Algorithmic aspects of hardware/software partitioning

PÉTER ARATÓ, ZOLTÁN ÁDÁM MANN, ANDRÁS ORBÁN
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology

One of the most crucial steps in the design of embedded systems is hardware/software partitioning, i.e. deciding which components of the system should be implemented in hardware and which ones in software. Most formulations of the hardware/software partitioning problem are $\mathcal{NP}$-hard, so the majority of research efforts on hardware/software partitioning has focused on developing efficient heuristics.

This paper considers the combinatorial structure behind hardware/software partitioning. Two similar versions of the partitioning problem are defined, one of which turns out to be $\mathcal{NP}$-hard, whereas the other one can be solved in polynomial time. This helps in understanding the real cause of complexity in hardware/software partitioning. Moreover, the polynomial-time algorithm serves as the basis for a highly efficient novel heuristic for the $\mathcal{NP}$-hard version of the problem. Unlike general-purpose heuristics such as genetic algorithms or simulated annealing, this heuristic makes use of problem-specific knowledge, and can thus find high-quality solutions rapidly. Moreover, it has the unique characteristic that it also calculates *lower bounds on the optimum solution.* It is demonstrated on several benchmarks and also large random examples that the new algorithm clearly outperforms other heuristics that are generally applied to hardware/software partitioning.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD); F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.1 [**Discrete Mathematics**]: Combinatorial algorithms

General Terms: Algorithms, Design

Additional Key Words and Phrases: hardware/software partitioning, hardware/software co-design, graph bipartitioning, graph algorithms, optimization

1. INTRODUCTION

Today's computer systems typically consist of both hardware and software components. For instance in an embedded signal processing application it is common to use both application-specific hardware accelerator circuits and general-purpose, programmable units with the appropriate software [Arató et al. 2003].

This is beneficial since application-specific hardware is usually much faster than software, and also more power-efficient, but it is also significantly more expensive.

Partial support of the Hungarian National Science Fund (Grant No. OTKA T043329 and T042559) is gratefully acknowledged.
Authors' address: H-1117 Budapest, Magyar tudósok körútja 2, Hungary; e-mail: arato@iit.bme.hu, {zoltan.mann, andras.orban}@cs.bme.hu

Software on the other hand is cheaper to create and to maintain, but slow, and general-purpose processors consume much power. Hence, performance or power critical components of the system should be realized in hardware, and non-critical components in software. This way, an optimal trade-off between cost, power and performance can be achieved.

One of the most crucial steps in the design of such systems is partitioning, i.e. deciding which components of the system should be realized in hardware and which ones in software. Clearly, this is the step in which the above-mentioned optimal trade-off should be found. Therefore, partitioning has dramatic impact on the cost and performance of the whole system [Mann and Orbán 2003]. The complexity of partitioning arises because conflicting requirements on performance, power, cost, chip size, etc. have to be taken into account.

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.

1.1 Previous work

Based on the partitioning algorithm, exact and heuristic solutions can be differentiated. The proposed exact algorithms include branch-and-bound [Binh et al. 1996], dynamic programming [Madsen et al. 1997; O'Nils et al. 1995], and integer linear programming [Mann and Orbán 2003; Niemann 1998; Niemann and Marwedel 1997].

The majority of the 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, and the exact algorithms for them have exponential runtimes. The $\mathcal{NP}$-hardness of hardware/software partitioning was claimed by several researchers [Binh et al. 1996; Eles et al. 1996; Kalavade 1995; Vahid and Gajski 1995], but we know only about one proof [Kalavade 1995] for a particular formulation of the hardware/software partitioning problem.

Many researchers have applied general-purpose heuristics to hardware/software partitioning. In particular, genetic algorithms have been extensively used [Arató et al. 2003; Dick and Jha 1998; Mei et al. 2000; Quan et al. 1999; Srinivasan et al. 1998], as well as simulated annealing [Eles et al. 1997; Ernst et al. 1993; Henkel and Ernst 2001; Lopez-Vallejo et al. 2000]. Other, less popular heuristics in this group are tabu search [Eles et al. 1997] and greedy algorithms [Chatha and Vemuri 2001; Grode et al. 1998].

Some researchers used custom heuristics to solve hardware/software partitioning. This includes the GCLP algorithm [Kalavade and Lee 1997; Kalavade and Subrahmanyam 1998] and the expert system of [Lopez-Vallejo and Lopez 1998; 2003], as well as the heuristics in [Gupta and de Micheli 1993] and [Wolf 1997].

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 [Abdelzaher and Shin 2000; Barros et al. 1993; Vahid 2002; Vahid and Gajski 1995]. The other group of partitioning-related heuristics is the Kernighan-Lin heuristic [Kernighan and Lin 1970], which was substantially improved by Fiduccia

and Mattheyses [1982], and later by many others [Dasdan and Aykanat 1997; Saab 1995]. These heuristics have been found to be appropriate for hardware/software partitioning as well [Lopez-Vallejo and Lopez 2003; Vahid 1997; Vahid and Le 1997].

Concerning the system model, further distinctions can be made. In particular, many researchers consider scheduling as part of partitioning [Chatha and Vemuri 2001; Dick and Jha 1998; Kalavade and Lee 1997; Lopez-Vallejo and Lopez 2003; Mei et al. 2000; Niemann and Marwedel 1997], whereas others do not [Eles et al. 1996; Grode et al. 1998; Madsen et al. 1997; O'Nils et al. 1995; Vahid and Le 1997; Vahid 2002]. Some even include the problem of assigning communication events to links between hardware and/or software units [Dick and Jha 1998; Mei et al. 2000].

