Int. J. Appl. Math. Comput. Sci., 2007, Vol. 17, No. 2, 249–267
DOI: 10.2478/v10006-007-0022-3

EVALUATING THE KERNIGHAN–LIN HEURISTIC FOR HARDWARE/SOFTWARE PARTITIONING

ZOLTÁN ÁDÁM MANN, ANDRÁS ORBÁN, VIKTOR FARKAS

Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H–1117 Budapest, Magyar tudósok körútja 2, Hungary
e-mail: {zoltan.mann,andras.orban}@cs.bme.hu, vikif@sch.bme.hu

In recent years, several heuristics have been proposed for the hardware/software partitioning problem. One of the most promising directions is the adaptation of the Kernighan-Lin algorithm. The Kernighan-Lin heuristic was originally developed for circuit partitioning, but it has been adapted to other domains as well. Moreover, numerous improvements have been suggested so that now several variants of the original algorithm exist. The aim of this paper is to systematically evaluate the possibilities of applying the Kernighan-Lin heuristic to hardware/software partitioning. It is investigated in detail which versions of the heuristic work well in this context. Since hardware/software partitioning also has several formulations, it is also discussed how the problem formulation affects the applicability of this heuristic. Furthermore, possibilities of efficient implementations of the algorithm—by using appropriate data structures—are also presented. These investigations are accompanied by numerous empirical test results.

Keywords: Kernighan/Lin heuristic, Fiduccia/Mattheyses heuristic, hardware/software partitioning, hardware/software codesign

1. Introduction

Today's computer systems typically consist of both hardware and software components. For instance, in an embedded signal processing application it is common to use both application-specific hardware accelerator circuits and general-purpose, programmable units with the appropriate software (Arató *et al.*, 2003b). This is beneficial since application-specific hardware is usually much faster than software, and also more power-efficient, but it is also significantly more expensive. On the other hand, software is cheaper to create and to maintain, but slow, and general-purpose processors consume much power. Hence, performance or power critical components of the system should be realized in hardware, and non-critical components in software. This way, an optimal trade-off between cost, power and performance can be achieved.

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

Traditionally, partitioning was carried out manually. Conversely, as the systems to be designed have become more and more complex, this method has become infeasible, and many research efforts have been undertaken to automate partitioning as much as possible. These results are reviewed in Section 2.1.

One of the most promising directions is the application of the Kernighan-Lin (KL) heuristic (Vahid and Le, 1997). Originally, this algorithm was developed for a formulation of the circuit partitioning problem (Kernighan and Lin, 1970). 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 for the success of the KL heuristic is that it is fast as a greedy algorithm, but it can escape from 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 (Fiduccia and Mattheyses, 1982). They presented—among other things—a powerful data structure to enable a linear-time implementation of their algorithm, which we 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 2.2.

The application of the KL heuristic in the context of hardware/software partitioning was suggested by Vahid (Vahid, 1997; Vahid and Le, 1997). 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 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, but only with partial success. For more details, see Section 2.3.

The aim of this paper is to remedy these shortcomings. More specifically, our goals are the following:

- to investigate the applicability of the suggested improvements to the KL algorithm in the context of hardware/software partitioning;
- to investigate how different formulations of the hardware/software partitioning problem influence the applicability and efficiency of the KL heuristic;
- to provide an efficient implementation of the KL algorithm for hardware/software partitioning.

More specifically, we proceed as follows: We start with the description of our partitioning model in Section 4, which focuses on two conflicting cost metrics, one of which has to be minimized while the other must not exceed a given value.

Our algorithm is described in Section 5. We first explain the skeleton of the algorithm, and then go through the details systematically. In particular, we investigate how the different cost metrics can be joined into a single gain value for each node; as it turns out, this choice has important implications for the efficiency of the whole algorithm. We also discuss different strategies for generating the initial partition, for tie-breaking and for locking. The opportunities for efficiently implementing the algorithm are discussed in Section 5.6. We 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. We also present an implementation based on the range tree data structure that achieves this lower bound for sparse graphs.

The implications of the partitioning problem formulation are elaborated in more depth in Section 6. In particular, we describe how more than two cost metrics can be incorporated into the algorithm without sacrificing efficiency. We also investigate how scheduling and other typical tasks of hardware/software co-design can be integrated into partitioning when using a KL-type algorithm. We sketch a possible hybrid algorithm for this purpose but leave the question of an efficient implementation open for future research.

In order to compare the different configurations of our algorithm with each other, as well as to compare our algorithm with other partitioning heuristics, we ran several empirical tests on benchmark problems. We present two versions of our algorithm (denoted by KL1 and KL2) with different speed/efficiency characteristics: KL1 produces satisfactory results extremely quickly, whereas KL2 produces excellent results in acceptable time.

2. Previous Work

In this section, we review previous work on hardware/software partitioning in general (Section 2.1), the KL family of algorithms (Section 2.2), and the application of the KL algorithm to hardware-software partitioning (Section 2.3).

2.1. Work on Hardware/Software Partitioning.

Hardware/software partitioning is a central task in hardware/software co-design (Wolf, 2003). Its main focus is deciding which components of the system should be realized in hardware and which ones in software. During partitioning, several conflicting design goals should be considered, such as performance, chip size, production costs, power consumption, etc.

Concerning the exact problem definition, there are significant differences between the suggested partitioning methods. In particular, many researchers consider scheduling as part of partitioning (Chatha and Vemuri, 2001; Dick and Jha, 1998; Kalavade and Lee, 1997; Lopez-Vallejo and Lopez, 2003; Mei *et al.*, 2000; Niemann and Marwedel, 1997), whereas others do not (Eles *et al.*, 1996; Grode *et al.*, 1998; Madsen *et al.*, 1997; O'Nils *et al.*, 1995; Vahid and Le, 1997; Vahid, 2002). Some even include the problem of assigning communication events to links between hardware and/or software units (Dick and Jha, 1998; Mei *et al.*, 2000).

