotes the control step it is performed in and l refers to the 3–address–code it is part of. Let $M = \{m_1, \ldots, m_D\}$ be the set of different

[1] Estimating the hardware effort of the controller is described in detail in [6].

```
1)  Initialize global ModulList
2)  For all  cs_i ∈ CS
3)    Initialize local ModulList
4)    For all  o_{i,l} ∈ O_l
5)      Compute sorted ModulList
6)      m_π := first element of sorted ModulList
7)      For all  elements of sorted ModulList
8)        m_π :=current ModulType of sorted ModulList
9)        If number(m_π)_{m_π ∈ mboxglob. Mod. List} >
10)         number(m_π)_{m_π ∈ lok. Mod. List}
11)       then
12)           Increment number of ModulType m_π in
13)           local ModulList by 1; goto 2)
14)       Incr. # of ModulType m_π in local ModulList by 1
15)   update global ModulList by means of local ModulList
16)  Initialize hardware effort A_mod of Modules
17)  For all  ModulTypes of global ModulList
18)    A_mod := A_mod + number(m_π) × A(m_π)
```

Fig. 2. Algorithm for estimating the number and types of modules

module types with D the total number. Furthermore, assume that for each module m of type π, i.e. m_π, there are one or more instances. Figure 2 shows the algorithm for estimating the number of necessary modules. The main loop (2) is iterated for all control steps cs_i in the set of all control steps CS. *Global* and *Local Module List* have the same structure since they both provide an entry for each module type and specifying their number of them that has been used so far. Now, the inner loop starting at (4) is iterated for all operators. The purpose of the *sorted module list* is to provide all possible modules i.e. those that might be able to execute operation $o_{i,l}$. The list is sorted in an ascending order of hardware effort for each module. Afterwards, an initial module is selected (6). In the following steps (9 to 13), all fitting module types are tested in terms of previous (i.e. within another control step) instantiation. If that is the case then this module type is assigned to the current operator and an additional instance of this module type is not necessary. This assignment is noted in the *local module list* and afterwards the next operator is selected (12). In the other case, the *local module list* is searched for an already instantiated but in the current control step not used instance of the specific module type. If the search fails, the first and as mentioned the least hardware consuming module is assigned to the current operator and the *local module list* is updated (14). After all operators of a specific control step have been served, the *global module list* is enlarged by as many entries as the *local module list* has (15). So, after all operators and all control steps are treated, the *global module list* contains the number of instances of all module types (18).

B.2 Data Registers

Data registers hold temporary results or data that is loaded from memory or going to be stored into memory. Since registers are expensive, as many as possible variables should share one register. Therefore, a life–time interval $t_L(X) := [cs_a, cs_b]$ for

```
1)  Register#_max = 0
2)  Calculate all pathes 𝒫 = P_1, ..., P_n of graph G = {V, E}
3)  For all  pathes P_i ∈ P
4)    Search all locations def(i)
5)    For all  definitions of variables def(i)
6)       Search last location use(i)
7)       T_L := T_L ∪ t_L(i)
8)    Register# = CliquePart(T_L)
9)    If
10)       Register# > Register#_max
11)    then
12)          Register#_max = Register#
13)  A_reg = Register#_max × A_single register
```

Fig. 3. Algorithm for estimating the number of data registers. Graph G is a CDFG.

each variable is calculated. In this context, cs_a is the control step in which variable x is defined for the first time i.e. a value is assigned to x. Furthermore, cs_b denotes the control step in which variable x is used for the last time. Taking into consideration that a variable can have several life time intervals within a program execution, the set of all life time intervals of a variable is defined. Using these definitions, figure 3 shows the algorithm for estimating the number of necessary registers. Starting with initializing the number of registers (1), the main loop iterates for all paths [2] (2), (3). Then, for all variables the set of life times are calculated (4) to (7). Afterwards, the minimum number of registers is calculated using the *clique partitioning* algorithm (8) [13]. In the followings steps (9) to (12) the number of registers is updated and finally the hardware effort of all necessary registers is calculated by multiplying the number of estimated registers by the amount of gates used for one register (13).

B.3 Multiplexer

The task of the multiplexer in a multiplex–based data path is to distribute the values in the data registers among the modules (ALUs, multipliers,...) and to forward results of operations to registers. These both types of multiplexer are called input and output multiplexer, respectively. The algorithm presented in the following (input multiplexer), assumes that a module estimation as well as estimation of data registers have already been carried out. Consequently, at this point it is known which value is contained in which register during each control step. Furthermore, it is known which operation is executed it which module. In the following, the estimation of input multiplexer is presented (fig. 4).