In a number of related papers, the target architecture is supposed to consist of a single software and a single hardware unit [Eles et al. 1996; Grode et al. 1998; Gupta and de Micheli 1993; Henkel and Ernst 2001; Lopez-Vallejo and Lopez 2003; Madsen et al. 1997; Mei et al. 2000; O'Nils et al. 1995; Qin and He 2000; Srinivasan et al. 1998; Stitt et al. 2003; Vahid and Le 1997], whereas others do not impose this limitation. Some limit parallelism inside hardware or software [Srinivasan et al. 1998; Vahid and Le 1997] or between hardware and software [Henkel and Ernst 2001; Madsen et al. 1997]. The system to be partitioned is generally given in the form of a task graph, or a set of task graphs, which are usually assumed to be directed acyclic graphs describing the dependencies between the components of the system.

1.2 Our approach

In this paper, we take a more theoretical approach than most previous works by focusing only on the algorithmic properties of hardware/software partitioning. In particular, we do not aim at partitioning for a given architecture, nor do we present a complete co-design environment. Rather, we restrict ourselves to the problem of deciding—based on given cost values—which components of the system to implement in hardware and which ones in software. This problem will be formalized as a graph bipartitioning problem. Using the graph-theoretic properties of the problem, we can develop more powerful algorithms—as will be shown later. Furthermore, the underlying problem definition is general enough so that the algorithms we propose can be used in many practical cases.

Our aims are the following:

—Clarifying complexity issues, such as: Is partitioning really $\mathcal{NP}$-hard? When is it $\mathcal{NP}$-hard? Why is it $\mathcal{NP}$-hard?

—Developing more powerful partitioning algorithms by capturing the *combinatorial structure* behind the partitioning problem. That is, instead of applying general-purpose heuristics to hardware/software partitioning, we develop algorithms based on the graph-theoretic properties of partitioning. This way, we hope to obtain more scalable algorithms.

Scalability is a major concern when applying general-purpose heuristics. Namely, 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. This effect can be overcome only if the small evaluated region contains high-quality solutions. This is exactly what we intend to achieve by making use of the combinatorial properties of the problem.

Specifically, we define two slightly different versions of the hardware/software partitioning problem. One of them is proven to be $\mathcal{NP}$-hard, whereas a polynomial-time exact algorithm is provided for the other one. We believe that this difference sheds some light on the origins of complexity of hardware/software partitioning.

Our main contribution is a novel heuristic algorithm for the $\mathcal{NP}$-hard version of the partitioning problem which is based on the polynomial-time algorithm for the other version of the problem. This heuristic has the property mentioned above that it only evaluates points of the search space that have a high quality in some sense. Consequently, this heuristic outperforms conventional heuristics, which is demonstrated with empirical tests on several benchmarks. Moreover, the new heuristic has the unique property that it can determine a *lower bound on the cost of the optimum solution*, and therefore it can estimate how far the result it found so far lies from the optimum. This is a feature that no previous partitioning algorithm possessed.

The rest of the paper is organized as follows. Section 2 provides formal definitions for the hardware/software partitioning problem. This is followed by the analysis of the defined problems in Section 3 and the description of our algorithms in Section 4. Empirical results are given in Section 5, and Section 6 concludes the paper. Finally, the proof of our theorems are presented in the Appendix.

2. PROBLEM DEFINITION

2.1 Basic model

In the basic model the system to be partitioned is described 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 this 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 processor (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 [Madsen et al. 1997]. 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 in 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_{ij}) 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. These are rather abstract and possibly conflicting cost metrics that should be optimized together. The most common approach is to assemble a single objective function $f(cost_1, \ldots, cost_l)$ containing all the metrics. We consider two versions of f that can be regarded as the two extremes—yielding two rather different versions of the partitioning problem.

In the first version, the aim of partitioning is to minimize the weighted sum of these three metrics, i.e. f is linear in all of its arguments. The weights are specified by the designer, and define the relative importance of the three metrics. More formally, we define the total cost of P as $T_P = \alpha H_P + \beta S_P + \gamma C_P$, where α, β, and γ are given non-negative constants, and the aim is to minimize T_P.

In the second version, one of the cost metrics is constrained by a hard upper limit. This case can also be modeled with an f function which adds an infinite penalty if the constraint is hurt. A possible interpretation can be the following: if software cost captures execution time, and communication cost captures the extra delay generated by communication, then it makes sense to add them. That is, we define the running time of the system with respect to partition P as $R_P = S_P + C_P$. We suppose that there is a real-time constraint, i.e. a constraint on R_P, and the aim is to minimize H_P while satisfying this constraint.

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

P1: Given the graph G with the cost functions h, s, and c, and the constants $\alpha, \beta, \gamma \geq 0$, find a hardware/software partition P with minimum T_P.
P2: 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.

2.2 Extensions to the basic model

The basic model of hardware-software partitioning captures many important characteristics of the problem. Its compactness allows us to develop efficient algorithms and helps us better understand the nature of the partitioning problem. However, the basic model can be extended in several ways to incorporate more details.

The problem graph can be extended with dependency information. In this case the communication graph should be rather directed and acyclic. Most authors (see Section 1.1 about previous work) follow this approach.

To respect dependency, the nodes ordered to a context should be scheduled properly, thus the additivity of execution times is not valid anymore. Scheduling requires the execution time of each node; this can be the software/hardware cost or an additional parameter of each node. Whether scheduling is regarded as part of the partitioning or done afterwards is still a question under discussion in the community (recall the different approaches from Section 1.1).

These extensions might be beneficial, but there is a risk that a too complex model can hide the true nature of the problem and only very small instances can be solved. First we show our algorithms for the basic model and then, in Section 4.4 we explain how they can be adapted to these extensions.

3. COMPLEXITY RESULTS

THEOREM 3.1. *The P1 problem can be solved optimally in polynomial time.*

PROOF. We can assume $\alpha = \beta = \gamma = 1$ because otherwise we multiply each h_i by α, each s_i by β, and each c_{ij} by γ. With this modification the problem becomes similar to the one solved in [Stone 1977]. Although Stone handles only one cost metric (time) instead of the linear combination of several cost metrics, the proof of this theorem is identical to [Stone 1977]. The details are omitted, only the main idea of the construction is given to help understand our later algorithms.

