MŰEGYETEM 1782

Partitioning Algorithms for Hardware/Software Co-Design

— Ph.D. dissertation —

Zoltán Ádám Mann

Supervisor: Professor Péter Arató

Budapest University of Technology and Economics
Department of Control Engineering and Information Technology

2004

Abstract

Hardware/software co-design (HSCD) is the discipline of automating the design of complex embedded systems with functionality in both hardware and software. The central task of HSCD is hardware/software partitioning, which aims at deciding which components of the system to implement in hardware and which ones in software. During partitioning, the conflicting requirements on performance, costs, energy consumption, etc., have to be taken into account.

The dissertation addresses the hardware/software partitioning problem from an algorithmic point of view. It proposes a novel formal framework for the definition of different variants of the problem. This is important because different variants of the problem can have very different complexity. Indeed, it is shown in the dissertation that although most variants are $\mathcal{NP}$-hard, there are also important cases that can be solved optimally in polynomial time. This was previously not known in the HSCD community.

Furthermore, the dissertation proposes three new heuristics for the $\mathcal{NP}$-hard versions of the problem. The first is a genetic algorithm that is significantly more effective than previous genetic approaches to hardware/software partitioning. The basis for this outstanding performance is the consistent handling of invalid individuals, i.e., partitions that violate some design constraint.

The second algorithm is an adaptation of the Kernighan–Lin (KL) heuristic which was originally developed for a different graph partitioning problem. The adaptation also involved transferring improvement possibilities that have been proposed for the original KL algorithm to this new domain, as well as addressing the problem of choosing an appropriate data structure for the efficient implementation of the algorithm.

While the first two algorithms are innovative ways of using previously known paradigms, the third one is based on a completely new view on the partitioning problem, making use of its combinatorial properties. It works by optimally solving a set of related, yet easier problems, thus generating candidate partitions from which the best is then selected. Beside excellent performance, an unprecedented advantage of this algorithm is that it can also generate lower estimates on the optimum of the problem.

It is demonstrated with empirical results how the three algorithms can cope with industrial benchmarks as well as large random problem instances. The strengths and weaknesses of the algorithms are investigated, which makes it possible to decide which one to use in a given situation. This vital question was previously not addressed in the HSCD literature. It is also presented how the new heuristics have already been applied in real-world projects, and how they can be used to create faster, cheaper, smaller, more power-efficient, and more reliable embedded computer systems.

Contents

Chapter 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 and the advent of ubiquitous computing, size, heat dissipation, and energy consumption are also becoming crucial aspects for a wide range of computer systems, especially embedded systems.

Embedded systems have become a part of our lives in the form of consumer electronics, car electronics, mobile phones, smart cards etc. These computer systems consist of both hardware and software; they together determine the operation of the system. Moreover, the differences between hardware and software as well as their interaction contribute significantly to the huge complexity of the systems.

To take into account all of these aspects in the design process is becoming next to impossible. According to the International Technology Roadmap for Semiconductors [30], the most crucially challenged branch of the computer industry is system design. The Roadmap clearly declares that Moore's law can hold on for the next decades only if innovative new ways of system design will be proposed to handle the growing complexity.

To address specifically the problem of designing complex mixed hardware/software systems, a considerable amount of research has taken place in the field of *hardware/software co-design* (HSCD[1]) since the early 1990s [40, 33, 41, 51, 62, 93, 70, 94]. The main objectives of HSCD are the exploitation of synergies between hardware and software, computer-aided exploration of the design space, and automatic optimization of key parameters of the designed system, such as performance and cost.

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 a significant amount of functionality of embedded systems, both a hardware implementation and a software implementation is possible. In such cases, the two implementation options

[1]The used abbreviations and notations are summarized in Appendix A.

typically have complementary advantages and disadvantages, as outlined in Table 1.1. This is the reason why it is beneficial to use both hardware and software components to implement a given system: performance-critical or power-critical components of the system should be implemented in hardware, whereas non-critical components in software. This way, an optimal trade-off can be found between performance, power, and costs.

HSCD is also facilitated by the tendency that several vendors offer customizable hardware boards (such as field-programmable gate arrays) with integrated processor cores [81].

	Hardware	Software
Speed	faster	slower
Production cost	more costly	cheaper
Energy consumption	lower	higher
Heat dissipation	lower	higher
Maintainability	hard	easier

Table 1.1: Typical advantages and disadvantages of hardware vs. software solutions

However, hardware/software co-design also poses significant challenges. The most important is *partitioning*, i.e., deciding which components of the system should be implemented in hardware and which ones in software. This is clearly the step in which the above-mentioned optimal trade-off between the conflicting requirements should be found. Unfortunately, finding 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.

Furthermore, the communication overhead between hardware and software also has to be taken into account during partitioning. Namely, if one of two communicating components is implemented in hardware, and the other one in software, then the communication between them incurs a significant overhead. If they are both in hardware or both in software, then the overhead is typically much lower. (An exception is when the two components are both implemented in software, but reside on different processors. In this case, the communication overhead is not negligible in terms of time. However, even in this case, other costs associated with the communication between components—such as implementation effort or occupied chip area of the communication logic—are significantly lower than in the case of communication between hardware and software.)

For this reason, the system to be partitioned is usually modeled with a graph, the nodes of which represent the components of the system, and the edges represent communication, calls, or dependencies between the components. Furthermore, the nodes and edges are assigned several cost values, such as execution time, size requirement, delay etc. It is also an important task of HSCD to obtain these cost values.

The main steps of a HSCD framework are outlined in Figure 1.1. As can be seen, the HSCD process starts from a high-level, yet formal system specification. Usually, it is beneficial to start from an executable specification, because this facilitates the estimation

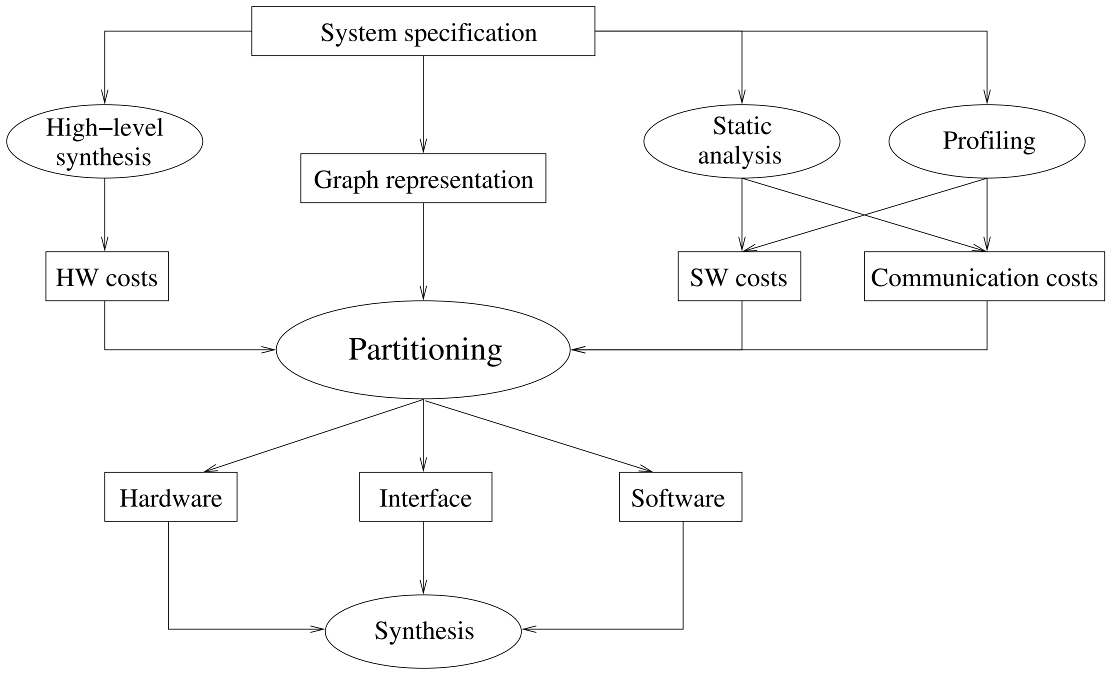


Figure 1.1: Block diagram of a possible hardware/software co-design framework

of the cost values. Therefore, a high-level programming language—e.g., C or Matlab—is often used for this purpose.

In order to perform partitioning, first the appropriate graph representation has to be extracted from the system specification. It should be noted at this point that there are several possibilities for the *granularity* of this graph representation. The nodes of the graph can represent low-level entities of the system, such as single instructions of the original executable specification, but much higher levels are also possible, e.g., the nodes can represent functions or procedures [44].

Beside extracting the graph, also the required cost values have to be determined for each node and edge. This task is by no means trivial; in fact, the computation of each cost metric is a challenge on its own. For example, hardware-related costs (hardware production costs, occupied chip area, etc.) are often computed using the techniques of high-level synthesis [14, 23, 35, 7, 65, 12]. Software-related costs on the other hand are often calculated using static analysis or measured by means of profiling [72, 75, 47, 64].

Based on all these input data, the partitioning algorithm decides which nodes to put into hardware and which ones into software. At the end of the HSCD flow, a synthesis phase is responsible for synthesizing the hardware part using, e.g., commercial hardware synthesis tools, the software part using standard compiler tools, and the arising interfaces between the two parts. The interfaces have to be generated on both sides, i.e., the software part has to be extended with appropriate device drivers, interrupt handlers etc., and the hardware part has to be extended with the appropriate communication logic [18, 88]. This final synthesis step is also an important research field on its own, as well as the validation and verification of the system [86].

In this work, the focus is solely on the algorithmic aspects of the central phase of HSCD, i.e., on partitioning. This is motivated by the enormous importance of partitioning: better partitioning algorithms enable the design of faster, more economical, and more power-efficient computer systems. Of course, partitioning is always a part of a HSCD framework and interacts with other parts of it. A possible way of integrating the partitioning algorithms presented in this dissertation into a HSCD framework is described in [11].

1.1 Previous work

Traditionally, partitioning was carried out manually, and even today, manual partitioning is often applied [76, 67, 58]. However, as the systems to be designed have become more and more complex, this method has become inappropriate, because it is very time-consuming and typically it leads to suboptimal results. For this reason, many research efforts have been undertaken to automate partitioning as much as possible.

Concerning the exact problem definition, there are significant differences between the suggested partitioning approaches. One of the main differences is whether partitioning only means deciding which components of the system to map to hardware and which ones to software, or it includes other tasks as well. In particular, many researchers consider scheduling (i.e., determining starting times for the components) as part of partitioning [24, 29, 52, 61, 69, 71], whereas others do not [31, 39, 62, 72, 92, 90]. Some formulations of the hardware/software partitioning problem even include the problem of assigning communication events to physical links between hardware and/or software units [29, 69].

Furthermore, in a number of related papers, the target architecture is supposed to consist of a single software and a single hardware unit [31, 39, 41, 44, 61, 62, 69, 72, 76, 82, 83, 92], whereas others do not impose this limitation. Some exclude parallelism inside hardware or software [82, 92] or between hardware and software [44, 62].

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.

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 [15, 17, 76, 48], middle granularity, where a node represents a basic block [50, 56, 75], and high granularity, where a node represents a function or procedure [2, 41, 71, 91], as well as flexible granularity, where a node can represent any of the above [44, 90].

When looking at the algorithms that have been suggested for hardware/software partitioning, one can differentiate between exact and heuristic methods. The proposed exact algorithms include branch-and-bound [21], dynamic programming [62, 72], and integer linear programming [66, 70, 71].

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 [51, 66], and the exact algorithms for them have exponential runtimes.

Many researchers applied general-purpose heuristics to hardware/software partitioning. In particular, genetic algorithms have been extensively used [6, 29, 69, 77, 82], as well as simulated annealing [32, 33, 44, 59]. Other, less popular heuristics in this group are tabu search [32] and greedy algorithms [24, 39].

Some researchers used custom heuristics to solve hardware/software partitioning. This includes the GCLP algorithm [52, 53] and the expert system of [60, 61], as well as the heuristics in [41] and [93].

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, 16, 90, 91]. The other group of partitioning-related heuristics was pioneered by the Kernighan-Lin algorithm [54], which was substantially improved by Fiduccia and Mattheyses [34], and later by many others [27, 78, 43, 57, 79, 46, 95]. These heuristics have been found to be appropriate for hardware/software partitioning as well [61, 89, 92]. More details on this topic can be found in Section 4.2.

1.2 Research goals

The aim of my research was to investigate the hardware/software partitioning problem from an algorithmic point of view. This consists of two main tasks:

- Investigation of the algorithmic complexity of the problem. It was claimed by several researchers that partitioning is $\mathcal{NP}$-complete, but only one particular formulation of the problem has been proven to be really $\mathcal{NP}$-hard. However, that formulation also included the scheduling problem, and the proof also depended on the hardness of scheduling. On the other hand, the literature also includes several other formulations of the partitioning problem that do not include scheduling, and the complexity of these problem formulations was not proven. For this reason, one of my aims was to investigate the complexity of different formulations of the partitioning problem, possibly finding polynomially solvable cases, or to formally prove $\mathcal{NP}$-hardness.

- The other main goal concerned the $\mathcal{NP}$-hard versions of the problem, since—as revealed by my investigations—several formulations of the problem are indeed $\mathcal{NP}$-hard even without scheduling. Nonetheless, these problems have to be solved somehow in practice. Therefore, I developed efficient heuristics for them. In contrast to most previous work in the field, the primary aim was not to include as many details of partitioning as possible in the algorithms, but rather to enhance efficiency for a more limited problem.

  A related task is to test, tune, and compare the heuristics on benchmark problems. It is important to determine the effectiveness and speed characteristics of the algorithms, because this is the basis for deciding which one to use in a given situation.

This vital question was previously not addressed explicitly in the HSCD literature. As it turns out, there is no clear winner from the algorithms: depending on the characteristics of a given problem instance (such as the ratio of communication costs to other costs or how tight the constraints are) it varies which one is the best algorithm.

The practical benefit of the new, enhanced partitioning heuristics is manifold, both for the design process and the resulting product. In the design process, the designers can work at a higher level of abstraction, thus enabling them to focus on the most important and most innovative aspects of the functionality of their designs. Non-functional requirements can be specified in a declarative way, and are guaranteed automatically, thus freeing the designers from tedious and stereotypical tasks. This leads to a shorter and cheaper design cycle, which is a vital business advantage.

Concerning the resulting product, the usage of a high-quality partitioning algorithm guarantees high performance and reasonable costs at the same time. Automatic partitioning typically also implies higher dependability because the design process is less error-prone. Furthermore, it extends the boundaries of the functionality that can be implemented with a given technology. All of this implies—beyond the business advantage—also a considerable positive impact on society.

One of the main challenges of the research was its interdisciplinary nature: it included methods from computer engineering, computer science, operational research, graph theory, and heuristic optimization.

1.3 Problem formulation

The model that will be used throughout this work was suggested by our research group [6, 10]. It only focuses on partitioning, and does not include scheduling and other tasks of a HSCD environment. We believe that decoupling the specific tasks of HSCD enables us to develop more powerful algorithms. Of course, this requires some simplification in the problem formulation and thus results in a loss of precision because the partitioning algorithm has only an estimate of the cost metrics. Conversely, the complexity of the problem is drastically reduced, and thus a bigger percentage of the search space can be searched. This way, similar or even better results are achieved than by considering all aspects together but scanning only a small fraction of the huge search space.

In our 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.[2] 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

[2]This is a significant advantage of this model, since acyclicity can be hardly guaranteed in real-world task graphs.

versa), then their communication incurs a communication penalty, the value of which is given for each edge as an *edge cost* (also called *communication 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, since on most architectures inter-context communication is orders of magnitude more costly than intra-context communication.

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 metric, 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 even memory blocks). However, we assume that the total hardware cost with respect to a given 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 (i.e., the cut edges).

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 [62].

The problem can now be formalized as follows. A simple undirected graph $G = (V, E)$, $V = \{v_1, \dots, v_n\}$, as well as the cost functions $s, h : V \to \mathbb{R}^+$ 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 $V_H \cap V_S = \emptyset$. (Note that $V_H = \emptyset$ or $V_S = \emptyset$ is also possible.) The set of cut edges with respect to partition $P = (V_H, V_S)$ 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 typically conflicting cost metrics that should be optimized together. There are several ways to formulate a well-defined optimization or decision problem based on these three cost metrics. In the following, five different problems are defined; each of them is an important design problem in its own right.[3]

In the first three versions, two of the cost metrics are added. A possible interpretation is 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 $R_P = S_P + C_P$. Now it is possible to constrain both R_P and H_P, or constrain one of them and minimize the other one. In the first version of the problem—which will be denoted by **P1**—there is both a real-time constraint, i.e., a constraint on R_P, and a hardware constraint, i.e., a constraint on H_P. This is a decision problem, i.e., it has to be decided if a partition satisfying both constraints exists. All other versions are optimization problems.

In the second version (called **P2**) H_P is constrained and the aim is to minimize R_P. In the third version (**P3**) 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.

In the fourth version (**P4**), S_P and C_P are not added. Rather, there is a constraint on S_P and a constraint on H_P and the aim is to minimize C_P while satisfying the two constraints.[4] A possible interpretation is the following. Hardware cost is the occupied area, which is constrained by the available chip size. Software cost is code size, which is constrained by available memory—on some architectures, e.g., in smart cards, the size of the memory can be the most stringent constraint on the software part [20]. Communication cost is implementation effort, which should be minimized (while obeying the two constraints), because implementing the necessary wrappers to enable communication between hardware and software is a tedious task.

In the fifth version (**P5**), the aim of partitioning is to minimize the weighted sum of the three cost metrics. The weights are specified by the designer, and define the relative importance of the three metrics. More formally, the *total cost* of P is defined 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.

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

[3]These problem definitions are summarized in Appendix C.1.

[4]It is assumed in this case that there are valid partitions, i.e., partitions that satisfy both constraints. It should be noted that in the case of **P2** and **P3** the analogous assumption is always true because the all-software partition is always valid for **P2** and the all-hardware partition is always valid for **P3**. Thus no extra assumption was needed in those versions of the problem.

P1 Given the graph G with the cost functions h, s, and c, and $R_0 \geq 0$, $H_0 \geq 0$, decide whether there is a hardware/software partition P with $R_P \leq R_0$ and $H_P \leq H_0$.

P2 Given the graph G with the cost functions h, s, and c, and $H_0 \geq 0$, find a hardware/software partition P with $H_P \leq H_0$ that minimizes R_P among all such partitions.

P3 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.

P4 Given the graph G with the cost functions h, s, and c, and $S_0 \geq 0$, $H_0 \geq 0$, such that there are hardware/software partitions with $S_P \leq S_0$ and $H_P \leq H_0$, find a partition P with $S_P \leq S_0$ and $H_P \leq H_0$ that minimizes C_P among all such partitions.

P5 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.

Sometimes, the following generalization of the problem will be considered (which can be defined for any of the above five versions): for some nodes it is prescribed which context they should be mapped to (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, it has to be assumed not only in the case of **P4**, but also in the case of **P2** and **P3** that there are valid partitions, because this is not guaranteed anymore.

At some places, the following notations will also be used. For a set $X \subseteq V$ let $s(X) = \sum_{v \in X} s(v)$ and similarly, $h(X) = \sum_{v \in X} h(v)$. Let $c(X, V \setminus X)$ denote the sum of the costs of the cut edges belonging to the partition $(X, V \setminus X)$.

1.4 About this dissertation

This work is organized as follows. First, the algorithmic complexity of the above problems is investigated (Chapter 2, Thesis 1). In particular, it will be shown that the first four versions of the partitioning problem are $\mathcal{NP}$-hard, whereas the fifth version can be solved in polynomial time. The possibilities for approximation algorithms will also be discussed.

In the remainder of the dissertation, three heuristic algorithms are proposed for the **P3** version of the partitioning problem. (These algorithms can also easily be adapted to the other versions of the problem.) In particular, Chapter 3 presents a genetic algorithm (Thesis 2), Chapter 4 describes an algorithm based on the Kernighan-Lin heuristic for graph bisection (Thesis 3), and a combinatorial algorithm, called the MFMC-based algorithm (Thesis 4), is presented in Chapter 5. Afterwards, the three heuristics are compared both from an analytical point of view and empirically on a number of benchmark problems

(Chapter 6). Finally, Chapter 7 presents some practical applications of the proposed algorithms.

Each chapter starts with a short summary of the thesis that the chapter is about.

Chapter 2

Complexity results

The contributions of this chapter can be summarized as follows:

Thesis 1: I proved that problem **P1** is $\mathcal{NP}$-complete, and problems **P2** and **P3** are $\mathcal{NP}$-hard. Problem **P4** is also $\mathcal{NP}$-hard, moreover, I also proved that if $\mathcal{P} \neq \mathcal{NP}$, then even no approximation algorithm can exist for **P4**. However, I gave a polynomial-time exact algorithm for the **P5** problem.

These results can be categorized into three groups: $\mathcal{NP}$-hardness results, which are presented in Section 2.1, polynomiality results, presented in Section 2.2, and issues related to approximation algorithms, presented in Section 2.3. Appendix B summarizes the used basic notions of complexity theory.

2.1 $\mathcal{NP}$-hardness results

Most natural formulations of the hardware/software partitioning problem are $\mathcal{NP}$-hard. At least, this was claimed by several researchers [21, 31, 51, 91], but only one particular formulation of the problem has been proven to be really $\mathcal{NP}$-hard [51]. However, that formulation also included the scheduling problem, and the proof also depended on the hardness of scheduling.

Moreover, as will be demonstrated in Section 2.2, the **P5** problem, which is also a natural formulation of the hardware/software partitioning problem, can be solved in polynomial time, and thus it is not $\mathcal{NP}$-hard, unless $\mathcal{P} = \mathcal{NP}$. Therefore it is not true that all formulations of the partitioning problem are necessarily $\mathcal{NP}$-hard. In fact, the complexity of those formulations that do not include scheduling is not clear. This shows the importance of the following theorem. It is the first $\mathcal{NP}$-hardness theorem for a formulation of the partitioning problem that does not include scheduling.

Theorem 1. *The **P1** problem is $\mathcal{NP}$-complete even if only graphs with no edges are considered.*

Proof. **P1** is in $\mathcal{NP}$, because if partitioning is possible with the given limits, then such a partition is a good witness for this.

To prove the $\mathcal{NP}$-hardness, the known $\mathcal{NP}$-complete Knapsack[1] problem [74] will be reduced to **P1**. Let an instance of the Knapsack problem be given. (There are n objects, the weights of the objects are denoted by w_i, the price of the objects by p_i, the weight limit by W and the price limit by K. The task is to decide whether there is a subset X of objects, so that $\sum_{i\in X} w_i \le W$ and $\sum_{i\in X} p_i \ge K$.) Based on this, an instance of **P1** can be defined as follows: $V=\{v_1,\ldots,v_n\}$, $E=\emptyset$. Let $h_i=p_i$, $s_i=w_i$ for each $i=1,\ldots,n$. (Since E is empty, there is no need to define c.) Introducing $A=\sum_{v_i\in V} p_i$, let $R_0=W$ and $H_0=A-K$.

Now it should be proven that this instance of **P1** is solvable if and only if the original Knapsack problem has a solution.

Assume first that this instance of **P1** has a solution $P=(V_H,V_S)$, where $V_H\cup V_S=V$ and $V_H\cap V_S=\emptyset$. This means that

$$R_P=\sum_{v_i\in V_S} w_i \le R_0 = W \tag{2.1}$$

and

$$H_P=\sum_{v_i\in V_H} p_i \le H_0 = A-K=\sum_{v_i\in V} p_i - K.$$

The last equation can also be formulated as:

$$K\le \sum_{v_i\in V} p_i - \sum_{v_i\in V_H} p_i = \sum_{v_i\in V_S} p_i. \tag{2.2}$$

Equations (2.1) and (2.2) prove that $X=V_S$ is a solution of the original Knapsack problem.

Now assume that X solves the original Knapsack problem. Therefore:

$$\sum_{v_i\in X} s_i = \sum_{v_i\in X} w_i \le W = R_0 \tag{2.3}$$

and

$$\sum_{v_i\in X} p_i \ge K = A - H_0 = \sum_{v_i\in V} p_i - H_0$$

that is

$$H_0\ge \sum_{v_i\in V} p_i - \sum_{v_i\in X} p_i = \sum_{v_i\in V\setminus X} p_i = \sum_{v_i\in V\setminus X} h_i. \tag{2.4}$$

Equations (2.3) and (2.4) verify that $P:=(V\setminus X, X)$ solves **P1**. □

This result easily implies the hardness of the **P2** and **P3** problems as well:

[1]See Appendix C.2 for details.

Theorem 2. ***P2*** *and* ***P3*** *are $\mathcal{NP}$-hard*[2] *even if only graphs with no edges are considered.*

Proof. **P1** can be reduced to **P2**: the solution of **P2** is a partition where $H_P \leq H_0$ and R_P is minimal; let this value be R_P^*. Clearly **P1** is solvable if and only if $R_P^* \leq R_0$.