The first loop (2) is iterated for all types of modules whereas the nested loop (3) is iterated for all instances of that particular module type. The number of input lines of a multiplexer n_E is calculated by means of $|O_{m_{\pi,i}}|$ i.e. the number of operators

[2] We have developed an algorithm for keeping the number of relevant paths small. Consequently, we do not have to cope with a polynomial path complexity problem as described in [12]. Our method is described in detail in [14].

1) $N_{mod} = 0$
2) **For all** ModulTypes $\in$ *globale ModulList*
3) **For all** Instances of ModulType m_π
4) $N_{mod} = N_{mod} + 1$
5) $n_E = 0$
6) **For both** Operands
7) $n_E = n_E + |O_{m_{\pi,i}}|$
8) $MuxList[N_{mod}] = n_E$
9) $A_{mux,in} = 0$
10) **For all** Elements m_i of *MuxList*
11) **If**
12) $MuxList[m_i] > 1$
13) **then**
14) $A_{mux,in} = A_{mux,in} + A_{mux}(n_E)$

Fig. 4. Algorithm for estimating the number and types of input multiplexer

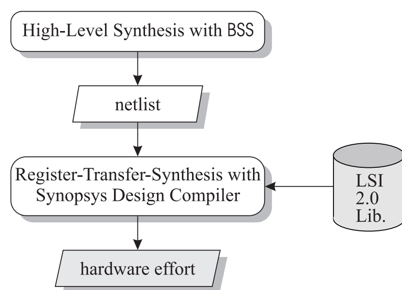


Fig. 5. Estimation results compared to synthesis results

that are intended to be executed in this instance of that module (known from module estimation). The number of modules N_{mod} and the according number of input lines are stored in $MuxList[]$. Once, the number of multiplexer and their input lines are known, the according hardware effort (in terms of gate equivalents) is calculated advising an according data base.

The estimation of the output multiplexer is performed in a similar way but omitted in this paper for lack space.

C. Estimating hardware effort: results

The result of our hardware estimation technique is the sum

$$A_{estim} = A_{mod} + A_{reg} + A_{mux} + A_{control}$$

where A_{mod}, A_{reg}, A_{mux}. $A_{control}$ are the hardware effort for modules, registers, multiplexers and the control unit, respectively. As mentioned above, $A_{control}$ is not described here. In order to guarantee a precise comparison of our estimation technique, the following steps are performed for each example: high–level synthesis using our system BSS is carried out and the resulting RTL description is fed into the SYNOPSYS design compiler by means of the LSI 2.0 library. Output is the hardware effort in terms of gate equivalents (1 gate equivalent (geq) is equal to 4 transistors). See fig. 5 for the synthesis steps. Later on, estimation results and synthesis results are compared. Before, the adaption of estimation to synthesis is discussed: as

application	method	DP1	DP2	DP3
bpic	estim	10,618	9,188	9,188
	synth	7,697	8,848	9,101
hash	estim	6,976	7,486	29,814
	synth	3,093	8,267	11,773
huffman	estim	22,887	9,944	26,463
	synth	6,904	7,272	7,538
labeling	estim	4,020	2,411	5,849
	synth	1,123	2,450	2,459
table	estim	2,380	14,906	14,982
	synth	1,088	15,057	16,483
trick	estim	13,290	27,700	29,130
	synth	3,449	12,837	15,084
turmits	estim	16,060	40,614	42,413
	synth	3,593	3,736	12,378

TABLE I
COMPARISON OF OUR ESTIMATION TECHNIQUE FOR HARDWARE EFFORT (IN GATE EQUIVALENTS) AND SYNTHESIS RESULTS FOR THREE DIFFERENT DESIGN POINTS (DP1 TO DP3) OF EACH APPLICATION

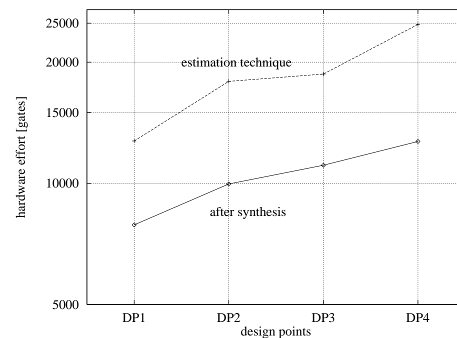


Fig. 6. Comparison of estimation results and synthesis results in terms of hardware effort for the application *sm1*

mentioned earlier, scheduling precedes hardware estimation. Since the scheduling method in the high–level synthesis system BSS is a kind of path–based scheduling, we carried out our path–based estimation technique ([14]). The controller estimation is tailored to the high–level synthesis system BSS since our experience has shown that a more general controller estimation technique leads to unacceptable deviations. The data path on the other hand is estimated n a more general way, as shown above.