We construct an auxiliary graph (see Figure 1) $G' = (V', E')$ based on G as follows: $V' = V \cup \{v_s, v_h\}$, $E' = E \cup E_s \cup E_h$, where $E_s = \{(v, v_s) : v \in V\}$ and $E_h = \{(v, v_h) : v \in V\}$. G' is also a simple, undirected graph, but in G' only the edges are assigned costs; the cost of edge $e \in E'$ is denoted by $b(e)$, and defined as follows:

$$b(e) = \begin{cases} c(e) & \text{if } e \in E \\ h_i & \text{if } e = (v_i, v_s) \in E_s \\ s_i & \text{if } e = (v_i, v_h) \in E_h \end{cases}$$

Note that the edges in E_s (i.e. those that connect the vertices to v_s) are assigned the h values, and the edges in E_h are assigned the s values, and not vice versa.

LEMMA 3.2 (STONE, 1977). *The value of the minimum cut in G' between v_s and v_h is equal to the optimum of the original graph bipartitioning problem.* □

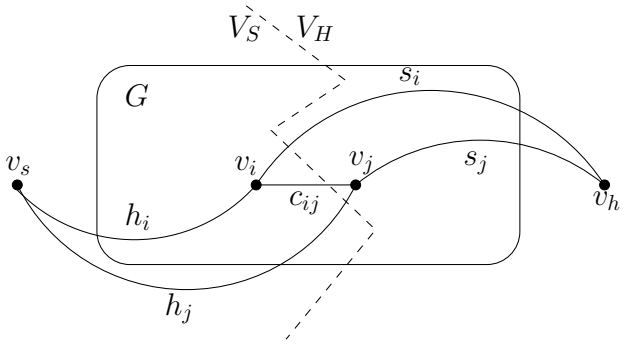


Fig. 1. The auxiliary graph

Algorithm 1 Polynomial-time algorithm for the P1 problem

1. Create the auxiliary graph.
2. Find a minimum cut in the auxiliary graph.

By Lemma 3.2, we have reduced the hardware/software partitioning problem to finding a minimum cut between two vertices in a simple undirected graph, for which polynomial-time algorithms are known [Ahuja et al. 1993]. Note that the size of G' is not significantly larger than that of G: if G has n vertices and m edges, then G' has $n+2$ vertices and $m+2n$ edges. This proves the theorem. □

THEOREM 3.3. *The P2 problem is $\mathcal{NP}$-hard in the strong sense.*

PROOF. The proof can be found in Appendix A. □

These two theorems show that—supposed that $\mathcal{P} \neq \mathcal{NP}$—the P2 problem is significantly harder than the P1 problem. This sheds some light on the origin of complexity in hardware/software partitioning: under the assumption of additivity of costs, the problem is easy if the different cost factors are combined using weighted sum to form a single objective function, whereas it becomes hard if they are bounded or optimized separately.

The other lesson learned from the above two theorems is that not all formulations of the partitioning problem are necessarily $\mathcal{NP}$-hard. The P1 problem, which is apparently easy, is also a meaningful formulation of the hardware/software partitioning problem that can capture a number of real-world variants of the problem. Hence, care has to be taken when claiming that partitioning is $\mathcal{NP}$-hard.

4. ALGORITHMS

4.1 Algorithm for the P1 problem

The proof of Theorem 3.1 suggests a polynomial-time algorithm for the P1 problem, as summarized in Algorithm 1.

Clearly, the first step of the algorithm can be performed in linear time. For the second step, many algorithms are known. We used the algorithm of Goldberg and Tarjan for finding maximum flow and minimum cut [Goldberg and Tarjan 1988; Cherkassky and Goldberg 1997], which works in $O(n^3)$ time, where n denotes the

number of vertices in the graph. Therefore, the whole process can be performed in $O(n^3)$ time[1]. Note that $O(n^3)$ is just a theoretic upper bound for the runtime of Algorithm 1. As will be shown in Section 5, the algorithm is extremely fast in practice.

Note that the condition that α, β, and γ are non-negative is important because no polynomial-time algorithm is known for finding the minimum cut in a graph with arbitrary edge costs (i.e. where the edge costs are not necessarily non-negative). In fact, this problem is $\mathcal{NP}$-hard.

It is important to mention that it is not essential that there are exactly three cost metrics to optimize. The same approach works for an arbitrary number of cost metrics as far as the linear combination of them should be minimized. (See Section 4.4 for more details.)

Finally we note that the algorithm can easily accommodate the following extension to the partitioning model: some components can be fixed to software, while others can be fixed to hardware (e.g. because the other implementation would not make sense or because of some existing components that should be integrated into the system). In this case, the components that are fixed to software are coalesced to form the single vertex v_s, and similarly, the components that are fixed to hardware are coalesced to form the single vertex v_h. If parallel edges arise, they can be unified to a single edge whose cost is the sum of the costs of the parallel edges. If a loop (i.e. an edge connecting a vertex to itself) arises, it can be simply discarded because it does not participate in any cut of the graph.

4.2 Heuristic algorithm for the P2 problem

We now show a heuristic for the P2 problem based on Algorithm 1. The idea is to run Algorithm 1 with several different α, β, and γ values. This way, a set of candidate partitions is generated, with the property that each partition is optimal for the P1 problem with some α, β, and γ parameters. Then we select the best partition from this set that fulfills the given limit on R_P.

As already mentioned, the scalability of a heuristic depends on whether the evaluated small fraction of the search space contains high-quality points. We believe that we can achieve this with the above choice of candidate partitions.

Obviously, the result of the run of Algorithm 1 is determined by the *ratio* of the three weights, and not by their absolute values. Therefore, we can fix one of the three, e.g. β, and vary only the other two. Thus, we have a two-dimensional search problem, in which the evaluation of a point involves running Algorithm 1 with the appropriate weights.

In order to keep our algorithm fast, we use two phases: in the first phase, we use coarse-grained steps in the two-dimensional plane to find the best valid partitioning approximately, and in the second phase we use a more fine-grained search in the neighborhood of the point found in the first phase (see Algorithm 2 for more details).

