A Hardware/Software Partitioner using a dynamically determined Granularity

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

Computer aided hardware/software partitioning is one of the key challenges in hardware/software co-design. While previous approaches have used a fixed granularity, i.e. the size of the partitioning objects was fixed, we present a partitioning approach that dynamically determines the partitioning granularity to adapt optimization steps to application properties and to intermediate optimization results. Experiments with simulated annealing optimization show a faster convergence and far better adaptability to cost function variations than in previous experiments with fixed granularity.

1 Introduction

Computer aided hardware/software partitioning is one of the most challenging tasks in co-design. With 100 million transistor systems-on-a-chip [1], it will become a key factor for design space exploration and, eventually, be used in automated co-synthesis flow to speed up the design process.

Even today, interactive co-synthesis has successfully been used in first industrial projects, where 50% saving (including development and product cost) could be reported [2].

However, computer aided partitioning must always be compared to manual design decisions. While many partitioning approaches just consider partitioning and allocation at the level of complete functions or even processes, at least data dominated applications such as high speed signal processing need a finer granularity, e.g., to make use of loop parallelization and pipelining as the designer would do. Manual rewriting of functions to get smaller functions just for the purpose of partitioning is a tedious and error–prone task. Partitioning with a finer granularity, however, suffers from a huge design space which might not be necessary for many system functions and is therefore bound to make optimization less efficient.

In a manual design the designer typically adapts his or her focus to the critical parts of a system. Applied to computer aided partitioning, this means to depart from the fixed granularity of current approaches and introduce a dynamically adapted granularity.

In this paper we present a new method for automated hardware/software partitioning since the granularity — i.e. the size of system parts that can either be implemented as hardware or as software — is determined *during* the partitioning process i.e. it is *dynamically determined*. This offers the opportunity to obtain a better adaptation of the partitioning process and a specific application and hence to obtain better partitioning results in terms of implementation cost and computation times used for partitioning.

The outline of this paper is as follows: In section 2 an overview of approaches to hardware/software co–synthesis and especially to hardware/software partitioning is given. Then section 3 gives an overview of our approach whereas section 4 describes the according algorithms in detail. Section 5 describes the results that confirm the usefulness of the approach. Finally a conclusion is given.

Design Automation Conference ®

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

2 Related work

The first approaches to automated hardware/software partitioning have been presented in the VULCAN II system [3] and in COSYMA [4]. Both approaches use a *fine–grain partitioning*(basic block). *Coarse–grain partitioning* on the other side means that whole functions or processes are moved from software to hardware or vice versa in order to find the best hardware/software tradeoff.
The approaches described in [5, 6, 7, 8, 10, 11, 12] use also fine–grain partitioning whereas [13, 14, 16, 17, 18] use coarse–grain partitioning. Some other approaches do not perform automatic partitioning but concentrate on interface synthesis [19, 20] or on co-simulation [21].
All these approaches have in common that their granularity is fixed. Our approach has previously also been limited to relatively fine grain. (basic block). The new partitioning approach is not any longer limited in this way since it determines the granularity *during* the partitioning process (i.e. *dynamically*) through moving very small and very large system parts — according to the peculiarities of a specific application and the given constraints.

3 Problem definition and fundamental approach

Since different applications have different peculiarities like

- programming style,
- size (lines of code) and
- type of application (control or data–dominated etc.)

a fixed granularity during partitioning can cause an expensive hardware/software system (production and development cost).

Example 1:

Given the specification of a real–time system consisting of the two tasks T1 and T2. Furthermore, assume that the target architecture consists of one standard processor core (SW) and an application specific hardware (HW). There are only four possibilities for partitioning (the arrow "$\rightarrow$" has the meaning "*is implemented on*"):

1) T1, T2 $\rightarrow$ SW
2) T1 $\rightarrow$ SW; T2 $\rightarrow$ HW
3) T1 $\rightarrow$ HW;T2 $\rightarrow$ SW
4) T1, T2 $\rightarrow$ HW

Case 1) and 4) are the trivial cases. Hardware/software systems usually are represented by cases 2) and 3). Maybe, the given real–time constraints could be met by only implementing one computation intensive loop — hidden in $T1$ or $T2$ — as hardware. Nervertheless, a whole task has to be implemented in hardware. This can lead to a hardware/software system that is many times more expensive than necessary.