As an example for the experiences we conducted, fig. 6 compares estimation and real synthesis by using different design points of an application from the domain of digital signal processing. Different design points means that either the whole application or only parts of it have been estimated/synthesized. Table I shows more results of other application, each time comparing three different design points. As can be seen, all estimation results are in the same order of magnitude. The estimation times are only a few seconds to a few minutes as opposed to results obtained by synthesis who took in most cases several hours. So, our estimation method is well suited for fast

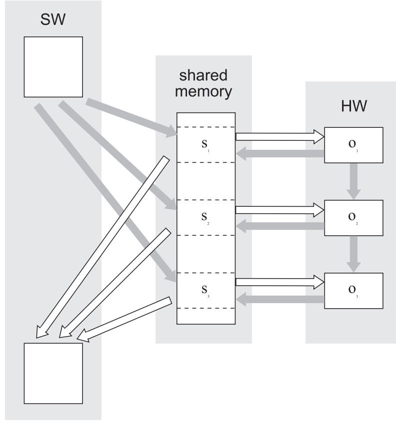


Fig. 7. Example for our communication model

design space explorations as well as high–level estimations to guide the designer during decisions like hardware/software partitioning.

As can be observed, in most cases the number of estimated gate equivalents is larger than the according number obtained by synthesis. This is due to the fact that all the optimization steps carried out in high–level synthesis, RTL synthesis and logic synthesis cannot — by principle — be taken into consideration during estimation.

The superiority of our approach in terms of accuracy and run-time as opposed to other estimation approaches is due to the following aspects: we perform a scheduling before estimation of hardware effort, we use an advanced method for path calculation [14], we rely on a fixed sequence of estimation (first modules, then data registers and finally multiplexer) and last but not least we separate data path and control path estimation meaning that control path estimation is tailored to the high–level synthesis system whereas data path estimation is more generic.

IV. Estimating Hardware/Software Communication

Our hardware/software model is as follows: a software program is executed on a standard processor core. At several points in the program a hardware routine is called, thereby transmitting control from software to hardware. After the hardware part has completed execution, control is given back to the software. When control is transmitted, also data has to be transferred from software to hardware and vice versa since both sides may operate on the same data. The estimation technique presented here, estimates this communication.

A. The estimation approach

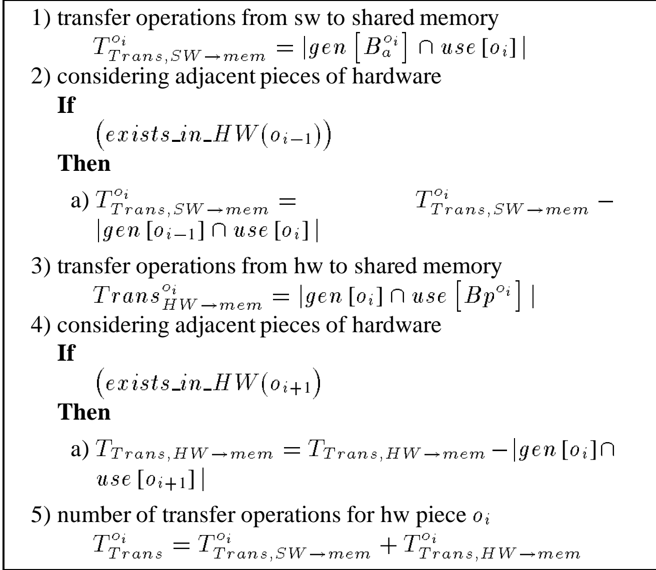
Fig.7 gives an example of our model for communication estimation. The boxes on the left hand side symbolize a software program, the boxes on the right hand side symbolize hardware i.e. algorithms or part of algorithms that are implemented as a synthesized hardware. In the middle there are different sections of shared memory where communication is performed through. All arcs are shown symbolize data transfers. For the purpose of simplicity, arcs representing control transmission, are omitted. Our communication model takes into consideration the data transfer into the shared memory and the data transfer from the shared memory into the processor or into the hardware. The effort for transmitting control from software to hardware or vice versa, is not reflected in our estimation technique since we found out that this part is pretty small. So, for one piece of hardware o_i that is called by software, the following communication overhead arises:

$$T^{o_i}_{Trans,SW\rightarrow HW} = T^{o_i}_{Trans,SW\rightarrow mem} + T^{o_i}_{Trans,mem\rightarrow HW} \quad (1)$$