In both phases, possible α and γ values are scanned with increments $d\alpha$ and $d\gamma$. Choosing the values for $d\alpha$ and $d\gamma$ constitutes a trade-off between quality and

[1]When Stone published his similar approach in [Stone 1977], the algorithms for finding a minimum cut in a graph were much slower. In fact, Stone claimed his partitioning algorithm to have $O(n^5)$ running time, thus it was rather impractical.

Algorithm 2 Heuristic algorithm for the P2 problem

Phase 1: //Scan the whole search space
FOR($\alpha = \alpha_{min}$; $\alpha < \alpha_{max}$; $\alpha = \alpha + d\alpha$)
  FOR($\gamma = \gamma_{min}$; $\gamma < \gamma_{max}$; $\gamma = \gamma + d\gamma$) {
    run Algorithm 1 with parameters α, β, and γ to obtain optimal partition P;
    IF($R_P < R_0$ AND $H_P <$ best_so_far) {
      save current solution;
      save previous and next α value (α_{prev}, α_{next});
      save previous and next γ value (γ_{prev}, γ_{next});
      reset $d\alpha$ and $d\gamma$;
    }
    ELSE
      $d\alpha = (1+\varepsilon)d\alpha$, $d\gamma = (1+\varepsilon)d\gamma$;
  }

Phase 2: //Scan the region around the best point found in Phase 1
reset $d\alpha$ and $d\gamma$;
perform same method as in Phase 1, with α going between α_{prev} and α_{next} and γ going between γ_{prev} and γ_{next} and using $\varepsilon' < \varepsilon$ instead of ε.

performance: if small increments are used, then the search is very thorough but slow, if the increments are high, the search becomes fast but superficial. As can be seen in Algorithm 2, we apply a searching scheme that adjusts the increments *dynamically*. More specifically, $d\alpha$ and $d\gamma$ are multiplied with $1+\varepsilon$ (where ε is a fixed small positive number) in each step when no better solution is found. This way, the algorithm accelerates exponentially in low-quality regions of the search space. On the other hand, $d\alpha$ and $d\gamma$ are reset whenever a better solution is found, thus the search slows down as soon as it finds a better solution. After our initial tests, we fixed $\varepsilon = 0.02$ and $\varepsilon' = 0.01$, which seemed to offer a good trade-off between speed and quality.

This way, the first phase can find the approximately best values for α and γ, but it is possible that the algorithm jumps over the best values. This is corrected in the second phase. Clearly, this approach works fine if the cost functions are smooth enough and have a relatively simple structure. We will come back to this issue in Section 5.

Finally, it should be mentioned how α_{min}, γ_{min}, α_{max} and γ_{max} are chosen. The following theorem, the proof of which is omitted for brevity, is useful for this purpose:

THEOREM 4.1. *(i) If* $\alpha \leq \beta \cdot \min_{v \in V} \frac{s(v)}{h(v)}$, *then the all-hardware partition is optimal with respect to* α, β *and* γ *(regardless of the value of* γ*).*
(ii) If $\alpha \geq \beta \cdot \max_{v \in V} \frac{s(v)}{h(v)}$, *then the all-software partition is optimal with respect to* α, β *and* γ *(regardless of the value of* γ*).*
(iii) Suppose that G *is connected, and let* c_{min} *denote the smallest edge cost. If* $\min(\alpha \cdot h(V), \beta \cdot s(V)) \leq \gamma \cdot c_{min}$, *then either the all-hardware or the all-software partition is optimal with respect to* α, β *and* γ.

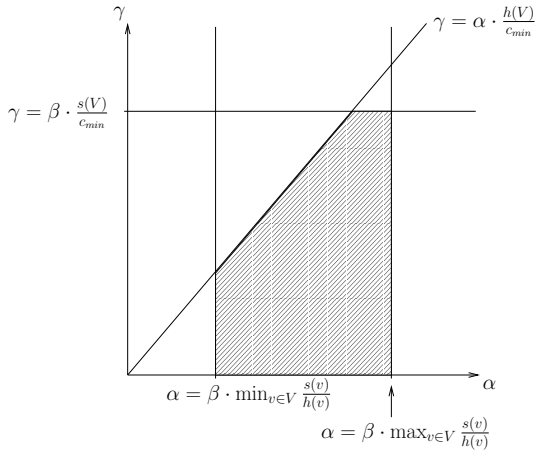


Fig. 2. The region to be scanned

Therefore, the region that has to be scanned looks as depicted in Figure 2.

4.3 Determining lower bounds

As mentioned earlier, Algorithm 2 can also incorporate the feature of determining lower bounds on the cost of the optimal solution of the given P2 problem instance. This is a unique feature of this algorithm, that is offered by no other competing heuristic. With the help of this feature, Algorithm 2 can maintain an estimate of how far the best solution it found so far is from the optimum. This is very advantageous because it helps evaluate the performance of the algorithm. Moreover, if the lower bound and the found best solution are not far from each other, this may indicate that there is no point in continuing the search. For instance, if the cost values assigned to the nodes and edges of the graph are measured values with a precision of 10%, then there is no point in continuing the search if the gap between the lower bound and the found best solution is under 10% of the lower bound. This way, we can reduce the runtime of the algorithm without any practical loss in the quality of the found solution.

Informally, Algorithm 2 is able to determine the lower bounds because every candidate partition that it evaluates is optimal for the P1 problem with some α, β, and γ values. Hence, each evaluated candidate partition tells us something about the costs of *all* partitions. This is formalized by the following theorem, the proof of which can be found in Appendix B.