P1 can be reduced to **P3**: the solution of **P3** is a partition where $R_P \leq R_0$ and H_P is minimal; let this value be H_P^*. Clearly **P1** is solvable if and only if $H_P^* \leq H_0$. □

In Section 2.3, it will be proven that the **P4** problem is also $\mathcal{NP}$-hard.

The proofs of the above two theorems show that, in the special case when the graph has no edges, the **P1** problem is equivalent to the decision version of the Knapsack problem, whereas the **P2** and **P3** problems are equivalent to its optimization version.

Although the Knapsack problem is $\mathcal{NP}$-hard, its complexity arises only because the numbers in it (the weights, prices, and limits) can be exponentially large. In fact, there exist pseudo-polynomial algorithms for the Knapsack problem, i.e., algorithms whose running time is polynomial in the numbers that appear in the problem instance. Hence, if the numbers that appear in the problem instance are polynomial in the size of the input, then the problem can be solved in polynomial time [74]. Thus, the same applies to hardware/software partitioning as well if communication costs can be neglected.

In the following it will be proven that in the general case—i.e., when the graph also has edges—the partitioning problem is even harder than the Knapsack problem. Specifically, it will be shown that it is $\mathcal{NP}$-hard in the strong sense, which means that it is $\mathcal{NP}$-hard even if only polynomial-size numbers are allowed. This will imply in particular that not even a pseudo-polynomial algorithm can exist for partitioning unless $\mathcal{P} = \mathcal{NP}$. Also note that this is the first such theorem for hardware/software partitioning.

Theorem 3. *The* ***P1*** *problem is $\mathcal{NP}$-complete in the strong sense.*

Proof. It has already been proven in Theorem 1 that **P1** is in $\mathcal{NP}$.

Now the decision version of the Minimum Bisection problem, which is known to be $\mathcal{NP}$-complete [37], will be reduced to **P1**.

An instance of the Minimum Bisection problem consists of a graph $G = (V, E)$ with n vertices, where n is even, m edges and a limit K (with $K \leq m$), and the goal is to find a cut (A, B), for which $|A| = |B| = \frac{n}{2}$ and the number of cut edges is at most K.

Now associate the following instance of the **P1** problem to it. The graph will be the same. 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 $R_0 := \frac{n}{2} + \frac{K}{m+1}$ and $H_0 := \frac{n}{2}$. Clearly all cost values appearing in this instance of **P1** are polynomial in n.

For $X, Y \subseteq V$, the set of edges between X and Y will be denoted by $E(X, Y)$.

Now it has to be proven that there exists a feasible bisection if and only if this instance of **P1** is solvable. Indeed, if (A, B) is a solution for the bisection problem (i.e., $|A| = |B| = \frac{n}{2}$

[2]**P1** is $\mathcal{NP}$-complete, but **P2** and **P3** are 'only' $\mathcal{NP}$-hard, because **P2** and **P3** are not decision problems and thus they are not in $\mathcal{NP}$.

and $|E(A,B)| \le K$), then (A,B) is also a solution for **P1**, since

$$\sum_{v\in B} s(v) + \sum_{e\in E(A,B)} c(e) = |B| + \frac{1}{m+1}|E(A,B)| \le \frac{n}{2} + \frac{K}{m+1} = R_0,$$

and

$$\sum_{v\in A} h(v) = |A| = \frac{n}{2} = H_0.$$

Vice versa, let us assume that the **P1** instance has a solution (V_H, V_S). Then

$$|V_H| = \sum_{v\in V_H} h(v) \le H_0 = \frac{n}{2} \tag{2.5}$$

and

$$\sum_{v\in V_S} s(v) + \sum_{e\in E(V_H,V_S)} c(e) \le R_0 = \frac{n}{2} + \frac{K}{m+1} < \frac{n}{2} + 1, \tag{2.6}$$

thus

$$|V_S| = \sum_{v\in V_S} s(v) \le \frac{n}{2}, \tag{2.7}$$

since it is an integer and c is non-negative. As both sides of the partition (V_H, V_S) are not larger than $\frac{n}{2}$ (expressed by (2.5) and (2.7)), $|V_H| = |V_S| = \frac{n}{2}$ must hold. This also implies—using again (2.6)—that $\sum_{e\in E(V_H,V_S)} c(e) \le \frac{K}{m+1}$, hence $|E(V_H, V_S)| \le K$. So (V_H, V_S) is indeed a solution for the bisection problem as well. □

Using the same reductions as in the proof of Theorem 2, Theorem 3 implies the following result:

Theorem 4. *The **P2** and **P3** problems are $\mathcal{NP}$-hard in the strong sense.* □

Concerning the complexity of the **P1**, **P2**, and **P3** problems, Theorem 3 and Theorem 4 are stronger results than Theorem 1 and Theorem 2. The reason why Theorem 1 and Theorem 2 have been presented at all, is that they show the connection of partitioning and the Knapsack problem. That is, in the special case when communication can be neglected and the numbers appearing in the problem instance are not too big, the problem can be solved efficiently using the algorithms for the Knapsack problem.

2.2 Polynomiality results

First it should be noted that the other extreme, i.e., when communication is the only significant cost, and hardware and software costs can be neglected, is easy, even if the cost values can be arbitrarily large. If no other restriction is made, then—regardless of which version of the problem is considered—mapping every node to hardware, or mapping every node to software are optimal solutions, because their cost is 0.

Now consider the generalization in which some nodes are prescribed to be in software and some others in hardware. More formally, let $\overline{V_S}$ denote the set of nodes that are prescribed to be in software, and $\overline{V_H}$ the set of nodes that are prescribed to be in hardware, where $\overline{V_H} \cap \overline{V_S} = \emptyset$, and at least one of $\overline{V_H}$ and $\overline{V_S}$ is not empty. In this case, the partitioning problem reduces to finding a minimum $s-t$ cut, or a minimum cut [4] in a modified graph, as shown below, and can thus be solved in polynomial time. This holds for all of the problems **P1**–**P5**.

First, let us consider the case when both $\overline{V_H}$ and $\overline{V_S}$ are non-empty. Then the graph should be modified as follows: $\overline{V_H}$ is coalesced to a single vertex v_h and $\overline{V_S}$ is coalesced to a vertex v_s. 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. After these modifications, the minimum-cost partition can be determined by finding the minimum cut between v_s and v_h.

If, on the other hand, either $\overline{V_H}$ or $\overline{V_S}$ is empty, then one can proceed as follows. Suppose for instance that $\overline{V_S} = \emptyset$. Then, $\overline{V_H}$ is again coalesced to a single vertex v_h, as above (parallel edges and loops are handled in the same way as above). However, the minimum-cost partition can now be determined by finding the globally minimum cut in this modified graph. This, too, can be solved in polynomial time. Thus, the following theorem has been proven:

Theorem 5. *If $\forall v \in V : h(v) = s(v) = 0$, then all the problems **P1**–**P5** can be solved in polynomial time, even if some nodes are prescribed to be in hardware and some others are prescribed to be in software.* □

Moreover, the time complexity of these algorithms is essentially the same as that of finding the minimum cut, or the minimum $s-t$ cut in a graph. The best known algorithms for finding the minimum cut in a graph have complexity of $\tilde{O}(n^2)$, and $\tilde{O}(nm)$ for finding the minimum $s-t$ cut [26].

Now I leave the topic of efficiently solvable special cases, and go on to show that the **P5** problem is solvable in polynomial time even in the general case.

Theorem 6. *The **P5** problem can be solved optimally in polynomial time.*

Proof. It can be assumed that $\alpha = \beta = \gamma = 1$ because otherwise each h_i is multiplied by α, each s_i by β, and each c_{ij} by γ.

With this modification the problem becomes similar to the one solved by Stone in [84]. Although Stone handles only one cost metric (time) and not the linear combination of several cost metrics, the proof of this theorem is identical to [84]. The details are omitted, only the main idea of the construction is given to help understand the algorithm that will be based on it.

An auxiliary graph (see Figure 2.1) $G' = (V', E')$ is constructed 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.

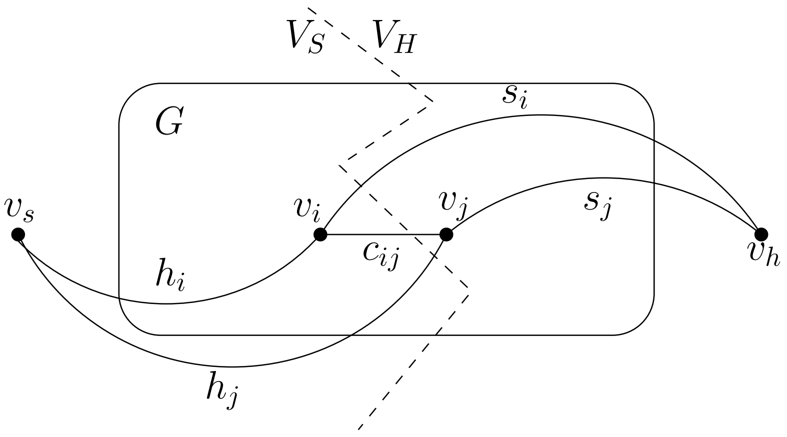


Figure 2.1: The auxiliary graph

Lemma 1 (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.* □

By Lemma 1, **P5** can be reduced to finding a minimum cut between two vertices in a simple undirected graph, for which polynomial-time algorithms are known [4]. 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. □

The above proof suggests a polynomial-time algorithm for the **P5** problem, as summarized in Algorithm 1.

Algorithm 1 Polynomial-time algorithm for the **P5** problem

```
1. Create the auxiliary graph.
2. Find a minimum cut between $v_s$ and $v_h$ in the auxiliary graph.
```

Clearly, the first step of the algorithm can be performed in linear time. For the second step, many algorithms are known. For my implementation, I used the max-flow-min-cut (MFMC) algorithm of Goldberg and Tarjan [38, 25], 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.[3] Note that $O(n^3)$ is just a theoretic upper bound for the runtime of Algorithm 1. Empirical experience has shown that the algorithm is extremely fast in practice (see Section 5.6.2).

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 [68].

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 5.7).

It should also be noted that the algorithm can easily accommodate the extension to the partitioning model, in which some components are fixed to software, while some others are fixed to hardware. In this case, the same method is used as in the proof of Theorem 6 with the sole difference that the vertices v_s and v_h need not be added to the graph, but rather they arise by coalescing the nodes that are prescribed to be in software, and by coalescing the nodes that are prescribed to be in hardware, respectively (just like in the proof of Theorem 5).

Finally, I would like to contrast Theorem 6 to the results of Section 2.1. It can be seen that—supposed that $\mathcal{P} \neq \mathcal{NP}$—the **P5** problem is significantly easier than the other problems. (It has not yet been proven that the **P4** problem is also $\mathcal{NP}$-hard, but this is also true, and it will be proven in Section 2.3.) 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 these theorems is that not all formulations of the partitioning problem are necessarily $\mathcal{NP}$-hard. The **P5** 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. Notwithstanding, in the rest of this work the main focus will be on the $\mathcal{NP}$-hard versions of the partitioning problem.

2.3 Approximability

Since **P2**, **P3**, and **P4** are $\mathcal{NP}$-hard optimization problems, it is a natural question whether they can be at least approximated in polynomial time. A *κ-approximation algorithm* (where $\kappa > 1$) for a minimization problem is a polynomial-time algorithm whose result is guaranteed to be at most κ times the optimum [45]. The question is whether there exist

[3]When Stone published his similar approach in [84], 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.

approximation algorithms for these problems. In the case of **P4**, the answer is negative if $\mathcal{P} \neq \mathcal{NP}$:

Theorem 7. *The* ***P4*** *problem is $\mathcal{NP}$-hard. Furthermore, if $\mathcal{P} \neq \mathcal{NP}$, then no approximation algorithm can exist for* ***P4****.*

Proof. According to Theorem 1, the **P1** problem is $\mathcal{NP}$-hard even in the special case when graphs with no edges are considered. Now, a Karp-reduction will be presented from this special case of **P1** onto **P4**. It will follow immediately that **P4** is $\mathcal{NP}$-hard. Moreover, the Karp-reduction will be such that the second claim will also easily follow.

Let an instance of **P1** with no edges be given. That is, it has to be decided whether the vertices can be partitioned in such a way that $\sum_{v_i \in V_S} s_i \leq R_0$ and $\sum_{v_i \in V_H} h_i \leq H_0$. Based on this, an instance of **P4** is constructed. The graph will be the same, except that two new nodes and an edge are added. The two new nodes will be called x and y, and the new edge will be called e. The two endpoints of e are x and y, and its communication cost is 1. The costs of node x are chosen in such a way that x will surely be in software: $s(x) = 0$ and $h(x)$ is a large number which is greater than the hardware limit. In the case of y: $h(y) = 0$ and $s(y) = M$, where M is a positive number with the following property:

$$R_0 + M \geq \sum_{v_i \in V} s_i. \tag{2.8}$$

The costs of the other nodes are unchanged. Finally, the limits in the newly created problem instance are $H_0' = H_0$ and $S_0' = R_0 + M$.

Now, it has to be proven that these limits are admissible in the sense that there is a partition that fulfills both limits. Indeed, according to Inequality (2.8), putting all nodes but y to software is such a valid partition.

Recall that in the case of **P4**, the aim is to minimize C_P, while obeying the two constraints. Clearly, the minimum of C_P in the newly created problem instance is either 0 or 1, depending on whether edge e is cut or not. Now it has to be proven that the original problem instance is solvable if and only if the optimal C_P is 0. Suppose first that the original problem instance is solvable. Then place the nodes in the new problem instance in the same way as they are placed in the solution of the original problem. According to the definition of S_0', there is enough room left on the software side to place node y into software as well, and thus e is not cut, $C_P = 0$.

On the other hand, if the optimal C_P is 0, then in an optimal solution y has to be in software, and hence, only R_0 room is left for the other nodes in software, and of course H_0 room is available on the hardware side. Therefore, the remaining nodes build a solution to the original problem. Thus, the above construction is indeed a Karp-reduction, and so **P4** is $\mathcal{NP}$-hard.

It remains to show that if $\mathcal{P} \neq \mathcal{NP}$, then no approximation algorithm can exist for **P4**. If there were a κ-approximation algorithm for **P4** with some $\kappa > 1$, then this algorithm would yield a result of 0 if the optimum is 0, and a result between 1 and κ if the optimum is 1. Thus, solving the newly created problem instance with this algorithm would tell us whether the original problem was solvable or not. Since the original problem was $\mathcal{NP}$-hard, this is not possible in polynomial time, unless $\mathcal{P} = \mathcal{NP}$. $\square$

Concerning the other two problems, it is still an open question whether there exist constant-factor approximation algorithms for them: no such algorithm is known, but a hardness result (in the sense of Theorem 7) is not known either. However, an $O(\log^2 n)$-approximation algorithm has been presented for the **P2** problem in [73]. Unfortunately, this algorithm has little practical relevance because of its high approximation factor and its high complexity.

In order to illustrate the hardness of approximating these problems, I will prove a simple negative result: that the greedy algorithm is not an approximation algorithm for the **P3** problem.[4] By greedy algorithm, the following is meant: it starts from the all-hardware partition, which is certainly valid in the case of **P3**. In each step, it checks which nodes can be moved from hardware to software without hurting the constraint on R_P, and moves the node with highest hardware cost from these. It stops when no more moves are possible.

Proposition 1. *The result of the greedy algorithm can be arbitrarily far from the optimum of the **P3** problem.*

Proof. Let G be a complete graph on n vertices. For each vertex v, let $s(v) = h(v) = 1$, and for each edge e, let $c(e) = 2$. Let $R_0 = n$. Then it is clear that the optimal solution is to put every node to software. The optimum is thus 0.

On the other hand, the greedy algorithm starts with all nodes in hardware, and it cannot move any node to software. This is because moving a node to software would lead to $R_P = 1 + 2(n-1) = 2n - 1 > R_0$. Therefore the algorithm stops without making a move, and its result is n. $\square$

2.4 Summary

To sum up the results of this chapter: **P5** is solvable optimally in polynomial time, but all other problems are $\mathcal{NP}$-hard in general. Moreover, no practically usable approximation algorithms are known for our $\mathcal{NP}$-hard optimization problems, and at least in the case of **P4**, it could be proven that no such approximation algorithm can be hoped for.

For these reasons, the rest of this work considers the usage of heuristics.

[4]Note that several optimization problems can be approximated with greedy algorithms, e.g., binpacking and many formulations of the scheduling problem [45]. This result shows that approximating hardware/software partitioning is not that easy. Furthermore, this result will be important in Chapter 4.

Chapter 3

Genetic algorithm

In this and the next two chapters, three different heuristic algorithms are presented for the **P3** problem. The contribution of this chapter can be summarized as follows:

Thesis 2: I developed a genetic algorithm for hardware/software partitioning. It is different from previous genetic algorithms in the following respects:

- The population can also contain individuals that violate the constraints; however, they are punished by the fitness function. This way, genetic operations can be applied without restriction, which enables more efficient optimization.
- I developed and tested different methods to make sure that the initial population contains valid individuals, but does not lack diversity.
- I implemented and tested several methods for cross-over and several different fitness functions.

3.1 GA in general

Since there are very good books and surveys on genetic algorithms (see, for instance, [28, 55] and references therein), I will only give here a very brief, rather practical introduction, and clarify notations and terminology.

In order to implement a genetic algorithm, one has to do the following. First, an initial population of individuals representing some possible solutions of the problem must be set up somehow. After that, in each iteration a new population is generated from the previous one using the genetic operations: recombination, mutation, and selection (the latter is sometimes called elitism). So in each step there are two populations. The new population is first partially filled using recombination, then the rest using selection. Mutation is then used on some individuals of the new population. See Algorithm 2. The order of the operations is important because this way individuals created by recombination and mutation surely get into the new population, even if they are not very fit, so they have the chance to propagate their possibly good properties. Note that it is also possible to define further genetic operations, but these are the most common ones, and they suffice for the aims of this chapter.

Algorithm 2 Skeleton of the GA

```
1. Create initial population
2. Calculate the fitness of each individual
3. Perform $recombination\_rate \cdot size\_of\_population$ recombinations, and insert the offsprings into the new population
4. Fill the remaining positions of the new population using selection
5. Perform $mutation\_rate \cdot size\_of\_population$ mutations in the new population
6. If stopping criterion met, stop
7. Replace the old population with the new one
8. Goto 2.
```

Mutation is important because it guarantees diversity and thus helps leaving local optima. Its implementation is simple: a randomly chosen gene of a randomly chosen individual is altered randomly. Of course there can be variants: one can alter one gene at a time or maybe more genes at once, one can allow only slight modifications of a gene or any modification etc.

The aim of recombination is to mix good properties of the individuals so that a sequence of genes corresponding to a good property can spread across the whole population. It is implemented by generating one or more (usually two) new individuals by the crossover of two individuals. Individuals are usually not chosen with uniform probability for recombination (as for mutation for instance), since it is more probable to get good genes from individuals having high fitness values.

Selection is also necessary, it is used to propagate good individuals and to eliminate bad ones. It is implemented by simply copying the best individuals into the new population. In some variants it is not necessarily the best ones who survive, but they are selected with a higher probability.

In most cases, individuals are represented as vectors of real numbers (sometimes integers or Boolean values). This way, an individual $x = (x_1, \dots, x_n)$ can be regarded as an element of the Euclidean space $\mathbb{R}^n$. If not every $x \in \mathbb{R}^n$ is a valid individual (i.e., there are constraints on the individuals), then the actual *solution space* is a subset $\mathcal{T} \subset \mathbb{R}^n$. Thus, a population can be formally defined as a finite subset $Pop \subset \mathcal{T}$. The number of individuals in a population ($|Pop|$) is usually a fixed number N. The fitness function is $f : \mathbb{R}^n \to \mathbb{R}$, but it usually suffices if $D_f = \mathcal{T}$. On the other hand, some versions of GA require that $R_f \subseteq \mathbb{R}^+$.

3.2 Application to the partitioning problem

In this section I present the way that the above general scheme can be used for the partitioning problem.

3.2.1 Individuals

The partitioning problem is fortunate from the point of view of a genetic algorithm. The applicability of genetic algorithms requires that the solutions of the optimization problem can be represented by means of a vector with meaningful components: this is the condition for recombination to work on the actual features of a solution.

Fortunately, there is an obvious vector representation in the case of the partitioning problem: each partition can be encoded as a bit vector of length n. The ith bit is 1 if v_i should be in hardware and 0 if it should be in software.

3.2.2 Population

The population is a set of individuals. The question is whether non-valid individuals, i.e., those violating the real-time constraint in the case of **P3**, should also be allowed in the population. Since non-valid individuals violate an important design constraint, it seems to be logical at first glance to work with valid partitions only. However, this approach would have several drawbacks: first, invalid individuals may contain valuable patterns that should be propagated, and second, it is hard to guarantee that genetic operations do not generate non-valid individuals even from valid ones. This holds for both mutation and recombination [12, 80, 87, 29]. For these reasons, I decided to allow non-valid individuals as well in the population. Of course the GA must produce a valid partition at the end, so it is important to insert some valid individuals into the initial population, and choose the fitness function in such a way that it punishes invalidity. More details on these topics are given in Section 3.2.3 and Section 3.2.4.

The size of the population, denoted by N, is an important parameter of the GA. Typically, increasing N leads to better results; however, at the cost of increased running time. Moreover, it is often the case that after a certain limit increasing N does not improve the results significantly anymore (i.e., it only increases the running time). Therefore, an appropriate trade-off between running time and result quality has to be determined empirically. This will be presented—along with the tuning of other parameters—in Section 3.3.

3.2.3 Initial population

In order to guarantee diversity in the population, the initial population usually consists of randomly chosen individuals. However, this method does not guarantee that there will be valid individuals in the initial population—in fact, if the problem space is big and constraints are tight, then the chances are very low for this. Thus it is possible that the algorithm will not find a valid partition at all. Therefore it has to be explicitly guaranteed that the initial population contains valid individuals as well.

In the case of **P3**, at least one valid partition is known: the all-hardware solution. Usually there are other valid partitions as well, but they are not known in advance. On the other hand, it would be a good idea to insert several different valid individuals into the initial population, so that optimization starts with many different good patterns. To this end, it would be beneficial if random valid individuals could be generated with approximately uniform distribution. Unfortunately, the set of valid solutions can have a very complex structure, and thus it is by no means clear how one could implement such a scheme.

Therefore I chose to use the following compromise: one part of the initial population is filled with randomly selected, not necessarily valid individuals, and the other part with valid individuals generated by a fast greedy algorithm. This way, there are valid individuals, but also a wide variety of other individuals in the initial population, from which the GA will then hopefully be able to mix valid individuals with high quality.

Clearly, the ratio between the two kinds of individuals in the initial population is a crucial parameter of the GA. This, too, had to be tuned based on empirical experience (see Section 3.3).

The algorithm responsible for generating the valid individuals for the initial population should fulfill the following requirements:

- It should produce a valid partition.
- It should be much faster than the GA itself since it is only a pre-optimization step that, on the other hand, will be run several times.
- It should be a randomized algorithm so that different runs may produce different solutions.

For these reasons, I implemented a randomized greedy algorithm which starts from the all-hardware solution, and moves in each step a randomly chosen node from hardware to software as long as this is possible without corrupting the real-time constraint (see Algorithm 3).

Algorithm 3 Generation of valid individuals for the initial population

```
1. Put all nodes to hardware.
2. Compute the set of nodes $X$ that can be moved from hardware to software without corrupting the real-time constraint.
3. If $X = \emptyset$, then exit and return the current partition.
4. Choose a vertex randomly from $X$ and move it to software.
5. goto 2.
```

3.2.4 Fitness function

Since I focused on the **P3** problem, the objective is to minimize hardware cost. Thus, a first possibility for the fitness function would be the following:

$$f(P) = \Theta - H_P.$$

Here, Θ is a sufficiently large constant, so that the fitness will always be positive (which is beneficial for the implementation). For instance, $\Theta = 1 + \sum_{v \in V} h(v)$ is a good choice. This is only needed because the fitness should be maximized, whereas H_P should be minimized.

However, the fitness function should punish invalid individuals. One possibility would be the following:

$$f(P) = \begin{cases} \Theta - H_P & \text{if } P \text{ is valid} \\ 0 & \text{otherwise.} \end{cases}$$

The reason why I chose a more sophisticated fitness function is that invalid individuals should be motivated to become 'less invalid,' i.e., those invalid individuals that almost fulfill the real-time constraint should have a higher fitness than those that are far from it. Therefore, I introduce a measure of invalidity, denoted by $exc(P)$, defined as the percentage by which the running time of the system exceeds the real-time constraint, i.e., $exc(P) = R_P/R_0$. Moreover, the fitness of an invalid individual should also be dependent on its hardware cost, since an invalid individual with a low hardware cost is more likely to contain valuable patterns than an invalid individual with a high hardware cost. Based on these insights, the fitness function is defined as follows:

$$f(P) = \begin{cases} \Theta' - H_P & \text{if } P \text{ is valid} \\ \Theta' - H_P - c * exc(P) - M & \text{if } P \text{ is invalid.} \end{cases}$$

Again, the role of Θ' is to make the fitness positive. However, Θ' must be bigger for this purpose than Θ was. The role of c is to define the relative importance of H_P and $exc(P)$ for invalid individuals. That is, it answers the question of 'how much decrease in H_P can compensate for a unit increase in $exc(P)$?' The term M denotes a base penalty for every invalid individual. Based on the value of M, two very different versions of the fitness function can be defined: a rigorous one, which ranks every valid individual in front of all invalid ones, and the less rigorous kind, which allows invalid individuals to beat valid ones. Again, it is up to the tests to decide which version works best in practice.