The first part stems from transferring data from software to shared memory whereas the second part stems from transferring data from the shared memory into the hardware. A similar communication effort arises when control is from hardware to software. In a real implementation this would mean a very large communication effort. What happens in reality is: data that is already in hardware (e.g. produced by hardware piece o_1 in the fig.) *and* used by another piece of hardware (e.g. o_2 in fig.7) will be directly transferred between these two hardware parts without the detour through the shared memory, as applicable. The problem is that estimating this more sophisticated way of communication is a very computation intensive task: each piece of hardware can potentially transmit data to each other piece of hardware, implying a number of 2^N of possible transfers, with N the number of all hardware pieces. This is a cumbersome way for estimation because of large estimation times. Fortunately, we observed that the greatest amount of communication arises between adjacent hardware pieces i.e. those hardware pieces that are executed consecutively. Therefore, we restrict our estimation only to those transfers. So, we make use of the so–called *locality of reference* which means that data is likely to be used where it is produced.[15] Fig.8 shows the algorithm for our estimation technique. The following conventions are valid: o_i is the current piece of hardware, o_{i-1} and o_{i+1} are the pieces of hardware that precede and succeed o_i in the control flow, respectively. Furthermore, $gen[o]$ means the set of all variables defined in piece o and $use[o]$ means the set of all variables used in piece o.[3] $B^{o_i}_a$ represents all pieces (no matter if software or hardware) that are executed *before* o_i and $B^{o_i}_p$ represents all pieces that are executed *after* o_i.

Line (1) estimates the communication overhead of data transfers from software into the shared memory according to equation 1. If there is a preceding piece of hardware (2), the communication is reduced by that portion of variables that are used and defined in the preceding piece, meaning that a direct transfer among the hardware pieces can be performed without the detour through the shared memory. The communication overhead from hardware to software is calculated in a similar manner (lines (3) to (4)). Finally, the total communication is calculated (5).

[3] gen and use have the same meaning as defined in [15].



1) transfer operations from sw to shared memory
$T^{o_i}_{Trans,SW \rightarrow mem} = |gen\left[B^{o_i}_a\right] \cap use\left[o_i\right]|$
2) considering adjacent pieces of hardware
If
$\left(exists_in_HW(o_{i-1})\right)$
Then
a) $T^{o_i}_{Trans,SW \rightarrow mem} = T^{o_i}_{Trans,SW \rightarrow mem} - |gen\left[o_{i-1}\right] \cap use\left[o_i\right]|$
3) transfer operations from hw to shared memory
$Trans^{o_i}_{HW \rightarrow mem} = |gen\left[o_i\right] \cap use\left[Bp^{o_i}\right]|$
4) considering adjacent pieces of hardware
If
$\left(exists_in_HW(o_{i+1})\right)$
Then
a) $T_{Trans,HW \rightarrow mem} = T_{Trans,HW \rightarrow mem} - |gen\left[o_i\right] \cap use\left[o_{i+1}\right]|$
5) number of transfer operations for hw piece o_i
$T^{o_i}_{Trans} = T^{o_i}_{Trans,SW \rightarrow mem} + T^{o_i}_{Trans,HW \rightarrow mem}$

Fig. 8. Algorithms to estimate hardware/software communication

B. Results

As a result of the above estimation technique, estimation times are very fast: even for larger applications (about 1000 lines of code) the estimation times are within a few seconds. The chart in fig.9 shows the percentage of savings in communication effort when using our technique (taking into consideration the principle of locality of reference) as opposed to a method that assumes for each piece of hardware a communication through the shared memory. As can be seen, the savings are between 20% and 100%. Further studies have shown that estimating the transfers between pieces of hardware that are not adjacent, lead only to a slight reduction in communication but to a much larger estimation times. Our approach for estimation communication therefore offers a good compromise between accuracy and estimation time and consequently it is well suited as a high level estimation technique to explore the design space or to guide hardware/software partitioning.