Furthermore, in a number of related papers, the target architecture is supposed to consist of a single software and a single hardware unit (Eles *et al.*, 1996; Grode *et al.*, 1998; Gupta and de Micheli, 1993; Henkel and Ernst, 2001; Lopez-Vallejo and Lopez, 2003; Madsen *et al.*, 1997; Mei *et al.*, 2000; O'Nils *et al.*, 1995; Qin and He, 2000; Srinivasan *et al.*, 1998; Stitt *et al.*, 2003; Vahid and Le, 1997), whereas others do not impose this limita-

tion. Some limit parallelism inside hardware or software (Srinivasan *et al.*, 1998; Vahid and Le, 1997) or between hardware and software (Henkel and Ernst, 2001; Madsen *et al.*, 1997).

The system to be partitioned is usually 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.

When looking at the algorithms that have been suggested for hardware/software partitioning, one can differentiate between exact and heuristic solutions. The proposed exact algorithms include branch-and-bound (Binh *et al.*, 1996), dynamic programming (Madsen *et al.*, 1997; O'Nils *et al.*, 1995), and integer linear programming (Mann and Orbán, 2003; Niemann, 1998; Niemann and Marwedel, 1997).

The majority of the proposed partitioning algorithms are 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 (Kalavade, 1995; Mann and Orbán, 2003), and the exact algorithms for them have exponential runtimes.

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

Some researchers used custom heuristics to solve hardware/software partitioning. This includes the GCLP algorithm (Kalavade and Lee, 1997; Kalavade and Subrahmanyam, 1998) and the expert system of (Lopez-Vallejo and Lopez, 1998; Lopez-Vallejo and Lopez, 2003), as well as the heuristics in (Gupta and de Micheli, 1993; Wolf, 1997). Hierarchical clustering is another family of well-known heuristics that has been often applied to partitioning problems (Abdelzaher and Shin, 2000; Barros *et al.*, 1993; Vahid, 2002; Vahid and Gajski, 1995).

2.2. Kernighan-Lin Algorithm and Its Variants. Originally, the KL algorithm was developed for circuit partitioning (Kernighan and Lin, 1970). Its aim is to partition a graph into two parts of equal size (i.e., to find the 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 cost 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 one. This is how the algorithm can escape from local optima. At the end of the pass, the algorithm reverts to the partition with the lowest cost 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 (1982). We will refer to this variant as the FM algorithm. 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 $2d_{\max}+1$, indexed from $-d_{\max}$ to $d_{\max}$. The element of the array with index i is a pointer to a linked list of 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 neighbours 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 (Hagen *et al.*, 1997). Krishnamurthy suggested a look-ahead mechanism to more precisely estimate the long-term gain of a move, thus breaking ties (Krishnamurthy, 1984). This mechanism has also been generalized for multiway partitioning (Sanchis, 1989). However, these efforts are beneficial primarily for hypergraphs. On the other hand, as was shown in (Hagen *et al.*, 1997), tie-breaking without look-ahead can also be efficient; in particular, the LIFO strategy is more efficient than the FIFO or 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 (Hoffmann, 1994), a dynamic locking mechanism was suggested: when a node is moved from part A to part B, it becomes locked, but its neighbours in part A become free. This is beneficial because then the neighbours 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 (Dasdan and Aykanat, 1997; Yeh, 1994), but 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 (Alpert and Kahng, 1995).

2.3. 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. We know of two such attempts.

The first one is due to Vahid and it is described in (Vahid, 1997; Vahid and Le, 1997). 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. Vahid uses a containment relation between the nodes to be able to calculate efficiently the time spent in a given node and its successors. He also mentions that other metrics, such as hardware size and software size, should also be taken into account, but does not elaborate on 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 and, consequently, running time which is 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.

On the other hand, 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, but nothing guarantees that. The work by Vahid also left open the questions (i) whether the improvements suggested in the KL literature can be used in the context of hardware/software partitioning as well, and (ii) if other formulations of the hardware/software partitioning problem also 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-Vallejo and Lopez (2003). Unfortunately, they gave hardly any details on their implementation. 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 any more that only the gains of the neighbours 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.

3. Challenges

The application of the KL heuristic in the context of 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 into 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. Conversely, there are often constraints that have a simi-

[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.

lar effect. For example, a real-time constraint typically limits the number of nodes that can be mapped to software, whereas 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 was discussed in Section 2.1, the scope of hardware/software partitioning can vary. 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 our purposes. Therefore, it is an important and non-trivial objective to keep the extended algorithm also as fast as possible. This also involves finding the right data structure for the implementation.

First, let us assume that partitioning only means deciding which nodes to map to hardware and which ones to software, i.e., we ignore the third challenge and focus on resolving the other three (addressed in Sections 5.1 to 5.6). Such a problem definition is presented in detail in Section 4. It will be investigated later in Section 6.2 how other steps, such as scheduling, can be included.

4. System Model

In order to illustrate our algorithm, we will use two cost metrics: an *execution-time* metric and a *hardware cost* metric. However, it is possible to include further cost metrics as well (see Section 6.1).

The execution-time metric is defined similarly as in (Vahid and Le, 1997), with the only difference that we do not use the containment hierarchy, so that the edges only model communication. That is, we are given a graph $G = (V, E)$ with vertices $v_1, \ldots, v_n$, and each vertex is assigned two execution times: $ts(v)$ is the execution time of vertex v if it is mapped to software, and $th(v)$ if mapped to hardware. Moreover, each edge is assigned a communication cost: $tc(v, w)$ is the overhead of the communication between vertices v and w if they are in different contexts (i.e., one in hardware and the other in software or vice versa). Just like in Vahid's work, the tc values can be computed from the bus widths, call frequencies, and amounts of transferred data per call.

We neglect the communication between vertices in the same context, since on most architectures this is much cheaper than the communication between hardware and software. Furthermore, we assume for simplicity that the overhead does not depend on the direction of the communication (whether from software to hardware or vice versa). Hence, the direction of the edges does not matter, so that the graph does not have to be directed. This is a benefit compared with many previous approaches that require the graph to be directed and acyclic, because acyclicity is often an unrealistic requirement.