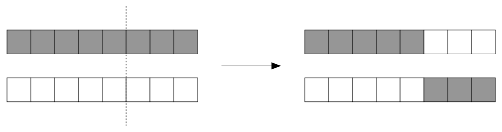
Instead of combining H_P and R_P into a single fitness function, it would also have been possible to construct a multi-objective genetic algorithm. This approach has already been suggested in the literature [29]. It has to be noted though that this is useful only if the constraints are not known in advance and the aim is to explore the solution space with the goal of finding several Pareto-optimal solutions instead of one optimum. However, in the problem formulations addressed in this dissertation, the constraints are given, and hence there is no point in performing a multi-objective search.

3.2.5 Genetic operations

Mutation, recombination, and selection are used.

Mutation is done in the new population; each gene of each individual is altered with the same probability. The number of mutations per generation (called mutation rate) is another parameter for testing.

In the case of recombination I implemented and tested both one-point crossover and two-point crossover (see Figure 3.1). In both cases, two parents are recombinated to yield


(a) One-point crossover

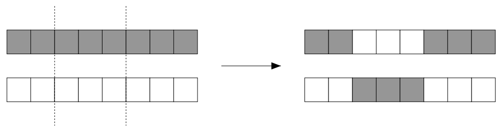

(b) Two-point crossover

Figure 3.1: Recombination of two individuals

two children. The point(s) at which the chromosomes are cut is (are) chosen randomly. Moreover, I implemented two different schemes for choosing the individuals for crossover: in the first one, all individuals are chosen with the same probability, in the second one, the probability of choosing a particular individual depends on its fitness, so that better individuals are chosen with a higher probability. So I tested four different versions of the recombination operation.

Selection is usually realized as filling some part of the new population with the best individuals of the old population. However, since the rigorous version of the fitness function ranks all invalid individuals behind the valid ones, this would mean that invalid individuals may be completely omitted from selection. Therefore, a given ratio of the selected individuals is taken from the invalid individuals, so that potential valuable patterns that are present in the invalid individuals will not be lost.

Both the ratio of selected individuals and the ratio of selected invalid individuals has to be determined empirically.

3.2.6 Stopping criterion

Typically, the best fitness of the population steeply increases in the first generations, but the improvement becomes much slower in the long run. In particular, after a certain number of steps, there is hardly any noticeable improvement anymore. The stopping criterion should be determined in such a way that the algorithm stops when there is no improvement anymore. However, this does not happen after a fix number of generations, and thus the stopping criterion should be *adaptive*.

In my implementation, the GA takes a given minimum number of steps (generations). After that, it stops after x steps if the best found partition did not improve in the last $\vartheta \cdot x$ steps. $0 < \vartheta < 1$ is also a parameter to test, just like the minimum number of steps.

3.3 Empirical evaluation

As can be seen from the above, the GA has several parameters that have a significant impact on its efficiency and performance. The best settings for these parameters had to be determined using empirical tests on practical problem instances. The parameters that had to be tested are the following:

- Type of recombination (4 versions)
- The fitness function (rigorous vs. non-rigorous, the value of M in the non-rigorous case,[1] the value of c)
- The size of the population
- The ratio of valid individuals in the initial population
- Mutation rate
- Recombination rate
- Ratio of selected invalid individuals
- Minimum number of generations, and ϑ

For each parameter, at least 3-4 different values had to be tried. The high number of parameters made it intractable to test all configurations thoroughly, since each parameter configuration should be tested on a number of benchmarks, with several different random seeds. Rather, I first fixed all parameters to a plausible value, and then tuned them one after the other, or at least in smaller, independent groups. For instance, the size of the population can be tuned independently from the mutation and recombination rates. Moreover, first the less sensitive parameters were tuned. Unfortunately, it is not easy to decide which parameters are independent and how sensitive they are. I based my decisions mainly on logic and previous experience with genetic algorithms.

For testing, I used the set of benchmarks presented in Section 6.2.1. Since in many cases the best choice of the parameters was not equivocal (i.e., it is possible that parameter configuration A works better on some benchmarks than parameter configuration B, but B works better on other benchmarks), I used the average of all measurement results to evaluate the quality of a given parameter configuration.

In order to conserve space, I omit the details of these experiments, only one example is shown in Figure 3.2. This figure shows how the average result depends on the ratio of selected individuals (i.e., the ratio of those individuals in the new population that are copied without any change, and not generated by means of recombination). On the x axis, different values for the selection rate are marked. The y axis shows the corresponding average normalized cost. Thus, lower values are better. It can be seen that the minimum of the curve is at around 0.1. Moreover, the curve hardly changes in the neighborhood of 0.1, hence there is no point in trying to locate the best choice more accurately. Thus, the selection rate can be fixed to 0.1, which means that the recombination rate should be 0.45 (because 90% of the population has to be generated by recombination, and each recombination yields two offsprings).

Using this methodology, I made the following findings:

[1] In the rigorous case, i.e., if M is sufficiently large, the exact value of M does not matter anymore.

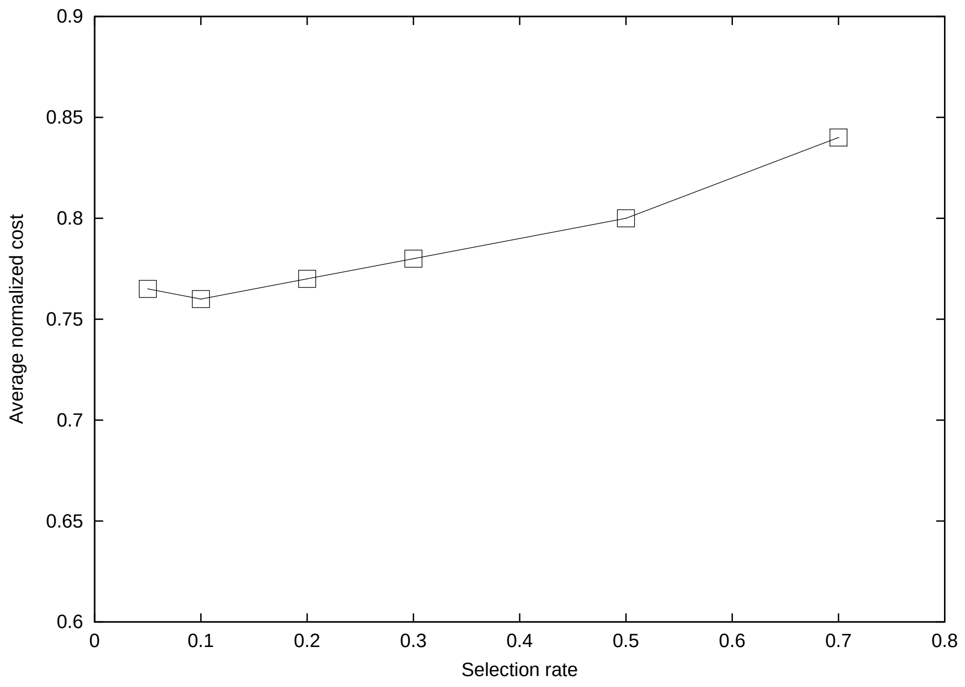


Figure 3.2: The effect of the selection rate on the cost of the found solution

- Recombination: those versions that select the individuals to recombinate based on their fitness gave clearly better results than those that do not. The difference between one-point and two-point crossover was very small: two-point crossover proved to be slightly better.
- Fitness function: the rigorous version clearly won.
- Size of the population: more than 300 was not useful in most test cases, thus I fixed it to 300.
- Ratio of valid individuals in the initial population: the best choice was around 0.7. This means that the individuals generated by the randomized greedy algorithm (Algorithm 3 on page 26) have a very important role in the operation of the GA.
- The best mutation rate was around 0.75%. This means that in a population of 300 individuals, about 2-3 mutations per generation take place on average.
- The best recombination rate was around 45%, which means that 90% of the population should be generated by recombination and 10% by selection (see above).
- Of all the selected individuals, about 80% should be chosen from the invalid ones. This is quite unexpected: it shows that a large percentage of the best patterns can be found in the invalid individuals with relatively high fitness.

- Stopping criteria: just like the size of the population, this setting also directly affects the trade-off between speed and result quality. The following values proved to be best: minimum number of generations: 500, $\vartheta = 0.4$.

After these tests, I fixed all parameters to their best values. Note that Chapter 6 presents a more detailed practical evaluation and a comparison between the GA and other partitioning algorithms. During that comparison, all parameters are fixed to their best values.

Finally, it should be noted that the whole tuning process was quite tedious, consuming more effort than the implementation of the algorithm. In recent years, it has emerged as an interesting research field on its own how tuning of heuristics can be automated or at least supported by computer software. Some promising approaches include the usage of function approximators [22] and experimental design [85, 3]. Hopefully that research will produce robust and reliable tools for the tuning of parameters of heuristics in a couple of years, but now the widely practiced methodology is still manual tuning.

3.4 Discussion

Several other genetic algorithms have already been suggested in the literature for hardware/software partitioning. The novelty of my GA lies in the consistent handling of invalid individuals. In particular:

- Invalid individuals are also allowed to participate in the population. In contrast, it would also be possible to allow only valid individuals, and whenever an invalid individual is generated by a genetic operation, it would be deleted. I also implemented and tested this scheme, but it gave much worse results: the population became degenerated after a few generations, and no further improvement took place.

- The fitness function is chosen in such a way that it punishes invalidity. This idea was used in previous works as well; however, no thorough evaluation of the different choices for the fitness function had been carried out. I tested several different settings for the fitness function, and experienced significant impact on solution quality and convergence speed. Thus I improved the efficiency of the algorithm significantly by taking the most successful version of the fitness function.

- Extra care is taken to include many different valid individuals in the initial population without sacrificing diversity. This is a feature that none of the previously proposed GAs possessed. However, my experiments showed clearly that this is very important. (Recall that 70% of the initial population should be valid!) In particular, if the initial population does not contain valid individuals, then it is possible that the whole GA does not find a valid partition. Both the way that the valid individuals are generated for the initial population and their ratio plays a significant role.

- The efficiency of the selection operation is significantly improved by the fact that the best invalid individuals are selected separately for the next generation. This is again a unique feature of this GA. The empirical tests have shown that this feature alone accounts for an about 15% improvement in result quality.

Finally, it should be pointed out that the presented algorithm can be extended to accommodate several generalizations of the problem:

- Some nodes can be prescribed to be in hardware, some other in software. The algorithm can be trivially extended.
- Several different cost metrics and constraints can be included. (For instance in component-based hardware/software co-design [9] it is common to define more than one *use case* of the system, and associate a separate real-time constraint with each use case.) Only the fitness function has to be extended accordingly.
- Partitioning into more than two contexts can be achieved. In this case, the genes would be k-valued instead of binary, for some $k > 2$.
- As discussed in Section 1.1, some partitioning approaches also include scheduling. Scheduling can be incorporated into the GA in two different ways:
  - When calculating the fitness of an individual, an external scheduling algorithm may be used to compute the R_P value for the given individual.
  - The GA itself can optimize the schedule as well. In this case, the chromosomes are extended to also include an encoding of the schedule.

Chapter 4

Kernighan-Lin-type algorithm

The contribution of this chapter can be summarized as follows:

Thesis 3: I developed a hardware/software partitioning heuristic based on the iterative improvement principle suggested by Kernighan and Lin. When compared to previous work presented in the literature, the most important novelties of my approach are the following:

- I investigated in detail how the numerous improvement possibilities that had been suggested for the original Kernighan-Lin algorithm can be adapted to hardware/software partitioning. In particular, I evaluated different tie-breaking strategies, locking schemes, and ways to generate the initial partition.
- I investigated how different models for hardware/software partitioning affect the applicability of a Kernighan-Lin-type algorithm.
- I provided an efficient implementation for the algorithm based on the range tree data structure.

4.1 Motivation

According to Vahid [92], one of the most promising directions in the research on hardware/software partitioning is the application of the Kernighan-Lin (KL) heuristic. Originally, this algorithm was developed for a formulation of the circuit partitioning problem [54]. Its aim is to partition a graph into two parts of equal size with a minimal number of cutting edges. It is a so-called iterative improvement algorithm, meaning that it starts from an arbitrary partition, and swaps pairs of nodes in order to improve the cost of the partition. The reason of the success of the KL heuristic is that it is fast as a greedy algorithm, but it can escape some local optima.

Since its inception, several improvements have been suggested to the KL heuristic. The most widely known is the work of Fiduccia and Mattheyses [34]. They presented—

among others—a powerful data structure to enable a linear-time implementation of their algorithm, which I will refer to as the FM algorithm. Other works investigated tie-breaking strategies and different locking schemes to enhance the efficiency of the algorithm. The KL algorithm along with these improvement possibilities is reviewed in Section 4.2.1.

The application of the KL heuristic in the context of hardware/software partitioning was suggested by Vahid [89, 92]. He extended the KL algorithm so that it optimizes an execution-time metric instead of the original cut metric. Although his algorithm achieved promising results, it did not use the full potential of the KL heuristic; in particular, it did not make use of the many improvements that are known for KL. He also tried to devise an efficient implementation by using a similar data structure as the one that had been suggested in the FM algorithm; however, with partial success only. For more details, see Section 4.2.2.

My aim was to remedy these shortcomings. More specifically, I present a KL-type algorithm for hardware/software partitioning with the following merits when compared to previous work in this field:

- Careful adaptation of the suggested improvements to the KL algorithm in the context of hardware/software partitioning
- Investigation of how different formulations of the hardware/software partitioning problem influence the applicability and efficiency of the KL heuristic
- Efficient implementation of the KL algorithm for hardware/software partitioning by making use of a more advanced data structure.

More specifically, I proceed as follows. After reviewing previous work in Section 4.2, I start with the description of the challenges that are associated with the adaptation of the KL heuristic to hardware/software partitioning (Section 4.3).

The skeleton of my algorithm is described in Section 4.4. Afterwards, the details are presented systematically. In particular, I investigate how the different cost metrics can be joined to a single gain value for each node; as it turns out, this choice has important implications on the efficiency of the whole algorithm (Section 4.5). I also discuss different strategies for generating the initial partition (Section 4.6), for tie-breaking (Section 4.7), and for locking (Section 4.8).

The opportunities for efficiently implementing the algorithm are discussed in Section 4.9. I prove that—under suitable conditions—any implementation of the algorithm requires at least $\Omega(n \log n)$ steps per pass, and thus no linear-time implementation is possible, unlike in the case of the FM algorithm. I also present an implementation based on the range tree data structure that achieves this lower bound for sparse graphs.

The implications of the problem formulation are elaborated in more depth in Section 4.10. In particular, I describe how more than two cost metrics can be incorporated into the algorithm without sacrificing efficiency. I also investigate how scheduling and other typical tasks of hardware/software co-design can be integrated into partitioning when using a KL-type algorithm.

In order to compare the different configurations of my algorithm with each other, I ran several empirical tests on benchmark problems. The results are presented in Section 4.11.

4.2 Previous work

4.2.1 The Kernighan-Lin algorithm and its variants

Originally, the KL algorithm was developed for circuit partitioning [54]. Its aim is to partition a graph into two parts of equal size (i.e., to find a so-called bisection of the graph) with a minimal number of cutting edges. The algorithm works by iterative improvement, that is, it starts from an arbitrary bisection, and swaps pairs of nodes in order to improve the cost of the partition.

The algorithm works in passes; in each pass every node moves exactly once. At the beginning of the pass, each node is *free*. In each step, a pair of free nodes is selected and swapped. The swapped nodes become *locked* (i.e., not free) afterwards. The algorithm is greedy in the sense that in each step it chooses the pair of nodes with the highest *gain*, where the gain of a pair of nodes is the decrease in cutsize achieved by swapping them.

The pass ends when there are no more free nodes. This also means that, as long as there are free nodes, a move is always done, even if it is a worsening move. This is how the algorithm can escape local optima. At the end of the pass, the algorithm reverts to the lowest cost partition observed during the pass. All nodes are unlocked and a new pass starts from this partition. The whole algorithm terminates when a pass does not find a better partition than its starting partition.

The most important variant of the KL algorithm was suggested by Fiduccia and Mattheyses [34]. Beside generalizing the KL algorithm to hypergraphs instead of graphs, they presented two vital improvements. First, they slightly relaxed the strict bisection constraint of KL, and suggested to move one node at a time instead of swapping a pair of nodes. Second, they presented an efficient data structure, the *gain bucket array*, with which a pass can be implemented in $O(n+m)$ time, where n is the number of nodes and m is the number of edges of the graph.

The gain bucket array concept depends on the fact that the gain of every node is an integer from the interval $[-d_{max}, d_{max}]$, where d_{max} is the highest degree of a node. Hence it is possible to index the nodes by their gains. In practice, this requires an array of size $2 \cdot d_{max}+1$, indexed from $-d_{max}$ to d_{max}. The element of the array with index i is a pointer to a linked list of the nodes that have gain i. Moreover, there is a separate pointer to the list corresponding to the highest gain. Beside this data structure, the efficiency of the algorithm depends on the observation that after moving a node only its own gain and the gain of its neighbors have to be updated, the other gains do not change.

A number of works focused on tie-breaking strategies in the KL or FM algorithm. Empirical tests have shown that often many nodes share the same gain value, in which case the tie-breaking strategy has an important role [43]. Krishnamurthy suggested a look-ahead mechanism to more precisely estimate the long-term gain of a move, and thus breaking ties [57]. This mechanism has also been generalized for multiway partitioning [79]. However, these efforts are beneficial primarily for hypergraphs. On the other hand, as shown in [43], tie-breaking without look-ahead can also be efficient, in particular, the LIFO strategy is more efficient than the FIFO or the random strategy, or even the look-ahead mechanisms.

Another area of intensive research has been the choice of the locking strategy. Many researchers felt that the original locking scheme was too rigid. In [46], a dynamic locking mechanism was suggested: when a node is moved from part A to part B, it becomes locked, but its neighbors in part A become free. This is beneficial because thus the neighbors have the possibility to follow this node. In order to prevent endless cycles, at most ten moves per pass are allowed for each node. Further relaxations were presented in [27, 95]; however, they are mainly beneficial for multiway partitioning, where the number of parts is high.

A good survey on these and other circuit partitioning algorithms can be found in [5].

4.2.2 KL in the context of hardware/software partitioning

The huge success of the KL algorithm and its variants in several domains suggested that these algorithms might be used for hardware/software partitioning as well. I know of two such attempts.

The first one is due to Vahid and it is described in two papers: [89] and [92]. Vahid's work mostly focused on replacing the cut metric of KL with a more appropriate and more complex execution-time metric. Essentially, the execution time of the system, with respect to a given partition, can be computed as the sum of the execution times of each node plus the sum of the transfer times along the edges. Here, each node's execution time depends on whether it is in hardware or software, and similarly, each edge's transfer time depends on whether the edge crosses the hardware/software boundary. This is almost the same model as ours, with the difference that Vahid does not neglect the execution time of the nodes in hardware, nor the communication penalty incurred by non-cut edges. Vahid uses a containment relation between the nodes to be able to calculate efficiently the time spent in a given node and its descendants. He also mentions that other metrics, such as hardware size and software size, should also be taken into account, but does not elaborate this issue.

Apart from this extended metric, Vahid used the concepts of the FM algorithm: single node moves in each step, and a gain bucket array for indexing move possibilities by their associated gain values. Unfortunately, since the execution times can be very large numbers, the gain bucket array also becomes huge. In fact, since only the logarithm of these numbers appears in the size of the input, the resulting algorithm has a space requirement that is exponential in the size of the input, and consequently, the running time of the algorithm is also exponential in the size of the input. To work around this problem, Vahid suggested normalizing the execution times to be integers between 0 and 1000. Unfortunately, this loss of precision can heavily degrade the algorithm: if, for instance, there is a node whose execution time is much bigger than that of the others, then the algorithm will not be able to distinguish between the other nodes, i.e., it will randomly move the nodes around.

However, even if we accept the workaround suggested by Vahid, the running time of one pass is still $O(n^2)$ in the worst case, which is of course much worse than the running time of $O(n+m)$ of the original FM algorithm.[1] Vahid argues that the worst case will likely not happen in practice; however, nothing guarantees that. The work of Vahid also left open the question of (i) whether the improvements suggested in the KL literature can be used in the context of hardware/software partitioning as well, and (ii) if also other formulations of the hardware/software partitioning problem allow the usage of a KL-type algorithm.

The second attempt to use a KL-type algorithm for hardware/software partitioning is the recent work of Lopez and Lopez-Vallejo [61]. Unfortunately, they gave hardly any details on how they adapted the algorithm. It seems that they addressed, at least partially, the second question above: they used a more sophisticated problem definition and consequently a more complex cost function. As the data structure for storing the gain values, they used the Map implementation of the Standard Template Library, yielding logarithmic time for accessing the gain values. However, because of the more complex cost function, it is not true anymore that only the gains of the neighbors of the recently moved node have to be updated. Rather, the gain of each node has to be recalculated after each move, which makes it unnecessary to store the gain values at all.

4.3 Challenges

The application of the KL heuristic in the context of hardware/software partitioning is a natural idea because of the ability of this heuristic to escape some local optima. Consider again Proposition 1 on page 22: it means that in the case of hardware/software partitioning local minima can be of very low quality, and escaping local minima is exactly where a KL-type algorithm can excel.

However, adapting the KL heuristic to hardware/software partitioning involves resolving the following four main challenges:

1. The original algorithm optimizes a single, quite simple cost function: the number of cut edges. In contrast, hardware/software partitioning typically deals with several conflicting cost functions. Alternatively, the different cost metrics are sometimes unified to a single cost function, but in this case, the cost function is much more complex than the cut metric of the original KL algorithm. Therefore, the algorithm has to be extended to handle the more complex cost metric(s).

2. The original KL algorithm maintains a very strict balance criterion, namely that the two parts have to be of equal size. The FM extension slightly relaxes the balance criterion: it aims at finding a partition with a given percentage of the nodes in one part; moreover, small deviations from this ratio are allowed. In the case of hardware/software partitioning, there is typically no explicit balance criterion. However, there are often constraints that have a similar effect. For example, a real-time constraint typically limits the number of nodes that can be mapped to software, whereas

[1] At least for sparse graphs, $O(n^2)$ is much worse than $O(n+m)$, and the graphs representing real designs are typically sparse.

a chip size constraint limits the number of nodes that can be mapped to hardware. Two such constraints together work out as an implicit balance criterion. Neither the original KL algorithm, nor the FM extension support such constraints.

3. As discussed in Section 1.1, the scope of hardware/software partitioning can vary significantly. Some partitioning approaches also include problems of very different nature, such as scheduling, routing, or interface synthesis. In contrast, the original KL algorithm addresses only the problem of partitioning graphs. It is questionable if it can also be extended to handle the other problems mentioned above.

4. FM is a very fast algorithm. However, as can be seen from the above, it has to be extended in several ways for the purposes of hardware/software partitioning. Therefore, it is an important but non-trivial objective to keep the extended algorithm also as fast as possible. This also involves finding the right data structure for the implementation.

The benefit of using the **P3** problem formulation is that it illustrates well how to deal with more than one, conflicting cost metrics. It also illustrates how cost metrics that are bound from above (R_P in this case) and cost metrics that have to be minimized (H_P in this case) can be handled together. However, it ignores the third of the above four challenges. It will be investigated later in Section 4.10 how other steps, such as scheduling, can be included.

4.4 The skeleton of the algorithm

As discussed in Section 4.3, there is typically no explicit balance criterion in hardware/software partitioning, hence my algorithm makes single node moves rather than node swaps, just as the FM algorithm. During the algorithm, two partitions are maintained:

- The current partition P_{curr}
- The partition that has been the best so far: P_{best}.

The skeleton of my algorithm is presented in pseudocode in Algorithm 4. The details are described in the next sections.

4.5 The gain function

The gain concept of the original KL algorithm has to be extended for the more complex cost metric. In the original algorithm, the gain had two roles: (i) it enabled fast updating of the cost of the current partition (without actually recomputing it) after a node had been moved; and (ii) it was the basis for choosing the next node to move. In my algorithm, these roles have to be separated, because each cost metric has to be updated separately

Algorithm 4 The skeleton of the KL-type algorithm

```
procedure onePass()
{
   calculate gains
   free all nodes
   while(there are free nodes) do
   {
      let $v$ be a free node with maximum gain
      move $v$ to the other part
      if $P_{curr}$ is better than $P_{best}$ then
      {
         let $P_{best} = P_{curr}$
      }
      perform locking
      update gains
   }
}

procedure KL()
{
   create initial partition, set $P_{curr}$ and $P_{best}$ to it
   repeat
   {
      onePass()
      let $P_{curr} = P_{best}$
   } until the pass did not improve $P_{best}$
   return $P_{best}$
}
```

but the choice of the next move should depend on all cost metrics. Therefore, I define the following three numbers instead:

- $\Delta R(v)$ is the amount by which moving v to the other context increases R. (Sometimes, $-\Delta R(v)$ will be called the software gain of node v.)
- $\Delta H(v)$ is the amount by which moving v to the other context increases H. (Sometimes, $-\Delta H(v)$ will be called the hardware gain of node v.)
- $gain(v) = f(\Delta R(v), \Delta H(v))$ is the basis for choosing the next node to move. (Note that—although it is not shown explicitly for the sake of readability—f can also depend on other variables, such as for instance the parameters of P_{curr}.)