THEOREM 4.2. *Suppose that P is an optimal solution of the P1 problem with the weights α, β, and γ. Let Q be any solution of the P2 problem (i.e. a partition that satisfies the bound $R_Q \leq R_0$). Then*

$$H_Q \geq H_P + \frac{\beta S_P + \gamma C_P - \max(\beta, \gamma) R_0}{\alpha} \tag{1}$$

Note that the right-hand side of (1) contains only known numbers. Therefore, the algorithm can compute a lower bound based on each evaluated candidate partition,

and use the best one of these lower bounds. Unfortunately, there is no guarantee that the lower bound will not be far off the optimum. However, as shown in Section 5, the gap between the found best partition and the lower bound was not big for practical benchmarks.

4.4 Adaptation of the algorithms to other partitioning models

Recall some possible extensions to our basic model from Section 2.2. These extensions can be easily handled by our algorithms without any change in our main idea: try to use the good candidates found by the first algorithm to guide the search of the second algorithm.

The algorithm for P1 can handle any number of cost metrics assigned to software/hardware side, provided their weighted sum should be minimized; for example every vertex v_i might have a software execution time st_i, a software implementation effort se_i, a hardware execution time ht_i, a hardware chip area ha_i and further on a cut edge e implies a communication penalty of $c(e)$. If our aim is to minimize $\alpha ST_P+\beta SE_P+\gamma HT_P+\delta HA_P+\varepsilon C_P$ with ST_P, SE_P, HT_P, HA_P defined obviously, then a similar auxiliary graph can be built as in the construction of Theorem 3.1, but the new edge weights $b(e)$ are as follows.

$$b(e) = \begin{cases} \varepsilon c(e), & \text{if } e \in E \\ \gamma ht_i + \delta ha_i, & \text{if } e = (v_i, v_s) \in E_s \\ \alpha st_i + \beta se_i, & \text{if } e = (v_i, v_h) \in E_h \end{cases}$$

Lemma 3.2 will remain true for this graph, hence this extended P1 problem can be similarly solved.

Moreover, scheduling of the tasks can also be incorporated in Algorithm 2. The subroutine of Algorithm 1 returns with a possible solution candidate. One can use any scheduling algorithm available in the literature to evaluate this candidate. The scheduling should be inserted just after the call to Algorithm 1 in line 4 of Algorithm 2. It makes the algorithm more complicated though, but it does not change our approach. Therefore in the test phase we were focusing on the evaluation of the basic algorithms to validate our concept.

5. EMPIRICAL RESULTS

We have implemented the above algorithms using the minimum cut algorithm of Goldberg and Tarjan [Goldberg and Tarjan 1988; Cherkassky and Goldberg 1997]. We had to modify the construction in the proof of Theorem 3.1 slightly because the used minimum cut algorithm works on directed graphs.

Generally, if we want to find the minimum cut in an undirected graph using an algorithm for directed graphs[2], then we have to change every undirected edge to two directed edges going in opposite directions. However, edges directed to the source or from the sink can be removed, because this does not change the value of the maximum flow, and hence it does not change the value of the minimum cut.

[2]There are also algorithms for finding the minimum cut in an undirected graph, which are even faster than the ones for directed graphs. However, we need a cut that separates two given vertices (a so-called *st*-cut), and for this problem, no faster algorithms are known for the undirected case.

n	Running time of Algorithm 1 [sec]
100	0.0007
1000	0.0198
3000	0.0666
5000	0.1264
7000	0.1965
10000	0.2896

Table I. Running time of Algorithm 1

In our case, this means that the edges in the original graph are introduced in two copies in the new graph, in opposite directions, but in the case of the additional edges (i.e. edges in E_s and E_h), only one copy is needed, directed to v_h, or from v_s, respectively.

The algorithms have been implemented in C, and tested on a Pentium II 400MHz PC running SuSE Linux. We conducted two sets of experiments: one for evaluating the performance of Algorithm 1, and one for evaluating the performance and effectiveness of Algorithm 2.

5.1 Experience with Algorithm 1

Since Algorithm 1 finds the optimal solution for the partitioning problem at hand, we only had to test its speed on practical problem instances. (Recall from Section 4.1 that it is an $O(n^3)$ algorithm, but this is only an asymptotic upper bound on its running time.)

For testing Algorithm 1, several random graphs of different size and with random costs have been used. In order to reduce the number of test runs and the amount of test data to process, we fixed the ratio of edges and vertices in the test graphs to 2, which means that on average, each vertex has four neighbors. Previous experience with real-world task graphs [Arató et al. 2003] has shown that this average is typical.

Table I shows measurement results concerning the running time of Algorithm 1 on graphs of different size. As can be seen, the algorithm is extremely fast: it finds the optimum in the case of a graph with 10000 vertices and 20000 edges in less than 0.3 seconds. Moreover, the practical running time of the algorithm seems to be roughly linear.

5.2 Experience with Algorithm 2

Since Algorithm 2 is a heuristic rather than an exact algorithm, we had to determine both its performance and the quality of the solutions it finds, empirically. For this purpose, we compared it with two other heuristics that are widely used for hardware/software partitioning: a genetic algorithm (GA) and an improved Kernighan/Lin–type heuristic (KL). In the case of the GA it is important to tune its parameters to match the characteristics of the problem domain. Details on this can be found in [Arató et al. 2003].

In the case of the KL algorithm, we built on the improvements suggested by [Vahid and Le 1997]. Specifically, [Vahid and Le 1997] defined the following changes: (i) redefined a move as a single node move, rather than a swap; (ii) described an efficient data structure; (iii) replaced the cut metric of the original KL heuristic by a

Name	n	m	Size	Description
crc32	25	34	152	32-bit cyclic redundancy check. From the Telecommunications category of MiBench.
patricia	21	50	192	Routine to insert values into Patricia tries, which are used to store routing tables. From the Network category of MiBench.
dijkstra	26	71	265	Computes shortest paths in a graph. From the Network category of MiBench.
clustering	150	333	1299	Image segmentation algorithm in a medical application.
rc6	329	448	2002	RC6 cryptographic algorithm.
random1	1000	1000	5000	Random graph.
random2	1000	2000	8000	Random graph.
random3	1000	3000	11000	Random graph.
random4	1500	1500	7500	Random graph.
random5	1500	3000	12000	Random graph.
random6	1500	4500	16500	Random graph.
random7	2000	2000	10000	Random graph.
random8	2000	4000	16000	Random graph.
random9	2000	6000	22000	Random graph.