P is called a hardware-software partition if it is a bipartition of V: $P = (V_H, V_S)$, where $V_H \cup V_S = V$ and $V_H \cap V_S = \emptyset$. ($V_H = \emptyset$ or $V_S = \emptyset$ is also possible.) The set of cut edges of partition P is defined as $E_P = \{(v, w) : v \in V_S,\ w \in V_H \text{ or } v \in V_H,\ w \in V_S\}$. Just as in Vahid's work, the execution time of the system, with respect to partition P, is $T_P = \sum_{v \in V_H} th(v) + \sum_{v \in V_S} ts(v) + \sum_{e \in E_P} tc(e)$.

Unlike Vahid's work, which focused mainly on optimizing a single metric, we define a second metric as well, which represents a conflicting design objective, because we believe that this way the problem becomes much more realistic. We define a hardware cost metric, which can represent any cost (e.g., production cost, heat dissipation, implementation effort etc.) associated with the hardware implementation of a node. Note that optimizing execution time typically results in mapping many nodes to hardware; optimizing the hardware cost has the opposite effect, thus the joint effect of these two metrics results in appropriate trade-offs between execution time and cost. Therefore we assume that each vertex v is assigned a hardware cost $h(v)$ and the hardware cost of the system with respect to partition P is $H_P = \sum_{v \in V_H} h(v)$.

As was mentioned in Section 3, one of the challenges in hardware/software partitioning is that there can be two different kinds of cost metrics: cost metrics that are constrained and cost metrics that have to be minimized. In order to illustrate how to handle these two kinds of cost metrics, we will assume that we are given a hard real-time constraint (i.e. an upper bound on T_P), and have to optimize the hardware cost. That is, the partitioning problem we are dealing with can be formulated as follows: Given the graph G with the cost functions th, ts, tc, and h, and the time limit $T_0 \geq 0$, find a hardware/software partition P with $T_P \leq T_0$ that minimizes H_P among all such partitions. It should be mentioned that this problem is provably $\mathcal{NP}$-hard (Arató *et al.*, 2003a).

We will call a partition *valid* if it satisfies $T_P \leq T_0$. We will assume that, for each node v_i, $th(v_i) \leq ts(v_i)$, i.e., each node is faster in hardware than in software. Of course, this is not a limitation in practice. Then, the least possible execution time corresponds to the all-hardware solution. Consequently, there exist valid partitions if and only if the all-hardware partition is valid. We will assume that this is true. This is no limitation either, because otherwise the partitioning problem does not make sense. Furthermore, this condition can be easily checked.

5. Algorithm

As discussed in Section 3, there is typically no explicit balance criterion in hardware/software partitioning, and hence our algorithm will make single node moves, rather than node swaps, just as the FM algorithm. During the algorithm, we always maintain two partitions:

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

Algorithm 1. Skeleton of the KL 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 other part   //P_curr changes accordingly
        if P_curr is better than P_best then
        {
            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 pass did not improve P_best
    return P_best
}
```

The skeleton of our algorithm is presented in pseudocode in Algorithm 1. The details are described in the next subsections. In particular, we discuss the choice of the gain function in Section 5.1, the creation of the starting partition in Section 5.3, possible tie-breaking strategies in Section 5.4, and different locking schemes in Section 5.5. Finally, possible data structures for the efficient implementation of the algorithm are discussed in Section 5.6.

5.1. Gain Function. The gain concept of the original KL algorithm has to be extended for our 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 our algorithm, these roles have to be separated, because each cost metric has to be updated separately but the choice of the next node should depend on all cost metrics. Therefore, we define the following three quantities instead:

- $\Delta T(v)$ is the amount by which moving v to the other context increases T. (Sometimes, we will call $-\Delta T(v)$ the software gain of node v.)
- $\Delta H(v)$ is the amount by which moving v to the other context increases H. (Sometimes, we will call $-\Delta H(v)$ the hardware gain of node v.)