The choice of the function f is crucial because it determines the order in which the nodes are moved. This function incorporates the sought trade-off between the conflicting cost measures. Specifically, by yielding the gain of a move with given $\Delta R(v)$ and $\Delta H(v)$ values, it defines the relative importance of the design goals.

Of course, f can be chosen in several ways. Since the algorithm always chooses the node with the highest gain, i.e., higher f values should indicate better moves, but $\Delta R(v)$ and $\Delta H(v)$ are lower for better moves, this means that f has to be monotonously decreasing in both of its arguments.

At this point it should be noted that my algorithm, at least in its current general form, contains the original FM algorithm as a special case. If $s(v) = h(v) = 0$ for each node v, $c(v,w) = 1$ for each edge (v,w), and $f(\Delta R(v), \Delta H(v)) = -\Delta R(v)$, then the gain of a node is exactly the amount by which moving it would decrease the number of cut edges.

But the goal is now different, and therefore f should be specialized in another way: the aim is to minimize H_P, while bounding R_P. The strictest solution for this is the following:

$$gain(v) = f(\Delta R(v), \Delta H(v), R_{P_{curr}}, R_0) = \begin{cases} -\infty & \text{if } R_{P_{curr}} + \Delta R(v) > R_0 \\ -\Delta H(v) & \text{otherwise} \end{cases} \tag{4.1}$$

That is, those moves that let the partition hurt the real-time constraint are infinitely bad, the other moves are ranked according to the gain in hardware cost associated with them. I will refer to this function as the *strict gain function.*

However, there are arguments in favor of less strict solutions. Suppose for example that a move would slightly hurt the real-time constraint but would result in a dramatic decrease in hardware cost. This move is infinitely bad according to the above function, yet it seems to be a good idea to allow such moves. In other words: a sufficiently large decrease in hardware cost can justify a small exceeding of the real-time constraint. One can hope that this way a much better part of the search space can be reached.

A logical possibility is to use a gain function of the following form:

$$gain(v) = f(\Delta R(v), \Delta H(v), R_{P_{curr}}, R_0) = -\Delta H(v) - p(R_{P_{curr}}, \Delta R(v), R_0) \tag{4.2}$$

Here, p is a *penalty function* that penalizes the exceeding of the time limit. Typically, p should depend on the percentage by which $R_{P_{curr}} + \Delta R(v)$ exceeds R_0, just like the fitness function in the case of the GA.[2] Note that it would also be possible to define p as a function of the *amount* (instead of percentage) by which R_0 is exceeded; however, it is much more informative to say that, for instance, the limit is exceeded by 10% than by 10 units. So, we can assume that p only depends on the quantity

$$exc = \frac{R_{P_{curr}} + \Delta R(v)}{R_0}$$

The same argument holds also for the other term of the expression: instead of $\Delta H(v)$ it is more informative to use the percentage by which H_P changes if this node is moved, i.e.,

[2]The similarity and the differences between this approach and the GA will be investigated in more detail in Section 6.1.

$\Delta H(v)/H_{P_{curr}}$. This way, the ultimate formula for the gain function becomes as follows (I will refer to such a function as a *permissive gain function*):

$$gain(v) = -\frac{\Delta H(v)}{H_{P_{curr}}} - p(exc). \tag{4.3}$$

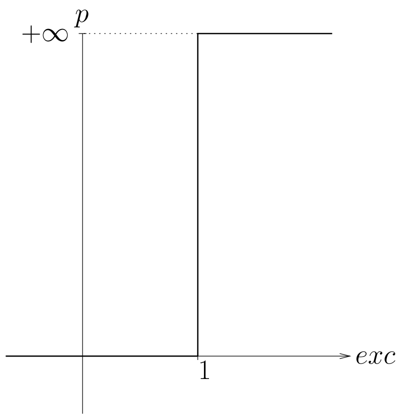


Figure 4.1: Strict penalty function

Note that Equation (4.1) is a special case of Equation (4.2), in which the penalty is ∞ if R_0 is exceeded, and 0 otherwise (see Figure 4.1). Strictly speaking, Equation (4.1) is not a special case of Equation (4.3) because of the division by $H_{P_{curr}}$ in Equation (4.3). However, when using the strict penalty function of Figure 4.1, the division by $H_{P_{curr}}$ does not change the ranking of the nodes. Thus, when looking only at the ranking of the nodes and not at the exact gain values, then Equation (4.1) is also a special case of Equation (4.3). This argument also shows why it was not necessary to divide $\Delta H(v)$ by $H_{P_{curr}}$ in Equation (4.1).

In the general case, it is also logical to set $p = 0$ if the limit is not exceeded. Moreover, a threshold $q > 1$ can be set, so that $p = \infty$ if even qR_0 is exceeded. Clearly, p should be monotonously increasing if exc is in the interval $[1, q]$. Such a function is shown in Figure 4.2.

Now that the different possibilities for the gain function have been introduced, a very important property of these functions will be investigated.

In the following, a function $\varphi : V \to \mathbb{R}$ is said to possess the $(*)$-*property* if the following holds: when moving a node v from one part to the other, $\varphi(w)$ does not change for any w that is not adjacent to v nor the same as v. In other words: only the φ value of v and its neighbors can change, all other φ values remain unchanged.[3]

Note that in the case of the original FM algorithm, the gain function, which was defined as the decrease of the cutsize, possesses the $(*)$-property. In fact, this was one of the crucial observations of Fiduccia and Mattheyses that lead to their highly efficient implementation.

[3]Hence the notation $(*)$: the nodes for which φ can change define a star-shaped (not necessarily induced) subgraph of G.

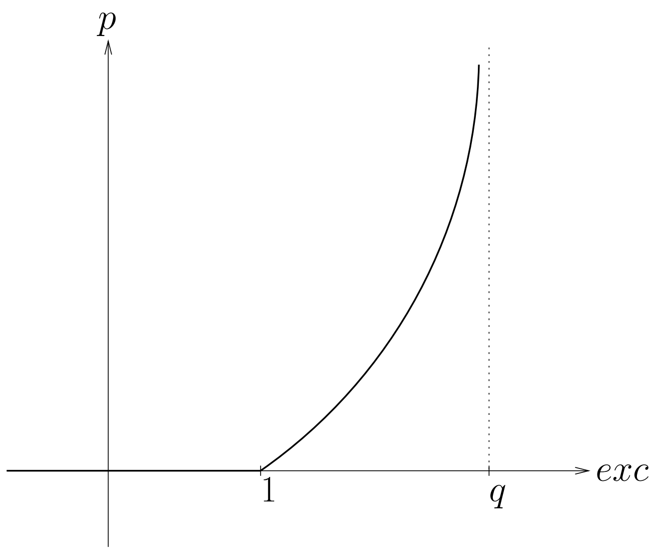


Figure 4.2: A possible permissive penalty function

The following theorem—which will have important consequences in Section 4.7 and Section 4.9—shows the connection between the $(*)$-property and the different gain notions defined here:

Theorem 8.
(i) ΔH possesses the $(*)$-property.
(ii) ΔR possesses the $(*)$-property.
(iii) gain does not necessarily possess the $(*)$-property. In fact, not even the strict gain function does.

Proof.
(i) From the definition of ΔH it is obvious that

$$\Delta H(v) = \begin{cases} h(v) & \text{if } v \in V_S \\ -h(v) & \text{if } v \in V_H. \end{cases}$$

Therefore, when moving v from one context to the other, only its own hardware gain changes, that of the other nodes remains unchanged, and thus ΔH possesses the $(*)$-property.
(ii) Similarly, it is clear from the definition of ΔR that

$$\Delta R(v) = \begin{cases} -s(v) + \sum_{w \in V_S, (v,w) \in E} c(v,w) - \sum_{w \in V_H, (v,w) \in E} c(v,w) & \text{if } v \in V_S \\ s(v) - \sum_{w \in V_S, (v,w) \in E} c(v,w) + \sum_{w \in V_H, (v,w) \in E} c(v,w) & \text{if } v \in V_H. \end{cases}$$

As can be seen, $\Delta R(v)$ depends on the context of v and that of its neighbors. Hence, when moving a node, the $\Delta R(v)$ value of itself and that of its neighbors can change; that of other nodes remains the same. That is, ΔR possesses the $(*)$-property.

(iii) As apparent from Equation (4.1), *gain* depends on $R_{P_{curr}}$. When a node is moved from one context to the other, $R_{P_{curr}}$ changes, and thus potentially the gain of all nodes can change. $\square$

Finally, I would like to point out that using a permissive gain function (i.e., one that does not give infinite penalty to non-valid partitions) requires extra precautions. Although one might get better results by temporarily allowing a small exceeding of the real-time constraint, it has to be guaranteed that in the end the output will be a valid partition, i.e., one that obeys the real-time constraint (just like in the case of the GA). For this purpose, the semantics of P_{best} should be changed slightly: P_{best} is always the best *valid* partition found so far. That is, P_{best} is updated only if a better valid partition has been found; when finding a partition with lower hardware cost but exceeding the time limit, P_{best} should not be updated.

4.6 The starting partition

In order to guarantee that P_{best} will always be a valid partition, it is also necessary to start from a valid partition (just like in the case of the GA; see Section 6.1 for a comparison). There are several possibilities to generate a valid initial partition:

- The all-hardware partition is valid, hence it is a good candidate for the starting partition. While this approach is very simple, it has the drawback that typically the all-hardware partition is far from the optimum. A starting partition of higher quality would make it more probable that the algorithm eventually finds a good partition.
- Any algorithm that produces a valid partition can be used to generate the starting partition. Since the whole KL algorithm is typically quite fast, the algorithm to create its starting partition should also be very fast. It can be for example a simple greedy algorithm, or any other fast partitioning heuristic. On the other hand, it is also interesting to test whether the KL algorithm can improve the partition found by other full-fledged partitioning algorithms. This way, KL can be incorporated into hybrid algorithms.
- A general way of improving the results of heuristics is to run them multiple times and take the best result. Of course, this is only useful for randomized algorithms; hence, it would be useful to start KL from a random partition. However, it is by no means obvious how one can quickly generate random valid partitions with an at least approximately uniform distribution. Here, I present a method that generates valid partitions randomly (see Algorithm 5)—but not with uniform distribution. The idea is to randomly map each node to either software or hardware, and check if the resulting partition is valid. If it is not, then the probability of mapping nodes to software is decreased. Sooner or later a valid partition is surely reached: in the worst case when the probability of mapping nodes to software becomes 0, the all-hardware solution is obtained. But the first valid partition that is found in this way is returned.

Algorithm 5 Randomly generate a valid initial partition

```
1. let $r = 1$ and $N$ be a positive integer; let $dr = r/N$
2. map each node independently with probability $r$ to software, with probability $1 - r$ to hardware
3. if the resulting partition is valid, return with it
4. otherwise: let $r = r - dr$ and goto 2.
```

The different methods for generating the initial partition are evaluated in Section 4.11, based on experimental results.

As can be seen from the above, there are many similarities, but also some differences between this algorithm and the GA. These questions are addressed in more depth in Section 6.1.

4.7 Tie-breaking

Whenever the node with maximum gain is not unique, a tie-breaking strategy has to be used to select one of the nodes with maximum gain. As discussed in Section 4.2.1, the tie-breaking strategies suggested in the KL literature can be roughly categorized into two groups: those that are based on look-ahead mechanisms and those based on previous behavior. Since all suggested look-ahead mechanisms are useful mainly for hypergraphs, we will consider here only the second group. It consists basicly of the following strategies ($\mathcal{M}$ denotes the set of nodes with maximum gain):

- Random, i.e., a node is randomly selected from $\mathcal{M}$
- LIFO, i.e., the node which was the last to get into $\mathcal{M}$ is selected
- FIFO, i.e., the node which was the first to get into $\mathcal{M}$ is selected

It has been reported that in general, the LIFO strategy outperforms the other two [43]. This can be intuitively explained as follows. One of the weaknesses of the FM algorithm is that it only moves one node at a time, and hence it often does not recognize that a much better partition could be reached through moving a highly connected subgraph from one of the parts to the other. Therefore, if a node is moved from one part to the other, then it is often beneficial to also let its neighbors follow it. The LIFO strategy encourages this, because in the FM algorithm the gain function possesses the $(*)$-property, hence the gain of a free node is changed only if one of its neighbors is moved, and thus the node selected by the LIFO strategy will be one whose neighbor was moved recently.

Now let us investigate to what extent this can be transferred to the case of hardware/software partitioning. Unlike in the original KL algorithm, the gain function is now real-valued. Hence, in this case it is very unlikely that more than one nodes have exactly the same gain value. Thus one could argue that no tie-breaking is needed. However, if there are some nodes with very similar gain values, it might be better to regard them as if they had the same gain value, and use one of the above tie-breaking strategies to select the winner from them.

Thus it seems that we face a one-dimensional clustering problem [49]: given n points with their gain values, it has to be determined which nodes have similar gains. However, the problem is actually simpler because only the best cluster has to be determined, i.e., the cluster with the highest gain value (denoted by $\mathcal{M}$ above). A possible method for this task is the following. Let us fix a constant $0 < \tau \leq 1$, and let B denote the highest gain value. Now define $\mathcal{M}$ as the set of nodes with gain values in the interval $[\tau \cdot B, B]$. τ should be chosen near 1, so that only a few nodes will be in the interval $[\tau \cdot B, B]$. (Note that the case $\tau = 1$ corresponds to the strategy of considering only the node with the highest gain value.)

A second difference between hardware/software partitioning and the original FM algorithm is that, according to Theorem 8 on page 42, now the ($*$)-property does not necessarily hold. Unfortunately, the LIFO strategy implicitly assumes the ($*$)-property in that the node whose gain changed the last time is the same as the one whose neighbor was moved the last time.

Therefore, the straight-forward adaptation of the LIFO tie-breaking strategy would presumably be less efficient than in the case of the FM algorithm. Hence, I suggest a more direct implementation of the actual aim of this tie-breaking strategy, i.e., to encourage moving the neighbors of recently moved nodes. More specifically, a variable $\ell(v)$ is defined for each node v, which stores the last time step when a neighbor of v was moved. (That is, a counter ctr is maintained, which is initialized to be zero at the beginning of each pass, and is incremented by one each time when a node is moved. Moreover, when node v is moved, the ℓ values corresponding to its neighbors are updated: for each neighbor w, let $\ell(w) = ctr$.) Now the LIFO strategy can be adapted as selecting the node with the highest ℓ value from $\mathcal{M}$, and similarly, FIFO is adapted as selecting the node with the lowest ℓ value from $\mathcal{M}$.

4.8 Locking schemes

From the many alternative locking schemes that have been suggested for KL (see Section 4.2.1 and [5]), the most promising for the purposes of hardware/software partitioning is the dynamic locking scheme proposed by Hoffman [46]. The other suggested locking schemes have been reported to be mainly beneficial for multiway partitioning with many parts, whereas hardware/software partitioning is inherently a bipartitioning problem.

Hoffman's method is based on the same intuition as the LIFO tie-breaking strategy discussed above: after moving a node from one part to the other, its neighbors should be encouraged to follow it. Hence, Hoffman suggests that after moving a node from part A to part B, its neighbors in A should be freed (the moved node however becomes locked). In order to prevent endless loops, a node can only be freed a given number of times—ten in Hoffman's work—during a pass.

Fortunately, this locking scheme can be adapted to hardware/software partitioning without any problems. However, it is questionable whether ten is the right number in this context as well.

4.9 Efficiency

One pass of the FM algorithm can be implemented in linear time, i.e., in $O(n+m)$ time. This depends on the following two crucial facts: (i) the possible moves can be indexed by their associated gain values, and thus stored in the gain bucket array data structure; (ii) the $(*)$-property holds for the gain function, i.e., after moving a node, only its own gain and the gain of its neighbors has to be updated.

Unfortunately, none of these two properties holds in the case of hardware/software partitioning: the gains can be large real numbers, which prohibits the usage of a gain bucket array, and the $(*)$-property does not hold (see Theorem 8 on page 42). In the following, it is investigated under which circumstances it is still possible to provide an efficient—but more tricky—implementation.

As it turns out, the efficiency of the algorithm depends very much on the gain function. First, let us consider the most general case: the gain is specified by some function f, on which no restrictions are imposed, except that it can be calculated in $O(1)$ time. In the worst case, the gain of all nodes can change after each move. Since a pass consists of $\Theta(n)$ moves (exactly n moves when using the original locking scheme, and at most cn moves when using dynamic locking, where c is a small constant), and after each move, the gain of all free nodes has to be calculated, and there are $n/2$ free nodes on average, this means that the duration of one pass is $\Theta(n^2)$. Also note that this can be achieved without any complicated data structures: the maximum gain can be selected while calculating the gains of all nodes, and there is no need to even store the calculated gain values, since only their maximum is needed, and all of them will be recomputed after the move anyway.

However, we are interested in a special class of gain functions, so that one can hope for a more efficient implementation. One could even hope for a linear-time implementation, as was the case with FM. However, this is not possible:

Theorem 9. *When using the strict gain function, every implementation has at least $\Omega(n \log n)$ time complexity.*

Proof. It will be shown that it is possible to sort numbers using the presented algorithm. Since it is known that sorting n numbers takes $\Omega(n \log n)$ time [26], the theorem will follow.

Assume that the numbers $x_1, \dots, x_n$ have to be sorted. A graph G is defined with n vertices $(v_1, \dots, v_n)$ and no edges. For each node v_i, let $s(v_i) = 0$ and $h(v_i) = x_i$. Let R_0 be any non-negative number, then $R_P \leq R_0$ will always hold (since R_P will always be 0). Assume that the initial partition is the all-hardware partition, and the original locking scheme is used. Then the gain of node v_i will always be $h(v_i) = x_i$. Since the algorithm moves the nodes in decreasing order of their gain values, observing the order in which the nodes are moved yields the decreasing order of the x_is. $\square$

Since the permissive gain function family contains the strict gain function as a special case, the same holds also for the permissive gain functions.

Now I present an implementation which is almost as efficient as this lower bound. First, the case of the strict gain function is considered.

According to Theorem 8 on page 42, not even the strict gain function possesses the $(*)$-property, and thus it is by no means obvious how the algorithm can be faster than the trivial $O(n^2)$. The key idea is that ΔH and ΔR do possess the $(*)$-property, and the strict gain function has such a simple structure that the node with highest gain can be found *without actually storing or even computing the gain values of each node explicitly.* Rather, only the ΔH and ΔR values are stored for each node. Because of the $(*)$-property, these can be updated efficiently: all updating steps together require $O(n+m)$ time per pass.

The gain depends beside $\Delta H(v_i)$ and $\Delta R(v_i)$ also on $R_{P_{curr}}$. In a given step of the algorithm, $R_{P_{curr}}$ has a concrete value. We are only interested in those nodes for which $R_{P_{curr}}+\Delta R(v_i) \leq R_0$ because all other nodes have gain $-\infty$. And from these 'good' nodes, the one with the highest hardware gain has to be selected.

The nodes can be thought of as points in the plane, where the x coordinate of v_i is $x_i := \Delta R(v_i)$ and its y coordinate is $y_i := \Delta H(v_i)$. Then, the task of selecting the node with the highest gain becomes this: select the point with lowest y coordinate in the $x \leq R_0 - R_{P_{curr}}$ half plane.

Thus a data structure is needed in which 2-dimensional points can be stored, such that queries of the form 'select the point with lowest y coordinate from the ones with $x \leq x_0$' can be executed efficiently, and it can be updated efficiently if the coordinates of a point change or a point is deleted. Fortunately, such a data structure is known in the computational geometry community: the range tree [19]. Actually, the range tree was developed to support queries of the form 'select all points in the rectangle x_1, y_1, x_2, y_2.' However, it can be trivially adapted to the above kind of query. With a range tree, all the needed operations can be performed in $O(\log n)$ time, thus yielding a time complexity of $O((n+m)\log n)$ for one pass of the algorithm ($O(n)$ searches and $O(n+m)$ updates).

As can be seen, the simple structure of the strict gain function and the powerful range tree data structure made it possible to find the node with highest gain without explicitly calculating the gain of each node, and only storing and updating the ΔH and ΔR values. Now let us investigate how this can be generalized to the permissive gain functions. The problem here is that the penalty function is not constant between R_0 and qR_0. In the case of the strict gain function it was possible to rephrase the problem as a simple geometric query because the penalty function had only two values, one of which was infinite.

Fortunately, we have some freedom in choosing the penalty function. It is not prescribed what it should look like between R_0 and qR_0, it is only required that it should be monotonously increasing from 0 to ∞. For the sake of efficiency, p will be required to be a 'staircase'-like function, i.e., it should consist of a small number of constant parts. Actually, this is not a big restriction, since every function can be approximated with staircase-like functions. For instance, the function in Figure 4.2 can be approximated with the staircase-like function in Figure 4.3.

For a staircase-like penalty function, the algorithm works as follows. Let (x_1, x_2) be an

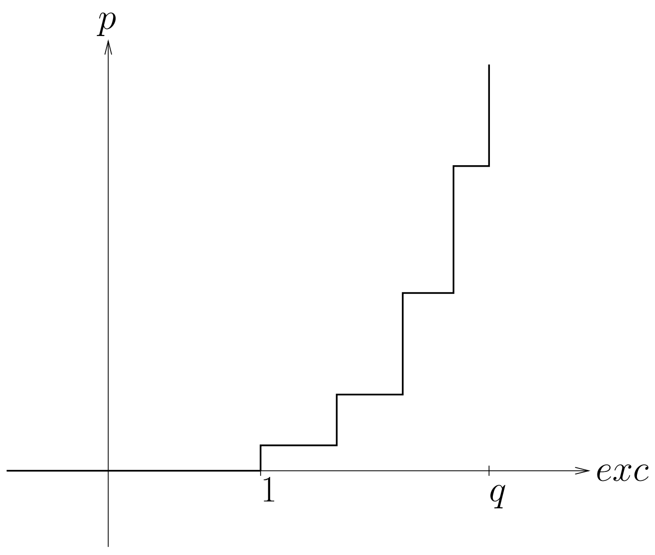


Figure 4.3: A staircase-like penalty function. This is an approximation of the function in Figure 4.2 on page 42.

interval in which p is constant. This means that in this interval the software gain does not influence the gain; the gain is proportional to the hardware gain. Therefore, our task is to select the point with lowest y coordinate from the 'track' $\{(x, y) : x_1 \leq x \leq x_2\}$. Again, a range tree can be used to efficiently implement such queries. Thus, a best node can be obtained for each interval where p is constant. Afterwards, the winner is selected from this handful of nodes based on their gains. That is, the gain has to be explicitly calculated only for these nodes. This way, assuming that there are $O(1)$ intervals with constant p, the whole pass can be implemented in $O((n+m)\log n)$ time for (staircase-like) permissive gain functions as well.

Note that for sparse graphs, i.e., if $m = O(n)$, this is the same as $O(n \log n)$, so that this algorithm has optimal performance—up to constant factors—for sparse graphs.

4.10 Extension possibilities

4.10.1 Considering more than two cost metrics

Until now, two cost metrics were considered: execution time and hardware cost. However, in some cases, other cost metrics have to be considered as well, such as power consumption or chip size. In general, assume there are k cost metrics (where k is a small constant): $c_1, \ldots, c_k$. In order to define a proper optimization problem, assume that there is a constraint on $k-1$ of the cost metrics, and the aim is to minimize the kth cost metric. That is, the problem is to find a partition P such that $c_1(P) \leq C_1, \ldots, c_{k-1}(P) \leq C_{k-1}$ and $c_k(P)$ is minimal among all such partitions.

Just as above, we can define for each node v_i and each cost metric c_j the change of that cost metric caused when moving v_i, denoted by $\Delta c_j(v_i)$. The gain of a node v_i is now defined as

$$gain(v_i) = f(\Delta c_1(v_i), \ldots, \Delta c_k(v_i)) = -\Delta c_k(v_i) - \sum_{j=1}^{k-1} p_j(\Delta c_j(v_i), c_j(P_{curr}), C_j),$$

where the p_js are appropriate penalty functions (p_j penalizes the exceeding of the constraint on c_j). Again, it is possible to define strict or permissive penalty functions.

If no further assumptions can be made concerning the gain function or the Δc_j functions, then the algorithm can be again implemented in a straight-forward way yielding a performance of $O(n^2)$. Conversely, if the Δc_j functions satisfy the $(*)$-property, and if the penalty functions are staircase-like, then it is again possible to devise an $O((n+m)\log n)$ implementation. In this case, the nodes can be modeled as points of the k-dimensional space with coordinates $(\Delta c_1(v_i), \ldots, \Delta c_k(v_i))$, and queries of the form 'select the point $(x_1, \ldots, x_k)$ with minimum x_k from the set $a_1 \le x_1 \le b_1, \ldots, a_{k-1} \le x_{k-1} \le b_{k-1}$' must be performed. This, too, can be implemented efficiently using k-dimensional range trees.