Example 2:

The target architecture assumed in this case is the same as in example 1. But now the granularity is fine–grain (e.g. basic block) and the specification of the hardware/software system is quite large (e.g. $>$ 1000 basic blocks). For the case that each basic block can possibly be implemented as hardware or as software there are 2^{1000} different implementations. Independent from the chosen optimization algorithm it is hard to find the global optimum i.e. the cheapest implementation that meets all given constraints.

The first example gives a characteristic disadvantage of coarse–grain partitioning whereas the second one is disadvantageous when using fine–grain partitioning.

The main idea of our approach is to combine the advantages of fine–grain and coarse–grain partitioning using a *dynamically determined granularity* since that:

a) permits the implementation of very small as well as very large system parts to be implemented as sw or as hw.
b) avoids large computation times.

Our approach consists of three following principal steps:

Step 1:

The high–level behavioural description (a superset of ANSI C) of the real–time application is transformed into a control flow graph (cfg) where each node represents a basic block or a single instruction. We call this our *base granularity*.

Step 2:

The smallest possible parts are integrated into *partitioning objects*. Thereby, the *smallest* partitioning object can consist of only one basic block or instruction whereas the *largest* one contains the *whole* real–time application. Partitioning objects can share same parts of the application i.e. they are *not* mutual exclusive.

- Small system parts as well as large system parts can be implemented as hw or as sw without distinctly increasing computation time.
- Constraints of different criticalness can easily be found due to more possibilities.
- The computation time to generate the partitioning objects is very small since this step needs no behavioural information.

Note that this step is performed by using *structural information* of the application only. Nevertheless the algorithm (detailed described in section 4.2) creates the objects in such a way that they may be promising candidates to be implemented as hardware or as software.

Step 3:

In contradiction to the last step this step uses behavioural information in order generate the so–called *macro instruction*. A macro instruction is composed of at least one partitioning object. It is generated an optimization algorithm that integrates partitioning objects into macro instructions controlled by an objective function. According to the hardness of the (time) constraints the algorithm therfore automatically picks up small or large partitioning objects.

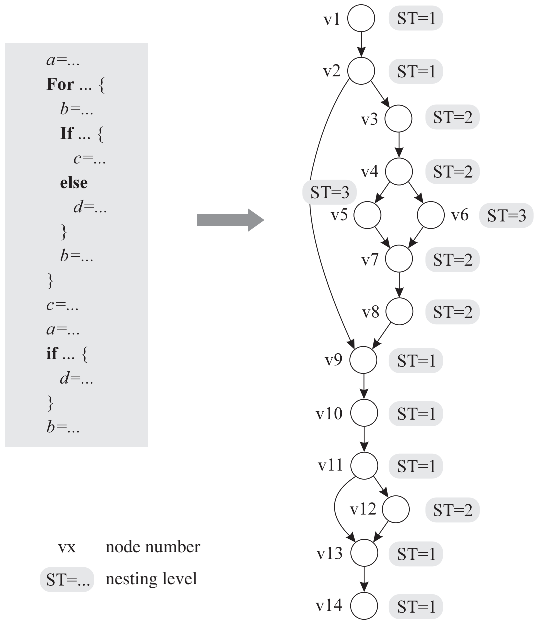

Figure 1: Example of a graph G with corresponding source code

The next section describes in detail each of the above steps. Thereby subsection 4.1 describes step 1, subsection 4.2 refers to step 2 and subsections 4.3, 4.4 and 4.5 belong to step 3.