Table II. Summary of the used benchmarks

more complex metric. We made use of these changes, with the only difference that our cost function is slightly different from theirs:

$$cost(P) = \begin{cases} \infty & \text{if } R_P > R_0 \\ H_P & \text{otherwise} \end{cases}$$

Note that [Vahid and Le 1997] used the DAG property of their graph representation to show that a move has only local effect, which is important for the performance of the algorithm. This is also true in our case: by moving a node from hardware to software or vice versa, only the gain value of its neighbors can change.

For testing, we used benchmarks from MiBench [Guthaus et al. 1997], our own designs, as well as bigger, random graphs. The characteristics of the test cases are summarized in Table II. n and m denote the number of nodes and edges, respectively, in the communication graph. *Size* denotes the length of the description of the graph (the performance of an algorithm is usually evaluated as a function of the length of the input). We calculated the size as $2n + 3e$ because each node is assigned two values—its hardware and software costs—and each edge is assigned three numbers—the IDs of its endpoints and its communication cost.

Where software costs were not available, they were generated as uniform random numbers from the interval [1, 100]. Where hardware costs were not available, they were generated as random numbers from a normal distribution with expected value $\kappa \cdot s_i$ and standard deviation $\lambda \cdot \kappa \cdot s_i$, where s_i is the software cost of the given node. That is, there is a correlation, as defined by the value of λ, between a node's hardware and software costs. This corresponds to the fact that more complicated components tend to have both higher software and higher hardware costs. We tested two different values for λ: 0.1 (high correlation) and 0.6 (low correlation). The value

of κ only corresponds to the choice of units for software and hardware costs, and thus it has no algorithmic implications. The communication costs were generated as uniform random numbers from the interval $[0, 2 \cdot \mu \cdot s_{max}]$, where s_{max} is the highest software cost. Thus, communication costs have an expected value of $\mu \cdot s_{max}$, and μ is the so-called communication to computation ratio (CCR). We tested two different values for μ: 1 (computation-intensive case) and 10 (communication-intensive case). Finally, the limit R_0 was taken from the interval $[0, \sum s_i]$. Note that $R_0 = 0$ means that all components have to be mapped to hardware, whereas $R_0 = \sum s_i$ means that all components can be mapped to software. All sensible values of R_0 lie between these two extremes. We tested two values for R_0: one generated as a uniform random number from the interval $[0, \frac{1}{2} \sum s_i]$ (strict real-time constraint) and one taken randomly from $[\frac{1}{2} \sum s_i, \sum s_i]$ (loose real-time constraint).

So we tested the three algorithms on the above set of problems, using two values for each of the three parameters (correlation between hardware and software costs, CCR, R_0). However, we found that the correlation between hardware and software costs did not have any significant impact on the performance of the algorithms. Therefore we include four plots, according to the combinations of the two remaining parameters, on the quality of the solutions found by the algorithms (see Figure 3). Since the objective was to minimize costs, smaller values are better. The lower bounds produced by Algorithm 2 are also shown. Based on the diagrams, the following observations can be made:

—For relatively small graphs, all three heuristics yield equal or very similar results, regardless of the parameter settings. Moreover, these results are very close to the lower bound computed by Algorithm 2, meaning that they are at least near-optimal.

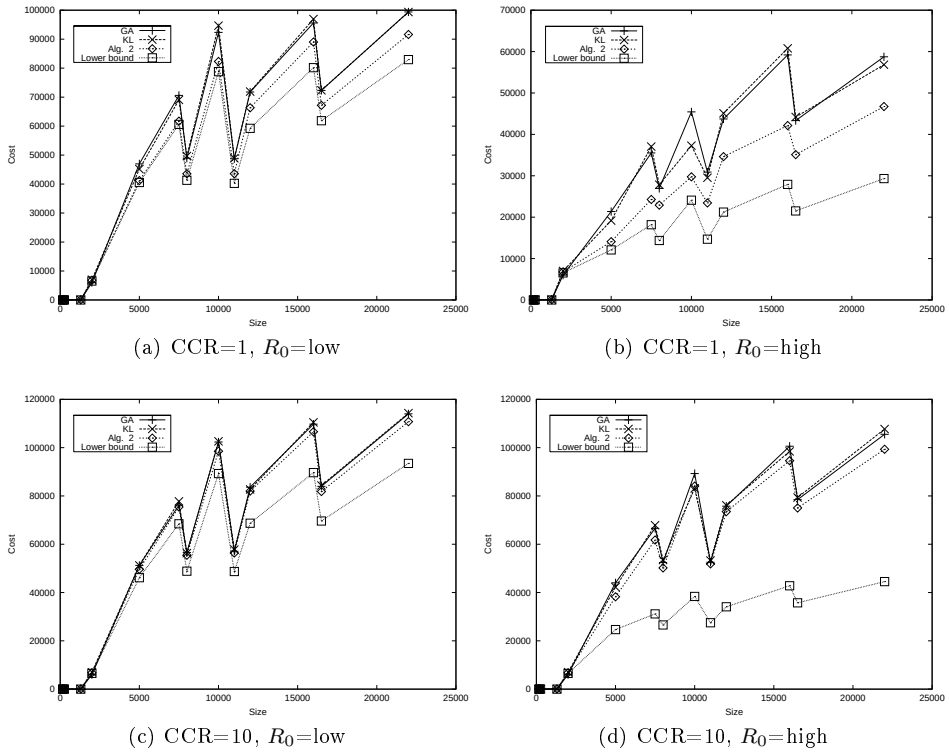
—For bigger graphs, Algorithm 2 consistently outperforms the other two heuristics. This is especially true in the low-CCR cases. In the high-CCR cases, the difference between the algorithms is not so striking. This is probably due to the easier nature of these problem instances (note that with growing CCR, the partitioning problem becomes essentially a simple minimum cut problem with polynomial complexity). Moreover, the difference between the results of Algorithm 2 and the other two heuristics is clearly growing.