4.10.2 Incorporating scheduling and other tasks

Until now, it has been assumed that the cost metrics are additive. For example, it was assumed that the hardware cost of a partition is the sum of the hardware cost of the nodes in hardware. Likewise, it was assumed that the execution time of the system can be calculated as the sum of the execution time of the nodes in software plus the sum of the communication overhead of the edges. In this respect, I followed the assumptions of Vahid [89]. However, Vahid's model is quite restrictive: it assumes a single hardware unit and a single software unit and excludes parallelism between the two. In a more general framework with several hardware and software units, some additional factors have to be considered, such as:

- In general, execution time is not additive. Rather, it depends on the number of available processing units, and on the precedence constraints between the nodes that constrain parallelism. Therefore, the calculation of the execution time involves scheduling the modules on the given processing units.
- Likewise, communication events have to be scheduled on the available communication links. Moreover, depending on the topology of the communication links, communication events may have to be routed along the communication links.
- Hardware costs are often not additive because of hardware sharing. That is, several modules can use the same hardware resource, thus reducing costs. It has to be noted that scheduling and hardware sharing are not independent: if two modules share a hardware resource, then they must not be active at the same time.

A hardware/software co-design framework has to consider all of these effects. However, as demonstrated in Section 1.1, there is no consensus in the literature whether all these aspects should be considered during partitioning. In some works, the whole co-design framework is a single optimization step, in which partitioning, scheduling, routing etc. are performed together; in others, partitioning only means deciding which modules to implement in hardware and which ones in software. In this latter group, powerful methods have been devised to decouple partitioning from the other problems (see e.g., [62]). Such decoupling results in a loss of precision because the partitioning algorithm has only an estimate of the cost metrics. Conversely, as discussed in Section 1.3, the complexity of the problem is drastically reduced, and thus a bigger percentage of the search space can be searched. This way, similar or even better results are achieved than by considering all aspects together but scanning only a small fraction of the huge search space.

My adaptation of the KL algorithm, as described so far, clearly belongs to the second group. I believe that the KL algorithm is intrinsically more appropriate for the second group because its strength lies in partitioning graphs by the efficient optimization of simple cost metrics. However, I will now briefly sketch how it can also accommodate scheduling of modules, scheduling of communication events, routing, hardware sharing etc. The cost metrics are then not additive anymore; rather, the cost metrics are calculated by external scheduling, routing etc. algorithms that can be included as black box into the algorithm. The $\Delta c_j(v_i)$ values that are needed by the algorithm are calculated by tentatively moving v_i to the other part, and then running the appropriate external algorithm in charge of calculating c_j (e.g., running a scheduler to calculate the execution time), and then moving the node back. In this case, the $(*)$-property is clearly lost, because after moving a node, all Δc_j values can change. For instance, moving a node from hardware to software can make it so slow that it invalidates every schedule that was considered so far.

Therefore, no 'tricky' implementation can be hoped for. The time complexity of the algorithm for one pass is $O(n^2\varrho)$, where $O(\varrho)$ is the time needed to run the external algorithms. This is because a pass consists of $O(n)$ moves, and after each move, the gain of $O(n)$ nodes has to be recomputed, and one such recomputation lasts $O(\varrho)$ time. Typically, ϱ will be non-negligible, since it is the time for solving a hard problem such as scheduling. The fastest list schedulers need $O(n)$ time, more advanced schedulers much more. Therefore, the complexity of the algorithm will be at least $\Omega(n^3)$, maybe significantly more.

To sum up: it is possible to include scheduling and hardware sharing into the KL algorithm; however, the algorithm will then lose its main advantages. (For example, it will calculate many schedulings, from which many will turn out to be equivalent and/or not needed.) But as demonstrated above, the algorithm is indeed very efficient for the other group of hardware/software partitioning formulations that make use of simplified cost metrics.

4.11 Empirical results

As can be seen from the previous sections, the algorithm has many variants and many parameters that can be tuned. I implemented several versions in order to compare them empirically. At this point, only the results of these tests will be sketched; the comparison of this algorithm to other partitioning methods is deferred to Chapter 6.

Here is a list of the implemented variants and parameters:

- Penalty function: strict vs. permissive
- Initial partition: all-hardware vs. greedy heuristic vs. the presented randomized algorithm (Algorithm 5 on page 44)
- Tie-breaking strategy: FIFO vs. LIFO vs. random. Moreover, the value of τ also had to be tuned.
- Locking scheme: original vs. dynamic
- In the case of dynamic locking: how many moves should be allowed?

As can be seen from this list, there are altogether 36 variants, plus an integer and a real-valued parameter. For testing I used the 17 benchmarks that will be presented in more detail in Chapter 6. In the case of the randomized algorithms, I ran the program 100 times and took the average result. The whole testing process was automated by a test script. The results can be summarized as follows:

- A general experience: my findings are mostly in accordance with the results reported for the original KL algorithm. That is, the winner is typically the same version for my algorithm as for plain KL. However, the results on hardware/software partitioning seem to be much more noisy than those for plain graph bisection. It is probably due to this fact that the difference between the different versions is often not as clear in the case of hardware/software partitioning than it is for the original algorithms.
- Concerning the penalty function: in most cases the permissive function yielded better results.
- Initial partition: the presented randomized algorithm (Algorithm 5 on page 44) offered the best results. Moreover, there was no significant difference between the results obtained from the all-hardware partition and the one obtained from the result of the greedy heuristic. This may be attributed to the fact that the KL algorithm itself is an improved greedy algorithm, and thus a greedy pre-optimization does not improve it.
- Tie-breaking: the LIFO strategy won, but not as clearly as reported for the original KL algorithm. The best value of τ was about 0.95. Note that I also tested the possibility of $\tau = 1$, and it was sub-optimal. This means that it is indeed useful

to choose from the best couple of moves using a tie-breaking strategy, instead of choosing simply the best one.

- Locking: the dynamic locking scheme proved to be significantly better than the original locking strategy. This difference was much clearer than the others mentioned above.
- Allowed number of moves: in contrast to the result reported in [46], I found that allowing more than 5 moves per node per pass does not improve the performance significantly anymore. With 5 allowed moves, the improvement over the original locking scheme was about 18% (averaged over all runs).

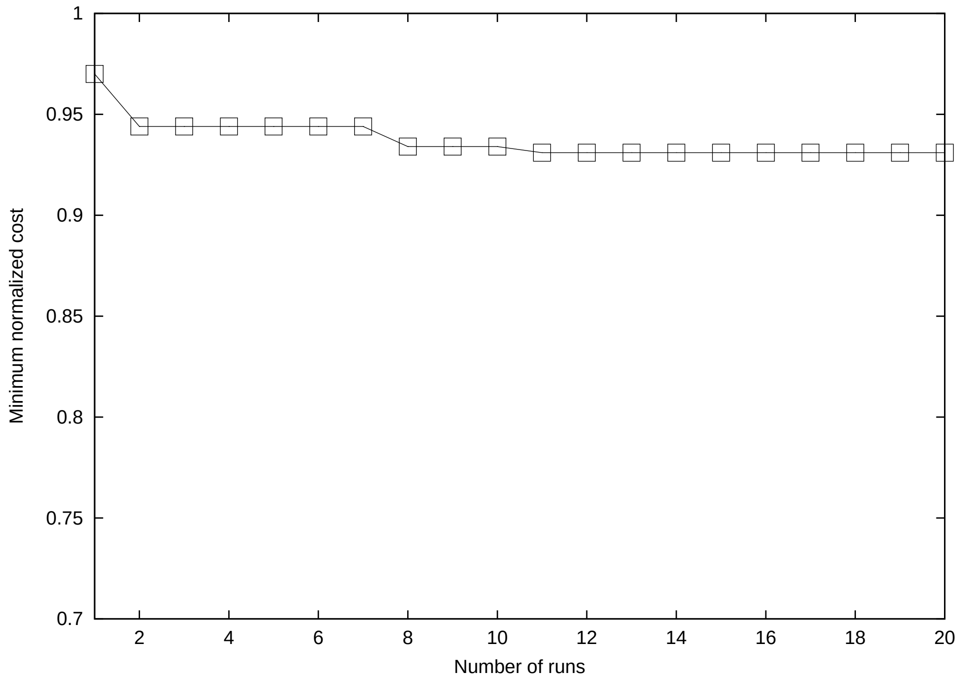


Figure 4.4: Improving the result by running the algorithm multiple times

It should also be mentioned that the randomized versions of the algorithm can be improved by running them multiple times, and taking the best result. A typical example is shown in Figure 4.4. As can be seen, 10-15 runs of the algorithm produce a result which is about 5% better than the first one. However, the improvement becomes very slow afterwards.

Altogether, the best configuration of the algorithm (permissive penalty function, Algorithm 5 for generating the initial partition, LIFO tie-breaking, dynamic locking) offered an improvement in result quality of about 31% over the plain version.

Chapter 5

MFMC-based algorithm

Thesis 4: I developed a heuristic algorithm for the **P3** problem, which is—in contrast to most other heuristics for hardware/software partitioning—not the application of a general-purpose heuristic, but rather an algorithm that makes use of the combinatorial properties of the partitioning problem.
This algorithm works by creating auxiliary graphs from the original graph, which are different instances of the **P5** problem. These problem instances are in turn solved by the exact polynomial-time algorithm for **P5** that was presented in Section 2.2. Thus, candidate partitions are generated, and the best one from these that does not violate the constraint is selected.
This way, only high-quality candidate partitions are evaluated, which makes it possible to find good solutions even if only a small fraction of the search space is scanned. A second consequence is that the algorithm is able to infer non-trivial lower bounds on the cost of the optimum solution—a feature that no other partitioning heuristic suggested in the literature possesses.

5.1 Motivation

The aim of this chapter is to present a more powerful partitioning algorithm by capturing the *combinatorial structure* behind the partitioning problem. That is, instead of applying general-purpose heuristics to hardware/software partitioning, I devise an algorithm based on the graph-theoretic properties of partitioning. This way, one can hope to obtain a more scalable algorithm.

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 I intend to achieve by making use of the combinatorial properties of the problem.

At this point I would like to refer back to Section 2.2, where I showed that the **P5** problem can be solved optimally in polynomial time. In fact, the proof was constructive, i.e., an efficient algorithm has been described (see Algorithm 1 on page 19).

The main contribution of this chapter is a completely novel heuristic algorithm for the **P3** problem which is based on the polynomial-time exact algorithm for the **P5** 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. 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 best result it found so far lies from the optimum. This is a feature that no previous partitioning algorithm possessed.

The idea is to run the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19) 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 **P5** problem with some α, β, and γ parameters. Then the best partition from this set that fulfills the given limit on R_P is selected.

As already mentioned, the scalability of a heuristic depends on whether the evaluated small fraction of the search space contains high-quality points. I believe that this can be achieved with the above choice of candidate partitions, because this way the candidate partitions are optimal for at least a related problem.

5.2 Description of the algorithm

Obviously, the result of the run of the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19) is determined by the *ratio* of the three weights, and not by their absolute values. Therefore, one of the three, e.g., β, can be fixed, and only the other two will be varied. This leads to a two-dimensional search problem, in which the evaluation of a point involves running the polynomial-time algorithm for the **P5** problem with the appropriate weights.

In order to keep the algorithm fast, I use two phases: in the first phase, coarse-grained steps are used in the two-dimensional plane to find the best valid partition approximately, and in the second phase a more fine-grained search is performed in the neighborhood of the point found in the first phase (see Algorithm 6 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 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 6, I 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

Algorithm 6 MFMC-based algorithm

```
*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 $\alpha$, $\beta$, and $\gamma$, let the resulting partition be $P$;
    **if**($R_P \leq R_0$ **and** $H_P <$ best_so_far) {
      save current solution;
      save previous and next $\alpha$ value ($\alpha_{prev}$, $\alpha_{next}$);
      save previous and next $\gamma$ value ($\gamma_{prev}$, $\gamma_{next}$);
      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 $\alpha$ going between $\alpha_{prev}$ and $\alpha_{next}$, $\gamma$ going between
$\gamma_{prev}$ and $\gamma_{next}$, and using $\varepsilon' < \varepsilon$ instead of $\varepsilon$.
```

solution is found, thus the search slows down as soon as it finds a better solution. (After an initial set of tests, I 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. This issue will be investigated in more detail, first from a theoretical point of view in Section 5.3, and then empirically in Section 5.6.

5.3 Monotonicity

Let $P(\alpha, \beta, \gamma)$ denote the partition for which $\alpha H_P + \beta S_P + \gamma C_P$ is minimal. (If there are more than one such partitions, then one of them.)

The aim of this section is to investigate $H_{P(\alpha,\beta,\gamma)}$ and $R_{P(\alpha,\beta,\gamma)}$ as a function of α, β, and γ. These are six separate tasks.

The intuitive expectation is that, by decreasing α, hardware becomes cheaper, and hence the algorithm puts more nodes into hardware, thus $H_{P(\alpha,\beta,\gamma)}$ should increase and $R_{P(\alpha,\beta,\gamma)}$ should decrease. Similarly, the opposite effect is expected when β is decreased. This would be a possible positive answer to the question raised at the end of the last section: whether the cost functions have a simple structure. For changing γ, there is no such expectation.

Let us start with a positive result:

Theorem 10. *$H_{P(\alpha,\beta,\gamma)}$ is monotonously decreasing in α.*

Proof. Let $P = P(\alpha, \beta, \gamma)$ and $P' = P(\alpha', \beta, \gamma)$, where $\alpha' < \alpha$. Now it will be shown that $H_{P'} \geq H_P$.

Since P is optimal with respect to (α, β, γ),

$$\alpha H_P + \beta S_P + \gamma C_P \leq \alpha H_{P'} + \beta S_{P'} + \gamma C_{P'}.$$

Similarly, since P' is optimal with respect to (α', β, γ),

$$\alpha' H_{P'} + \beta S_{P'} + \gamma C_{P'} \leq \alpha' H_P + \beta S_P + \gamma C_P.$$

Using the abbreviations $A = \beta S_P + \gamma C_P$ and $B = \beta S_{P'} + \gamma C_{P'}$, these can also be written as

$$\alpha H_P + A \leq \alpha H_{P'} + B, \tag{5.1}$$

$$\alpha' H_{P'} + B \leq \alpha' H_P + A. \tag{5.2}$$

Using (5.1) and (5.2),

$$A \leq B + \alpha(H_{P'} - H_P) \leq (A + \alpha'(H_P - H_{P'})) + \alpha(H_{P'} - H_P) = A + (\alpha' - \alpha)(H_P - H_{P'}).$$

Therefore,

$$0 \leq (\alpha' - \alpha)(H_P - H_{P'}).$$

Since $\alpha' - \alpha < 0$, this is only possible if $H_P - H_{P'} \leq 0$. □

Unfortunately, in the other five cases, no monotonicity can be guaranteed. Here is a proof for one of the negative results:

Theorem 11. *$R_{P(\alpha,\beta,\gamma)}$ is neither monotonously increasing nor monotonously decreasing in α.*

Proof. Two examples will be shown. In each example, α is decreased, β and γ remain unchanged. In the first example, $R_{P(\alpha,\beta,\gamma)}$ decreases, in the second example, it increases.

Example 1. The graph consists of a single vertex v with $s(v) = 1$ and $h(v) = 1$. At the beginning, $\alpha = 3$ and $\beta = 2$ (the value of γ does not matter, since there are no edges). In this case, the optimum is clearly to put v to software, yielding $R = 1$. Now α is decreased to 1. The new optimum is obviously to put v to hardware, yielding $R = 0$.

Example 2. The graph consists of two vertices u and v, connected by edge e. The costs are as follows:

$s(u)$:	1
$s(v)$:	1
$h(u)$:	1
$h(v)$:	100
$c(e)$:	2

First, $\alpha = 100$, $\beta = 10$ and $\gamma = 1$. Then the optimum is to put both nodes to software, because this partition has total cost 20, whereas putting any node to hardware induces a total cost of at least 100. This yields $R = 2$.

Now, α is decreased to 1. The software-only partition has still total cost 20; however, putting u to hardware and v to software now only has total cost 13. The other two partitions have much higher costs because in those cases, v is in hardware, inducing a total cost of at least 100. Thus the optimum is to put u to hardware and v to software, yielding $R = 3$. □

In order to conserve space, the proof of the other four negative results is omitted.

As can be seen, the intuition proved only partially true, and the cost functions do not necessarily offer the conjectured simple structure. However, as shown in Section 5.6, in practical cases there is a simple structure.

5.4 Extreme values for α, β, and γ

In this section, the description of the MFMC-based algorithm (Algorithm 6 on page 55) is continued, and the aim is to clarify the following question: which region of the $\alpha - \gamma$ plane should be scanned for meaningful solutions? The intuition is that after certain extreme values of α or γ, there is no point in continuing the search, because no new solution will be found.

To formalize this, the following notations will be useful. Let P^*H denote the trivial partition in which all nodes are in hardware, and P^*S the trivial partition in which all nodes are in software. Moreover, the following simple lemma will be used:

Lemma 2. *For any non-negative numbers $x_1, \ldots, x_k$ and positive numbers $y_1, \ldots, y_k$,*

$$\min_{1 \le i \le k} \frac{x_i}{y_i} \le \frac{\sum_{i=1}^k x_i}{\sum_{i=1}^k y_i} \le \max_{1 \le i \le k} \frac{x_i}{y_i}$$

Proof.

$$\frac{\sum_{i=1}^k x_i}{\sum_{i=1}^k y_i} = \frac{\sum_{i=1}^k \frac{x_i}{y_i} y_i}{\sum_{i=1}^k y_i} \le \frac{\sum_{i=1}^k \left(\max_{1 \le j \le k} \frac{x_j}{y_j}\right) y_i}{\sum_{i=1}^k y_i} = \left(\max_{1 \le i \le k} \frac{x_i}{y_i}\right) \frac{\sum_{i=1}^k y_i}{\sum_{i=1}^k y_i} = \max_{1 \le i \le k} \frac{x_i}{y_i}$$

The other inequality can be proven in the same way. □

This lemma remains true if some of the y_is can be 0 ($c/0$ is defined to be $+\infty$ for $c > 0$). The only constraint is that $\forall 1 \le i \le k : x_i > 0$ or $y_i > 0$ must hold. Accordingly, it will be assumed in the following that $\forall v \in V : h(v) > 0$ or $s(v) > 0$ holds (so that Lemma 2 can be used for these numbers). Of course, this is not a serious limitation in a practical setting.

Now it is possible to formulate the first result which states that there is indeed no point in scanning α values that are lower than a given extreme:

Theorem 12. *If $\alpha \le \beta \cdot \min_{v\in V} \frac{s(v)}{h(v)}$, then the partition P^*H is optimal with respect to α, β, and γ (regardless of the value of γ).*

Proof. Since $\alpha \le \beta \cdot \min_{v\in V} \frac{s(v)}{h(v)}$, using Lemma 2 it follows that

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \le \beta \cdot \frac{s(X)}{h(X)}.$$

Therefore,

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \cdot h(X) \le \beta \cdot s(X) + \gamma \cdot c(X, V \setminus X).$$

Adding $\alpha \cdot h(V \setminus X)$ to both sides:

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \cdot h(V) \le \alpha \cdot h(V \setminus X) + \beta \cdot s(X) + \gamma \cdot c(X, V \setminus X).$$

Using the notation $Y = V \setminus X$, this can be rewritten as

$$\forall Y \subsetneq V : \quad \alpha \cdot h(V) \le \alpha \cdot h(Y) + \beta \cdot s(V \setminus Y) + \gamma \cdot c(Y, V \setminus Y).$$

Here, the left-hand side is exactly the total cost of P^*H, the right hand side is the total cost of any other partition. $\square$

Similarly, the next theorem shows that there is no point in scanning α values that are greater than a given extreme:

Theorem 13. *If $\alpha \ge \beta \cdot \max_{v\in V} \frac{s(v)}{h(v)}$, then the partition P^*S is optimal with respect to α, β, and γ (regardless of the value of γ).*

Proof. Since $\alpha \ge \beta \cdot \max_{v\in V} \frac{s(v)}{h(v)}$, using Lemma 2 it follows that

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \ge \beta \cdot \frac{s(X)}{h(X)}.$$

Therefore,

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \cdot h(X) + \gamma \cdot c(X, V \setminus X) \ge \beta \cdot s(X).$$

Adding $\beta \cdot s(V \setminus X)$ to both sides:

$$\forall \emptyset \neq X \subseteq V : \quad \alpha \cdot h(X) + \beta \cdot s(V \setminus X) + \gamma \cdot c(X, V \setminus X) \ge \beta \cdot s(V).$$

Here, the right-hand side is exactly the total cost of P^*S, the left-hand side is the total cost of any other partition. $\square$

Finally, the next theorem shows that there is no point in scanning too high γ values. Unlike in the previous theorems, here it must be assumed that the graph is connected. However, this is again no serious limitation when considering real communication graphs.

Theorem 14. *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 P^*H or P^*S is optimal with respect to α, β, and γ.*

Proof. Assume that neither of P^*H and P^*S is optimal. Then, since G is connected, the optimal partition $P(\alpha, \beta, \gamma)$ cuts at least one edge. Hence, its total cost is at least $\gamma \cdot c_{min}$. On the other hand, according to the property that $\min(\alpha \cdot h(V), \beta \cdot s(V)) \leq \gamma \cdot c_{min}$, either P^*H or P^*S offers a smaller total cost than this. □

Putting these three theorems together, one can conclude that the region that should be scanned is defined by the following inequalities:

$$\alpha \geq \beta \cdot \min_{v \in V} \frac{s(v)}{h(v)} \tag{5.3}$$

$$\alpha \leq \beta \cdot \max_{v \in V} \frac{s(v)}{h(v)} \tag{5.4}$$

$$\gamma \leq \alpha \cdot \frac{h(V)}{c_{min}} \tag{5.5}$$

$$\gamma \leq \beta \cdot \frac{s(V)}{c_{min}} \tag{5.6}$$

Suppose that β is fixed. This region is bounded if $c_{min} > 0$ and $h_{min} > 0$ hold. Again, this can be assumed without imposing any serious limitations in the case of real communication graphs. Then the region to be scanned looks like the one depicted in Figure 5.1.

In order to see this, one only has to prove that the point in which the line $\gamma = \alpha \cdot \frac{h(V)}{c_{min}}$ intersects the horizontal line $\gamma = \beta \cdot \frac{s(V)}{c_{min}}$ is indeed between the two vertical lines $\alpha = \beta \cdot \min_{v \in V} \frac{s(v)}{h(v)}$ and $\alpha = \beta \cdot \max_{v \in V} \frac{s(v)}{h(v)}$.

In the intersection point $\alpha \cdot \frac{h(V)}{c_{min}} = \gamma = \beta \cdot \frac{s(V)}{c_{min}}$, and hence, $\alpha = \beta \cdot \frac{s(V)}{h(V)}$. According to Lemma 2, this is indeed between $\beta \cdot \min_{v \in V} \frac{s(v)}{h(v)}$ and $\beta \cdot \max_{v \in V} \frac{s(v)}{h(v)}$ and so the intersection point is really between the two vertical lines.

5.5 Determining lower bounds

As mentioned earlier, the MFMC-based algorithm can also incorporate the feature of determining lower bounds on the cost of the optimal solution of the given **P3** problem instance. This is a unique feature of this algorithm, that is offered by no other competing heuristic. With the help of this feature, the MFMC-based algorithm 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

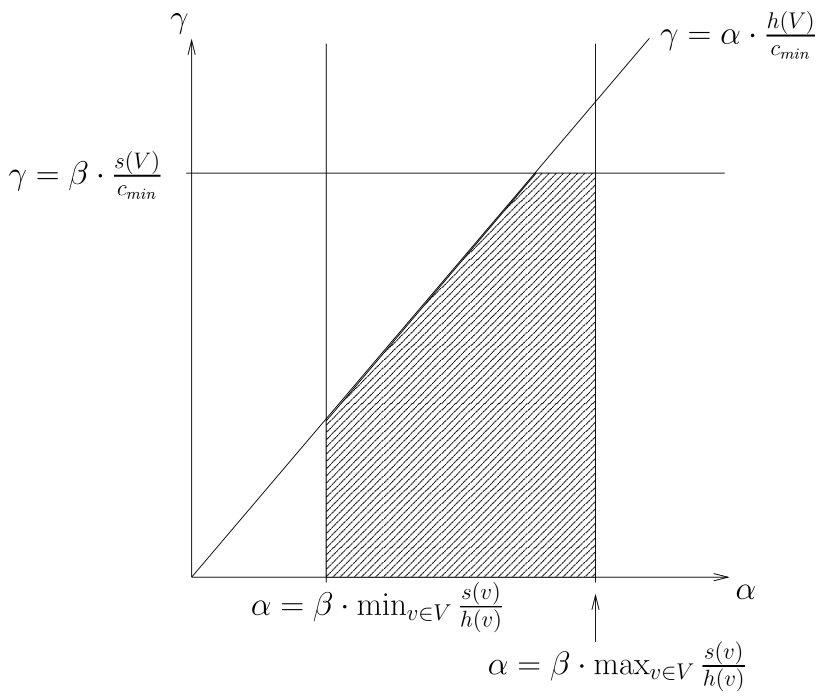

γ
$\gamma = \alpha \cdot \frac{h(V)}{c_{min}}$
$\gamma = \beta \cdot \frac{s(V)}{c_{min}}$
α
$\alpha = \beta \cdot \min_{v \in V} \frac{s(v)}{h(v)}$
$\alpha = \beta \cdot \max_{v \in V} \frac{s(v)}{h(v)}$

Figure 5.1: The region to be scanned

and edges of the graph are measured values with a precision of 5%, then there is no point in continuing the search if the gap between the lower bound and the found best solution is under 5% 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, the MFMC-based algorithm is able to determine the lower bounds because every candidate partition that it evaluates is optimal for the **P5** problem with some α, β, and γ values. Hence, each evaluated candidate partition yields some information about the costs of *all* partitions. This is formalized by the following theorem.