4 Partitioning method

4.1 Defining a base granularity

First the behavioural description given in a high–level language is transformed into a cfg $G = \{V, E\}$. Each node $v_i \in V$ corresponds to a part of the application that itself belongs to the base granularity (a single instruction or a basic block). There are three different types of nodes. A node that

- contains straight forward code. Designation: v_i^B with $v_i^B \in V^B$, $V^B \subseteq V$.
- contains the beginning of a control construct (e.g. the beginning of a loop or branch). Designation: v_i^S with $v_i^S \in V^S$, $V^S \subseteq V$.
- contains the end of a control construct. Designation: v_i^E with $v_i^E \in V^E$, $V^E \subseteq V$.

An edge $e_{i,j} \in V$ gives the direction of the control flow from node v_i to node v_j. A node $v_i = pred(v_j)$ is called the predecessor of node v_j if there is an edge $e_{i,j}$. Accordingly a node $v_j = succ(v_i)$ is called the successor of node v_i if there is an edge $e_{i,j}$. The set of all successor of a node v_i is *SUCC*(v_i), the set of all predecessors is *PRED*(v_i).

Furthermore there are two dedicated nodes: a start node and an end node. For the algorithm in the next subsection a so–called nesting level ST as an attribute of a node is needed. It is defined as follows:

$$ST(v_i) := \begin{cases} 1 & : \ v_i \text{ is start node} \\ ST(\text{pred}(v_i)) + 1 & : \ \text{pred}(v_i) \in V^S \\ \max\{ST(\text{pred}(v_i))\} - 1 & : \ v_i \in V^E \\ ST(\text{pred}(v_i)) & : \ \text{else} \end{cases}$$

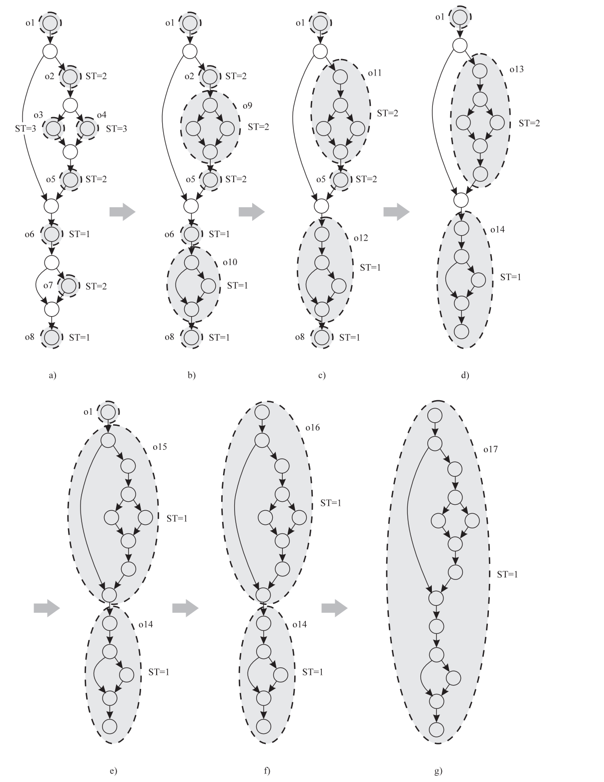

Figure 2: Steps of generating the partitioning objects for an example

4.2 Generating the partitioning objects

Before the algorithm for generating the partitiong objects is described the demands that led to the definition are given:

a) It is desirable to put small parts like basic blocks or instructions into a partitioning object.
b) Larger partitioning objects should contain whole control contructs (e.g. nested loops) or possibly functions/procedures
c) Only a few moves (a move is the action to put an object from sw to hw or vice versa) should be necessary to obtain a good co–design.