V. Applying our fast estimation techniques for hardware/software partitioning

In the previous sections, we pointed out that our estimation techniques are well suited for usage in design space explorations and for guiding the process of partitioning a system into hardware and software parts.

As an example for the application of our fast estimation techniques, we present a dynamically weighted cost function that aims at minimizing both, hardware effort and execution time of a hardware/software system. However, the whole optimization algorithm is not presented here since it is already published in [16]. Here, we concentrate on the new part of the dynamically weighted cost function and present new encouraging results in section VI.

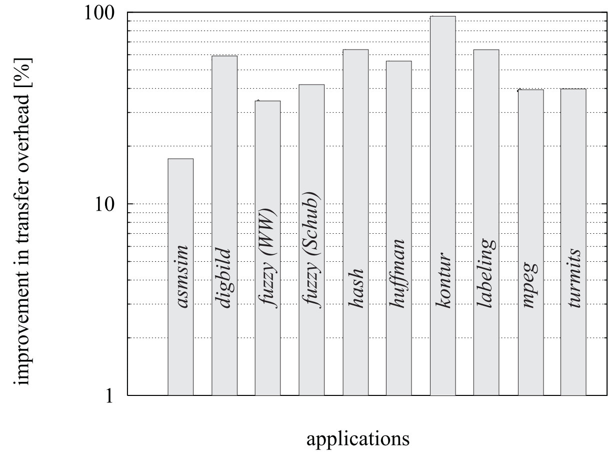


Fig. 9. Percentage of savings in communication of our technique as opposed to a technique that is slightly faster but more conservative

A. Dynamically weighted cost function

It is assumed that for the purpose of hardware/software partitioning an optimization algorithm is used which optimizes by iterative improvement. That means, for each proposed hardware/software partitioning the cost function calculates the cost in terms of the design goals (in our case meeting real–time constraints and minimizing the hardware effort). The optimization algorithm proposes a new hardware/software partitionings until no more improvements are possible. Our optimization procedure has been presented in [16]. Here we concentrate on our new cost function that is dynamically weighted in order to minimize the hardware effort while meeting real–time constraints. The cost function is defined as follows:

$$Cost = \underbrace{a \cdot cost_T(T_{sys})}_{time\ component} + \underbrace{b \cdot w_{area} \cdot \frac{Area}{\overline{A}}}_{area\ component}. \qquad (2)$$

We will discuss the single components in the following, especially the weighting factor w_{area} which varies dynamically, dependent on how close the time constraint is met. A static weighting by using the two constant factors a and b only is *not* sufficient for our purpose of meeting a real–time constraint *and* minimizing the hardware effort at the same time. But before, let us discuss another problem: since *time* and *area* have different physical units, we have to normalize our components. As for the hardware component, we calculate the area of a piece of hardware (using our estimation method presented in section III) and divide it by the average area of all other pieces of hardware $\overline{A}$. Consequently, we get a relative number that is in average close to 1.
The time component is normalized as follows:

$$cost_T(T_{sys}) = \frac{|T_{sys} - T_{constr}|}{T_{constr}} + 1$$

Here, T_{constr} is the real–time constraint and T_{sys} is the current execution time of the system. Since our aim is

$$T_{sys} \leq T_{constr}$$

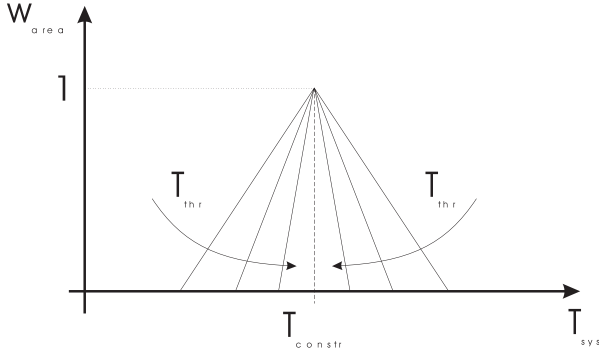


Fig. 10. Factor w_area as a function of T_{sys}

we define the deviation $|T_{sys} - T_{constr}|$ as cost and put it in relation to T_{constr} in order to get a unit–less number that can be combined with the area component. The constant 1 is added for the case where $T_{sys} = T_{constr}$, meaning that in that case the area component would totally dominate unless we add 1.