- $\text{gain}(v) = f(\Delta T(v), \Delta H(v))$ 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 $\Delta T(v)$ and $\Delta H(v)$ values, it defines the relative importance of the design goals.

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

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

But our aim is different: we would like to minimize H_P, while bounding T_P. The strictest solution for this is the following:

$$\text{gain}(v) = \begin{cases} -\infty & \text{if } T_{P_{\text{curr}}} + \Delta T(v) > T_0, \\ -\Delta H(v) & \text{otherwise.} \end{cases} \quad (1)$$

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

There are also arguments in favour of less strict solutions. Suppose, e.g., that a move would slightly violate the real-time constraint but would result in a dramatic decrease in the 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 small exceeding of the real-time constraint. One can hope that in 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:

$$\text{gain}(v) = -\Delta H(v) - p(T_{P_{\text{curr}}}, \Delta T(v), T_0). \quad (2)$$

Here, p is a *penalty function* that penalizes the exceeding of the time limit. Typically, p depends on the percentage by which $T_{P_{\text{curr}}} + \Delta T(v)$ exceeds T_0. Note that it would also be possible to define p as a function of the *amount* (instead of percentage) by which T_0 is exceeded. However, it is much more informative to say that, e.g., the limit is exceeded by 10% than by 10 units. So, we can assume that p only depends on the quantity

$$exc = \frac{T_{P_{\text{curr}}} + \Delta T(v)}{T_0}.$$

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

$$\text{gain}(v) = -\frac{\Delta H(v)}{H_{P_{\text{curr}}}} - p(exc). \quad (3)$$

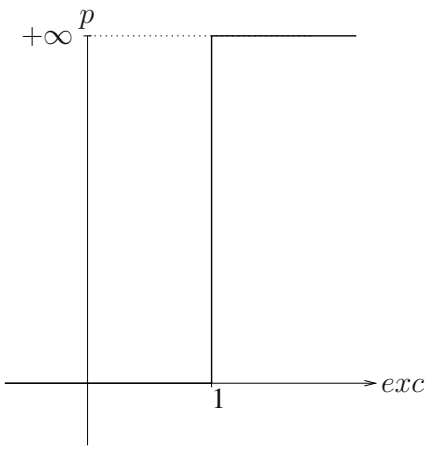

Fig. 1. Strict penalty function.

Note that (1) is a special case of (2), in which the penalty is ∞ if T_0 is exceeded, and 0 otherwise (see Fig. 1). Strictly speaking, (1) is not a special case of (3) because of the division by $H_{P_{\text{curr}}}$ in (3). On the other hand, when using the strict penalty function of Fig. 1, the division by $H_{P_{\text{curr}}}$ does not change the ranking of the nodes. Thus, if we are only concerned with the ranking of the nodes and not the exact gain values, then (1) is also a special case of (3).

In the general case, it is also logical to set $p = 0$ if the limit is not exceeded. Moreover, we can set a threshold $q > 1$, and let $p = \infty$ if even qT_0 is exceeded. Clearly, p should be monotonously increasing if exc is in the interval $[1, q]$. Such a function is shown in Fig. 2.

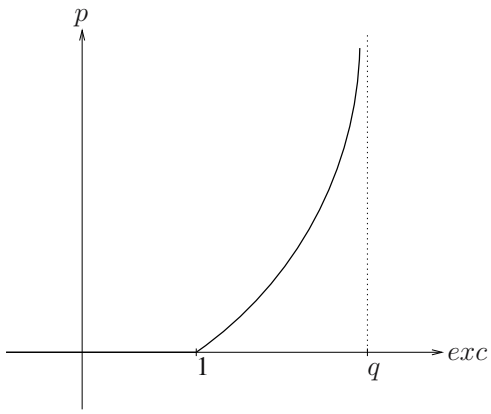

Fig. 2. Possible permissive penalty function.

In the rest of this paper, we will use the strict gain function (cf. (1)) as well as the permissive gain function (cf. (3)) with suitable penalty functions.

Finally, we would like to point out that using a gain function that does not give an infinite penalty to non-valid partitions requires extra precautions. Although one might get better results by temporarily allowing 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. 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; if we find a partition with a lower hardware cost but exceeding the time limit, then P_{best} should not be updated.

5.2. Updating Gain Values. At the beginning of each pass, the gains of all nodes are calculated. Moreover, after each move, the gains are updated, see Algorithm 1. The initial computation of the gains is straightforward. Now we focus on the updating step.

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 neighbours can change, and all other φ values remain the same.[2] Note that, in the case of the original FM algorithm, the gain function, which was defined in (Fiduccia and Mattheyses, 1982) as the decrease in the cutsize, possesses the $(*)$-property.

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

If the gain function possesses the $(*)$-property, then the update of the gains after a move can be very efficiently implemented, see Algorithm 2. If $d(v)$ denotes the number of neighbours of node v, then the updating after moving v requires $O(1 + d(v))$ time, therefore all updating steps of a pass require $\sum_{v \in V} O(1 + d(v)) = O(n + m)$ time, where m denotes the number of the edges in the graph.

Algorithm 2. Updating the gains after moving node v, assuming the $(*)$-property.