Some definitions: o_i with $o_i \in V^O$ is called a partitioning object. o_i itself is a set that contains of nodes $v_i \in V$. So, V^O is a set of sets. Furthermore, an temporarily set V^{O^*} is used. It contains the partitioning objects that have been generated during the actual iteration step of the algorithm. There is the relation $V^{O^*} \subseteq V^O$. To simplify the algorithm the term $\{o_a, o_b\} \rightsquigarrow o_c$ has the meaning that a new partitioning object is created by copying the contents of o_a and o_b to o_c. Thereby $o_a,\ o_b,\ o_c$ represent partitioning objects. In the following the algorithm for generating the partitioning objects is given:

1) Transform elements $v_i \in V$ into partitioning objects

a) $n = 0$

For all $v_i \in V$:

If

$$\bigwedge_{v_i \in V} \left(|pred(v_i)| \leq 1 \ \wedge \ |succ(v_i)| \leq 1\right) \textbf{ then}$$

a) $\{v_i\} \rightsquigarrow o_n$

b) $n = n + 1$

2) **For all** $o_i \in V^{O^*}$:

a) Integrate sequential system parts:

$$\bigvee_{o_j \in PRED(o_i) \ \wedge \ |pred(o_j)| \leq 1} \{o_i, o_j\} \rightsquigarrow o_n$$

b) Integrate control constructs

$$\bigvee_{o_j \in PRED(o_i) \ \wedge \ ST(o_j) < ST(o_i)} \ \bigwedge_{o_k \in SUCC(o_j)} \left(succ(o_k) \equiv succ(o_i)\right)$$

$$\Rightarrow \left\{ \bigcup_{o_k}, o_i, o_j, succ(o_i) \right\} \rightsquigarrow o_n$$

c) $n = n + 1$

3) updating V^O, V^{O^*} and removing redundancy

4) ready, if $|V^{O^*}| = 1$, else proceed with step 2)

The whole algorithm is performed as long as the condition of step 4) is false. This means that there is a partitioning object that contains the whole application. Only one of the conditions in 1), 2)a), 2)b) becomes true for possibly new generated partitioning objects.

The effect after performing the algorithm is shown in figure 2. The example application is the graph from fig. 1. Each of the graphs in the figure represents the state after one iteration through the algorithm. The respectively generated partitioning objects are set off by the grey color. Note that the complete set V^O contains all meanwhile generated objects (in the example o_1 to o_{17}).

4.3 Optimization

Describing the optimization algorithm that generate the so–called macro instructions, is necessary since the following steps will depend on its peculiarities. In order to solve the problem of finding the best macro instructions to be implemented in hardware — this is a combinatorial optimization problem — we have chosen the simulated annealing algorithm [22] for the following reasons:

- It is mathematically well investigated.
- It offers the possibility of a quality/computation time trade–off.
- It is independent from a specific problem.

Our implementation uses the so–called *annealing schedule* described in [23] since it offers one of the best quality/computation time trade–offs. The interface between the annealing schedule and the specific problem formulation consists of:

- $generate()$
  If this function is called the generation of a new state is requested i.e. in hardware/software partitioning a new co–design C has to be generated. This is done by a move m of a partitioning object from sw to hw or vice versa. A characteristic peculiarity of each co–design C are its cost $Cost : \mathcal{C} \rightarrow \mathbf{R}^+$. Thereby $\mathcal{C}$ is the set of all possible co–designs.
- $accept()$
  If the annealing algorithm has decided to accept this move then $accept()$ returns the value *"TRUE"*.
- $reject()$
  If the annealing algorithm has decided not to accept this move then $accept()$ returns the value *"FALSE"*.

4.4 Selection of a move

The following prerequisites led to the definition of the selection algorithm for a new move:

- Exactly one partitioning object is moved from software to hardware or vice versa during one call of the $generate()$ function.
- The algorithm assumes that hardware and software parts execute in mutual exclusion.
- The algorithm should select a new move in dependency upon the current co–design.

