Finding optimal hardware/software partitions*

Zoltán Ádám Mann, András Orbán, Péter Arató
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
{zoltan.mann, andras.orban}@cs.bme.hu, arato@iit.bme.hu

Abstract

Most previous approaches to hardware/software partitioning considered heuristic solutions. In contrast, this paper presents an exact algorithm for the problem based on branch-and-bound. Several techniques are investigated to speed up the algorithm, including bounds based on linear programming, a custom inference engine to make the most out of the inferred information, advanced necessary conditions for partial solutions, and different heuristics to obtain high-quality initial solutions. It is demonstrated with empirical measurements that the resulting algorithm can solve highly complex partitioning problems in reasonable time. Moreover, it is about ten times faster than a previous exact algorithm based on integer linear programming. The presented methods can also be useful in other related optimization problems.

1 Introduction

The requirements towards today's computer systems are tougher than ever. Parallel to the growth in complexity of the systems to be designed, the time-to-market pressure is also increasing. In most applications, it is not enough for the product to be functionally correct, but it has to be cheap, fast, and reliable as well. With the wide spread of mobile systems, size, heat dissipation, and energy consumption are also becoming crucial aspects for a wide range of computer systems [21], especially embedded systems.

These computer systems typically consist of both hardware and software. In this context, hardware means application-specific hardware units, *i.e.*, hardware designed and implemented specifically for the given system, whereas software means a program running on a general-purpose hardware unit, such as a microprocessor.

For much of the functionality of embedded systems, both a hardware and a software implementation is possible. Both possibilities have advantages and disadvantages: software is typically cheaper, but slower; moreover, general-purpose processors consume more power than application-specific hardware solutions. Hardware on the other hand is fast and energy-efficient, but significantly more expensive.

Consequently, it is beneficial to implement performance-critical or power-critical functionality in hardware, and the rest in software. This way, an optimal trade-off can be found between performance, power, and costs [4]. This is the task of hardware/software partitioning: deciding which parts of the functionality should be implemented in hardware and which ones in software. Unfortunately, finding

*This paper has been published in *Formal Methods in System Design, volume 31, issue 3, pages 241-263, Springer, 2007*.

such an optimal trade-off is by no means easy, especially because of the large number and different characteristics of the components that have to be considered. Moreover, the communication overhead between hardware and software has to be taken into account as well [33].

Traditionally, partitioning was carried out manually. However, as the systems to design have become more and more complex, this method has become infeasible, and many research efforts have been undertaken to automate partitioning as much as possible. Most presented algorithms are heuristics, but some are exact (*i.e.* non-heuristic, optimal) algorithms. Section 2 presents a survey of the related work.

Although heuristic partitioning algorithms are typically very fast and produce near-optimal or even optimal results for small systems, their effectiveness (in terms of the quality of the found solution) degrades drastically as the size of the problem increases [3]. This is due to the fact that—in order to be fast—such heuristics evaluate only a small fraction of the search space. As the size of the problem increases, the search space grows exponentially (there are 2^n different ways to partition n components), which means that the ratio of evaluated points of the search space must decrease rapidly, leading to worse results. Consequently, if the system that has to be partitioned is big and constraints on its cost, performance etc. are tight (and they usually are), then chances are high that a heuristic partitioner will find no valid partition. What is even worse, the designer will not know if this is due to the weak performance of the partitioning algorithm or because there is no valid partition at all. This shows that in partitioning, it makes sense to strive for an optimal solution, because it makes a major difference whether or not all constraints on a design can be fulfilled.

The above problems can be overcome using exact algorithms. However, since most formulations of the hardware/software partitioning problem are $\mathcal{NP}$-hard, such algorithms have exponential runtimes which makes them inappropriate for large problem instances. Consequently, previous exact algorithms were used either for very small problem instances only [9, 32, 37], or have been used inside heuristics, thus sacrificing their optimality [35, 36].

Most previous algorithms were tested on systems with some dozens of components. The aim of this paper is to present an exact partitioning algorithm that can optimally partition systems with hundreds of components in reasonable time. We believe that such an algorithm can be used in practical, industrial projects. Notice though that developing such an algorithm is by no means straight-forward because partitioning a system of, say, 300 components optimally means scanning a search space of size 2^{300}, which is enormous. In this paper, several techniques are presented with which this task becomes feasible.

Specifically, we start with the integer linear program (ILP) formulation of hardware/software partitioning that we presented in an earlier work [3]. This ILP program can be solved using any ILP solver; however, our tests have shown that this requires too much time [3]. Hence, the search strategy of the ILP solver has to be tuned. Note that general-purpose ILP solvers use general solution algorithms. Using problem-specific knowledge, an improvement is often possible.

Most ILP solvers use branch-and-bound for exploring the solution space [43]. They use the bound received by solving the LP-relaxation of the problem for cutting off unpromising branches of the search tree. This suggests that the solution process can be accelerated if, in a specialized branch-and-bound algorithm we can exploit extra knowledge on the partitioning problem, in addition to LP-relaxation which is applicable to all ILP programs.

Specifically in Section 5, we present a problem-specific inference engine that applies every information extractable during the branch-and-bound procedure to infer additional knowledge. Several constraints and implications on variable values are stored in an appropriate graph structure that can be used in later steps of the branch-and-bound search procedure to reason about unsubstituted variables.

Besides the inference mechanism, we identify several necessary conditions in Section 6 that must

hold in the optimal solution. These conditions help cut off large subtrees of the search tree speeding up the whole branch-and-bound procedure.

Furthermore, the effectiveness of branch-and-bound can be significantly improved using a high-quality initial solution. Thus, further acceleration is possible by making use of efficient heuristics to produce such an initial solution. We utilize three different heuristics described in Section 7: the first one is a genetic algorithm, the second is a minimum cut-based algorithm, while the third one uses hierarchical clustering.

As a result of all these techniques, the algorithm is significantly faster than a simple ILP solver. Our experience with industrial benchmark problems showed an acceleration of about a factor of 10 for the biggest problem instances (see Section 8).

2 Previous work

In a number of related papers, the target architecture is supposed to consist of a single software and a single hardware unit [13, 17, 18, 20, 31, 32, 34, 37, 40, 44, 45, 49], whereas others do not impose this limitation [10, 12, 25, 36]. Some limit parallelism inside hardware or software [44, 49] or between hardware and software [20, 32]. The system to be partitioned is usually given in the form of a task graph, or a set of task graphs, usually assumed to be directed acyclic graphs describing the dependencies between the components of the system.

Concerning the scope of hardware/software partitioning, further distinctions can be made. In particular, many researchers consider scheduling as part of partitioning [10, 12, 24, 31, 34, 36], whereas others do not [13, 17, 32, 37, 49, 47]. Some even include the problem of assigning communication events to links between hardware and/or software units [12, 34].

The proposed methods also vary significantly concerning model granularity, *i.e.* the semantics of a node. There have been works on low granularity, where a node represents a single instruction or a short sequence of instructions [6, 8, 40, 21], middle granularity, where a node represents a basic block [22, 27, 38], and high granularity, where a node represents a function or procedure [18, 36, 48, 2], as well as flexible granularity, where a node can represent any of the above [20, 47].

The majority of the previously proposed partitioning algorithms is heuristic. This is due to the fact that partitioning is a hard problem, and therefore, exact algorithms tend to be quite slow for bigger inputs. More specifically, most formulations of the partitioning problem are $\mathcal{NP}$-hard [23, 5], and the exact algorithms for them have exponential runtimes.

Many researchers have applied general-purpose heuristics to hardware/software partitioning. In particular, genetic algorithms have been extensively used [3, 12, 34, 41, 44], as well as simulated annealing [13, 14, 15, 20, 28]. Other, less popular heuristics in this group are tabu search [13, 14] and greedy algorithms [10, 17]. Some researchers used custom heuristics to solve hardware/software partitioning. This includes the GCLP algorithm [24, 25] and the expert system of Lopez-Vallejo and Lopez [29, 31], as well as the heuristics of Gupta and de Micheli [18] and Wolf [50]. There are also some families of well-known heuristics that are usually applied to partitioning problems. The first such family of heuristics is hierarchical clustering [1, 7, 30, 47, 48]. The other group of partitioning-related heuristics is the Kernighan-Lin heuristic [26], which was substantially improved by Fiduccia and Mattheyses [16], and later by many others [11, 42]. These heuristics have been found to be appropriate for hardware/software partitioning as well [31, 46, 49].

A couple of exact partitioning algorithms have also been proposed. Branch-and-bound has been presented for partitioning in the design of ASIPs (Application-Specific Instruction set Processors) [9]. Although the design of ASIPs is a related problem, it is very different in its details from the partitioning

problem that we address, thus that algorithm is also very different from ours.

Algorithms based on the dynamic programming paradigm were used by Madsen et al [32] and by O'Nils et al [37]. However, both algorithms were applied to only very small problem instances (with some dozens of components only), and so it is quite doubtful whether they are scalable enough for larger problems as well. Conversely, we focus our efforts on making our algorithm scalable for systems with several hundreds of components.