Theorem 15. *Suppose that P is an optimal solution of the **P5** problem with the weights α, β, and γ. Let Q be any solution of the **P3** problem (i.e., a partition that satisfies the bound $R_Q \leq R_0$). Then*

$$H_Q \geq H_P + \frac{\beta S_P + \gamma C_P - \max(\beta, \gamma) R_0}{\alpha} \tag{5.7}$$

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

$$H_Q \geq H_P + \frac{\beta S_P + \gamma C_P - \beta S_Q - \gamma C_Q}{\alpha}. \tag{5.8}$$

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 (5.8) proves the theorem. □

Note that the right-hand side of (5.7) contains only known numbers. Therefore, the algorithm can compute a lower bound based on each evaluated candidate partition, and use the best one of the lower bounds determined so far. Unfortunately, there is no guarantee that the lower bound will not be far off the optimum. However, as shown in Chapter 6, the gap between the found best partition and the lower bound is not big for practical benchmarks.

5.6 Implementation and empirical results

5.6.1 Implementation

I implemented the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19) and the MFMC-based algorithm (Algorithm 6 on page 55) using the minimum cut algorithm of Goldberg and Tarjan [38, 25]. I had to modify the construction in the proof of Theorem 6 (page 18), which is the basis of Algorithm 1, slightly because the used minimum cut algorithm works on directed graphs.

Generally, in order to find the minimum cut in an undirected graph using an algorithm for directed graphs,[1] every undirected edge has to be replaced with 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. 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 in the terminology of Theorem 6 on page 18), only one copy is needed, directed to v_h, or from v_s, respectively.

The algorithms have been implemented in C. The detailed tests and comparisons with other algorithms are presented in Chapter 6. In the rest of this section, two aspects of the empirical results are presented which do not affect the other heuristics: the speed of the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19) on practical problem instances and the smoothness of the cost functions.

[1]There are also algorithms for finding the minimum cut in an undirected graph, which are even faster than the ones for directed graphs. However, what is needed here is a cut that separates two given vertices (a so-called $s-t$-cut), and for this problem, no faster algorithms are known for the undirected case.

5.6.2 Experience with Algorithm 1

Since Algorithm 1 finds the optimal solution for the **P5** problem, I only had to test its speed on practical problem instances (and not the cost of the found solution). Recall from Section 2.2 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, I 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 [6] has shown that this average is typical.

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 5.1: Running time of Algorithm 1

Table 5.1 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.

5.6.3 Smoothness

Another question that I addressed empirically is whether the two-dimensional search approach of Algorithm 6 is adequate. Recall from Section 5.2 that this search approach works well if the cost functions have a simple and smooth structure. Recall also from Section 5.3 that it can be proven that $H_{P(\alpha,\beta,\gamma)}$ is monotonously decreasing in α, but that the other conjectured monotonicity results do not hold.

On the other hand, the empirical results have shown that the structure of these functions is indeed smooth and simple, and they are 'almost monotonous.' A typical example can be seen in Figure 5.2 showing the hardware cost of the optimal partition in the **P5** problem for different values of the hardware weight α and communication weight γ (the software weight β was fixed to 100 in this example).

Notice that $H_{P(\alpha,\beta,\gamma)}$ is indeed monotonously decreasing in α, just as proven in Section 5.3. In contrast, it is not monotonous in γ. However, it is almost monotonous in the following sense: for small values of α, $H_{P(\alpha,\beta,\gamma)}$ is monotonously increasing in γ, whereas

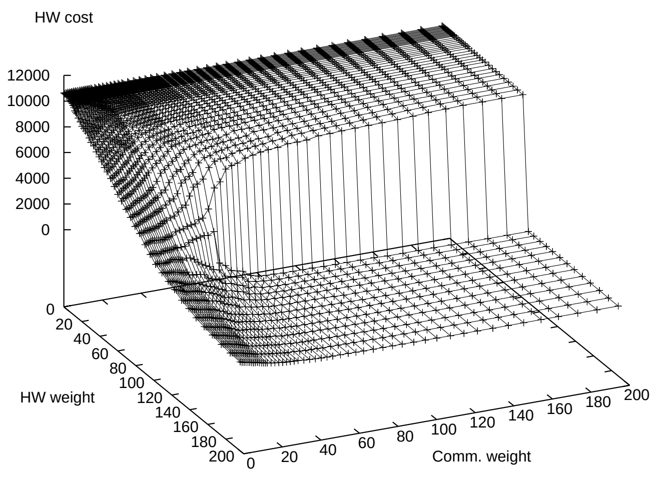


Figure 5.2: Hardware cost of the optimal partition in the **P5** problem as the function of the weights α and γ

for big values of α it is monotonously decreasing. Another interesting observation is that for small values of γ, $H_{P(\alpha,\beta,\gamma)}$ can be very precisely tuned by modifying α, whereas this is not possible for large values of γ. This is not surprising, since for large values of γ the problem is dominated by the edge costs, and thus it is highly non-linear. It is obviously a consequence of this fact, that the best (α,γ) point found by the MFMC-based algorithm typically has a relatively small γ coordinate.

In some test cases I also ran a modified version of the MFMC-based algorithm 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 also justifies the search strategy of the MFMC-based algorithm.

5.7 Extension possibilities

The MFMC-based algorithm essentially has two main components: (i) the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19) is used to generate candidate solutions; and (ii) these candidates are evaluated based on their merit concerning the **P3** problem to choose the best. Accordingly, the MFMC-based algorithm can be extended in two main ways:

- Extensions to the polynomial-time algorithm for the **P5** problem (Algorithm 1 on page 19). Algorithm 1 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 each edge e might be assigned a communication cost $c(e)$. 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 in the obvious way, then a similar auxiliary graph can be built as in the construction of Theorem 6 (page 18), 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 1 (page 19) remains true for this graph, hence this extended problem can be solved similarly as the original **P5** problem. This way, the whole MFMC-based algorithm can be easily extended to include more than three cost functions.

- Extensions to the evaluation of the candidate solutions. For instance, scheduling of the nodes can also be incorporated in the MFMC-based algorithm. 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 6. Of course, such an extension makes the algorithm more complicated and significantly slower.

Chapter 6

Comparison of the algorithms

This chapter aims at comparing the three presented heuristic algorithms:

- The GA, presented in Chapter 3
- The KL-type algorithm, presented in Chapter 4
- The MFMC-based algorithm, presented in Chapter 5

The main goal is to investigate under which circumstances which algorithm works best. Such an understanding of the performance characteristics is of course vital for the practical applicability of heuristic algorithms. However, the HSCD literature does not contain an unbiased, systematic treatment of this important issue.

The comparison is performed in two stages: first Section 6.1 provides a more theoretical comparison based on the analysis of the algorithms. Section 6.2 on the other hand complements this with empirical experience on the practical performance of the algorithms.

6.1 Analytical comparison

In this section, the proposed algorithms are analyzed and compared based on those aspects that can be deduced without empirical tests. Each of the next subsections details a different aspect.

6.1.1 Initial partitions

The MFMC-based algorithm does not need an initial partition, the KL-type algorithm starts from one initial partition, whereas the GA starts with a set of initial partitions. It follows that the performance of the KL-type algorithm depends very much on its initial partition, much more than the GA. In the case of the GA, the aim is not to create one very good initial partition, but rather to generate many different partitions, from which the genetic operations are later able to mix much better ones.

This is also the reason why I used two different methods to generate initial partitions for the KL-type algorithm and the GA, respectively. The algorithm that is used to generate the initial partition for the KL-type heuristic (Algorithm 5 on page 44), aims at finding a valid partition, which is almost invalid, i.e., its R_P value is very near to the limit. This is because typically, good partitions place as many nodes as possible into software in order to decrease hardware costs; hence the best solutions typically lie near the limit. In contrast, the algorithm that generates the random valid individuals for the initial population of the GA (Algorithm 3 on page 26), does not aim at getting as near as possible to the real-time constraint. This is simply not necessary because there are also many invalid individuals in the population, so that the genetic operations will be able to generate individuals which are near the limit.

On the other hand, the fact that the KL-type algorithm depends so heavily on its initial partition, combined with the experience that it is a very fast algorithm, makes it promising to use it as a post-optimization stage in combination with any other algorithm: the resulting partition of the other algorithm can be used by the KL-type algorithm as initial partition, so that it may further improve it.

6.1.2 Stopping

The stopping criterion of the KL-type algorithm and the MFMC-based algorithm is quite clear: the KL-type algorithm ends when a pass fails to find a better solution, and the MFMC-based algorithm ends when it finished searching the set of meaningful α, β, γ values. In the first case, it does not make sense to go on since all subsequent passes would yield the same result; in the second case, it is simply not possible to continue the algorithm.

However, in the case of the GA, the stopping criterion is much more arbitrary. Specifically, the GA is stopped when there has been no improvement recently. However, this does not mean that the algorithm could not have run further. Actually, it can be proven that under quite mild conditions genetic algorithms converge to the optimum [55]. That is, if the algorithm runs infinitely long, it reaches the optimum with probability 1.

The other two algorithms do not possess this property because they only scan a well-defined subset of the whole search space and the optimum is not guaranteed to be in that subset. However, the KL-type algorithm can be extended to possess it by running it again and again, every time with a new random initial partition.

However, it should be noted that even if the GA or the KL-type algorithm reaches the optimum, it will not recognize this fact. This is in contrast with the MFMC-based algorithm which always has—based on the lower bounds it produces—an estimate of how far it is from the optimum.

Let us also consider the other extreme, in which we have very little time, and thus cannot wait until the end of the algorithms. Fortunately, all three algorithms maintain a best-so-far partition, so that all of them can be stopped prematurely, and yet they yield a valid partition, the quality of which gets only better if more time is allowed. (Such algorithms are often called anytime algorithms because they can be stopped at any time.)

6.1.3 Parallelization

What if an unlimited set of processors is available? Is it possible to significantly accelerate the algorithms?

In the case of the GA, the answer is clearly yes. Most of the time the algorithm performs recombinations and fitness computations. Since these can be done independently for the individuals, the algorithm can be very well parallelized.

In the case of the MFMC-based algorithm, parallelization is also possible, but it requires slight changes to the algorithm. Most of the time is spent in the calls to Algorithm 1, which are basicly independent of each other, and can thus be done in parallel. The only problem is the search strategy of the algorithm, which adjusts the increments $d\alpha$ and $d\gamma$ dynamically, based on the last result. Thus, the next point to consider depends on the last result, and so there is some dependence between the calls to Algorithm 1. However, this is not a real problem, because it is of course allowed to evaluate more (α, γ) points. If really an unlimited set of processors is available, then the easiest solution is to evaluate all (α, γ) points of the grid defined by $d\alpha$ and $d\gamma$ (without ever incrementing $d\alpha$ and $d\gamma$). These pairs can then be evaluated independently. If, on the other hand, the number of processors is limited, then the region to be scanned should be divided into sub-regions of approximately the same size, each sub-region should be mapped to one of the processors, and scanned using the original search strategy.

In the case of the KL-type algorithm, parallelization is rather problematic.[1] It is inherently a serial algorithm because every partition that the algorithm evaluates is computed from the previous one. The only possibility for parallelization is offered by the gain recomputations, because these are independent of each other. As discussed in detail in Section 4.9 and Section 4.10, either all gains have to be recomputed after a move, or just the gain of the neighbors of the lastly moved node. In the latter case, the algorithm can be hardly accelerated, since typical communication graphs are sparse, and thus only a small number of gain recomputations take place after a move. However, if the cost metrics or the gain function is such that all gains have to be recomputed after each move, then the algorithm can be accelerated by a factor of roughly n if a sufficient number of processors is available.

6.1.4 Stability

By stability, I mean the property of not 'forgetting' already visited promising parts of the search space and being able to return to them [96]. A typical example is the GA: since it maintains a whole set of solutions at the same time, it can also evaluate different regions of the search space simultaneously. Thus, it does not have to give up its position in one part of the search space in order to evaluate another part. Rather, the population provides a kind of 'memory' in which the best positions of the search space can be remembered.

[1]Of course, even the KL algorithm can be parallelized by using multiple runs starting from different initial partitions. That is, each processor runs its own KL, and at the end the overall best result is taken. However, this is not an intrinsic property of the KL algorithm: every randomized heuristic can be parallelized this way.

The same applies to the MFMC-based algorithm as well, because it *systematically* scans its—limited—search space.

However, the KL-type algorithm does not possess this property. This is typical generally for algorithms based on local search. Such algorithms are 'memoryless': it is possible that they follow a seemingly promising trace, lose their starting point meanwhile, and thus fail to find the best solution for which another path should have been followed. The situation is slightly better in the case of my KL-type algorithm than generally with local search algorithms, because my algorithm maintains two partitions (P_{curr} and P_{best}), thus it has some very limited memory, but much less than the GA.

6.1.5 Extension possibilities

Although this aspect has already been mentioned in the description of every algorithm, I would like to summarize which extensions are supported by which algorithm:

- Some nodes prescribed to be in software, some others prescribed to be in hardware: all three algorithms can be easily extended.
- More than two cost functions: all three algorithms can be extended.
- Non-linear cost functions (this includes scheduling, hardware sharing etc.): all three algorithms can be extended. However, such extensions typically make the algorithms significantly slower.
- Multiway partitioning: the GA and the KL-type algorithm can be easily extended. In the case of the MFMC-based algorithm, this is not straight-forward. The construction of Theorem 6 (page 18) can be generalized for multiway partitioning as well, but the resulting problem (minimum multiway cut) is $\mathcal{NP}$-hard. However, there are effective approximation algorithms for this problem [63], which can be used instead of the conventional minimum cut algorithm.
- More than one constraint (example: more than one use case of the system; a separate real-time constraint is associated with each use case): all three algorithms can be extended.

Moreover, each of the three algorithms can also be easily adapted to the **P2** or **P4** problems.

6.1.6 Tuning

The MFMC-based algorithm has hardly any parameters. The GA and the KL-type algorithm on the other hand have many parameters with which their performance can be tuned. This can be both an advantage and a disadvantage. It is advantageous because the algorithm can be fine-tuned for each problem instance to yield the best result. However, this tuning process is very tedious and time-consuming, and thus it is questionable whether it can be expected from the designer that he or she should always perform it.

6.1.7 Generating candidate partitions

Each of the three algorithms works by evaluating a number of partitions and choosing the best one from them.[2] They differ only in the way they generate the candidate partitions to consider. The time spent for this can be regarded as the overhead of the algorithms. This overhead has to be paid for the intelligent choice of candidate partitions.

In the case of the GA, this overhead is incorporated by the genetic operations, from which recombination is the most powerful and also the most time-consuming. But recombination is still very fast, because it can be done in $O(n)$ time. It is a problem-independent operation that necessarily generates many low-quality individuals as well.

The mechanism to generate the next partition is more problem-specific in the case of the KL-type algorithm. Thus it can be assumed that it will less often generate low-quality partitions. In the case of the MFMC-based algorithm, this mechanism is much more sophisticated than in the other two cases, and it is also more time-consuming. Thus the MFMC-based algorithm evaluates typically less partitions in a given time frame than the other two algorithms.

6.1.8 Relative importance of the cost metrics

Both in the case of the GA and the KL-type algorithm it was an important decision how the hardware cost of a partition and the measure of its exceeding of R_0 should be combined into a single metric. This meant in particular that the relative importance of these two factors had to be determined. The best solution was in both cases to allow the exceeding of R_0 but to penalize it. However, the penalty function in the two cases was quite different. The reason of the difference is similar to the one discussed in Section 6.1.1. That is, the goal of the penalty function in the case of the KL-type algorithm is to not allow the algorithm to get far from the valid region of the search space, because it would be hard to get back, and thus computational power would be wasted in an unpromising part of the search space. Therefore a steep, highly non-linear penalty function is used. In the case of the GA, it is not a problem if some individuals are far from the valid region because they will simply not survive.

In the case of the MFMC-based algorithm, the whole question did not appear. In this case, non-valid partitions are simply discarded, even if they contain 'valuable patterns.' The algorithm simply does not have a mechanism for transferring such patterns to later solutions.

6.1.9 Determinism

The GA is definitely a random algorithm. The KL-type algorithm is also randomized if it starts from a random partition. The MFMC-based algorithm on the other hand is a

[2]While this may sound trivial, it is not necessarily true for all possible partitioning algorithms. For instance an algorithm based on the branch-and-bound paradigm would spend most of its time working with partial solutions.

deterministic algorithm.

Deterministic algorithms have the advantage of reproducible results. In contrast, randomized algorithms have the advantage that they can be run several times, and the best result can be taken. This way, if sufficient time is available, the result of random algorithms can be improved.

6.2 Empirical comparison

In this section, the proposed heuristics are compared based on empirical experience gained from running them on numerous benchmark problems.

6.2.1 The benchmarks

In order to have a representative mix of benchmark problems, I used three different sources:

- The MiBench benchmark suite [42]
- The own designs of our research group
- Large random graphs

The characteristics of the test cases are summarized in Table 6.1. 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. This is necessary because the performance of an algorithm is usually evaluated as a function of the length of the input. I calculated the size as $2n+3m$ 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.

It has to be noted that most previous algorithms in the literature were tested on graphs with only some dozens of vertices, like the crc32, patricia, or dijkstra benchmarks. The next five benchmarks (segment, fuzzy, rc6, mars, and ray) are significantly larger, and they are typical of current industrial problems. However, the systems to be designed become more and more complex, and therefore I also included some really large random benchmarks as well, which are above today's typical problem sizes. Note also that the graphs corresponding to the real designs are sparse, i.e., they have few edges. The densest is the dijkstra example with $m/n = 2.73$, and the sparsest is the rc6 benchmark with $m/n = 1.39$. Therefore, the random graphs were also generated with similar m/n ratios.

In order to illustrate the non-trivial nature of the partitioning problem, Figure 6.1 depicts the communication graph of the segment benchmark. The plot was created with the automatic graph-drawing tool `dot` [36]. For the sake of visibility, the cost values are not shown.

In the case of our own designs, all cost values were available. However, in the case of the benchmarks from MiBench only a software implementation was available,[3] thus the

[3]In these cases, the communication graphs were generated from the available C programs with a tool developed by our research group.

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.
segment	150	333	1299	Image segmentation algorithm in a medical application.
fuzzy	261	422	1788	Clustering algorithm based on fuzzy logic.
rc6	329	448	2002	RC6 cryptographic algorithm.
mars	417	600	2634	MARS cipher.
ray	495	908	3714	Ray-tracing algorithm for volume visualization.
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 6.1: Summary of the used benchmarks

software costs could be determined using profiling, but the other cost values were not available. And of course in the case of the random graphs, no cost values were available at all.

Therefore, I made use of the following methodology to generate the missing cost values:

- 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

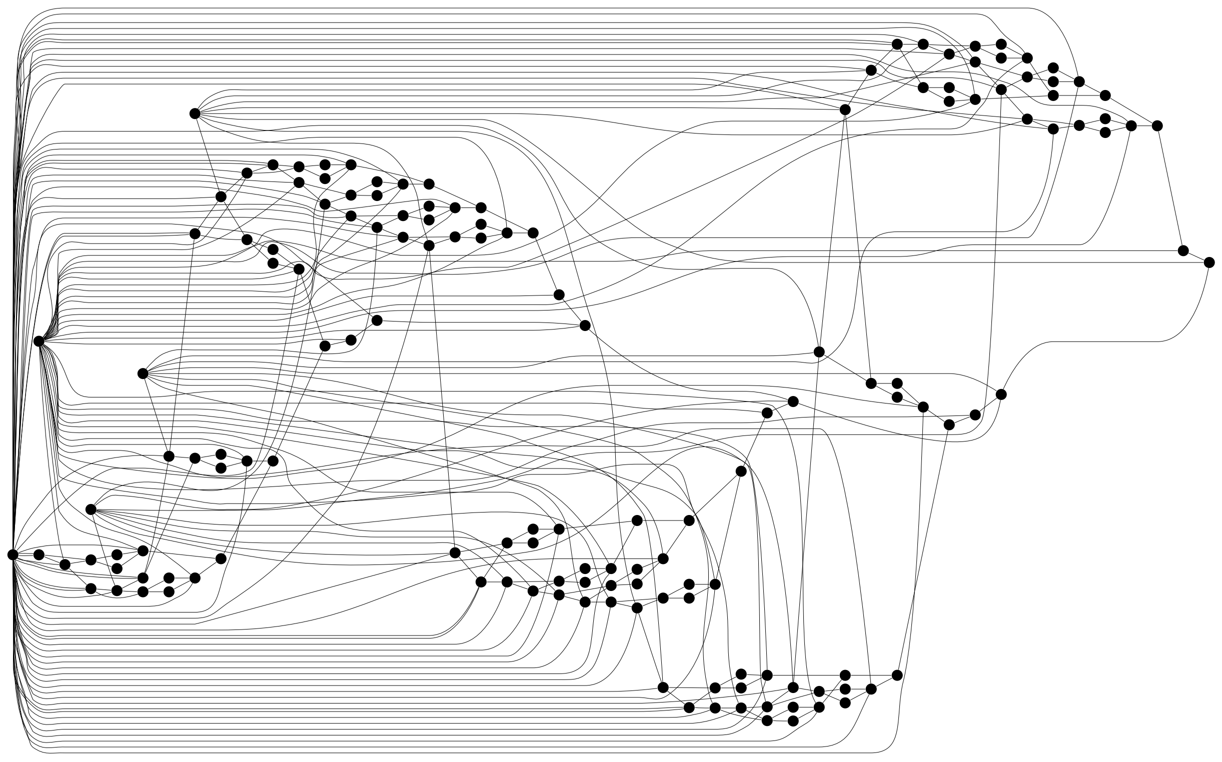


Figure 6.1: The communication graph of the segment benchmark

defined by the value of λ, between a node's hardware and software costs. (When $\lambda = 0$, then $h_i = \kappa s_i$ will hold for each node. But when λ is higher, the h_i values can deviate from the respective κs_i values, and thus the correlation becomes lower.) This corresponds to the fact that more complicated components tend to have both higher software and higher hardware costs. I 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.

- Where communication costs were not available, they 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). I tested two different values for μ: 1 (computation-intensive case) and 10 (communication-intensive case).

- Finally, the limit R_0 can be arbitrarily defined for every benchmark. 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. I tested two types of 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).

6.2.2 Optimal solutions

It would make the test results more informative if the results of the heuristics could be compared to the optimal solution of the problems. However, since **P3** is an $\mathcal{NP}$-hard problem, it is not likely that the optimum of large problem instances can be found within reasonable time.

I used an exact algorithm based on integer linear programming (ILP), which was presented in [6], to try to solve at least the smaller problems optimally. As expected, the ILP-based algorithm could solve the three smallest benchmark problems quite quickly, and in a couple of hours, it could even solve some of the bigger ones as well. It could not solve the ray problem within one day, and also some configurations of the mars problem proved to be too hard to solve optimally in this period of time. Of course, none of the large random instances could be solved optimally within a day of computation time.

To sum up: for the benchmarks up to the rc6 problem, the results of the heuristics can be compared to the optimum, but for bigger problems the optimum is not known. In the latter case, the best alternative is to compare the results to the lower bound generated by the MFMC-based algorithm.

6.2.3 Test configuration

All algorithms were implemented in C and compiled and linked using `gcc` v3.2. The tests were performed on a PC with a 400MHz PII Celeron processor, 128 kB cache, and 128 MB main memory. The operating system was SuSE Linux 8.1 with Kernel 2.4.19-4GB. For time-related measurements, GNU `time` v1.7 was used.

6.2.4 Results

Cost of the found solution

In the first set of experiments, I compared the cost of the solution found by the three algorithms. Note that in the case of the GA and the KL-type algorithm, the best configuration was used, as described in Section 3.3 and Section 4.11, respectively. The results are summarized in Figures 6.2–6.5. I included four plots because I tested two different values for R_0 and two different values for the CCR. I also tested two different values for λ but I found that the change of λ had no significant impact on the relative performance of the algorithms.