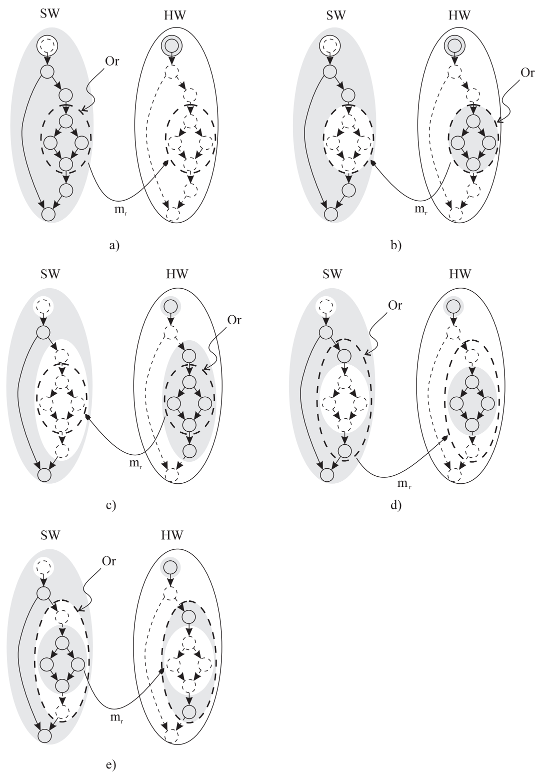

Figure 3: All possible base configurations and characteristic moves

In the following algorithm the term $o_r \overset{m_r}{\longmapsto} H$ has the meaning that partitioning object o_r is moved from software to hardware using the move m_r. The algorithm is defined as follows:

1) Random selection of an object $o_r \in V^O$

2) Case 1: Selected object is in SW

If

$$\left(\bigwedge_{v_i \in o_r} v_i \in S \right)$$

Then

a) Valid move: $\left(\bigcup_{v_i \in o_r} v_i \right) \overset{m_r}{\longmapsto} H$

3) Case 2: Selected object is in HW

If

$$\left(\bigwedge_{v_i \in o_r} v_i \in H \right)$$

Then

a) **If**

$$\left(\bigvee_{o_j \in V^O} \left((o_j \supset o_r) \wedge \left(\bigvee_{v_i \in o_j, v_i \notin o_r} v_i \in H \right) \right) \right)$$

Then

No move possible. Proceed with step 1)

Else

Valid move: $\left(\bigcup_{v_i \in o_r} v_i \right) \overset{m_r}{\longmapsto} S$

4) Case 3: Selected object is in parts on SW and in parts on HW

If

$$\left(\bigvee_{v_i \in o_r} \bigvee_{v_j \in o_r} (v_i \in S \wedge v_j \in H) \right)$$

Then

a) $o_k = \bigcup_{v_i \in o_r,\, v_i \in H} v_i$

b) $o_l = o_r \setminus o_k$

c) Valid move: $\left(\bigcup_{v_i \in o_l} v_i \right) \overset{m_r}{\longmapsto} H$

5) **If**

None of the conditions in steps 2), 3), 4) is fulfilled

Then

Proceed with step 2)

Else

Ready, since a valid move is found

If a valid move is found exactly one of the cases Case 1, Case 2 or Case 3 is valid. All possible base configurations and moves are given in fig. 3. The assignments of cases to figures are as follows: Case 1 = "TRUE" corresponds to fig.3a), Case 2 a) = "TRUE" corresponds to fig.3b), Case 2 a) = "FALSE" corresponds to fig.3c) and Case 3 = "TRUE" corresponds to fig.3d). A situation like in fig.3e) is impossible since no move can generate such an initial configuration (it would hurt the conventions about the target architecture).

Based on this base configurations and according moves every valid co–design can be reached (see [24]).