The usage of integer linear programming (ILP) was first suggested in the early work of Prakash and Parker [39]. Their approach handled the whole co-design problem—including scheduling and optimization of the communication topology—in the form of a single integer program. This resulted in an algorithm that was very slow even for small problem instances and practically unusable for bigger ones. In contrast, our aim is to develop a fast exact algorithm for a more limited problem.

Integer linear programming was later also used by Niemann et al [35, 36]. However, that algorithm is only a heuristic, although it makes use of an exact ILP solver. We assume that the pure ILP algorithm would not have been scalable enough, and therefore the authors combined it with a heuristic to make it faster. However, this way the optimality of the algorithm is sacrificed.

For a more detailed survey on hardware/software co-design, see [51].

3 Problem formalization

We use the model that was presented in [5]. Here, we review its most important characteristics.

The system to be partitioned is modeled by a *communication graph*, the nodes of which are the components of the system that have to be mapped to either hardware or software, and the edges represent communication between the components. Unlike in most previous works, it is not assumed that this graph is acyclic in the directed sense. The edges are not even directed, because they do not represent data flow or dependency. Rather, their role is the following: if two communicating components are mapped to different contexts (*i.e.* one to hardware and the other to software, or vice versa), then their communication incurs a communication penalty, the value of which is given for each edge as an edge cost. This is assumed to be independent of the direction of the communication (whether from hardware to software or vice versa). If the communication does not cross the hardware/software boundary, it is neglected.

Similarly to the edge costs mentioned above, each vertex is assigned two cost values called hardware cost and software cost. If a given vertex is decided to be in hardware, then its hardware cost is considered, otherwise its software cost. We do not impose any explicit restrictions on the semantics of hardware costs and software costs; they can represent any cost metrics, like execution time, size, or power consumption. Likewise, no explicit restriction is imposed on the semantics of communication costs. Nor do we impose explicit restrictions on the granularity of partitioning (*i.e.* whether nodes represent instructions, basic blocks, procedures or memory blocks).

However, we assume that the total hardware cost with respect to a partition can be calculated as the sum of the hardware costs of the nodes that are in hardware, and similarly, the software cost with respect to a partition can be calculated as the sum of the software costs of the nodes that are in software, just as the communication cost with respect to a partition, which is the sum of the edge costs of those edges that cross the boundary between hardware and software.

While the assumption of additivity of costs is not always appropriate, many important cost factors do satisfy it. For example, power consumption is usually assumed to be additive, implementation effort is additive, execution time is additive for a single processing unit (and a multi-processor system can also be approximated by an appropriately faster single-processor system), and even hardware size

is additive under suitable conditions [32].

Furthermore, although it is a challenging problem how the cost values can be obtained, it is beyond the scope of this paper. Rather, we focus only on algorithmic issues of partitioning.

We now formalize the problem as follows. An undirected graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $s, h : V \to \mathbb{R}^+$ and $c : E \to \mathbb{R}^+$ are given. $s(v_i)$ (or simply s_i) and $h(v_i)$ (or h_i) denote the software and hardware cost of node v_i, respectively, while $c(v_i, v_j)$ (or $c_{i,j}$) denotes the communication cost between v_i and v_j if they are in different contexts. P is called a hardware-software partition if it is a bipartition of V: $P = (V_H, V_S)$, where $V_H \cup V_S = V$ and $V_H \cap V_S = \emptyset$. ($V_H = \emptyset$ or $V_S = \emptyset$ is also possible.) The set of crossing edges of partition P is defined as: $E_P = \{(v_i, v_j) : v_i \in V_S, v_j \in V_H \text{ or } v_i \in V_H, v_j \in V_S\}$. The hardware cost of P is: $H_P = \sum_{v_i \in V_H} h_i$; the software cost of P is: $S_P = \sum_{v_i \in V_S} s_i$; the communication cost of P is: $C_P = \sum_{(v_i, v_j) \in E_P} c(v_i, v_j)$. Thus, a partition is characterized by three metrics: its hardware cost, its software cost, and its communication cost.

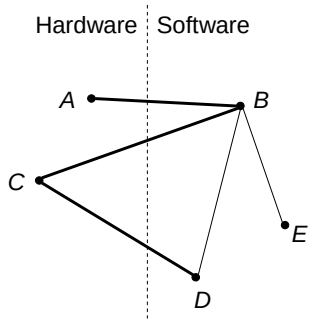

Figure 1: Example communication graph

An example for a communication graph and a possible hardware/software partition can be seen in Figure 1. The crossing edges are bold. Suppose that for each vertex, hardware and software costs are both 1, and the communication cost of each edge is 1. Then the hardware cost of the given partition is 2, its software cost is 3, and its communication cost is also 3.

Now we consider a version of the hardware/software partitioning problem, in which two of the three metrics are added. For instance, if software cost captures execution time, and communication cost captures the extra delay generated by communication, then it makes sense to add them. (Another example would be the following: if h denotes hardware implementation effort, and c denotes the effort of implementing communication modules, then it makes sense to add them.) That is, we define the running time of the system with respect to partition P as $R_P = S_P + C_P$.

A very important and frequently studied problem is the following: design the cheapest system respecting a given real–time constraint, that is the overall system execution time is limited by a constant R_0. Since the dominant cost factor of the system is the cost of the hardware units, so the aim of the designer is to minimize the hardware cost H_P[1] while satisfying the real–time constraint $R_P \leq R_0$.

One might note that the hardware execution time is neglected in this model and it does not contribute to the overall system execution time. The following transformation shows that in case of additive costs this can be assumed without loss of generality: let first assume that to each node v both a software execution time $t_s(v)$ and a hardware execution time $t_h(v)$ are assigned and the system execution time is the sum of the software times plus the hardware times and the communication times.

[1]Note that in this problem formulation S_P and C_P are time-dimensional, but H_P is not.

We now show an equivalent system with zero hardware times. Consider a system with hardware time everywhere zero and with a modified software execution time $t_s(v) - t_h(v)$ for a node v (here we further assume that $t_s(v) - t_h(v) \geq 0$ which generally holds). This system behaves exactly the same way, but for each partition the system running time is decreased by $T_h := \sum_{v \in V} t_h(v)$ which is a constant. So by prescribing a limit of $R_0 - T_h$, the modified problem becomes equivalent to the original one and has zero hardware times.

To sum up, the partitioning problem we are dealing with can be formulated as follows:

> Given the graph G with the cost functions h, s, and c, and $R_0 \geq 0$, find a hardware/software partition P with $R_P \leq R_0$ that minimizes H_P among all such partitions.

In an earlier work [3], we proved that this problem is $\mathcal{NP}$-hard. Moreover, the following ILP formulation was suggested.

$h, s \in (\mathbb{R}^+)^n, c \in (\mathbb{R}^+)^e$ are the vectors representing the cost functions (n is the number of nodes, e is the number of edges). $E \in \{-1, 0, 1\}^{e \times n}$ is the transposed incidence matrix:

$$E_{i,j} := \begin{cases} -1 & \text{if edge } i \text{ starts in node } j \\ 1 & \text{if edge } i \text{ ends in node } j \\ 0 & \text{if edge } i \text{ is not incident to node } j \end{cases}$$

The definition of E suggests a directed graph, although so far we spoke about undirected graphs only. The undirected incidence matrix of G would result in a slightly more complex ILP program, so an arbitrary direction of the edges should be chosen and the directed incidence matrix should be used.

Let $x \in \{0, 1\}^n$ be a binary vector indicating the partition:

$$x_i := \begin{cases} 1 & \text{if node } i \text{ is realized in hardware} \\ 0 & \text{if node } i \text{ is realized in software} \end{cases}$$

Finally, let $y \in \{0, 1\}^e$ be a binary vector indicating which edges are crossed by the partition:

$$y_i := \begin{cases} 1 & \text{if edge } i \text{ crosses the hardware/software boundary} \\ 0 & \text{if edge } i \text{ does not cross the hardware/software boundary} \end{cases}$$

Using these notations, the integer program is as follows:

$$\min hx \tag{1a}$$

$$\begin{aligned} s(\mathbf{1} - x) + cy &\leq R_0 && \text{(1b)} \\ Ex &\leq y && \text{(1c)} \\ -Ex &\leq y && \text{(1d)} \\ x &\in \{0, 1\}^n && \text{(1e)} \end{aligned}$$

It can be easily proven that the optimal solution of this integer program is also the optimum of the hardware/software partitioning problem [3]. An interesting property of this integer program is that there is no integrality constraint on y, just on x. However, it can also be proven that y will be integral in the optimal solution of the ILP problem [3].

Although the cost values h_i, s_i, $c_{i,j}$ can be arbitrary positive numbers, it will be assumed for the sake of simplicity in the rest of this paper that they are integers. This assumption is not very strict and can be exploited in finding good necessary conditions to speed up our algorithm. See Section 6 for details.