```
recalculate and store gain of v
foreach free neighbour w of v do
{
    recalculate gain of w
    store gain of w
}
```

Conversely, if the $(*)$-property does not hold, then the gain of all nodes has to be recomputed after each move. Therefore, there is no need to store the gain values at all. Rather, the node with maximum gain can be selected whilst computing the gain values (see Algorithm 3). In this case, the number of steps is $O(n)$ after each move, yielding a total of $O(n^2)$ per pass.

Algorithm 3. Recomputing the gains and selecting the maximum after a move when the $(*)$-property does not hold.

```
max_gain = −∞
best_node = undefined
foreach free w ∈ V do
{
    recalculate gain(w)
    if gain(w) > max_gain
    {
        max_gain = gain(w)
        best_node = w
    }
}
```

As can be seen, the $(*)$-property has important implications on the efficiency of the algorithm, since $O(n^2)$ is much bigger than $O(n + m)$ for sparse graphs, and typical communication graphs are sparse. The following theorem—which will have important consequences in Section 5.4 and Section 5.6—shows the connection between the $(*)$-property and our different gain notions:

Theorem 1.
(i) ΔH possesses the $(*)$-property.
(ii) ΔT possesses the $(*)$-property.
(iii) The gain functions defined in Section 5.1 do 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, and that of the other nodes remains the same. Thus ΔH possesses the $(*)$-property.

(ii) Again, from the definition of ΔT, it is obvious that

$$\Delta T(v) = \begin{cases} th(v) - ts(v) + \sum\limits_{\substack{w \in V_S \\ (v,w) \in E}} c(v,w) - \sum\limits_{\substack{w \in V_H \\ (v,w) \in E}} c(v,w) \\ \qquad\qquad \text{if } v \in V_S, \\ ts(v) - th(v) + \sum\limits_{\substack{w \in V_H \\ (v,w) \in E}} c(v,w) - \sum\limits_{\substack{w \in V_S \\ (v,w) \in E}} c(v,w) \\ \qquad\qquad \text{if } v \in V_H. \end{cases}$$

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

(iii) As can be seen from (1), 'gain' depends on $T_{P_{\text{curr}}}$. When a node is moved from one context to the other, $T_{P_{\text{curr}}}$ changes, and thus potentially the gain of all nodes can change. ■

Section 5.6 describes how the efficiency of the algorithm can still be enhanced, although the $(*)$-property does not hold.

5.3. 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. There are several possibilities to generate a valid initial partition:

- As has been mentioned earlier, it can be assumed that the all-hardware partition is valid, and 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 likely 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, e.g., a simple greedy algorithm. This works as follows: It starts from the all-hardware partition. In each step, it checks which nodes can be moved from hardware to software without hurting

the real-time constraint, and moves the node with the highest hardware cost from these. It stops when no more moves are possible.

- 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. Unfortunately, it is by no means obvious how one can quickly generate random valid partitions at least with an approximately uniform distribution. Here, we present a method that generates valid partitions randomly (see Algorithm 4)—but not with uniform distribution. The idea is to randomly map each node to either software or hardware, and to check if the resulting partition is valid. If it is not, then we decrease the probability of mapping nodes to software. Sooner or later we surely reach a valid partition: in the worst case, when the probability of mapping nodes to software becomes zero, we get the all-hardware solution. But we return the first valid partition that is found in this way.

Algorithm 4. Randomly generate a valid initial partition.

1. Let $r = 1$ and N be a positive integer; let $dr = 1/N$.
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.

5.4. 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 was discussed in Section 2.2, 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 behaviour. Since all suggested look-ahead mechanisms are useful mainly for hypergraphs, we will consider here only the second group. It consists basically of the following strategies (M denotes the set of nodes with maximum gain):

- random, i.e., a node is randomly selected from M,
- LIFO, i.e., the node which was the last to get into M is selected,
- FIFO, i.e., the node which was the first to get into M is selected.

It has been reported that, in general, the LIFO strategy outperforms the other two (Hagen *et al.*, 1997). This can be intuitively explained as follows: One of the problems with 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 neighbours 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 neighbours is moved, and thus the node selected by the LIFO strategy will be the one whose neighbour was moved recently.

Now let us investigate to what extent this can be transferred to our case. Unlike in the original KL algorithm, our gain function is real-valued. Hence, in our case it is very unlikely that more than one node 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: given n points with their gain values, it has to be determined which nodes have similar gains. Fortunately, the problem is actually simpler because we only need to determine the best cluster, i.e., the cluster with the highest gain value (denoted by 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. We define M as the set of nodes with gain values in the interval $[\tau B, B]$. (Note that the case $\tau = 1$ corresponds to the strategy of considering only the node with the highest gain value.) τ should be chosen near 1, so that only a few nodes will be in the interval $[\tau B, B]$.

The second difference between our case and the FM algorithm is that, according to Theorem 1, 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 neighbour was moved the last time.

Therefore, the straightforward adaptation of the LIFO tie-breaking strategy would presumably be less efficient than in the case of the FM algorithm. Hence, we suggest a more direct implementation of the actual aim of this tie-breaking strategy, i.e., to encourage moving the neighbours of recently moved nodes. More specifically, we define a variable $\ell(v)$ for each node v, which stores the last time step when a neighbour of v was moved. (That is, we maintain a counter ctr, 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, we update the ℓ values corresponding to its neighbours: for each neighbour w, let $\ell(w) = ctr$. All this can be done in linear time per pass.) Now the LIFO strategy can be adapted as selecting the node with the highest ℓ value from M and, similarly, FIFO is adapted as selecting the node with the lowest ℓ value from M.

5.5. Locking Schemes. From the numerous alternative locking schemes that have been suggested for KL (see Section 2.2 and (Alpert and Kahng, 1995)), the most promising for our purposes is the dynamic locking scheme proposed by Hoffman (1994). 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 neighbours should be encouraged to follow it. Hence, Hoffman suggests that after moving a node from part A to part B, its neighbours 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. On the other hand, it is questionable whether ten is the right number in this context as well.

5.6. 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 neighbours have to be updated.

Unfortunately, these two properties do not hold 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 1). In the following, we investigate under which circumstances we can still provide an efficient—but more tricky—implementation.

As it turns out, the efficiency of our 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 we do not pose any restrictions, 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. As it turns out, this is not possible.

Theorem 2. *When using the strict gain function, every implementation has at least $\Omega(n \log n)$ time complexity.*

Proof. We will show that it is possible to sort numbers using our algorithm. Since it is known that sorting n numbers takes $\Omega(n \log n)$ time (Cormen *et al.*, 2001), the theorem will follow.