4.5 Cost function

After each generated move a new co–design C emerges. The simulated annealing algorithm requests the cost of a co–design C in order to decide on acception or rejection. The cost function reads:

$$Cost = \underbrace{A \cdot cost_T(T_{HW/SW})}_{\text{time component}} + \underbrace{B \cdot w_{area} \cdot \frac{Area}{\overline{A}}}_{\text{area component}}. \quad (1)$$

The total cost of a co–design are composed of a time component and an area component. Thereby A and B are heuristically chosen factors which guarantee that the area component in every case is small compared to the time component since "time" is a constraint whereas "area" is an optimization goal. w_{area} is balancing function that delivers values between 0 and 1. This allows a separation of the two phases *"meet the time constraint"* and *"minimize the hardware effort"*. For lack of space w_{area} is not described here. Furthermore $Area$ is the gate count of all partitioning objects that have currently been implemented as hardware. $\overline{A}$ is the average gate count of a partitioning object. $Area$ and $\overline{A}$ are estimated by a high–level area estimation tool [24].

The time cost are obtained as follows:

$$cost_T = e^{\frac{T_{HW/SW} - T_{cst}}{T_0}} \cdot |T_{cst} - T_{HW/SW}| \cdot \frac{1}{T_N} \quad (2)$$

The exponential factor enforces the simulated annealing algorithm to punish non–valid co–designs ($T_{HW/SW} \geq T_{cst}$) and so it accelerates the convergence of the optimization procedure. T_0 and T_N are heuristically chosen.

Each partitioning object has a couple of attributes:

t^O_{HW} This is the execution time for the case that partitioning object o is implemented as hardware.

t^O_{com} This is the communication time (transfer of data and control) from software to hardware and vice versa for the case object o is implemented as hardware.

t^O_{SW} This is the execution time for the case that partitioning object o is implemented as software.

it This is the number of times the partitioning object o is executed (profiling data for a typical set of stimuli).

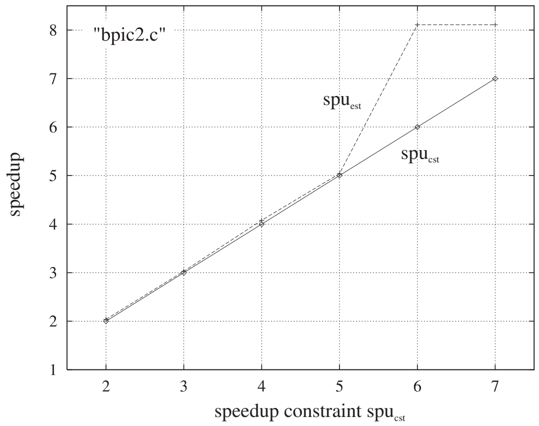

Figure 4: Speedup constraint spu_{constr} and estimated speedup spu_{est} for different design points of an application

t^O_{HW} is estimated by a path–based estimation technique and t^O_{SW} uses a run–time analysis tool

The incremental cost (timing) of a partitioning object o are calculated as follows:

$$t^o = (t^o_{HW} + t^o_{com} - t^o_{syn,com} - t^o_{SW}) \cdot it^o \qquad (3)$$

After each move the run–time of the whole co–design (i.e. the hardware/software system) is updated:

If

(Partitioning object o is moved from SW to HW)

Then

$T_{HW/SW} = T_{HW/SW} + t^{o_i}$

Else

$T_{HW/SW} = T_{HW/SW} - t^{o_i}$

Together with equations 1, 2 and 3 the cost of a move can be calculated *incrementally*. This is useful since it saves a lot of computation time during partitioning. Furthermore note that all the attributes of a partitioning object have been pre–calculated — that means *before* the simulated annealing is performed. For further information about the target architecture, please refer to [25].

5 Experimental results

In this section we demonstrate the efficiency of the proposed algorithm by means of a set of applications that partly come from real industrial projects. The sizes of the applications reach from about 30 lines of C–code to about 500 lines of C–code. All steps of the partitioning algorithm are fully automated as well as the necessary estimation methods.