This ILP program can be solved using any ILP solver, yielding an algorithm for solving the partitioning problem. However, our experience has shown that this is quite slow [3].

4 Branch-and-bound framework

An ILP solver typically uses branch-and-bound to intelligently search the space of possible solutions. Note that there are 2^n possible partitions, which makes it intractable to check all partitions even for graphs of moderate size.

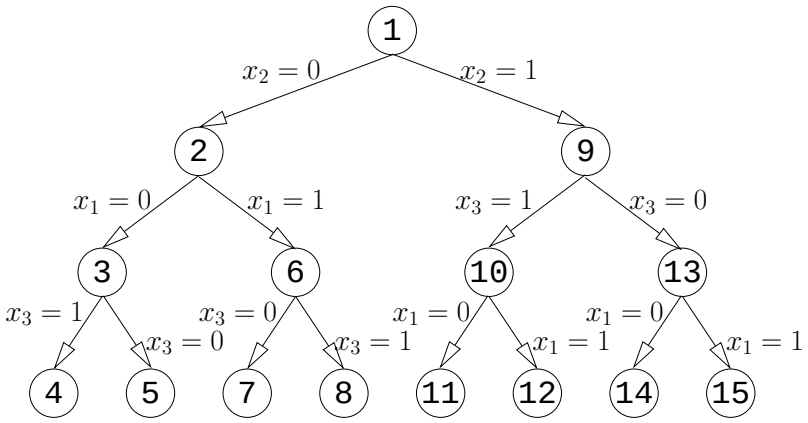

Figure 2: Example search tree

The branch-and-bound algorithm traverses a *search tree*. Each leaf of the search tree corresponds to a possible solution, *i.e.* an assignment of values to the variables.[2] The internal nodes of the search tree correspond to partial solutions, *i.e.* assignment of values to some of the variables. In each internal node, a new, unfixed variable is chosen, and branching is performed according to the value of this variable. In our case, each variable has two possible values (0 and 1), hence each internal node of the search tree has two children. If the original communication graph has n vertices, the search tree has 2^n leaves and $2^n - 1$ internal nodes.

An example with three variables is shown in Figure 2. The nodes of the search tree are numbered according to the order in which the branch-and-bound algorithm visits them. Note that, when considering a new variable, the algorithm can either first assign the value 0 to it, and only later the value 1, or vice versa. Furthermore, the algorithm does not have to follow the same variable order in each branch of the search tree. These decisions can be made based on a static order, randomly, or according to some heuristic run-time decisions.

Branch-and-bound works well if large portions of the search tree can be cut off. For instance, using the example of Figure 2, suppose that we are currently in node 9, *i.e.* x_2 is fixed to 1, but the other variables are not fixed. We have already evaluated four different partitions (nodes 4, 5, 7, and 8). Suppose that the best valid partition that has been encountered so far had a hardware cost of H_0. This means that we are looking for partitions with $R_P \leq R_0$ and $H_P < H_0$. Suppose furthermore that some mechanism tells us that these two constraints cannot be fulfilled with $x_2 = 1$. Then the branch starting at node 9 can be cut off, thus saving us a lot of time.

In the case of ILP, this mechanism is typically *LP-relaxation*. The LP-relaxation of an ILP program is defined as the same integer program without integrality constraints. This yields an LP problem instance, which can be quickly solved using a standard LP solver, because LP is much simpler than ILP. Clearly, if not even the LP-relaxation can be solved in a given node of the search tree (*i.e.* after

[2]By variables, the x_i variables are meant in this section. This is because there is no integrality constraint on y, and thus branching is only performed with respect to the x_is.

some variables were fixed), then the ILP is not solvable either, so that the current branch can be cut off.

As we will see later, other mechanisms can also be used at this point beside LP-relaxation, and this way the algorithm can be further accelerated.

Algorithm 1 Skeleton of the branch-and-bound algorithm

```
procedure backtrack
{
    while both possible values of the lastly fixed variable have been checked
    {
        undo last fixation;
        if there are no more fixed variables
        {
            STOP;    //finished searching the tree
        }
    }
    change the lastly fixed variable to its other possible value;
}

procedure branch-and-bound
{
    repeat     // the stopping condition is in the backtrack procedure
    {
        if all variables are fixed    //found a leaf of the search tree
        {
            evaluate found partition;
            backtrack;
            skip to next iteration;
        }
        check feasibility;
        if not feasible    //cut off current branch
        {
            backtrack;
            skip to next iteration;
        }
        //we are not in a leaf, nor can we cut the branch off
        //this means we have to go deeper in the search tree
        choose a variable that is not yet fixed;
        fix it to one of its possible values;
    }
}
```

The skeleton of our branch-and-bound algorithm is shown in Algorithm 1. This algorithm is used as a framework for incorporating further algorithms, as described in the next sections.

The algorithm may need some explanation. The idea of the *backtrack* routine can be illustrated easily on the example of Figure 2. Suppose for instance that we are currently in node 5, and we

perform a backtrack. We have to go back on the tree (and undo the fixations), as long as both possible values of the lastly fixed variable are checked. This holds for node 3 because both of its children have already been visited. But it does not hold for node 2, because only one of its children has been visited. So we stop the cycle of the backtrack procedure at this point, and change the current variable, which is x_1 in the case of node 2, to its other value, which is 1. This is how we get to the next node, which is node 6.

As already mentioned, the feasibility check involves solving an LP problem instance. This LP is obtained from the original ILP by abandoning the integrality constraints and taking into account the variable fixations and any other inferred information. Moreover, the following inequality can be added to the LP: $hx \leq H_0 - 1$ (where H_0 is the best objective value found so far), because all costs are assumed to be integers, and we are only interested in finding solutions that are better than the best one found so far. Actually, it would be sufficient to test if the resulting LP can be solved at all (instead of minimizing hx) in order to decide whether to go deeper in the search tree or to backtrack. However, we did not choose this alternative for the following reasons:

- It is in general not easier to test the solvability of a set of inequations than to optimize a linear function subject to those constraints [43].
- Optimizing hx has the advantage that if accidentally all variables are integral in the optimum of the LP[3], then the optimum of the ILP in that branch is found as well.
- Even if not all variables are integral in the optimum of the LP, their value carries important information. Namely, the strategy of fixing a variable in the next round that is not integral in the optimum of the current LP, is known to yield faster termination than the strategy of fixing a random variable.

Note that there are two possibilities for creating the LP problem instance. It can be either built from scratch, or by modifying the last one. If backtracking has been performed since the last LP, then the new LP should be built from scratch. However, if the difference since the last LP is only additional information that can be captured with linear equalities or inequalities then it is quicker to supplement the last LP with these new pieces of information. Conversely, if we keep on adding new rows to the LP, this adds to its complexity. In our algorithm, we use a trade-off: after a predefined number of modified LPs, the next LP is built from scratch.

5 Improvement by inference

Each *round* of the branch-and-bound starts by fixing a variable to either 0 or 1. In a general-purpose ILP solver, this would be followed by the solution of the slightly changed LP that takes into account the new information. However, problem-specific knowledge can be incorporated into the algorithm at this point in order to infer additional information. The following rules can be used for this (in the following, u and v denote two vertices of the communication graph connected by edge e; x_u, x_v, and y_e are the corresponding variables):

Rule 1: If x_u and x_v are both fixed to the same value, then $y_e = 0$.

Rule 2: If x_u and x_v are fixed to different values, then $y_e = 1$.

[3]For reasons that are beyond the scope of the paper, this happens much more often than one might expect, especially in the lower levels of the search tree when the majority of the variables are already fixed to integer values.

Rule 3: If x_u is fixed to 0, but x_v is not yet fixed, then $y_e = x_v$.

Rule 4: If x_u is fixed to 1, but x_v is not yet fixed, then $y_e = 1 - x_v$.

All of the above rules are obvious based on problem-specific knowledge (*i.e.* based on the understanding of the hardware/software partitioning problem). On the other hand, these rules are very useful because they all reduce the number of variables and thus the complexity of the problem. It would be possible to use a full-fledged, general-purpose inference engine (such as the Sicstus Prolog CLP library) to make use of these rules. However, these rules are very special, so that they can be handled more efficiently in a proprietary way. This is going to be described next.

5.1 Knowledge representation

As can be seen from the above rules, our knowledge on a variable can be one of the following: (i) no knowledge; (ii) the variable is fixed to a value; (iii) the variable has the same value as another variable; (iv) the variable has the other value than another variable.

The first two kinds of knowledge can be easily represented and handled. For the last two kinds of knowledge, we will present a novel, very compact data structure, called the *star representation*. But first, some notations are defined.