—The results found by GA and KL are very similar, but in most cases the GA is slightly better.

—The results found by Algorithm 2 are in most cases not very far from the lower bounds it produced—the difference was 31% on average. Of course the difference keeps growing with bigger graphs, but quite slowly. This proves the high quality of both the solutions and the lower bounds found by our algorithm.

—The choice of the R_0 parameter does not seem to significantly impact the relative performance of the algorithms. However, the lower bounds produced by Algorithm 2 do seem to be sensitive to this parameter: they are clearly better for low R_0 values.

An even bigger difference between the three algorithms is their running time, which is shown in Figure 4 (Here, only one plot is shown because the explained parameter settings do not have a significant impact on the running times). We can



(a) CCR=1, R_0=low (b) CCR=1, R_0=high

(c) CCR=10, R_0=low (d) CCR=10, R_0=high

Fig. 3. Algorithm 2 vs. GA and KL: quality of found solution

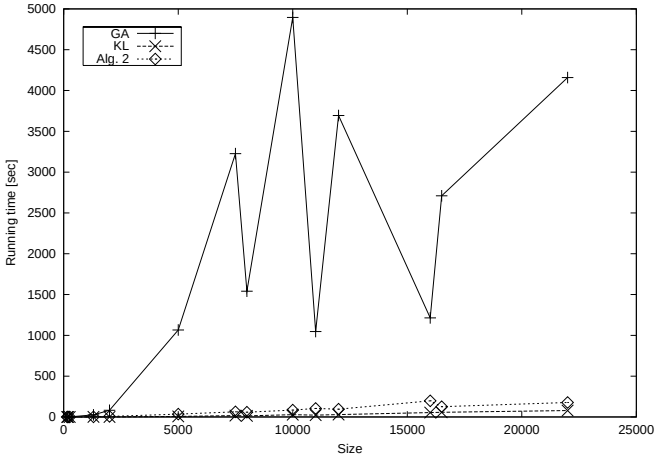


Fig. 4. Algorithm 2 vs. GA and KL: running time

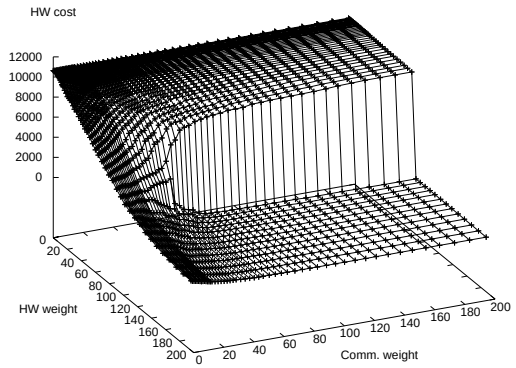


Fig. 5. Hardware cost of the optimal partition in the P1 problem as the function of the weights α and γ

observe the following:

—Again, for relatively small graphs, the speed of the algorithms is comparable. However, for bigger graphs, KL and Algorithm 2 are much faster than GA, and again, the difference keeps growing with bigger graphs. For the biggest graphs, GA is about 20 times slower than Algorithm 2.

—The running time of GA oscillates wildly. In some cases, it took over an hour for the GA to terminate. However, even the shortest GA runs were much slower than the other two algorithms.

—The fastest of the three is clearly the KL algorithm: for small graphs, it is about 5 times faster than Algorithm 2, but for bigger graphs the difference decreases. For the biggest graphs, KL was about 2.5 times faster than Algorithm 2.

—The speed of both KL and Algorithm 2 is acceptable because both could solve even the biggest problems in 2-3 minutes, and the smaller ones in a couple of seconds.

Another question that we addressed empirically is whether or not the two-dimensional search approach of Algorithm 2 is adequate.

The above results show that the algorithm performs very well on large benchmarks. This can be attributed to the smoothness of the costs as functions of the weights α, β, and γ. An example can be seen in Figure 5 showing the hardware cost of the optimal partition in the P1 problem for different values of the hardware weight α and communication weight γ (the software weight β was fixed to 100). Notice the smoothness and the simple structure of this function. Actually, it can be proven that this function is monotonously decreasing in α.

In some test cases we also ran a modified version of Algorithm 2 in which the two-dimensional search space is searched uniformly in small steps, without augmenting $d\alpha$ and $d\gamma$. The test results showed no improvement in the results; however, the speed of the algorithm worsened significantly. This justifies the search strategy of

Algorithm 2.

To sum up: Algorithm 2 offers a clear advantage over the other two heuristics concerning the quality of the found solution. It is at the same time significantly faster than GA, and somewhat slower than KL, but still fast enough to be applicable in practice. Moreover, it produces high-quality lower bounds.

6. CONCLUSION

In this paper, we defined two slightly different versions of the hardware/software partitioning problem (P1 and P2). We proved that the P1 problem can be solved in polynomial time, but the P2 problem is $\mathcal{NP}$-hard. The polynomial-time algorithm for the P1 problem (Algorithm 1) makes use of the graph-theoretic properties of the hardware/software partitioning problem. It has a worst-case running time of $O(n^3)$ steps, but our empirical experiments showed that on practical examples it is very fast.

Based on this algorithm, we also proposed a new heuristic for the P2 problem (Algorithm 2) which works by running Algorithm 1 with several different weights to obtain high-quality candidate partitions, from which it chooses the best one satisfying the given constraint.

Algorithm 2 possesses the unique feature that it can calculate lower bounds for the optimum solution and hence it can evaluate how far its currently found best solution lies from the optimum.