Speedup and timing constraints

Speedup and timing constraint are defined as

$$spu_{H/S} = \frac{T_{SW/HW}}{T_{SW}} \geq spu_{cst},$$

where T_{SW} is the execution time of an application for an all–software solution, $T_{SW/HW}$ is the execution time of the same application but for a hardware/software implementation (after partitioning) and spu_{cst} is the given constraint speedup. Fig. 4 shows the behaviour of the obtained speedup and speedup constraint for different design points of an application. The graph "spu_{est}" gives the result obtained by the partitioner. It can be seen that in all cases the constraint of the above inequation is met ($spu_{est} \geq spu_{cst}$). Table 1 shows some more results. It also shows the real speedup

Applic.	Meas.	Time constraints spu_{cst}			
		2.00	3.00	5.00	10.00
"bin"	spu_{est}	→	3.90	5.20	13.77
	spu_{syn}	→	3.27	5.36	11.63
	geq	→	2632.0	12997.0	12690.0
"bp2"	spu_{est}	2.04	3.03	5.04	max.
	spu_{syn}	1.62	3.10	4.69	spu
	geq	32990.5	35861.5	36914.0	8.45
"digb"	spu_{est}	2.70	3.00	max.	max.
	spu_{syn}	4.25	4.79	spu	spu
	geq	20395.0	21507.0	3.90	3.90
"frac"	spu_{est}	→	→	→	15.78
	spu_{syn}	→	→	→	19.98
	geq	→	→	→	19527.5
"huff"	spu_{est}	2.00	max.	max.	max.
	spu_{syn}	2.78	spu	spu	spu
	geq	7272.0	3.80	3.80	3.80
"sm1"	spu_{est}	→	→	7.42	10.29
	spu_{syn}	→	→	9.13	11.46
	geq	→	→	11.089.5	12706.5

Table 1: Results of estimated speedup, synthesized speedup and hardware effort

"spu_{syn}" that has been obtained as follows: after hardware/software partitioning the software part has been mapped to a standard processor core (SPARC) and the hardware part (including interfaces) has been synthesized using high–level synthesis. Afterwards we have taken the output (slif netlist) of the high–level synthesis and optimized it using the SYNOPSYS design compiler. After simulation of software and hardware parts we got the real speedup called "spu_{syn}".

The table shows that in all cases the speedup values are very close to the constraint. The partitioner meets the given speedup constraint in all cases. Due to the inaccuracy of the estimation tools (hardware run–time, software run–time, communication time) that perform estimation at a high level of abstraction there are some deviations between real synthesis results (spu_{syn}) and constraint (spu_{cst}). It shows that the estimation tools have a good accuracy.

An arc in the table means that the according design point is the same as the one the arc points to. Other applications cannot be sped up more than a maximum value due to the peculiarities of an application.

Computation time

Due to the variable size of the partitioning objects that can cover the whole application or only a single instruction, the computation time could almost be kept independent from the size of the benchmark. Of course, the number of all possible partitioning objects is larger than it would be when using a fixed–size granularity (we measured about four times more partitioning objects than there would have been if the granularity had been fixed to basic block level). As a result the computation times have been in most cases within a couple of seconds.

Hardware cost

Also an important task in embedded systems design is the minimization of the hardware cost. The attribute "geq" (gate equivalents) gives the hardware cost for every design point. Large speedups around 10.0 lead to a small hardware cost (application specific hardware of the hardware/software system) of only 30.000 or less gate equivalents is due to the cost function that also takes into account a hardware component (see eq. 1). The hardware results are obtained by using the SYNOPSYS design compiler.

Fig. 5 shows the percentage of hardware that could be saved compared to the same cost function but without the hardware component. It can be seen that in some cases savings of up to 50% have been achieved.