Assume that the numbers $x_1, \ldots, x_n$ have to be sorted. We define a graph with n vertices $(v_1, \ldots, v_n)$ and no edges. For each node v_i, let $ts(v_i) = th(v_i) = 0$ and $h(v_i) = x_i$. Let T_0 be any non-negative number, then $T_P \leq T_0$ will always hold (since T_P will always be 0). Assume that the initial partition is the all-hardware partition. In each step of the first pass, a node will be moved from hardware to software until each node is in software. The gain of a free node v_i will always be $h(v_i) = x_i$. Since the algorithm moves the nodes in the decreasing order of their gain values, observing the order in which the nodes are moved yields the decreasing order of the x_is. ■

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 we 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 1, not even the strict gain function possesses the $(*)$-property, and thus it is by no means obvious how the algorithm can be faster than trivial $O(n^2)$. The key is that ΔH and ΔT do possess the $(*)$-property, and the strict gain function has such a simple structure that the node with the highest gain can be found without actually storing or even computing the gain values of each node explicitly. Rather, we only store the ΔH and ΔT values for each node. Because of the $(*)$-property, these can be updated efficiently: all updating steps require $O(n+m)$ time per pass.

The gain depends, beside $\Delta H(v_i)$ and $\Delta T(v_i)$ also on $T_{P_{\text{curr}}}$. In a given step of the algorithm, we have a concrete $T_{P_{\text{curr}}}$ value. We are only interested in those nodes for which $T_{P_{\text{curr}}} + \Delta T(v_i) \leq T_0$ 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 $x_i := \Delta T(v_i)$ 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 the lowest y coordinate in the $x \leq T_0 - T_{P_{\text{curr}}}$ half plane.

Thus a data structure is needed in which two-dimensional points can be stored, such that queries of the form 'select the point with the 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: it is the range tree (de Berg *et al.*, 2000). 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 very easily be adapted to the kind of query that we face. 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 our 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 the highest gain without explicitly calculating the gain of each node, and only storing and updating the ΔH and ΔT values. Now let us investigate how this can be generalized to permissive gain functions. The problem here is that the penalty function is not constant between T_0 and qT_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 T_0 and qT_0, it is only required that it should be monotonously increasing from 0 to ∞. For the sake of efficiency, we require p to be a 'staircase'-like function, i.e. it should consist of a small number of constant parts. Actually, this is not a fundamental restriction, since every function can be approximated with staircase-like functions. For instance, the function in Fig. 2 can be approximated with the staircase-like function in Fig. 3. The reason for using staircase-like penalty functions is that one stair of such a function is similar to the strict penalty function, and so it is possible to apply the method used for the strict penalty function to each stair.

More specifically, our algorithm works as follows for a staircase-like penalty function: Let (x_1, x_2) be an 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, we have to select the point with the 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. We can thus obtain a best node in 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.

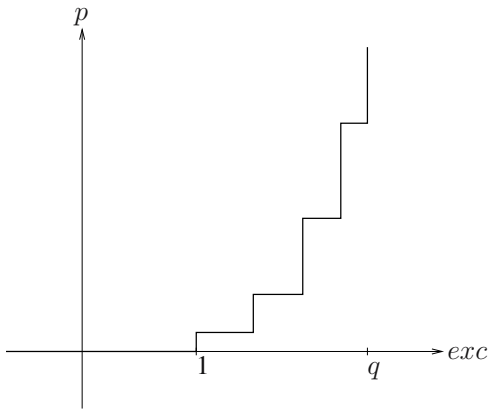

Fig. 3. Staircase-like penalty function. This is an approximation of the function in Fig. 2.

6. Extension Possibilities

As was demonstrated in Section 2.1, the hardware/software partitioning problem has several formulations. Until now, we have focused on one of the simpler problem formulations, in which partitioning only aims at deciding which components should be mapped to hardware and which ones to software, and only two cost metrics were considered. In this section, we investigate how more than two cost metrics can be taken into account (Section 6.1), and how scheduling and other tasks can be incorporated (Section 6.2).

6.1. Considering More Than Two Cost Metrics. Until now, we have focused on two cost metrics: 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 k-th cost metric. That is, the problem is to find 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, for each node v_i and each cost metric c_j we can define the change in 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

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

where the p_js are appropriate penalty functions (p_j penalizes the violation 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 straightforward 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 modelled 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.

6.2. 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 (where the execution time of a node depends on whether it is implemented in hardware or in software) plus the sum of the communication overhead of the edges. In this respect, we followed the assumptions of Vahid (1997). On the other hand, Vahid's model is somewhat restrictive: it assumes a single hardware unit and a single software unit and limits parallelism between the two. In a more general framework with several hardware and software units, some additional factors have to be considered, e.g.:

- 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. Parallelism is especially relevant in hardware, where usually several operations can be executed concurrently.
- 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. Conversely, as was demonstrated in Section 2.1, there is no consensus in the literature on 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 the latter group, powerful methods have been devised to decouple partitioning from the other problems (see, e.g, (Madsen *et al.*, 1997)). Such decoupling 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.

An example of the loss of precision is estimating execution time with the sum of the execution times of the nodes, without taking into account the concurrent execution of the operations in hardware. How rough this estimate is depends on several factors. For example, concurrency plays a major role in the case of fine-grained operations (e.g., single instructions), because many of them can be concurrently executed. However, if the nodes of the graph represent bigger modules (e.g., complex functions), then the level of parallelism decreases because it is not possible to execute many such operations concurrently.

Another aspect is the order of magnitude of the execution times. Usually, hardware execution times are significantly lower than software execution or communication times. Therefore, a loss in the precision in the calculation of the hardware execution time hardly effects the overall execution time, which is dominated by software execution and communication time. In such cases, it is even possible to assume hardware execution times to be zero. With this assumption, the scheduling of hardware operations becomes dispensable (Arató *et al.*, 2003a). Of course, on architectures where this assumption is not justified, scheduling does play an important role.

Our adaptation of the KL algorithm, as described so far, clearly belongs to the group of 'pure' partitioning approaches, i.e., without considering scheduling, etc. We believe that the KL algorithm is intrinsically more appropriate for this group because its strength lies in the efficient optimization of simple cost metrics. However, we will now briefly sketch how it can also accommodate the scheduling of modules, the scheduling of communication events, routing, hardware sharing, etc. The cost metrics are then not additive any more; rather, the cost metrics are calculated by external scheduling or routing algorithms that can be included as a black box into our algorithm. The $\Delta c_j(v_i)$ values that are needed by our 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. For details, see Algorithm 5. 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 that node so slow that every schedule considered so far becomes invalid.

Algorithm 5. Extended gain calculation.