Let Var denote the set of all variables, *i.e.* $Var = \{x_i : i = 1, 2, \ldots, n\} \cup \{y_j : j = 1, 2, \ldots, e\}$. Two relations R_1 and R_2 are defined on Var: for any $z_1, z_2 \in Var$, $R_1(z_1, z_2) \Leftrightarrow$ we know that $z_1 = z_2$. Similarly, $R_2(z_1, z_2) \Leftrightarrow$ we know that $z_1 = 1 - z_2$. Clearly, R_1 and R_2 have to be disjoint. It is easy to see that both R_1 and R_2 are symmetric relations, and therefore they define two undirected graphs on Var: $H_1 = (Var, R_1)$ and $H_2 = (Var, R_2)$. Furthermore, R_1 and R_2 have the following additional properties:

Rule 5: $R_1(z_1, z_2) \wedge R_1(z_2, z_3) \Rightarrow R_1(z_1, z_3)$ (*i.e.* R_1 is transitive)

Rule 6: $R_2(z_1, z_2) \wedge R_2(z_2, z_3) \Rightarrow R_1(z_1, z_3)$

Rule 7: $R_1(z_1, z_2) \wedge R_2(z_2, z_3) \Rightarrow R_2(z_1, z_3)$

Rule 8: $R_2(z_1, z_2) \wedge R_1(z_2, z_3) \Rightarrow R_2(z_1, z_3)$

Let $R_{1,2} = R_1 \cup R_2$ and $H_{1,2} = (Var, R_{1,2})$ be the corresponding undirected graph. In this graph, each edge is marked either with the label R_1 or with the label R_2, according to which relation it belongs to. Rules 5-8 imply the following for this graph:

Proposition 1. *(i) Let $z_1, z_2, \ldots, z_k$ ($k > 1$) be a path in $H_{1,2}$. Then $(z_1, z_k) \in R_{1,2}$. Furthermore, the label of (z_1, z_k) can be deduced from the labels of the edges of the path.*

(ii) Each connected component of $H_{1,2}$ is a complete graph.

(iii) A(n arbitrary) spanning forest of $H_{1,2}$ with the associated labels represents it in the sense that all other information present in $H_{1,2}$ can be deduced from the spanning forest.

Proof. We prove (i) by induction. For $k = 2$ the statement is trivial. For the induction step assume that it holds for $k - 1$. That means that $(z_1, z_{k-1}) \in R_{1,2}$ and $(z_{k-1}, z_k) \in R_{1,2}$. Furthermore, the labels of (z_1, z_{k-1}) and (z_{k-1}, z_k) are known. Rules 5-8 together imply that $(z_1, z_k) \in R_{1,2}$ and define its label.

(ii) is an easy consequence of (i). Let C be a connected component of $H_{1,2}$ and $u, v \in C$ two arbitrary vertices in it. Since C is connected, there is a path $u = z_1, z_2, \ldots, z_k = v$ between u and v. Applying (i) to this path yields that (u, v) is an edge in C, hence C is complete.

(iii) From (i) it follows that each connected component of $H_{1,2}$ can be represented with a spanning tree, thus $H_{1,2}$ can be represented with a spanning forest. □

Let us consider an example with seven variables: $z_1, \ldots, z_7$. Assume that the following information has been inferred: $R_1(z_1, z_2)$, $R_2(z_2, z_3)$, $R_1(z_4, z_5)$, $R_2(z_5, z_7)$, $R_2(z_6, z_7)$. This information is shown in Figure 3.

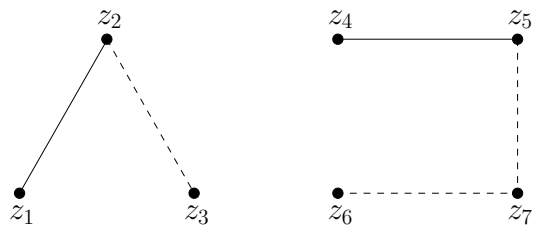

Figure 3: Example: directly inferred information. Solid lines indicate the relation R_1, dashed lines indicate the relation R_2

The corresponding $H_{1,2}$ graph contains this information, but also those pieces of information that can be inferred from this knowledge. For instance, it can be inferred using Rule 7 that $R_2(z_1, z_3)$ holds. The corresponding $H_{1,2}$ graph is shown in Figure 4. This graph possesses indeed the property of Proposition 1/(ii).

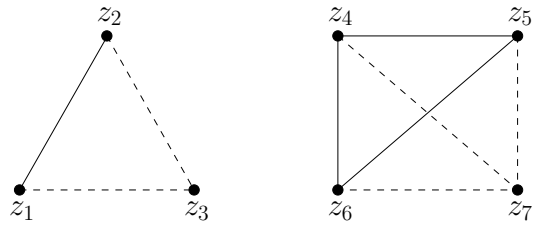

Figure 4: The $H_{1,2}$ graph corresponding to the example of Figure 3

As can be seen from the above, the graph $H_{1,2}$ contains all the information that has been inferred or can be inferred indirectly. Proposition 1/(iii) suggests that spanning trees of the connected components of the graph can be used for the compact representation of all this knowledge. It is also clear that at least a spanning tree has to be stored from each component because otherwise some information would be lost. Therefore, storing a spanning tree for each component is optimal concerning the size of the data structure. It only remains to decide which spanning trees to use. A complete graph has several spanning trees, which have of course the same number of edges, *i.e.* the size of the data structure does not depend on the choice of the spanning tree. However, the time needed to retrieve the information from the data structure does depend on the choice of the spanning tree. Specifically, determining the relation between two variables involves examining the path between them in the spanning tree, and so the time needed to retrieve the information correlates to the length of the paths in the spanning tree. It

follows that the spanning tree has to be chosen in such a way that all paths in it are short. This is accomplished by a star, in which each path consists of at most two edges. This way, the relation between two variables can be retrieved in $\mathcal{O}(1)$ time[4]. We refer to this representation as the *star representation*. Note that there are several possible star representations, because each component has several star-shaped spanning trees. However, all representations yield the same, optimal performance, so we can choose one arbitrarily. Figure 5 shows a possible star representation of the previous example.

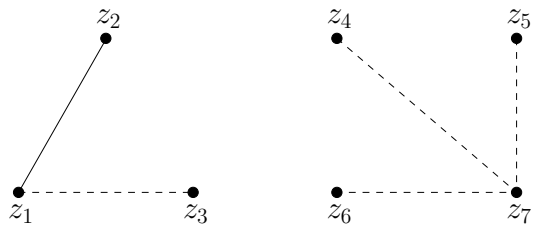

Figure 5: A possible star representation corresponding to the example of Figure 4

The following rules describe how the data structure is updated whenever new information is available:

Rule 9: If $z_1 = z_2$ is inferred, and z_1 and z_2 are in different components of $H_{1,2}$, then unify the two stars to a single one.

Rule 10: If $z_1 = 1 - z_2$ is inferred, and z_1 and z_2 are in different components of $H_{1,2}$, then unify the two stars to a single one.

Rule 11: If the value of z gets fixed, then fix all variables in the component of z appropriately.

Note that the newly inferred knowledge cannot lead to a contradiction in the $H_{1,2}$ graph—such as $R_1(z_1, z_2)$ and $R_2(z_1, z_2)$ holding at the same time—because all inferred information corresponds to a consistent labeling of the variables.

Unifying two stars involves detaching all nodes in the smaller star from its root and attaching them using the appropriate relation to the root of the bigger star, plus attaching the root of the smaller star to the root of the bigger star. Therefore, this operation takes time proportional to the size of the smaller star.

Continuing our example, Figure 6 shows the unification of the stars of Figure 5 after having inferred that $z_3 = 1 - z_6$.

To sum up: the star representation is a data structure with optimal size, $\mathcal{O}(1)$ information retrieval time, and linear update time.

5.2 Inference algorithm

Each time the branch-and-bound algorithm fixes a variable, the new information is propagated to the other variables using the above rules, until no more information can be inferred.

[4]This is possible, if we store to each node the set of its neighbors in the star plus a flag whether this node is the root of the star. Note that for the non-root nodes, the set of their neighbors in the star consists of just one element, namely the root. Given two nodes, we can determine their relation by stepping to the roots of the respective stars (if one of the nodes is a root on its own, then no step is needed in that case). If the two roots are different, then there is no relation between the two nodes. Otherwise we have found the two-edge path between the two nodes, which enables the deduction of the relation according to Proposition 1/(i).

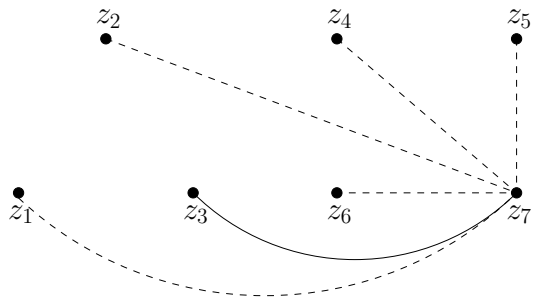

Figure 6: Unification of the stars of Figure 5

The inferred information is stored in a stack-like data structure, the *inference stack*, so that the inference steps can be easily undone when backtracking. Moreover, for easier access, the currently known information is also stored separately for each variable.