So far, the cost function has been built straight forward. However, we would never gain our aim of meeting a real–time constraint and minimizing the hardware effort unless we would care about w_{area}. A desirable behavior of w_{area} would be that it is 0 when we are far away from meeting the real–time constraint. This means that in such cases timing behavior is optimized exclusively. But when T_{sys} is close to T_{constr}, the area component should start to become more and more influence until T_{sys} is almost equal to T_{constr} where w_{area} should be maximum. This is a dynamically weighting, meaning that w_{area} is dependent on the timing component. The following definition fulfills the desirable behavior:

$$w_{area}(T_{sys}) = \begin{cases} \frac{T_{sys}-T_{thr}}{T_{constr}-T_{thr}} & : T_{thr} \leq T_{sys} \leq T_{constr} \\ \frac{2 \cdot T_{constr}-T_{thr}-T_{sys}}{T_{constr}-T_{thr}} & : T_{constr} \leq T_{sys} \leq \\ & : 2 \cdot T_{constr} - T_{thr} \\ 0 & : else \end{cases}$$

The value of T_{thr} is obtained by experience. In a more apparent way, w_{area} is shown in figure 10. Unless the system timing T_{sys} is far away from T_{constr}, $w_{area} = 0$. If T_{sys} is close to T_{constr} (T_{thr}), w_{area} starts to increase until it becomes maximum (i.e. 1). Now the area component is dominating over the time component. This is desirable since the time at this stage has already been optimized. So, the dynamically weighting offers the possibility of separating area and time dependent on the current state — without switching to another cost function (this might be very computation intensive). By means of the factors a and b in equation 2 it is possible to control the extent to which the area component should dominate during the end–phase of the optimization procedure.

The experiments presented in the next section will show that this technique is superior to techniques that combine time and area in a fixed manner or that do not consider area as a minimization goal.

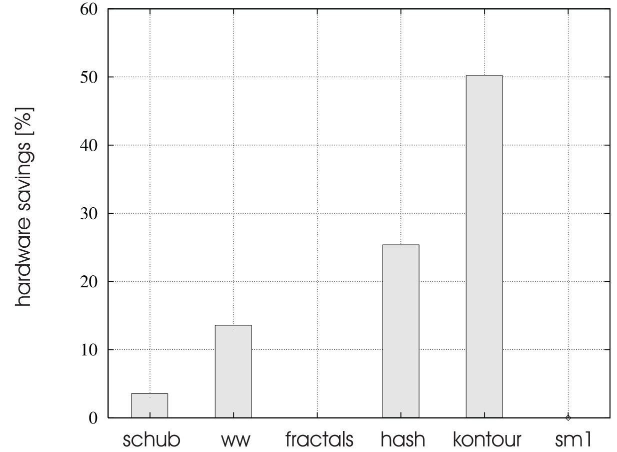


Fig. 11. Savings in hardware effort using the dynamically weighted cost function as opposed to a cost function without dynamic weighting of the hardware component.

VI. Results

In the sections III and IV results solely demonstrating the advantage of the particular estimation technique, have been presented. In this section we present results that make use of the techniques described there as well of the estimation techniques already published ([17, 14]). These techniques are applied to a cost function that aims at meeting real–time constraints while minimizing the hardware effort of a system–on–a–chip. The main feature of that cost function is the dynamically weighted area component as described in the previous section.

The applications have a size between about 50 lines of code and about 600 lines of C code. They are a smoothing algorithm, contour detection, an HDTV chromakey algorithm, a hash function and a simulation program for a spacecraft. The results are summarized in table II.

The six applications are listed in columns three to eight. Rows one and three show the according hardware effort (in gate equivalents) and rows two and four hold the execution times (in clock cycles) of each application. There are two areas in the table: the first two rows belong to a cost function that does not feature a dynamically weighted component for the hardware effort whereas rows three to four belong to to the cost function described in section V. It can bee seen that the hardware effort with the dynamically weighted hardware component (*"with"*) is in most cases smaller than in the case without (*"w/o"*). Thereby — and that is very important — the execution time of the application is only slightly different or it is even the same. This reflects that our cost function can find better solutions as opposed to a cost function that has no dynamically weighted hardware component. The percentage in hardware savings is summarized in fig. 11.