Note that the y axis of the figures represents the cost of the found solution, therefore lower values are better. Based on the results, the following observations can be made:

- For relatively small graphs, all three heuristics yield equal or very similar results, regardless of the parameter settings. However, for bigger graphs, there are clear differences between the algorithms, and these are dependent on the parameters (CCR and R_0).

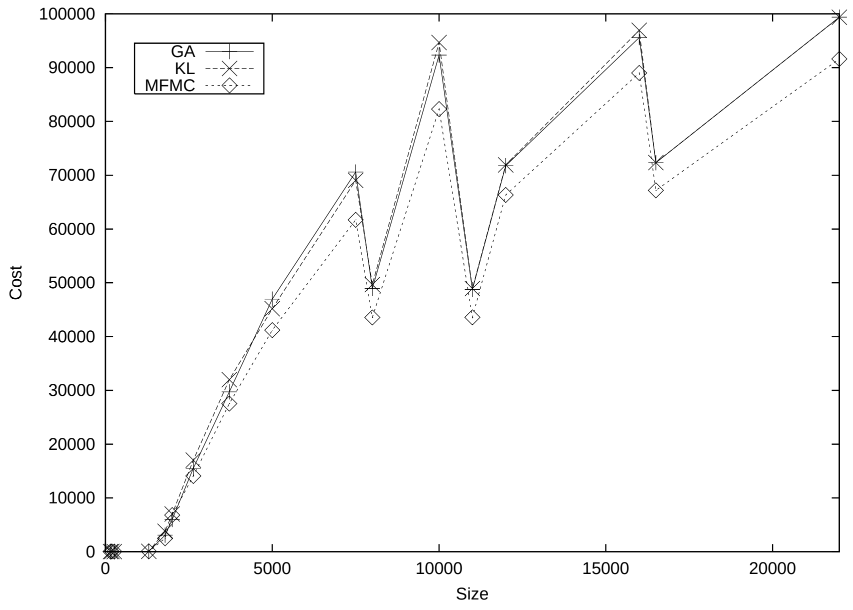


Figure 6.2: Cost of the solution found by the three algorithms. CCR=low, R_0=low

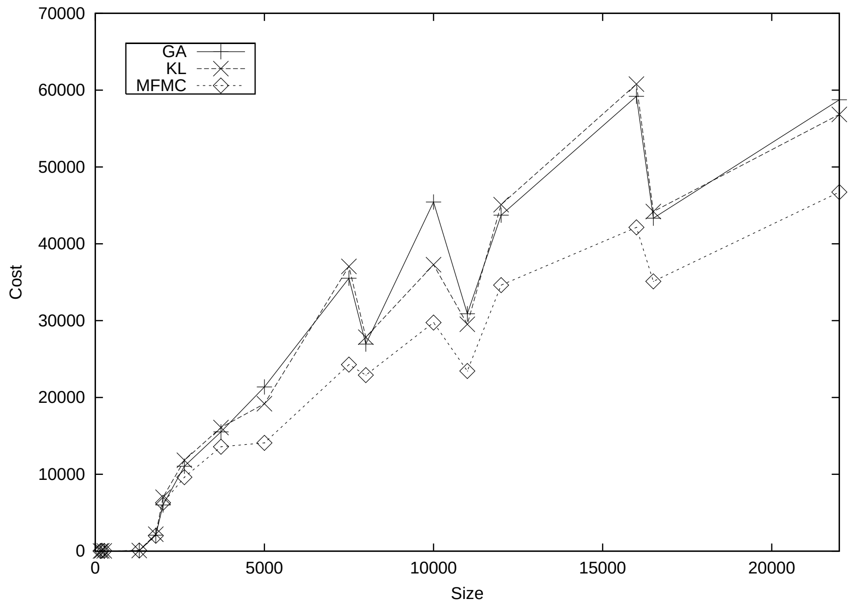


Figure 6.3: Cost of the solution found by the three algorithms. CCR=low, R_0=high

- In the low-CCR cases (Figures 6.2 and 6.3), the MFMC-based algorithm clearly offers the best results. Its advantage is bigger in the R_0=high case (Figure 6.3). Here, the difference was about 19% for the largest graphs. The performance of the GA and the

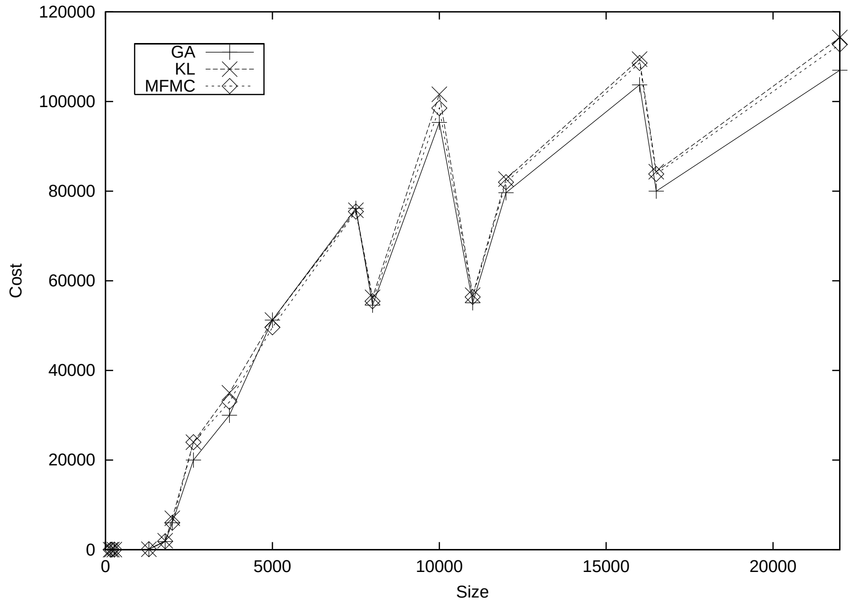


Figure 6.4: Cost of the solution found by the three algorithms. CCR=high, R_0=low

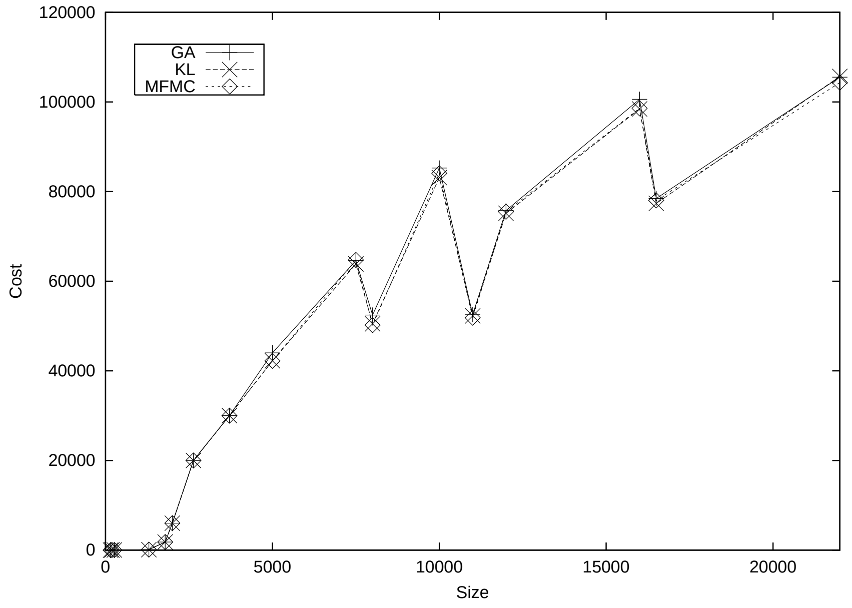


Figure 6.5: Cost of the solution found by the three algorithms. CCR=high, R_0=high

KL-type algorithm is similar.

- In the high-CCR cases (Figures 6.4 and 6.5), the difference between the algorithms is smaller. 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). In particular, in the CCR=high, R_0=high case (Figure 6.5), there is no clear winner. In the CCR=high, R_0=low case (Figure 6.4), however, clearly the GA performed best. Its result was about 6% better than the ones of the other two algorithms for the largest graphs. The other two algorithms performed very similarly.

Running time

The settings for CCR and R_0 did not significantly impact the running time of the algorithms, hence only one plot is shown (Figure 6.6). The following observations can be made:

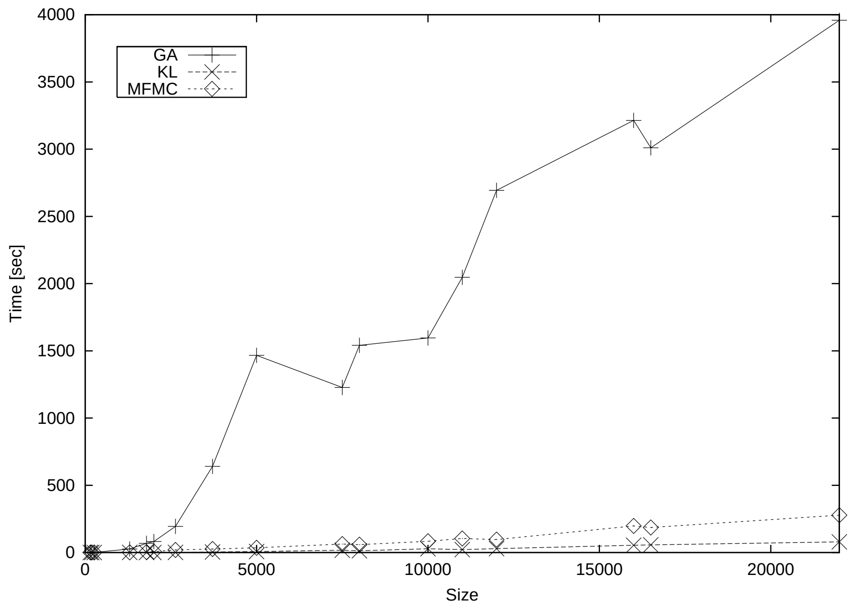


Figure 6.6: Running time of the three algorithms

- Again, for relatively small graphs, the speed of the algorithms is comparable. However, for bigger graphs, KL and the MFMC-based algorithm are much faster than GA, and the difference keeps growing with bigger graphs. For the biggest graphs, GA is about 15 times slower than the MFMC-based algorithm.
- 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-type algorithm. On average, it was about 3 times faster than the MFMC-based algorithm.

- The speed of both the KL-type algorithm and the MFMC-based algorithm is clearly acceptable because both could solve even the biggest problems in a couple of minutes, and the smaller ones in a couple of seconds. The higher running time of the GA might cause problems in some environments. Notice though that for the smaller benchmarks, the GA, too, was quite fast. It is safe to assume that in a couple of years, when it will be necessary to partition graphs of several thousand nodes, the computers on which the heuristics are run will also be significantly faster, so that the GA is also fast enough for most applications.

Quality/speed trade-off

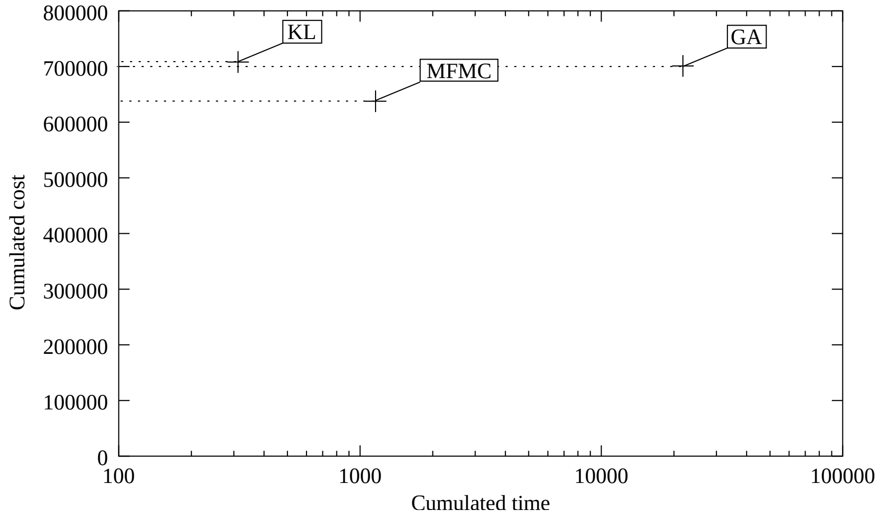


Figure 6.7: Cost/speed trade-off of the three algorithms. CCR=low, R_0=low

The presented measurement results concerning result quality and running time can be combined into more informative diagrams. Figures 6.7–6.10 show the cost/speed tradeoff of the three algorithms. Each algorithm is represented by a point in the cost–speed plane. The coordinates of this point were calculated by adding the running time of all the runs, and the cost of the result of all the runs, respectively, of the given algorithm. Note that on both axes smaller values are better. Note also the logarithmic scale on the 'time' axis.

Again, four plots are presented according to the settings for CCR and R_0. These diagrams clearly show the characteristics of the different algorithms, which can be summarized as follows:

- The KL-type algorithm is always the fastest, and as such, it is in all cases a Pareto-optimal choice. (This means that none of the other algorithms dominate it, i.e., none of the other algorithms offers both better speed and better result cost.) Concerning the cost of the resulting solution, it is usually not the best. However, in

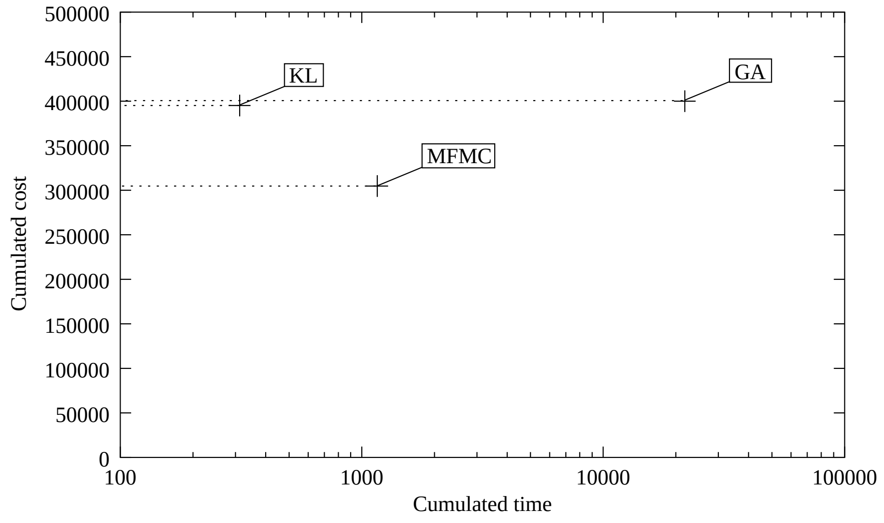


Figure 6.8: Cost/speed trade-off of the three algorithms. CCR=low, R_0=high

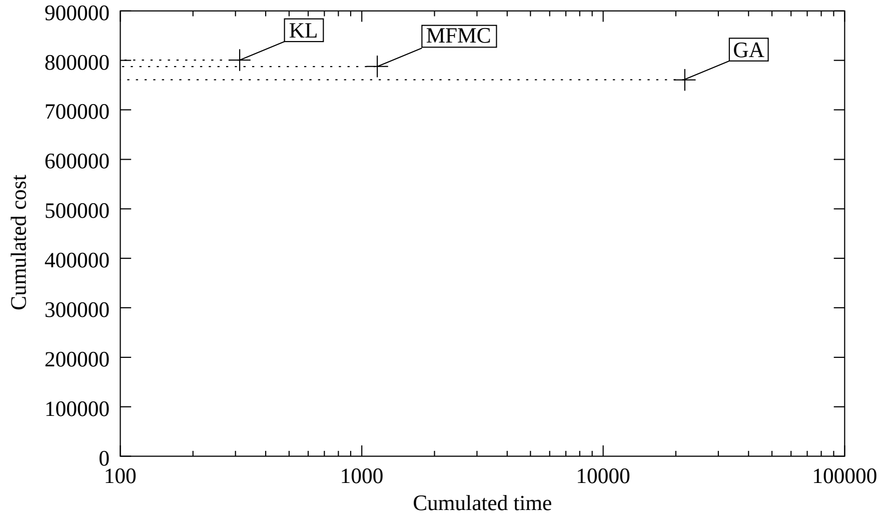


Figure 6.9: Cost/speed trade-off of the three algorithms. CCR=high, R_0=low

the CCR=high, R_0=high case (Figure 6.10), where all of the three algorithms offer approximately the same result quality, the KL-type algorithm is the rational choice because of its low running-time.

- The MFMC-based algorithm offers the best result quality in the low-CCR cases (Figures 6.7 and 6.8), and therefore it is a Pareto-optimal choice in these cases. Note that it is also Pareto-optimal in the CCR=high, R_0=low case (Figure 6.9) because it offers better result quality than the KL-type algorithm (although it is slower) and it offers lower running time than the GA (although its result quality is worse). In the

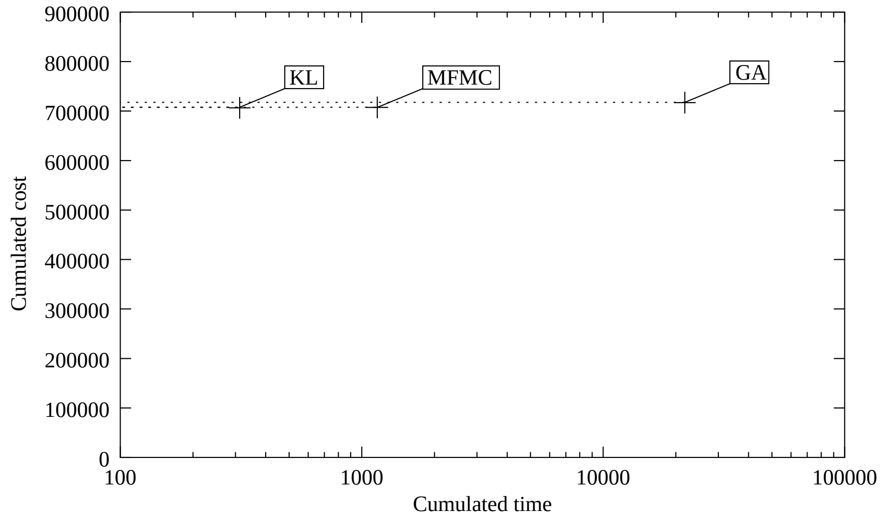


Figure 6.10: Cost/speed trade-off of the three algorithms. CCR=high, R_0=high

CCR=high, R_0=high case (Figure 6.10), it is dominated by the KL-type algorithm because of the difference in running time.

- In the low-CCR cases (Figures 6.7 and 6.8) the GA is dominated by the MFMC-based algorithm. In the CCR=high, R_0=high case (Figure 6.10), it is dominated by both the KL-type and the MFMC-based algorithm. However, in the CCR=high, R_0=low case (Figure 6.9) the GA offers the best result quality, and thus it is Pareto-optimal.

It should be noted that the presented results on the effect of CCR and constraint tightness on the relative performance of the algorithms is unprecedented in the HSCD literature. Most papers on hardware/software partitioning try to convince the reader that a given algorithm is better than all other approaches. However, my results show a more sophisticated picture: depending on some key parameters of the problem as well as the relative importance of algorithm speed vs. result quality, different algorithms should be chosen. This is vital information for the practitioners of hardware/software partitioning. Also novel is the kind of diagrams used in Figures 6.7-6.10 that enable the compact visualization of the relative performance of the algorithms.

Lower bound

I also investigated the gap between the result of the MFMC-based algorithm and the lower bound that it produces. Again, the results are shown in four diagrams (Figures 6.11–6.14). The following can be observed:

- For small graphs, the difference between the result found by the algorithm and the lower bound is very small. This is true for all four configurations. This proves both the high quality of the found solution and that of the lower bound.

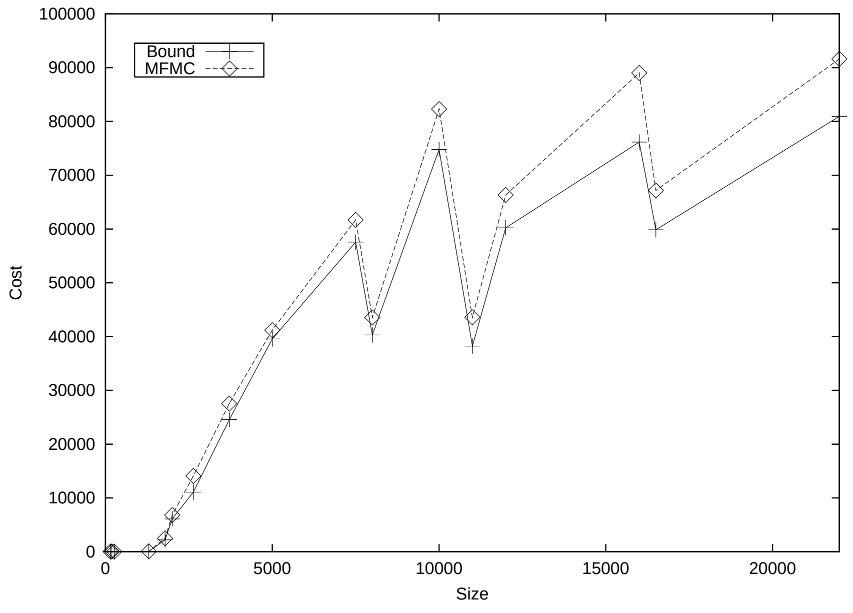


Figure 6.11: Lower bound vs. result of MFMC-based algorithm. CCR=low, R_0=low

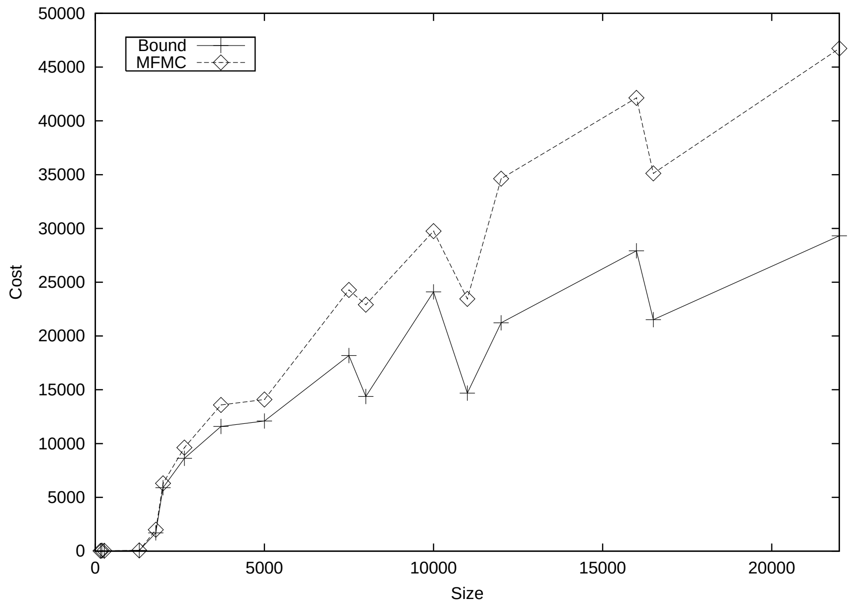


Figure 6.12: Lower bound vs. result of MFMC-based algorithm. CCR=low, R_0=high

- With bigger graphs, the gap clearly grows. This is not surprising since the algorithm is only a heuristic for a tough problem.
- The speed by which the gap grows is quite different between the four configurations.