Finally, a technique called *constructive disjunction* (CD [52]) is applied. This means a controlled branching of given depth (typically 1). For instance, let z be a variable whose value is not yet known. CD works by first trying to substitute $z = 0$, checking if this leads to a contradiction with the cost constraints, and then also checking the $z = 1$ setting for contradiction. If both options lead to a contradiction, then the current branch can be cut off. If exactly one of the two options leads to a contradiction, then z can be set to the other value. In this case, CD was successful. If none of the two options leads to a contradiction, then nothing has been inferred, z remains unset, CD was not successful.

CD is motivated by the following facts:

- According to Rule 11, fixing a variable can lead to substantial new knowledge.
- Even the fixation of a single variable can increase H_P or R_P significantly. This can happen either because the corresponding node or edge of the original communication graph has a high cost, or because we fix a node of the communication graph in such a way that many edges become crossed by the partition.

In our algorithm, CD is applied to all variables that are roots in the star representation (isolated vertices also count as roots). Whenever a variable is fixed during CD, this information is propagated to the other variables using the previously described rules. The inference stack can be used to store the information inferred during CD as well, because it enables easy undoing. If CD is successful for at least one variable, then it makes sense to restart it for all variables, otherwise it does not.

Although CD involves a non-negligible time penalty, it can be very useful in pruning large regions of the search space, so that it is generally accepted in the artificial intelligence community as a very effective technique.

Finally, we note that general solvers for constraint logic programming (CLP) typically use branch-and-bound and inference. This means that inference itself is a quite powerful technique that can be used even without LP-relaxation. Our method can be regarded as a combination of an ILP-solver and a CLP-solver. As another consequence, it is not necessary to solve a linear program in each step. Since solving an LP is more time-consuming than the other steps of our algorithm, it makes sense to perform LP optimization less frequently.

6 Necessary conditions (lower bounds on costs)

As already discussed, branch-and-bound works well if large portions of the search tree can be cut off. For this, the algorithm must be able to recognize that a given partial solution will surely not yield a valid solution that is better than the best one found so far. In other words, necessary conditions are needed that a promising partial solution has to fulfill. If the current partial solution does not fulfill a necessary condition then the current branch can be cut off. Of course, such necessary conditions should be easily checkable. Until now, only one such necessary condition has been mentioned, namely the one based on LP-relaxation. In this section, some additional, problem-specific necessary conditions are described.

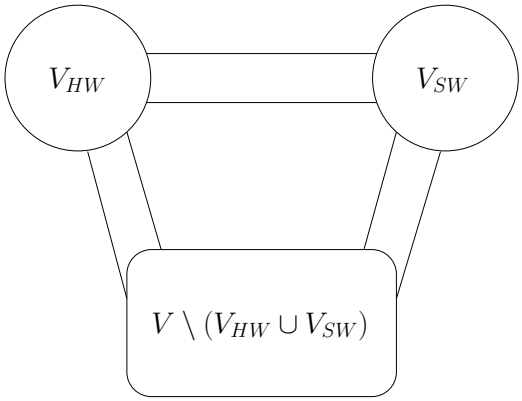

Figure 7: A partial solution

Suppose we have a current partial solution such as the one shown in Figure 7. V_{HW} denotes the set of nodes of the communication graph that are fixed to hardware, and similarly, V_{SW} denotes the set of nodes that are fixed to software. Let $H_{curr} = \sum_{v_i \in V_{HW}} h_i$ be the hardware cost of the nodes fixed to hardware, and $S_{curr} = \sum_{v_i \in V_{SW}} s_i$ be the software cost of the nodes fixed to software. These costs will surely be incurred. Similarly, the edges between V_{HW} and V_{SW} will surely be cut by the partition, which incurs a communication cost of $C_{curr} = \sum(c_{i,j} : v_i \in V_{HW}, v_j \in V_{SW}$ or $v_i \in V_{SW}, v_j \in V_{HW})$. The nodes in $V' = V \setminus (V_{HW} \cup V_{SW})$ are not yet fixed, therefore it is not known whether their hardware cost or their software cost will be incurred. Similarly, it is not yet known whether the edges incident to these nodes will be cut by the partition or not. Our aim is to find as high as possible lower bounds on the costs.

To find advanced, non-trivial necessary conditions, the techniques that we introduced in another paper [5] can be used. In that paper, we addressed a slightly different partitioning problem as well, in which there is no bound on R_P, but rather the objective is to minimize $T_P = \alpha H_P + \beta S_P + \gamma C_P$, where α, β, and γ are given non-negative weight factors. That problem is much easier and we have presented a fast, polynomial-time exact algorithm for it, which relies on finding the minimum cut in an auxiliary graph. Moreover, the algorithm could also be simply generalized to the case in which some components of the system are fixed to hardware or software.

Let us now turn back to the problem at hand. Suppose that the minimum of T_P is calculated (taking into account the already fixed nodes in V_{HW} and V_{SW}) for different α, β, and γ values. This minimum will be denoted by $T(\alpha, \beta, \gamma)$. Using the above-mentioned polynomial-time algorithm, such T values can be calculated quickly. Then, the following are necessary conditions (recall that R_0 denotes the bound on R_P, and H_0 is the hardware cost of the best valid partition found so far):

Condition 1: $T(0,1,1) \leq R_0$

Condition 2: $T(1,1,1) \leq H_0 + R_0 - 1$

Condition 1 holds because $T(0,1,1)$ is the minimum achievable R_P value, given the already fixed nodes. Testing Condition 1 decides optimally[5] whether the current partition can somehow be extended to a valid partition (*i.e.* satisfying R_0). Similarly, Condition 2 holds because $T(1,1,1)$ is the minimum achievable $H_P + R_P$ value, given the already fixed nodes (and because of the integrality of the cost values). Note that these conditions imply other, easier conditions, as follows:

Condition 3: (implication of Condition 1) $S_{curr} + C_{curr} \leq R_0$

Condition 4: (implication of Condition 2) $H_{curr} + S_{curr} + \sum_{v_i \in V'} \min(h_i, s_i) + C_{curr} \leq H_0 + R_0 - 1$

Another trivial necessary condition that does not follow from any of the previous conditions is also used:

Condition 5: $H_{curr} \leq H_0 - 1$

To sum up, Conditions 1, 2 and 5 should be used.

7 Improvement by heuristics

The speed of the branch-and-bound algorithm can be further improved using any heuristic algorithm for the partitioning problem, as shown in this section.

The main idea is the following. The speed of the algorithm depends heavily on how early branches can be cut off. If the algorithm notices a contradiction early, *i.e.* near the root of the search tree, then a large part of the search tree can be pruned. Clearly, if H_0 is low, then chances are good for finding a contradiction. Normally, H_0 is initialized with the value ∞, and each time a new, better solution is found, H_0 is decreased accordingly. Because of the above observations, our aim is to make H_0 as small as possible, as soon as possible.

Suppose that before running the branch-and-bound algorithm, a heuristic algorithm is run. The heuristic algorithm produces a valid partition with hardware cost H_{heur}, which is not necessarily optimal. Nevertheless, the branch-and-bound algorithm can be started with H_0 initialized to this value afterwards. This way, the property that the optimum solution is provided is not lost. However, if the heuristic algorithm is fast, and yields high-quality results (which is exactly what makes a good heuristic algorithm), then we can accelerate the branch-and-bound algorithm significantly by entailing only a small time penalty for running the heuristic algorithm beforehand.

In the following, we describe the usage of three heuristic algorithms for this purpose: a genetic algorithm, a minimum-cut-based algorithm, and a clustering-based algorithm. Of course, the idea of pre-optimization with a heuristic can be generally applied to ILP problems. In fact, ILP solvers do use general-purpose heuristics for this purpose. In this section, however, the usage of application-specific heuristics is discussed, which can yield better results. The effect of these heuristics on the overall performance of the algorithm is evaluated in Section 8. Another consequence is that two of the presented heuristics (the genetic algorithm and the clustering-based algorithm) can be combined in a more sophisticated way with the exact algorithm, not just in the form of pre-optimization.

[5] That is, Condition 1 is the strongest possible condition for this.

7.1 Genetic algorithm

In an earlier work [3], we presented a genetic algorithm (GA) for the hardware/software partitioning problem. It maintains a population of (not necessarily valid) hardware/software partitions, and improves them using the standard genetic operators recombination, mutation, and selection. This heuristic algorithm can be used without any modifications as a pre-optimization step before the branch-and-bound algorithm.

It is also possible to establish a stronger coupling between the two algorithms. Suppose that the two algorithms run in parallel, and maintain a common best-so-far solution. Whenever the genetic algorithm finds a better solution, H_0 can be decreased, which will—as discussed above—help the branch-and-bound algorithm prune parts of the search tree earlier. Conversely, when the branch-and-abound algorithm finds a better solution, this new, high-quality partition is injected into the population of the genetic algorithm, thus hopefully improving its effectiveness.