```
foreach free v ∈ V
{
    for j=1 to k do
    {
        move v to the other context
        run external algorithm to calculate c_j^new(v)
        move v back
        let Δc_j(v) = c_j^new(v) − c_j(v)
    }
    compute gain(v) from Δc_1(v), ..., Δc_k(v)
}
```

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 takes $O(\varrho)$ 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 (e.g., a force-directed scheduler takes $O(n^3)$ time, see also (Arató *et al.*, 2005b)). Therefore, the complexity of our algorithm will be at least $\Omega(n^3)$, maybe significantly more.

We can conclude that 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 numerous schedulings, many of which will turn out to be equivalent and/or not needed.) Therefore, it is an important future research direction to investigate whether the efficiency of the KL algorithm can be maintained in such an extended version of it. However, we have demonstrated that it is indeed very efficient for the other group of hardware/software partitioning formulations that make use of simplified cost metrics.

7. Empirical Results

As can be seen from the previous sections, the algorithm has many variants and many parameters that can be tuned. We implemented several versions in a C++ program in order to compare them empirically.

The testing process consisted of two phases. In the first phase, our aim was to select the 'best' configuration, or a handful of 'best' configurations, i.e., a set of configurations that work well on typical problem instances. These experiments are described in Section 7.2. In the second phase, we compared our best configurations with a number of other partitioning heuristics (see Section 7.3). The benchmark problem instances used are presented in Section 7.1.

7.1. Benchmarks Used. In order to have a representative mix of benchmark problems, we used three different sources:

- the MiBench benchmark suite (Guthaus *et al.*, 1997),
- own designs of our research group,
- large random graphs.

The characteristics of the test cases are summarized in Table 1. Here n and m denote the numbers of nodes and edges, respectively, in the communication graph.

Table 1. Summary of the benchmarks used.

Name	n	m	Description
crc32	25	34	32-bit cyclic redundancy check. From the Telecommunications category of MiBench.
patricia	21	50	Routine to insert values into Patricia tries, which are used to store routing tables. From the Network category of MiBench.
dijkstra	26	71	Computes shortest paths in a graph. From the Network category of MiBench.
segment	150	333	Image segmentation algorithm in a medical application.
fuzzy	261	422	Clustering algorithm based on fuzzy logic.
rc6	329	448	RC6 cryptographic algorithm.
mars	417	600	MARS cipher.
ray	495	908	Ray-tracing algorithm for volume visualization.
random1	1000	1000	Random graph.
random2	1000	2000	Random graph.
random3	1000	3000	Random graph.
random4	1500	1500	Random graph.
random5	1500	3000	Random graph.
random6	1500	4500	Random graph.
random7	2000	2000	Random graph.
random8	2000	4000	Random graph.
random9	2000	6000	Random graph.

It has to be noted that most previous algorithms 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. Conversely, the systems to be designed become more and more complex, and therefore we also included some really large random benchmarks, 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 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, and thus the 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, we 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 κs_i and standard deviation $\lambda\kappa s_i$, where s_i is the software cost of the given node. That is, there is a correlation, as defined by the value of λ, between a node's hardware and software costs. (If $\lambda = 0$, 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. We tested two different values for λ: 0.1 (high correlation) and 0.6 (low correlation). The value of κ only corresponds to the choice of units for software and hardware costs, and thus it has no algorithmic implications.
- Where communication costs were not available, they were generated as uniform random numbers from the interval $[0, 2\mu\, s_{\max}]$, where $s_{\max}$ is the highest software cost. Thus, communication costs have an expected value of $\mu s_{\max}$, and μ is the so-called communication to computation ratio (CCR). We tested two different values for μ: 1 (computation-intensive case) and 10 (communication-intensive case).
- Finally, the limit T_0 can be arbitrarily defined for every benchmark. Note that $T_0 = \sum th_i$ means that all components have to be mapped to hardware, whereas $T_0 = \sum ts_i$ means that all components can be mapped to software. All sensible values of T_0 lie between these two extremes. We tested two values for T_0: one generated as a uniform random number from the interval $\left[\sum th_i, \frac{1}{2}(\sum th_i + \sum ts_i)\right)$ (strict real-time constraint) and one taken randomly from $\left[\frac{1}{2}(\sum th_i + \sum ts_i), \sum ts_i\right]$ (loose real-time constraint).

Half of the benchmarks were used to find the best configurations of our algorithm, and the other half to compare the best configurations found to other partitioning heuristics.

7.2. Determining the Best Configurations. As has already been mentioned, there are several possible configurations for our algorithm. Here is a list of the implemented variants and parameters:

- Penalty function: strict vs. permissive.
- In the case of the permissive penalty function: what exactly should the penalty function be like?
- Initial partition: all-hardware vs. greedy heuristic vs. Algorithm 4.
- 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 configurations, plus an integer and a real-valued parameter, plus the choice of a function. In the latter three cases discretization is needed in order to have a finite set of configurations.

Selecting the best configuration essentially means a search in a high-dimensional space, which is a non-trivial problem. A further difficulty is that there is often no clear 'better-than' relation between the different configurations, i.e., it is possible that configuration A is better on one of the benchmarks than configuration B, but it is worse on another benchmark. To sum up: running all configurations on all benchmarks results in a huge set of data that is hard to analyze.

Therefore we used a more careful methodology. When comparing two configurations, we used the following guidelines:

- Very similar results were not considered different.
- When comparing configuration A to configuration B, the basic criterion was the number of benchmarks on which A was better than B.
- The actual difference in result quality was considered only if
  - the configuration that was worse according to the previous guideline won by significantly more on the benchmarks on which it won than it lost on the benchmarks on which it lost; or

Table 2. Cost of the resulting partition for different configurations.