In our empirical tests on several benchmarks we compared Algorithm 2 with two established partitioners: a genetic algorithm and an improved Kernighan/Lin–type algorithm. We found that our algorithm consistently outperformed the other two heuristics, while being slightly slower than KL and significantly faster than GA. We attribute the good scalability of our algorithm to the fact that it only evaluates high-quality points of the search space (only those that are optimal solutions of the P1 problem for some weights) and hence it makes better use of the combinatorial properties of the search problem.

Generalization of our algorithms for multi-way partitioning and proving or disproving approximation bounds for Algorithm 2 remain interesting future research directions.

APPENDIX

A. $\mathcal{NP}$–HARDNESS RESULTS

THEOREM A.1. *The P2 problem is $\mathcal{NP}$-hard.*

PROOF. The proof can be found in [Mann and Orbán 2003]. □

However, the P2 problem is $\mathcal{NP}$–hard in the *strong sense* as well, i.e. even if the vertex and edge costs have to be polynomial in n. In the following we show a reduction of the MINIMUM BISECTION problem to P2.

THEOREM A.2. *The P2 problem is $\mathcal{NP}$-hard in the strong sense.*

PROOF. We reduce the decision version of the MINIMUM BISECTION problem, which is known to be $\mathcal{NP}$-complete [Garey and Johnson 1979], to P2.

Given an instance of the minimum bisection problem on $G(V, E)$ with n vertices, where n is even, m edges and a limit K, our goal is to find a cut (A, B), for which $|A| = |B| = \frac{n}{2}$ and the cutsize is at most K ($K \leq m$).

Now associate the following instance of the P2 problem to it. Let $h(v_i) = s(v_i) = 1$ for each $v_i \in V$ and let $c(v_i, v_j) = \frac{1}{m+1}$ for each $(i, j) \in E$. Define $R_0 := \frac{n}{2} + \frac{K}{m+1}$. Clearly this instance has polynomial costs in n.

For $X, Y \subseteq V$ we denote by $m(X, Y)$ the number of edges between X and Y and by $c(X, Y)$ the total cost of edges between X and Y.

We claim that there exists a feasible bisection iff the optimum for the P2 problem is at most $\frac{n}{2}$. Indeed, if (A, B) is a solution for the bisection problem ($|A| = |B| = \frac{n}{2}$ and $m(A, B) \leq K$), then (A, B) is also a feasible solution for P2, since $s(B) + c(A, B) = |B| + \frac{1}{m+1} m(A, B) \leq \frac{n}{2} + \frac{K}{m+1} = R_0$. The hardware cost of (A, B) in P2 is $h(A) = |A| = \frac{n}{2}$, thus the optimum is at most $\frac{n}{2}$.

Vice versa, if in the optimal partition (V_H, V_S) of P2 the hardware cost is at most $\frac{n}{2}$, then $h(V_H) = |V_H| \leq \frac{n}{2}$ and $s(V_S) + c(V_H, V_S) \leq \frac{n}{2} + \frac{K}{m+1} < \frac{n}{2} + 1$, thus $s(V_S) = |V_S| \leq \frac{n}{2}$, as it is an integer and c is non–negative. As both sides of the partition (V_H, V_S) are not larger than $\frac{n}{2}$, $|V_H| = |V_S| = \frac{n}{2}$ must hold. This also implies—using again the condition for the running time—that $c(V_H, V_S) \leq \frac{K}{m+1}$, hence $m(V_H, V_S) \leq K$. So (V_H, V_S) is indeed a solution for the bisection problem as well. □

B. LOWER BOUND

THEOREM B.1. *Suppose that P is an optimal solution of the P1 problem with the weights α, β, and γ. Let Q be any solution of the P2 problem (i.e. a partition that satisfies the bound $R_Q \leq R_0$). Then*

$$H_Q \geq H_P + \frac{\beta S_P + \gamma C_P - \max(\beta, \gamma) R_0}{\alpha} \tag{2}$$

PROOF. Since P is optimal with respect to the weights α, β, and γ, it follows that

$$\alpha H_P + \beta S_P + \gamma C_P \leq \alpha H_Q + \beta S_Q + \gamma C_Q$$

and hence

$$H_Q \geq H_P + \frac{\beta S_P + \gamma C_P - \beta S_Q - \gamma C_Q}{\alpha} \tag{3}$$

Of course, this is also a lower bound on H_Q, but the right-hand side cannot be computed because S_Q and C_Q are not known. However, since Q is a valid partition, it follows that

$$S_Q + C_Q = R_Q \leq R_0$$

and therefore

$$\beta S_Q + \gamma C_Q \leq \max(\beta, \gamma) S_Q + \max(\beta, \gamma) C_Q = \max(\beta, \gamma) R_Q \leq \max(\beta, \gamma) R_0$$

Substituting this into (3) proves the theorem. □

REFERENCES

ABDELZAHER, T. F. AND SHIN, K. G. 2000. Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers 49*, 1, 81–87.

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall.

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

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

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

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

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

Cherkassky, B. V. and Goldberg, A. V. 1997. On implementing push-relabel method for the maximum flow problem. *Algorithmica 19,* 4, 390–410.

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

Dick, R. P. and Jha, N. K. 1998. 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.

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

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

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

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

Garey, M. R. and Johnson, D. S. 1979. *A guide to the theory of NP–completeness.* Freeman, San Francisco.

Goldberg, A. V. and Tarjan, R. E. 1988. A new approach to the maximum flow problem. *Journal of the ACM 35,* 921–940.

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

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

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

Henkel, J. and Ernst, R. 2001. 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.

Kalavade, A. 1995. System-level codesign of mixed hardware-software systems. Ph.D. thesis, University of California, Berkeley, CA.

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

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

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

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

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

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

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

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

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

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

Niemann, R. and Marwedel, P. 1997. 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.

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

Qin, S. and He, J. 2000. An algebraic approach to hardware/software partitioning. Tech. Rep. 206, UNU/IIST.

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

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

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

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

Stone, H. 1977. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering 3,* 1 (Jan), 85–93.

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

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

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

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

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

Received Month Year; revised Month Year; accepted Month Year