Unfortunately, our empirical tests have shown that the latter combination of the two algorithms is not very effective. Although the branch-and-bound algorithm benefits at first from the results found by the GA, the GA is simply not able to further improve the injected partitions, because they are of much better quality than the other individuals in the population of the GA. Therefore, we kept to the original idea of running the GA first, followed by the branch-and-bound algorithm.

7.2 Heuristic based on minimum cut

In another work [5], we have described a heuristic algorithm for the hardware/software partitioning problem, that is based on finding the minimum cut in a graph. (Note that this algorithm is not the same as the cut-based algorithm mentioned in Section 6, but it is based on that algorithm).

This algorithm, too, can be used as a pre-optimization step before the branch-and-bound algorithm.

7.3 Heuristic based on hierarchical clustering

A further possibility is hierarchical clustering, *i.e.* unifying vertices of the graph based on local closeness metrics until the size of the resulting graph is sufficiently small. Because of the local decisions typically employed in hierarchical clustering, this method is very fast, but of course sub-optimal. Obviously, it is an important question when the clustering process should stop. If many vertices are unified, then the resulting graph is small, so that it can be easily partitioned, but many degrees of freedom are lost. Conversely, if only few vertices are unified, then the resulting graph is quite big, so that its partitioning is still non-trivial, but the loss in degrees of freedom (and hence in potential solution quality) is not so significant.

Whenever two nodes v_i and v_j of the communication graph are unified, the software cost of the resulting new node will be s_i+s_j, and its hardware cost will be h_i+h_j. If parallel edges occur, they can be substituted with a single edge, the communication cost of which is the sum of the communication costs of the parallel edges. If a loop (*i.e.* a cycle of length 1) occurs, it can be simply abandoned because it does not participate in any cut of the graph. It follows that a partition P in the clustered graph can be expanded to a partition P' in the original graph with the same hardware, software, and communication costs.

Hierarchical clustering and the branch-and-bound algorithm can be combined as follows. First, the input graph is clustered until its size reaches a range that makes it possible to quickly find its optimal solution using branch-and-bound. This works, because the branch-and-bound algorithm is rather fast on small graphs. So in the second step, the optimal partition of the clustered graph is

obtained using the branch-and-bound algorithm. As discussed above, this defines a valid partition of the original graph with the same hardware cost, so that we have obtained a non-trivial H_0 value for the original graph. In the third step, the original partitioning problem is solved using the branch-and-bound algorithm, starting with this H_0 value. Clearly, this strategy also yields an optimal solution for the original problem, and if the clustering algorithm is good enough, then the whole process is significantly accelerated.

This idea can be generalized to a recursive, multi-level algorithm. Let $G_0 = G$ be the original graph, which has $n_0 = n$ vertices. Furthermore, let $\mu \in (0, 1)$ be a constant. The graphs G_i ($i = 1, 2, \ldots, k$; the number of vertices in G_i is denoted by n_i) are defined recursively as follows: G_i is obtained from G_{i-1} using hierarchical clustering, so that $n_i = \lfloor \mu n_{i-1} \rfloor$. k is chosen in such a way that G_k is the first graph in the series that is in the range of quickly solvable problem sizes. Now, the algorithm works as follows. First, the optimal partition in G_k is calculated. According to the choice of k, this can be quickly done. In a general step, after the optimal partition in G_i has been calculated, this value is used to initialize H_0, and to solve the partitioning problem in G_{i-1} optimally using this aid. That is, partitioning problems of increasing complexity are solved, while better and better aids (*i.e.* H_0 values) are available. At the end, G_0 is partitioned, which is exactly the original problem. An example for this process with $n_0 = 16$, $\mu = 0.5$, and $k = 2$ is shown in Figure 8. (Edges and costs are not shown for the sake of simplicity.)

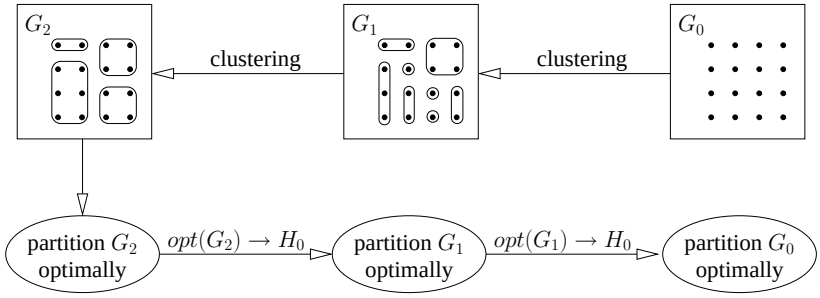

Figure 8: Hierarchical clustering and branch-and-bound combined in a multi-level algorithm

8 Empirical results

We implemented the presented algorithm in a C program and compared it to an implementation based on a commercial ILP solver [3]. Note that, since both algorithms are exact, only their speed had to be compared, and not the quality of the solutions they find. The empirical experiments were conducted on a Pentium III 1100MHz PC running SuSE Linux.

Table 1 lists the benchmarks that were used for testing. The vertices in the graphs correspond to high-level language instructions. Software and communication costs are time-dimensional, whereas hardware costs represent the occupied area. More details on the cost functions used in the benchmarks can be found in [5].

Note that the first three benchmarks – which are taken from MiBench [19] – have approximately the same size as the graphs that most previous algorithms were tested on[6]. The segment and fuzzy benchmarks are our own designs, and they are significantly larger. We also added some very complex

[6]In order to justify the usage of only small benchmarks, some authors argue that a typical program spends most of its execution time in some loops that are relatively small. While this may be true in general, there also exist programs in which

Name	n	e	Description
crc32	25	34	32-bit cyclic redundancy check. From the Telecommunications category of MiBench.
patricia	21	50	Routine to insert values into Patricia tries, which are used to store routing tables. From the Network category of MiBench.
dijkstra	26	71	Computes shortest paths in a graph. From the Network category of MiBench.
random1	200	200	Random communication graph.
random2	200	250	Random communication graph.
segment	150	333	Image segmentation algorithm in a medical application.
random3	300	300	Random communication graph.
random4	300	375	Random communication graph.
fuzzy	261	422	Clustering algorithm based on fuzzy logic.
rc6	329	448	RC6 cryptographic algorithm.
random5	400	400	Random communication graph.
random6	400	500	Random communication graph.
random7	500	500	Random communication graph.
mars	417	600	MARS cipher.
random8	500	625	Random communication graph.
random9	600	600	Random communication graph.
ray	495	908	Ray-tracing algorithm for volume visualization.
random10	600	750	Random communication graph.

Table 1: Summary of the used benchmarks.

benchmarks from the domains of cryptography and volume visualization as well as large random graphs to test the limits of the applicability of the algorithms.

Our algorithm adds three techniques to a pure ILP-solver-based algorithm: inference, lower bounds, and pre-optimization. In order to see the improvement by each of these techniques, we tested four versions of the algorithm:

ILP: Standard ILP solver[7], *i.e.* branch-and-bound algorithm without the suggested improvements

inference: Branch-and-bound algorithm with inference but without lower bounds and pre-optimization

inference+LB: Branch-and-bound algorithm with inference and lower bounds but without pre-optimization

there is a large number of such time-consuming loops, or there are also relatively big time-consuming loops. Furthermore, in very complex systems even the relatively small parts can be huge. Our larger benchmarks are examples of such systems.

[7] As ILP solver, we used version 3.2 of the package `lp_solve`, which is available from *ftp://ftp.es.ele.tue.nl/pub/lp_solve*. It uses branch-and-bound, the search tree is traversed in a depth-first-search fashion, and branching is performed each time according to a randomly chosen non-integer variable. For cutting off parts of the search tree, it uses the bound from LP-relaxation.

CUT+inference+LB: Branch-and-bound algorithm with inference, lower bounds, and pre-optimization based on minimum-cut[8]