Another important result is that all design points could be obtained within a few minutes. As a consequence, our estimation techniques are well suited for fast design space analysis to explore a hardware/software system.

area comp.	measured	"‘*schub*"’	"‘*ww*"’	"‘*fractals*"’	"‘*hash*"’	"‘*kontour*"’	"‘*sm1*"’
w/o	HW Effort	27448.0	49348.0	17605.5	64117.0	118295.0	16830.5
	T_{sys}	227077	433387	322341	5602	2374158	277429
with	HW Effort	26418.5	42793.0	17605.5	48150.0	59623.5	16830.5
	T_{sys}	226758	433472	322341	5602	2378059	277429

TABLE II
HARDWARE EFFORT USING THE DYNAMICALLY WEIGHTED COST FUNCTION AS OPPOSED TO A COST FUNCTION WITHOUT DYNAMIC WEIGHTING OF THE HARDWARE COMPONENT.

VII. CONCLUSION

In this paper we have presented a fast high–level estimation technique for the hardware effort as well as a fast estimation techniques for estimating hardware/software communication. In the second part of this paper we applied these estimation techniques together with a fast hardware run–time and software run–time estimation technique ([17, 14]) to a cost function that combines a run–time constraint and a constraint to minimize the hardware effort. This was possible through a technique of dynamically weighting the hardware effort component. The results have shown that this cost function in conjunction with the fast estimation techniques leads to cheaper implementation cost (less hardware effort) while still meeting the timing constraints. So, our technique is well suited for fast design space explorations that can support a designer by decisions like hardware/software partitioning.

REFERENCES

[1] *TI’s 0.18 Micron Process Technology Packs 125 Million Transistors on a Single Chip*, Texas Instruments, Published in the Internet, http://www.ti.com/corp/docs/-pressrel/1996/96025b.htm, 1996.

[2] J. Henkel, F. Vahid, L. Ramachandran, *Hardware/Software Co–design of Embedded Systems,* Held as tutorial at Europ. Design and Test Conf. 97, Paris, March 17th. 1997.

[3] 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.

[4] F. Vahid, D.D. Gajski, J. Gong, *A Binary–Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning,* IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1994, pp. 214–219, 1994.

[5] F. Vahid, D.D. Gajski, *Incremental Hardware Estimation during Hardware/Software Functional Partitioning,* IEEE Trans. on VLSI Systems, Vol.3, No.3, pp. 459–464, Sept. 1995.

[6] C. Stüdemann, *Implementierung eines Schätzverfahrens für den Hardwarebedarf aus der Verhaltensbeschreibung einer Hardware*, Master Thesis, Institut für DV–Anlagen, Technische Universität Braunschweig, 1996.

[7] R.J. Jain, A.C. Parker, N. Park, *Predicting System–Level Area and Delay for Pipelined and Nonpipelined Designs,* IEEE Trans. on CAD, Vol.11, No.8, pp. 955-965, Aug. 1992.

[8] A. Sharma, R. Jain, *Estimating Architectural Resources and Performance for High–Level Synthesis Applications*, IEEE Trans. on VLSI Systems, Vol.1, No.2, pp. 175–190, June 1993.

[9] S.Y. Ohm, F. Kurdahi, N. Dutt, *Comprehensive Lower Bound Estimation from Behavioral Descriptions*, IEEE/ACM Proc. of International Conference on CAD (ICCAD) 1994, pp. 182–186, 1994.

[10] J.M. Rabaey, M. Potkonjak, *Estimating Implementation Bounds for Real Time DSP Application Specific Circuits*, IEEE Trans. on CAD, Vol.13, No.6, pp. 669–683, June 1994.

[11] S. Narayan, D.D. Gajski, *Area and Performance Estimation from System–Level Specifications*, University of California Irvine, Dept. of Information and Computer Science, Technical Report ICS-92-16, 1992.

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

[13] C.J. Tseng, D.P. Siewiorek, *Automated Synthesis of Data Paths in Digital Systems*, IEEE Trans. on CAD, Vol. 5, No. 3, Juli 1986.

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

[15] A.V. Aho, R. Sethi, J.D. Ullmann, *Compilers Principles, Techniques, and Tools,* Addison–Wesley Publishing Group, 1986.

[16] J. Henkel, R. Ernst, *A Hardware/Software Partitioner using a dynamically determined Granularity*, IEEE/ACM Proc. of 34th. Design Automation Conference (DAC) 1997, pp. 691–696.

[17] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Harware–Software Cosynthesis,* IEEE/ACM Proc. of International Conference on Computer Design (ICCD) 1993, pp. 452-457, 1993.