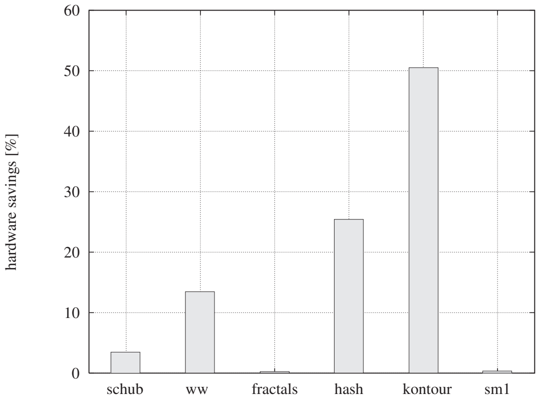

Figure 5: Savings in hardware cost by using an explicit hardware cost component in the cost function

6 Conclusion

We have presented a hardware/software partitioning approach that determines the granularity dynamically i.e. during the partitioning process. Therefore the granularity is best suited for the specific peculiarities of an application. This is the major difference to known approaches. The experimental results have shown that a large speedup is obtained and in every case the given speedup constraints could be met. The new partitioning approach is integrated into the COSYMA environment [4]. Due to the dynamically determined granularity the computation times of the partitioner are kept small. Nevertheless an improvement in system speedup can be obtained by extending the target architecture to a concurrent execution of hardware and software parts.

REFERENCES

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

[2] C. Kuttner, *Hardware–Software Codesign Using Processor Synthesis*, IEEE Design & Test of Computers, Vol. 13, No. 3, pp. 43–53, 1996.

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

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

[5] E. Barros, W. Rosenstiel, X. Xiong, *A Method for Partitioning UNITY Language in Hardware and Software,* Proc. of IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1994, pp. 220–225, 1994.

[6] A. Jantsch, P. Ellervee, J. Oeberg et. al., *Hardware/Software Partitioning and Minimizing Memory Interface Traffic,* IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1994, pp. 220–225, 1994.

[7] P. Athanas and H.F. Silverman, *Processor Reconfiguration Through Instruction–Set Metamorphosis,* IEEE Computer Magazine, pp. 11–18, March 1993.

[8] Z. Peng, K. Kuchcinski, *An Algorithm for Partitioning of Application Specific System,* IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC) 1993, pp. 316–321, 1993.

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

[10] J. Madsen, P. V. Knudsen, *LYCOS Tutorial*, Handouts from Eurochip course on Hardware/Software Codesign, Denmark, 14.–18. Aug. 1995.

[11] R. Niemann, P. Marwedel,*Hardware/Software Partitioning using Integer Programming,* IEEE/ACM Proc. of EDAC'96, pp.473–479, 1996.

[12] I. Karkovski, R. H. J. M. Otten,*An Automatic Hardware–Software Partitioner Based on the Possibilistic Programming,* IEEE/ACM Proc. of EDAC'96, pp.467–472, 1996.

[13] 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.

[14] F. Vahid, D. D. Gajski, *Clustering for improved system–level functional partitioning*, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 28–33, 1995.

[15] D.D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994.

[16] 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.

[17] J. K. Adams, D. E. Thomas *Multiple–Process Behavioral Synthesis for Mixed Hardware–Software Systems*, IEEE/ACM Proc. of 8th. International Symposium on System Synthesis, pp. 10–15, 1995.

[18] 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.

[19] 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.

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

[21] F. Balarin, M. Chiodo, D. Engels et al., *POLIS: A design environment for control–dominated embedded systems*, Technical Report, UC Berkeley, 1996.

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

[23] J. Lam, J.-M. Delosme, *Performance of a New Annealing Schedule*, IEEE/ACM Proc. of 25th. Design Automation Conference (DAC), pp. 306–311, 1988.

[24] J. Henkel, *Automatisierte Hardware/Software–Partitionierung im Entwurf integrierter Echtzeitsysteme*, PhD thesis, Technische Universität Braunschweig, 1996.

[25] 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.

[26] 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.