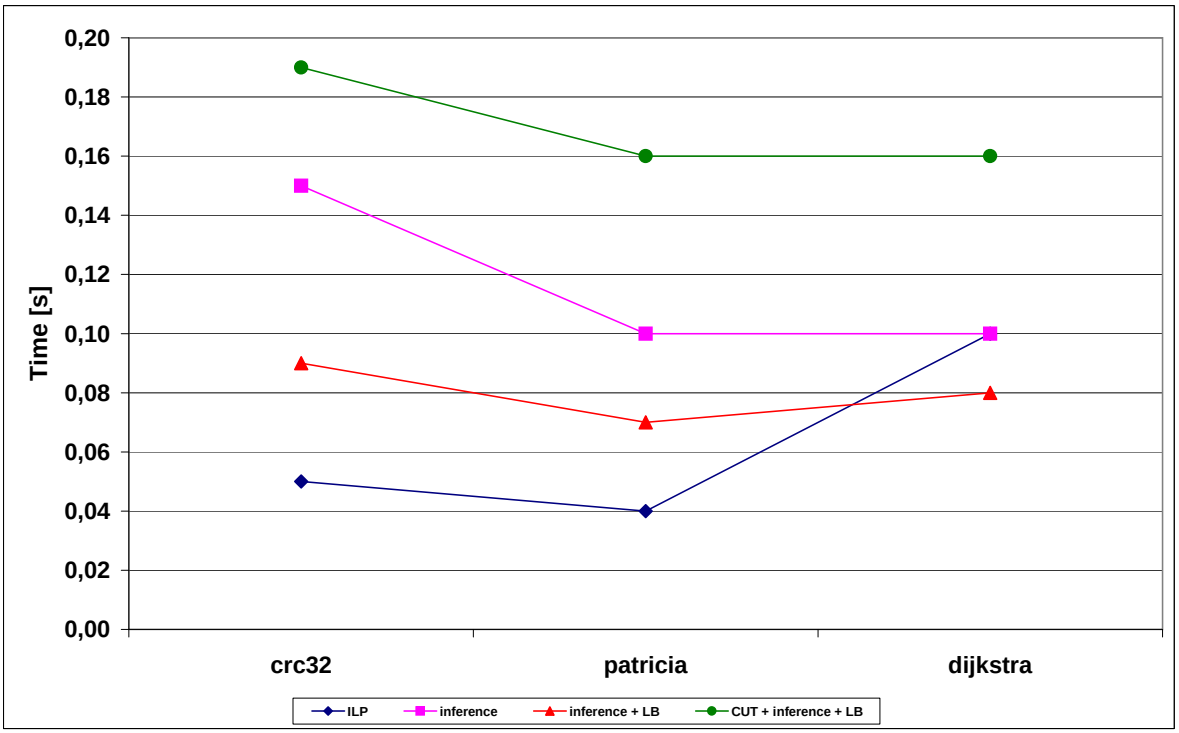

Figure 9: Algorithm runtimes on the easiest benchmarks

Figures 9-11 show the running time of the algorithms (the averate of five measurements) on benchmarks of low, medium, and high complexity. As can be seen from the results, the ILP algorithm is very fast on the three smallest benchmarks, but its running time explodes quickly. On the other hand, all versions of the branch-and-bound algorithm are slightly slower on the smallest benchmarks (but still fast – under 0.2 second), but much more efficient on the bigger examples. On benchmarks of medium complexity, the best version of the branch-and-bound algorithm is about 10 times faster than the ILP-based algorithm. On the largest benchmarks, only the best version of the branch-and-bound algorithm finished in acceptable time (within 1-2 hours), and in two cases also the algorithm without pre-optimization, the other versions did not finish within 10 hours. Of course, even the running time of the best version of the branch-and-bound algorithm falls victim to combinatorial explosion, but it can be seen that it can indeed solve very complex problems in reasonable time.

Figure 10 shows clearly that for non-trivial problem instances, each one of the proposed techniques significantly improves the performance of the algorithm.

Finally, we would like to revisit the issue of heuristic versus exact algorithms, which was mentioned in the Introduction. Table 2 shows the deviation of the result of the genetic algorithm from the optimum, in percentage. We chose the genetic algorithm for this purpose because it is typical of the kinds of algorithms that are widely used for hardware/software partitioning (see Section 2). In this set of experiments, we tried different R_0 values for all benchmarks, because the performance of the GA

[8]Of course, we also tested the other two pre-optimization heuristics, but they yielded somewhat worse results than the minimum-cut-based heuristic, therefore we omit those results here.

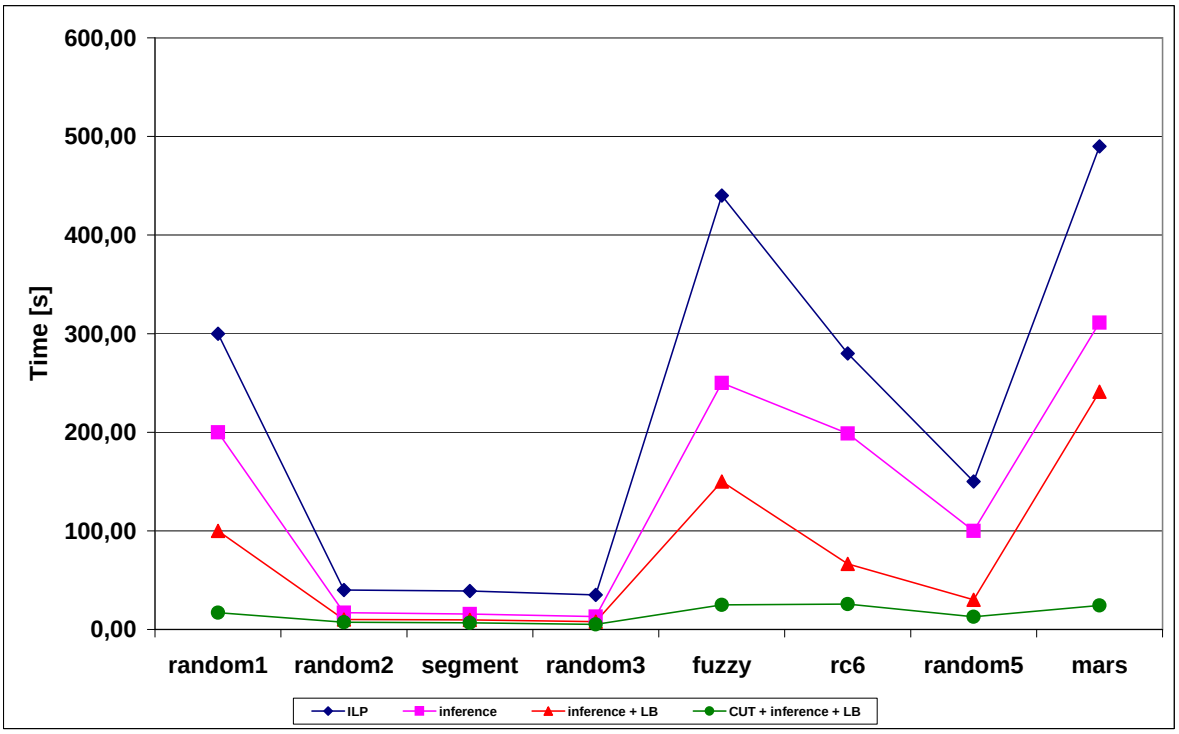

Figure 10: Algorithm runtimes on benchmarks of medium complexity

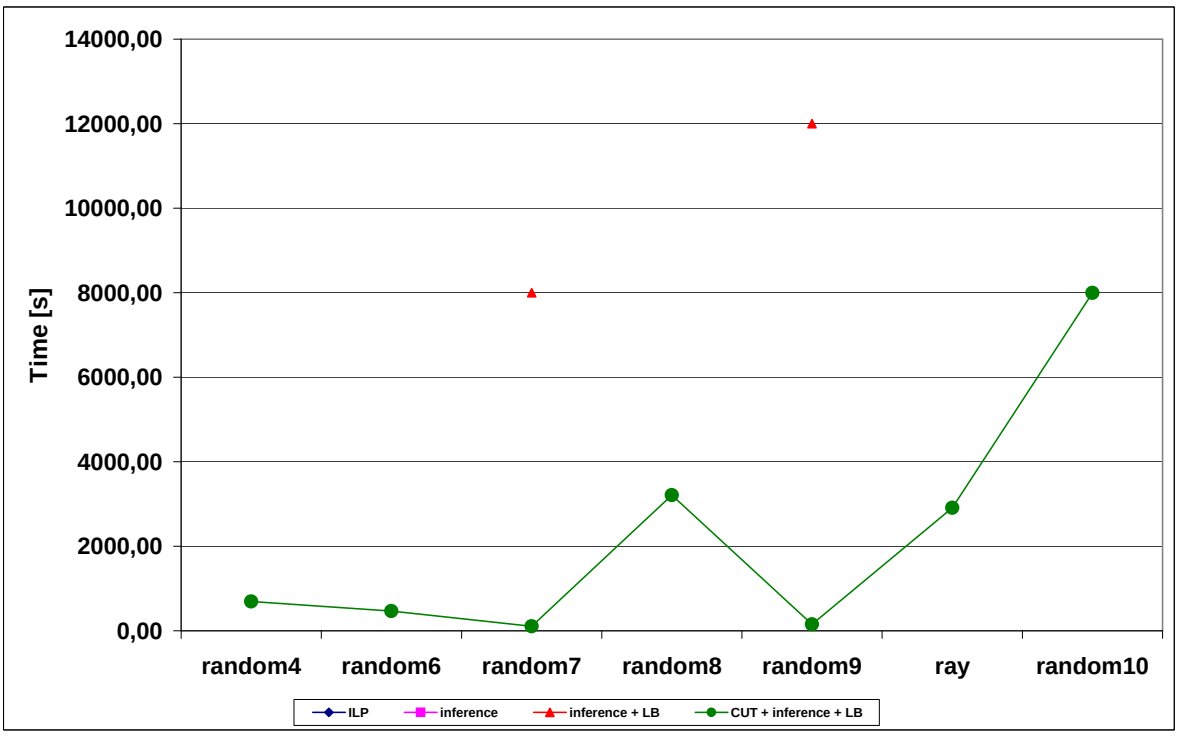

Figure 11: Algorithm runtimes on the hardest benchmarks

Benchmark	Minimum deviation	Average deviation	Maximum deviation
crc32	0	0	0
patricia	0	0	0
dijkstra	0	0.6%	3.2%
segment	1.4%	4.7%	16.1%
fuzzy	3.9%	14.9%	57.0%
rc6	22.5%	50.9%	126.1%
mars	19.1%	42.2%	93.0%
ray	58.3%	89.4%	171.6%

Table 2: The deviation of the result of GA from the optimum

depends significantly on this parameter [3]. The three columns of the table refer to the best, average, and worst of the results for the different R_0 values. As can be seen from the results, the GA finds almost always the optimal solution in the case of the three small benchmarks. However, as we move on to the bigger benchmarks, the relative performance of the GA also significantly worsens. In particular, the worst-case result for the ray benchmark is 171.6%, which is unacceptable in most applications. Hence, the exact algorithms offer significant rewards for their longer running times. And in the case of the small benchmarks, where the GA finds virtually always the optimum, the exact algorithms are also very fast.