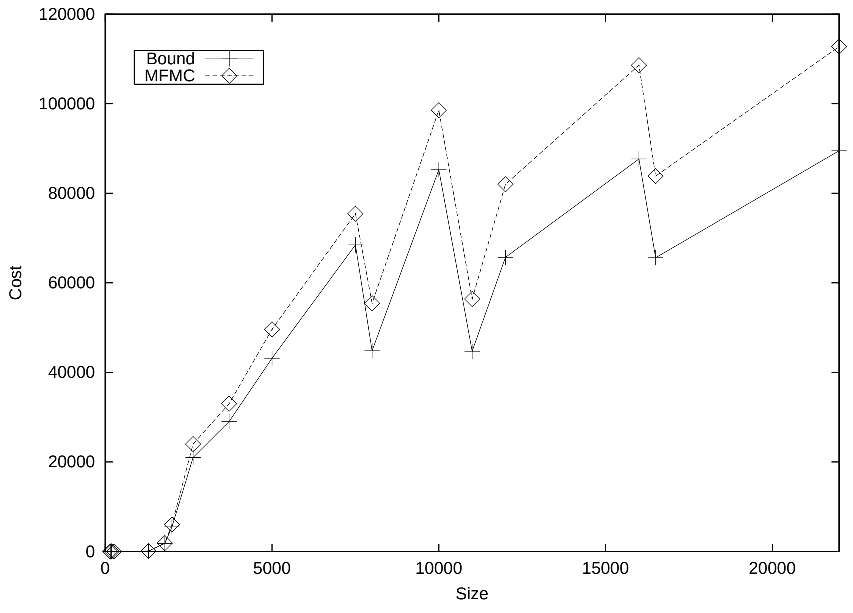


Figure 6.13: Lower bound vs. result of MFMC-based algorithm. CCR=high, R_0=low

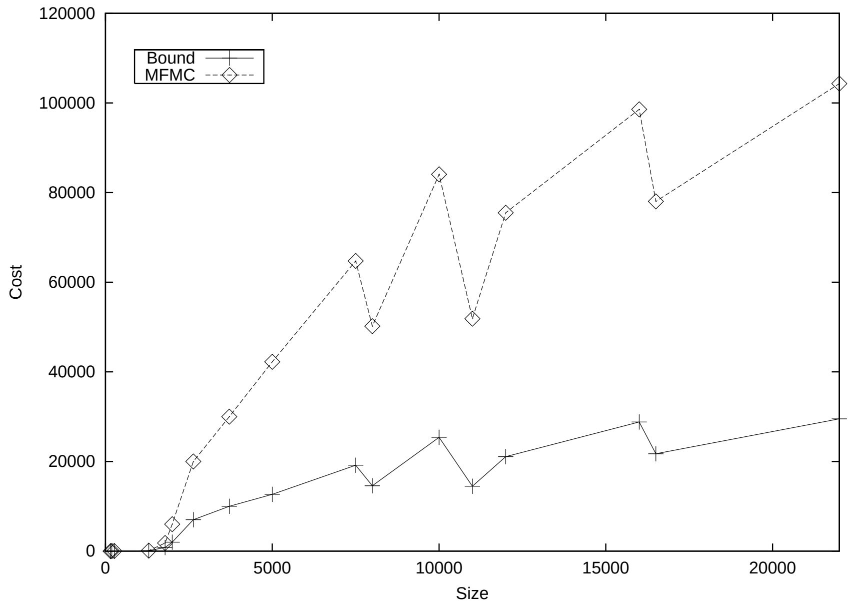


Figure 6.14: Lower bound vs. result of MFMC-based algorithm. CCR=high, R_0=high

In the R_0=low cases (Figures 6.11 and 6.13) the gap grows quite slowly. In particular, in the CCR=low, R_0=low case (Figure 6.11), the gap is only about 12% even for the biggest graph. On the other hand, the gap grows quite rapidly in the R_0=high cases

(Figures 6.12 and 6.14). In particular, in the CCR=high, R_0=high case (Figure 6.14), the gap is about 72% for the biggest graph.

It would be very interesting to know if this big gap is due to poor result quality or poor lower bounds. In other words: it would be interesting to know where the optimum lies. However, with the techniques currently available, it is not possible to answer this question for those graphs for which it would be interesting.

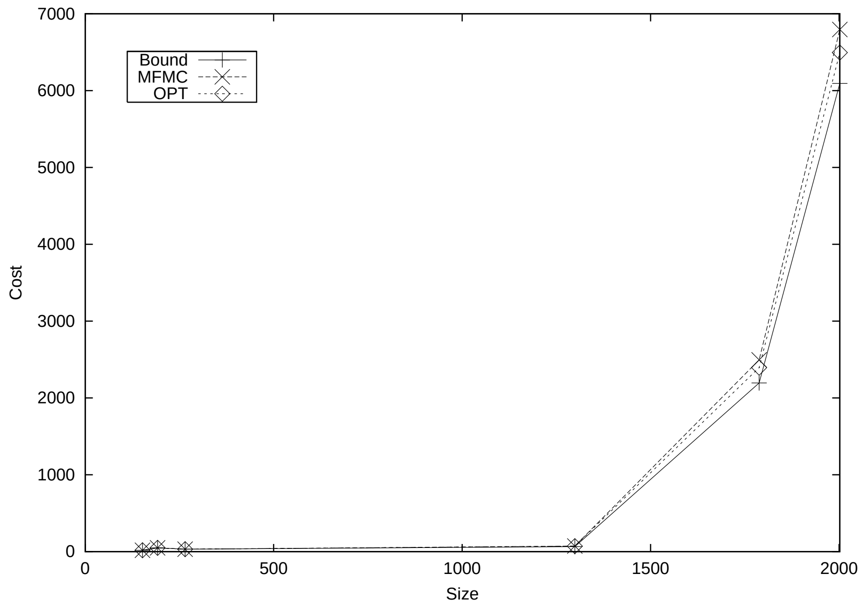


Figure 6.15: Lower bound vs. result of MFMC-based algorithm vs. the optimum

For the smaller graphs, I compared the lower bound and the result of the algorithms also with the optimum. The results are shown in Figure 6.15. Only one diagram is given because the curves are similar for all four configurations. The following can be observed:

- For the three smallest benchmarks all three algorithms find the optimum, and the lower bound also equals the optimum.
- For the three bigger benchmarks for which the exact algorithm can still produce results in acceptable time, the gap between both the result of the algorithm and the optimum and between the optimum and the lower bound grows. Both gaps grow very slowly, but the gap between the optimum and the lower bound grows a little faster. Whether this also holds for bigger graphs is not known.

Chapter 7

Applications

This chapter describes in which practical settings the presented algorithms have already been applied.

The results of the first thesis (the complexity results) found mainly indirect applications: the polynomial-time exact algorithm of Section 2.2 became the basis for the MFMC-based heuristic algorithm presented in Chapter 5; the hardness results helped to better understand the problem and lead my attention to the presented heuristics.

My heuristic partitioning algorithms have been applied in the following contexts:

- In an exact partitioning algorithm based on branch-and-bound [13]. That algorithm, which I will refer to as BB, is able to optimally solve partitioning problems of several hundred nodes in reasonable time. It also significantly outperforms the similar ILP-based algorithm. In BB, my genetic algorithm and the MFMC-based algorithm have been applied as pre-optimization heuristics: they generate initial solutions and thus initial upper bounds for the branch-and-bound algorithm. As demonstrated in [13], both of these algorithms are very useful for decreasing the running time of the BB algorithm without sacrificing optimality.

  Moreover, the polynomial-time algorithm for the **P5** problem also has an important role in the BB algorithm: it is used to determine whether a given partial solution can be extended to a valid complete solution with lower cost than the best one found so far. Thus, this algorithm helps in cutting off unpromising parts of the search space and so improving the performance of the BB algorithm.

- In a project funded by Akita Industrial Development Center (Japan), our research group participated in the development of some medical applications. Our task was to create efficient implementations for several neural algorithms using hardware/software co-design. The neural algorithms included Kohonen's self-organizing map, an image segmentation algorithm, and a clustering algorithm based on fuzzy logic. The most important step in the implementation of these algorithms was to find a good trade-off between the conflicting requirements on chip size, cost, and performance. In the first phase of the project, this was done manually, but as later the presented partitioning algorithms became ready, this work was substantially eased [8].

- In the EasyComp project, funded by the European Union, our research group participated in the development of new component-based system design methodologies. In particular, we developed a component-based process for hardware/software co-design [9]. The main idea of this methodology is to compose complex systems out of abstract building blocks in an intuitive way using a graphical tool. In order to cope with design complexity, implementation-specific details of the components are hidden from the designer. In particular, the components may have more than one implementation, for instance a hardware implementation and a software implementation; our tool shields the designer from such low-level issues. This means in particular, that partitioning decisions have to be made automatically. This is where the heuristic partitioning algorithms presented in this work are used [11].

As can be seen from the above, my partitioning algorithms have proven to be useful in practical applications: they made our work in other projects significantly easier, and I hope that in future they will also help others design more powerful and more cost-effective electronic systems.

Acknowledgments

I am grateful to my supervisor Péter Arató and my colleague András Orbán for supporting my work. Furthermore, I would like to thank Viktor Farkas, Sándor Juhász, Tibor Kandár, Dávid Papp, András Recski, and Áron Sisak for their valuable help.

For financial support, I would like to thank the following organizations: Hungarian National Science Fund (OTKA grants T030178, T043329, T042559), European Union (IST 1999-14191), Akita Industrial Development Center, Timber Hill LLC.

Appendix A

Notations

$\emptyset$	The empty set
$(*)$-property	When moving a node, only its own gain value and the gain of its neighbors can change, all other gain values remain unchanged.
α	The weight of the hardware cost in T_P
β	The weight of the software cost in T_P
$c(e), c_{i,j}$	Communication cost of edge e or edge (v_i, v_j)
C_P	The communication cost of partition P, i.e., the sum of the communication costs of the edges in E_P.
CCR	Communication-to-computation ratio
D_f	Domain of function f
$d\alpha$	The increment of α, used in the MFMC-based algorithm
$d\gamma$	The increment of γ, used in the MFMC-based algorithm
$\Delta H(v)$	The amount by which moving v to the other context increases H
$\Delta R(v)$	The amount by which moving v to the other context increases R
E_P	Given a partition P, E_P is the set of edges that connect vertices in different parts of P.
$exc(P)$	The percentage by which R_P exceeds the constraint, i.e., R_P/R_0
FIFO	The FIFO tie-breaking strategy means that, from the set of nodes with the highest gain, the one that was the first to enter this set is selected.
FM	Fiduccia–Mattheyses

$G=(V,E)$	A graph with vertex set V and edge set E
GA	Genetic algorithm
$gain(v)$	The basis for choosing the next node to move in the KL-type algorithm
γ	The weight of the communication cost in T_P
$h(v_i), h_i$	Hardware cost of node v_i
H_P	The hardware cost of partition P, i.e., the sum of the hardware costs of the vertices that are in hardware.
HSCD	Hardware/software co-design
ILP	Integer linear programming
KL	Kernighan–Lin
LIFO	The LIFO tie-breaking strategy means that, from the set of nodes with the highest gain, the one that was the last to enter this set is selected.
m	The number of edges in the graph
MFMC	Maximum flow, minimum cut
n	The number of nodes in the graph
N	Size of the population in the GA
p	The penalty function used in the KL-type algorithm
$P=(V_H,V_S)$	A hardware/software partition P; V_H is the set of nodes in hardware, V_S is the set of nodes in software
$P(\alpha,\beta,\gamma)$	The optimal partition in the **P5** problem with respect to the weights α, β, and γ
P_{best}	During the course of the KL-type algorithm, P_{best} stores the partition that has been the best so far.
P_{curr}	During the course of the KL-type algorithm, P_{curr} stores the current partition.
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^+$	The set of positive real numbers
$\mathbb{R}^n$	The n-dimensional real space

R_0	A real-time constraint, expressed as an upper bound on R_P
R_f	Range of function f
R_P	Shortcut for $S_P + C_P$. This is the running time of the system with respect to partition P if software costs represent running time and communication costs represent time penalty of communication between hardware and software.
$s(v_i), s_i$	Software cost of node v_i
S_P	The software cost of partition P, i.e., the sum of the software costs of the vertices that are in software.
$s-t$-cut	A cut of the graph that separates two given vertices (s and t)
T_P	The total cost of partition P, defined as $T_P = \alpha H_P + \beta S_P + \gamma C_P$.
v_i	The ith node of the graph
V_H	The set of nodes that are mapped to hardware
V_S	The set of nodes that are mapped to software

Appendix B

Basics of complexity theory

This chapter contains a brief introduction to the basic notions of complexity theory that are used in this dissertation. More details can be found, e.g., in [74].

B.1 Asymptotic growth rates

- $f(n) = O(g(n))$ means that $f(n)$ is at most order $g(n)$, that is, $f(n) \leq a + bg(n)$ for all $n \geq 0$ and some non-negative constants a and b.
- $f(n) = \Omega(g(n))$ means that $f(n)$ is at least order $g(n)$, that it, $g(n) = O(f(n))$.
- $f(n) = \Theta(g(n))$ means that $f(n)$ is exactly order $g(n)$, that is, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- $f(n) = \tilde{O}(g(n))$ means that $f(n)$ is—apart from logarithmic factors—at most order $g(n)$, that is, $f(n) = O(g(n) \log^k n)$ for some non-negative constant k.

B.2 $\mathcal{P}$ and $\mathcal{NP}$

An algorithm solves a problem in *polynomial time* if there is a polynomial g such that the algorithm can solve every instance of the given problem in at most $g(d)$ steps, where d is the length of the description of the given problem instance.

Let us now concentrate on decision problems. A decision problem has *yes-instances*, for which the output is 'yes,' and *no-instances*, for which the output is 'no.'

$\mathcal{P}$ denotes the set of decision problems that can be solved in polynomial time. These are the efficiently solvable problems. Unfortunately, there are many problems for which it is not known whether or not they are in $\mathcal{P}$.

$\mathcal{NP}$ denotes the set of decision problems for which the positive answer can be checked in polynomial time using some appropriate polynomial-size extra information (also called proof or witness). For instance, the problem whether a graph has a Hamiltonian cycle is in $\mathcal{NP}$, because, if the answer is positive, then the Hamiltonian cycle itself is a good proof for it. Indeed, a Hamiltonian cycle can be compactly represented, for example as a sequence of nodes, and it can be quickly verified that the given sequence of nodes is really a Hamiltonian cycle. Most of the naturally arising decision problems are easily shown to be in $\mathcal{NP}$.

It is obvious that $\mathcal{P} \subseteq \mathcal{NP}$ but it is not known whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \subsetneq \mathcal{NP}$. If $\mathcal{P} = \mathcal{NP}$ were true, this would mean that virtually all naturally arising problems could be solved in polynomial time. This is unlikely and therefore it is conjectured that $\mathcal{P} \subsetneq \mathcal{NP}$, which means that there are hard problems in $\mathcal{NP}$ that cannot be solved in polynomial time.

Candidates for this are the *$\mathcal{NP}$-complete* problems because they are the hardest problems in $\mathcal{NP}$. In order to define $\mathcal{NP}$-completeness, first the notion of *Karp-reduction* has to be clarified. Let P_1 and P_2 be two decision problems. A Karp-reduction of P_1 to P_2 is a polynomially computable function k such that for each problem instance x of P_1 the following holds: x is a yes-instance of P_1 if and only if $k(x)$ is a yes-instance of P_2. This means that, if we can solve P_2, then we can also solve P_1 with some little extra work: we calculate the function k for the given input, and then ask the 'oracle' for P_2. This is why it is called a reduction: in order to solve P_1, it is essentially sufficient to solve P_2.

If there exists a Karp-reduction of P_1 to P_2 then we say that P_1 can be reduced to P_2. An easy consequence is that, if $P_2 \in \mathcal{P}$ and P_1 can be reduced to P_2, then $P_1 \in \mathcal{P}$.

Now we can define $\mathcal{NP}$-completeness. A problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and every problem in $\mathcal{NP}$ can be reduced to it.

There are many problems that are proven to be $\mathcal{NP}$-complete. No polynomial-time algorithm is known for any of these problems; in fact, if any of these problems were in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$ would follow. Therefore it is conjectured that there is no polynomial-time algorithm for $\mathcal{NP}$-complete problems.

A problem is said to be *$\mathcal{NP}$-hard* if all problems in $\mathcal{NP}$ can be reduced to it. (That is, a problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and it is $\mathcal{NP}$-hard.) The notion of Karp-reduction can also be defined for optimization problems instead of decision problems. That is, it makes sense to say that a decision problem can be reduced to an optimization problem. This way, $\mathcal{NP}$-hardness can also be defined for optimization problems, and again, the $\mathcal{P} \subsetneq \mathcal{NP}$ conjecture implies that there are no polynomial-time algorithms for $\mathcal{NP}$-hard optimization problems.

In order to prove that a given problem is $\mathcal{NP}$-hard, it is not necessary to show that all problems in $\mathcal{NP}$ can be reduced to it. It is sufficient to show that one $\mathcal{NP}$-hard problem can be reduced to it.

B.3 Pseudo-polynomiality and strong $\mathcal{NP}$-hardness

In many problems, there are some numbers, and the problem is difficult only because these numbers can be exponentially high with respect to the length of the description of the input. For instance, the Knapsack problem is difficult because the numbers in it can be exponentially high. There are algorithms for the Knapsack problem that are polynomial with respect to these numbers. Such an algorithm is called *pseudo-polynomial*. In general, such algorithms have an exponential worst-case complexity (with respect to the length of the description of the input). However, in many practical cases, the numbers are not very high. If the numbers can be bound with a polynomial of the length of the description of the input, then such algorithms become polynomial.

On the other hand, there are problems that are structurally difficult, without any exponentially high numbers. An example is the optimization version of the Minimum Bisection problem, in which there are no numbers at all (the aim is here to cut the graph into two parts of equal size with a minimum number of cut edges). Another example is the weighted version of the same problem (the edges have weights, and the aim is to minimize the sum of the weights of the cut edges): here, there are numbers in the problem, but the problem is hard even if the numbers are small.

A problem is *$\mathcal{NP}$-hard in the strong sense* if it is $\mathcal{NP}$-hard even if the numbers in it are bound by a polynomial of the length of the description of the problem.

The $\mathcal{P} \subsetneq \mathcal{NP}$ conjecture implies that there are even no pseudo-polynomial algorithms for problems that are $\mathcal{NP}$-hard in the strong sense.

Appendix C

Problem definitions

C.1 Variants of the hardware/software partitioning problem

P1 Given the graph G with the cost functions h, s, and c, and $R_0 \geq 0$, $H_0 \geq 0$, decide whether there is a hardware/software partition P with $R_P \leq R_0$ and $H_P \leq H_0$.

P2 Given the graph G with the cost functions h, s, and c, and $H_0 \geq 0$, find a hardware/software partition P with $H_P \leq H_0$ that minimizes R_P among all such partitions.

P3 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.

P4 Given the graph G with the cost functions h, s, and c, and $S_0 \geq 0$, $H_0 \geq 0$, such that there are hardware/software partitions with $S_P \leq S_0$ and $H_P \leq H_0$, find a partition P with $S_P \leq S_0$ and $H_P \leq H_0$ that minimizes C_P among all such partitions.

P5 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.

C.2 Other problems

C.2.1 The KNAPSACK problem

There are n objects, each one has a weight w_i and a price p_i. There is a knapsack with a given capacity (i.e., limit on the weight that can be carried in it) W.

In the optimization version of the problem, the aim is to pick some objects so that they fit into the knapsack and their total price is maximal. Formally, the aim is to find a subset X of objects, so that $\sum_{i \in X} w_i \leq W$ and $\sum_{i \in X} p_i$ is maximal.

In the decision version of the problem, a minimum total price K is also given. The aim is to decide whether some objects can be picked so that they fit into the knapsack and their total price is at least K. Formally, the aim is to decide whether there exists a subset X of objects, so that $\sum_{i \in X} w_i \leq W$ and $\sum_{i \in X} p_i \geq K$.

C.2.2 The Minimum Bisection problem

We are given a graph $G = (V, E)$ with n vertices, where n is even, and m edges. A *bisection* is a bipartition of the graph into two parts with the same number of vertices. The cost of a bisection is measured by the number of edges that connect vertices in different parts of the bipartition (i.e., the number of cut edges).

In the optimization version of the problem, the aim is to find a bisection with minimum cost, i.e., with minimum number of cut edges.

In the decision version of the problem, a limit K is also given ($K \leq m$). The aim is to decide whether there is a bisection with at most K cut edges.

Bibliography

[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] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, in press.

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

[5] C. J. Alpert and A. B. Kahng. Recent developments in netlist partitioning: A survey. *VLSI Journal*, 19(1-2):1–81, 1995.

[6] 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.

[7] P. Arató, Z. Á. Mann, and A. Orbán. Genetic scheduling algorithm for high- level synthesis. In *Proceedings of the IEEE 6th International Conference on Intelligent Engineering Systems*, 2002.

[8] 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.

[9] P. Arató, Z. Á. Mann, and A. Orbán. Component-based hardware-software co-design. In *17th International Conference on Architecture of Computing Systems, Lecture Notes in Computer Science (LNCS 2981)*, pages 169–183. Springer-Verlag, 2004.

[10] 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.

[11] P. Arató, Z. Á. Mann, and A. Orbán. Extending component-based design with hardware components. *Elsevier Journal of Science of Computer Programming, Special Issue on New Software Composition Concepts*, 56(1-2):23–39, 2005.

[12] P. Arató, Z. Á. Mann, and A. Orbán. Time-constrained scheduling of large pipelined datapaths. *Journal of Systems Architecture*, accepted.

[13] P. Arató, Z. Á. Mann, and A. Orbán. Finding optimal hardware/software partitions. *IEEE Transactions on Computers*, submitted.

[14] P. Arató, T. Visegrády, and I. Jankovits. *High-Level Synthesis of Pipelined Datapaths.* John Wiley & Sons, Chichester, United Kingdom, 2001.

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

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

[17] 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.

[18] A. Basu, R. Mitra, and P. Marwedel. Interface synthesis for embedded applications in a co-design environment. In *11th IEEE International conference on VLSI design*, pages 85–90, 1998.

[19] M. de Berg, O. Schwarzkopf, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2nd edition, 2000.

[20] I. Zs. Berta and Z. Á. Mann. Smart cards—present and future. *Híradástechnika, Journal on C5*, (12):24–29, 2000.

[21] 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.

[22] J. Boyan and A. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2000.

[23] R. Camposano. From behaviour to structure: high-level synthesis. *IEEE Design and Test of Computers*, 10:8–19, 1990.

[24] 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.

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

[26] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, 2nd edition, 2001.

[27] 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.

[28] L. Davis. *Handbook of genetic algorithms.* Van Nostran Reinhold, 1991.

[29] 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.

[30] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. 2003 technology roadmap for semiconductors. *IEEE Computer*, 37(1):47–56, 2004.

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

[32] 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.

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

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

[35] D. Gajski. *High-Level Synthesis.* Kluwer Academic Publishers, 1992.

[36] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[37] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):216–227, 1980.

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

[39] 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.

[40] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Stanford University, December 1993.

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

[42] 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.

[43] L. Hagen, J. H. Huang, and A. B. Kahng. On implementation choices for iterative improvement partitioning algorithms. *IEEE Transactions on CAD*, 16(10):1199–1205, 1997.

[44] 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.

[45] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, Boston, MA, 1997.

[46] A. G. Hoffmann. The dynamic locking heuristic – a new graph partitioning algorithm. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 173–176, 1994.

[47] X. Hu, T. Zhou, and E. Sha. Estimating probabilistic timing performance for real-time embedded systems. *IEEE Transactions on VLSI Systems*, 9(6), 2001.

[48] 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.

[49] A. K. Jain and R. C. Dubes. *Algorithms for clustering data.* Prentice Hall, 1988.

[50] 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.

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

[52] 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.

[53] 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.

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

[55] W. Kinnebrock. *Optimierung mit genetischen und selektiven Algorithmen.* Oldenburg, 1994.

[56] 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.

[57] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[58] S. J. Krolikoski, F. Schirrmeister, B. Salefski, J. Rowson, and G. Martin. Methodology and technology for virtual component-driven hardware/software co-design on the system level. In *ISCAS*, 1999.

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

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

[61] 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.

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

[63] Z. Á. Mann. Metric-based approximation algorithms for graph cut problems. Master's thesis, Eötvös Loránd University, 2004.

[64] Z. Á. Mann and K. Kondorosi. Tracing system-level communication in distributed systems. *Software: Practice & Experience*, 34:727–755, 2004.

[65] Z. Á. Mann and A. Orbán. Integrating formal, soft and diagrammatic approaches in high-level synthesis and hardware-software co-design. In *Proceedings of Informatik 2001, Workshop on Integrating Diagrammatic and Formal Specification Techniques*, volume I, pages 649–654, September 2001.

[66] 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.

[67] P. L. Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, and O. Cayrol. Hardware, software and mechanical cosimulation for automotive applications. In *IEEE International Workshop on Rapid Systems Prototyping*, 1998.

[68] S. T. McCormick, M. R. Rao, and G. Rinaldi. Easy and difficult objective functions for max cut. *Math. Program., Ser. B*, 94(2-3):459–466, 2003.

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

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

[71] 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.

[72] 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.

[73] A. Orbán. Approximation of the minimum bisection and the hardware–software partitioning problem. Master's thesis, Eötvös Loránd University, 2004.

[74] C. H. Papadimitriou. *Computational complexity.* Addison Wesley, 1994.

[75] 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.

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

[77] 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.

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

[79] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[80] A. E. Smith and D. M. Tate. Genetic optimization using a penalty function. In *Int. Conf. Genetic Algorithms*, 1993.

[81] G. Spivey, S. S. Bhattacharyya, and Kazuo Nakajima. Logic Foundry: A rapid prototyping tool for FPGA-based DSP systems. Technical report, Department of Computer Science, University of Maryland, 2002.

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

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

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

[85] K. Sugihara. A case study on tuning of genetic algorithms by using performance evaluation based on experimental design. Technical Report ICS-TR-97-01, Dept. of Information and Computer Sciences, Univ. of Hawaii at Manoa, 1997.

[86] J. Sziray. A test model for hardware and software systems. *Journal of advanced computational intelligence and intelligent informatics*, 8(5):523–529, 2004.

[87] J. Teich, T. Blickle, and L. Thiele. An evolutionary approach to system-level synthesis. In *Int. Workshop Hardware/Software Codesign*, 1997.

[88] S. R. Thatté. Automated synthesis of interface adapters for reusable classes. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1994.

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

[90] 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.

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

[92] 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.

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

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

[95] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin. A general purpose, multiple-way partitioning algorithm. *IEEE Transactions on CAD*, 13(12):1480–1487, 1994.

[96] G. Ziegler, Z. Á. Mann, A. Orbán, Zs. Palotai, L. Grad, and A. Lőrincz. Three-level memory for optimization. *Journal of Applied Soft Computing*, submitted.