9 Conclusions

This paper described a first attempt towards a practical exact algorithm for the hardware/software partitioning problem. We have shown that a branch-and-bound scheme can be used as a framework, into which further algorithms can be integrated. Specifically, we have integrated the following techniques into the algorithm: (i) lower bounds based on LP-relaxation; (ii) a custom inference engine; (iii) non-trivial necessary conditions based on a minimum-cut algorithm; (iv) different heuristics as a pre-optimization step. The presented methods can also be useful in other related optimization problems.

The empirical results have shown that the resulting algorithm is indeed capable of solving large problem instances in reasonable time. In particular, it is clearly more practical than a standard ILP solver.

The presented algorithm can be simply generalized to include more than one constraint (for instance, real-time constraints for different use cases of the system). Another generalization enables the system designer to prescribe that some nodes have to be in hardware, and some in software.

Several future research directions can be identified as well. A number of other algorithmic ideas can be incorporated, for instance, the Kernighan/Lin heuristic, more advanced heuristics for choosing the next variable to fix, using a custom LP solver etc. Furthermore, it should also be investigated in more depth what characteristics of a problem instance make it difficult or easy to solve optimally.

References

[1] T. F. Abdelzaher and K. G. Shin. Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers*, 49(1):81–87, 2000.

[2] J. K. Adams and D. E. Thomas. Multiple-process behavioral synthesis for mixed hardware/software systems. In *Proceedings of the IEEE/ACM 8th International Symposium on System Synthesis*, 1995.

[3] P. Arató, S. Juhász, Z. Á. Mann, A. Orbán, and D. Papp. Hardware/software partitioning in embedded system design. In *Proceedings of the IEEE International Symposium on Intelligent Signal Processing*, 2003.

[4] P. Arató, Z. Á. Mann, and A. Orbán. Hardware-software co-design for Kohonen's self-organizing map. In *Proceedings of the IEEE 7th International Conference on Intelligent Engineering Systems*, 2003.

[5] P. Arató, Z. Á. Mann, and A. Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 10(1):136–156, 2005.

[6] P. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, pages 11–18, March 1993.

[7] E. Barros, W. Rosenstiel, and X. Xiong. Hardware/software partitioning with UNITY. In *2nd International Workshop on Hardware-Software Codesign*, 1993.

[8] E. Barros, W. Rosenstiel, and X. Xiong. A method for partitioning UNITY language in hardware and software. In *Proceedings of the IEEE/ACM European Conference on Design Automation*, 1994.

[9] N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi. A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts. In *Proceedings of the 33rd Design Automation Conference*, 1996.

[10] K. S. Chatha and R. Vemuri. MAGELLAN: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *Proceedings of CODES 01*, 2001.

[11] A. Dasdan and C. Aykanat. Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(2):169–177, February 1997.

[12] R. P. Dick and N. K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of hierarchical heterogeneous distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, 1998.

[13] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. Hardware/software partitioning of VHDL system specifications. In *Proceedings of EURO-DAC '96*, 1996.

[14] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2(1):5–32, January 1997.

[15] R. Ernst, J. Henkel, and T. Benner. Hardware/software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, 1993.

[16] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, 1982.

[17] J. Grode, P. V. Knudsen, and J. Madsen. Hardware resource allocation for hardware/software partitioning in the LYCOS system. In *Proceedings of Design Automation and Test in Europe (DATE '98)*, 1998.

[18] R. K. Gupta and G. de Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, 1993.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, 1997.

[20] J. Henkel and R. Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Transaction on VLSI Systems*, 9(2):273–289, 2001.

[21] E. Hwang, F. Vahid, and Y. C. Hsu. FSMD functional partitioning for low power. In *Proceedings of the Design Automation and Test in Europe Conference*, 1999.

[22] A. Jantsch, P. Ellervee, and J. Oeberg. Hardware/software partitioning and minimizing memory interface traffic. In *Proceedings of the IEEE/ACM European Conference on Design Automation*, 1994.

[23] A. Kalavade. *System-level codesign of mixed hardware-software systems*. PhD thesis, University of California, Berkeley, CA, 1995.

[24] A. Kalavade and E. A. Lee. The extended partitioning problem: hardware/software mapping, scheduling and implementation-bin selection. *Design Automation for Embedded Systems*, 2(2):125–164, 1997.

[25] A. Kalavade and P. A. Subrahmanyam. Hardware/software partitioning for multifunction systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(9):819–837, September 1998.

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

[27] P. V. Knudsen and J. Madsen. PACE: a dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the IEEE/ACM 4th International Workshop on Hardware/Software Codesign*, 1996.

[28] M. Lopez-Vallejo, J. Grajal, and J. C. Lopez. Constraint-driven system partitioning. In *Proceedings of DATE*, pages 411–416, 2000.

[29] M. Lopez-Vallejo and J. C. Lopez. A knowledge based system for hardware-software partitioning. In *Proceedings of DATE*, 1998.

[30] M. Lopez-Vallejo and J. C. Lopez. Multi-way clustering techniques for system level partitioning. In *Proceedings of the 14th IEEE ASIC/SOC Conference*, pages 242–247, 2001.

[31] M. Lopez-Vallejo and J. C. Lopez. On the hardware-software partitioning problem: system modeling and partitioning techniques. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):269–297, July 2003.

[32] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. LYCOS: The Lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2(2):195–236, 1997.

[33] Z. Á. Mann and A. Orbán. Optimization problems in system-level synthesis. In *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003.

[34] B. Mei, P. Schaumont, and S. Vernalde. A hardware/software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of ProRISC*, 2000.

[35] R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, 1998.

[36] R. Niemann and P. Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems, special issue: Partitioning Methods for Embedded Systems*, 2:165–193, March 1997.

[37] M. O'Nils, A. Jantsch, A. Hemani, and H. Tenhunen. Interactive hardware-software partitioning and memory allocation based on data transfer profiling. In *International Conference on Recent Advances in Mechatronics*, 1995.

[38] M. F. Parkinson and S. Parameswaran. Profiling in the ASP codesign environment. In *Proceedings of the IEEE/ACM 8th International Symposium on System Synthesis*, 1995.

[39] S. Prakash and A. C. Parker. SOS: synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, 1992.

[40] S. Qin and J. He. An algebraic approach to hardware/software partitioning. Technical Report 206, UNU/IIST, 2000.

[41] G. Quan, X. Hu, and G. Greenwood. Preference-driven hierarchical hardware/software partitioning. In *Proceedings of the IEEE/ACM International Conference on Computer Design*, 1999.

[42] Y. G. Saab. A fast and robust network bisection algorithm. *IEEE Transactions on Computers*, 44(7):903–913, July 1995.

[43] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

[44] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. Hardware software partitioning with integrated hardware design space exploration. In *Proceedings of DATE*, 1998.

[45] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: a first approach. In *Proceedings of DAC*, 2003.

[46] F. Vahid. Modifying min-cut for hardware and software functional partitioning. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1997.

[47] F. Vahid. Partitioning sequential programs for CAD using a three-step approach. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):413–429, July 2002.

[48] F. Vahid and D. Gajski. Clustering for improved system-level functional partitioning. In *Proceedings of the 8th International Symposium on System Synthesis*, 1995.

[49] F. Vahid and T. D. Le. Extending the Kernighan/Lin heuristic for hardware and software functional partitioning. *Design Automation for Embedded Systems*, 2:237–261, 1997.

[50] W. Wolf. An architectural co-synthesis algorithm for distributed embedded computing systems. *IEEE Transactions on VLSI Systems*, 5(2):218–229, June 1997.

[51] W. Wolf. A decade of hardware/software codesign. *IEEE Computer*, 36(4):38–43, 2003.

[52] J. Würtz and T. Müller. Constructive disjunction revisited. In *20th German Annual Conference on Artificial Intelligence*, 1996.