Benchmark	strict–FIFO–original	strict–FIFO–dynamic	strict–LIFO–original	strict–LIFO–dynamic	strict–random–original	strict–random–dynamic	permissive–FIFO–original	permissive–FIFO–dynamic	permissive–LIFO–original	permissive–LIFO–dynamic	permissive–random–original	permissive–random–dynamic
crc32	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**	16
dijkstra	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**
fuzzy	223	223	224	223	230	229	224	223	221	**220**	224	222
mars	1009	978	710	710	951	920	1009	920	**704**	**704**	1019	971
random2	3955	3955	3840	3812	3889	3866	3796	3841	3789	**3775**	3821	3807
random4	5708	5625	5625	**5609**	5629	5630	5625	5625	5616	**5609**	5645	5650
random6	7412	7193	7361	7147	7307	7220	**7049**	7068	7120	7067	7241	7091
random8	9715	9677	9642	9538	9819	9708	9603	9540	9538	**9520**	9656	9613

  - the number of wins for A was about the same as the number of wins for B.

- The running time of the algorithms was not considered because all versions are very fast.

Based on these guidelines, we managed to compare the different configurations in an intuitive way.

In a first set of experiments, we tuned the non-discrete parameters, independently from the others. Not taking into account some interdependencies naturally results in a loss of precision, but it enabled us to always focus on a manageable number of configurations at a time. This way, we made the following findings:

- Concerning the permissive penalty function: We tested several different polynomials (actually, their staircase-like approximations). The best results were achieved with a fourth-order polynomial.

- Allowed number of moves: in contrast to the result reported in (Hoffmann, 1994), we found that allowing more than five moves per node per pass does not improve the performance significantly anymore.

- The effect of the value of τ on the result quality was not clear at all because the deviation of the results was too high. On the other hand, the setting $\tau = 1$ was clearly not 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. Eventually we fixed τ to be 0.95, which seemed to be a plausible value with quite good performance.

Next, we divided the set of configurations into two subsets: those starting with a random initial partition and those that start with a deterministically selected initial partition (i.e., all-hardware or by the deterministic greedy heuristic). The reason is that—according to previous experience with KL-type algorithms—the effectiveness of the algorithm depends heavily on the initial partition, and hence running the algorithm several times with random initial partitions increases the chance of finding a good initial partition and thus good results. Therefore, if the algorithm starts with a random initial partition, its effectiveness can be presumably significantly improved by running it multiple times and taking the best result, which is, of course, not possible with a deterministic algorithm. (A second source of randomization can be the random tie-breaking strategy, but our first experiences showed that it is much less promising than the random initial partition.)

First, we searched for the best configuration with a deterministic initial partition. The results are summarized in Table 2. Each column of the table corresponds to a configuration of a gain function, tie-breaking strategy and locking scheme: e.g., 'strict–FIFO–original' means a strict gain function, the FIFO tie-breaking strategy, and the original locking scheme. In these experiments, a mix of low and high CCR as well as of low and high T_0 values was used, because we were not interested in the effect of these factors at this point (this effect will be investigated in Section 7.3). In each case, the initial partition was the all-hardware partition. We do not include here the results obtained from the partition found by the deterministic greedy heuristic, because there was no significant difference between the two choices of the initial partition. 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.

As can be seen from the table, the best results were found when using a permissive gain function, the LIFO tie-breaking strategy, and the dynamic locking scheme. We will denote this configuration by KL1. A closer look also reveals that the permissive gain function was in most cases better than the strict gain function, the LIFO tie-breaking strategy was generally better than the other two tie-breaking strategies, and the dynamic locking scheme was also in most cases better than the original locking scheme. This justifies that KL1 is indeed the best configuration. Furthermore, it shows that the improvements that had been suggested for the original KL algorithm are also useful in the context of hardware/software partitioning.

Next, we turned our attention to the configurations with random initial partitions. In these cases, we ran the algorithms fifty times on each benchmark and took the best result. When searching for the best of these configurations, we made very similar observations as with the deterministic configurations: the permissive gain function, LIFO tie-breaking strategy, and dynamic locking won again. We denote the resulting algorithm by KL2. That is, KL2 is essentially fifty runs of KL1 from random initial partitions, after which the best result is returned.

7.3. Comparison with Other Heuristics. In this section, we compare KL1 and KL2 to other hardware/software partitioning heuristics:

- A genetic algorithm (GA) (Arató *et al.*, 2003a).
- A combinatorial algorithm (MFMC) that works by creating a number of auxiliary graphs, determining their minimum cuts, and returning the best partition found this way (Arató *et al.*, 2005a).
- An algorithm based on hierarchical clustering (HC) that works by repeatedly coalescing vertices of the graph based on local closeness metrics until the resulting graph is small enough so that it can be partitioned optimally using branch-and-bound relatively quickly (Arató *et al.*, 2007).

All algorithms were implemented in C/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.

The comparison was based on two metrics: the running time of the algorithms and the cost of the best solution they found. The results are summarized in Figs. 4–7. The four figures correspond to the possible combinations of CCR and T_0 values. (Our previous experience showed that these factors might have significant impact on the relative performance of different partitioning heuristics. Therefore, we applied different CCR and T_0 values to each benchmark used in order to obtain these figures.) Each algorithm is represented by a point in the diagrams. The x-coordinate of the point is the cumulated running time of the given algorithm on the given set of benchmarks. The y-coordinate of the point is the cumulated cost of the resulting partition found by the algorithm for the benchmarks. On both axes, smaller values are better. Also note the logarithmic scale of the x-axis.

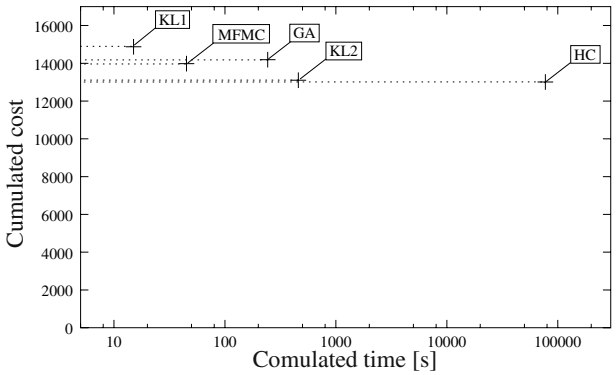

Fig. 4. Running time vs. result cost tradeoff of the heuristics (CCR=low, T_0=low).

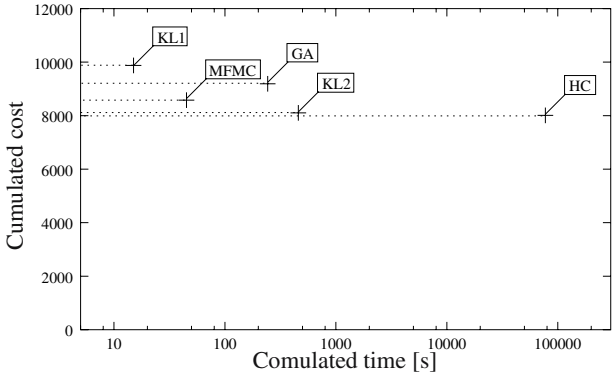

Fig. 5. Running time vs. result cost tradeoff of the heuristics (CCR=low, T_0=high).

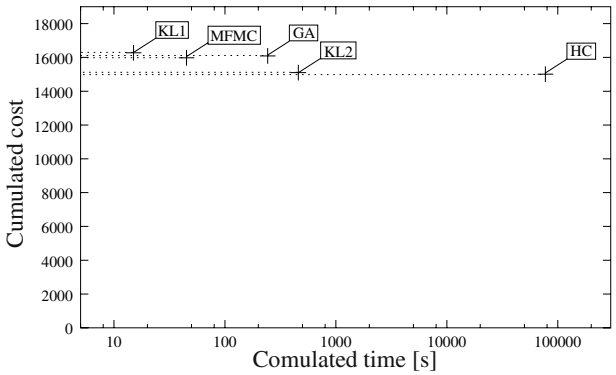

Fig. 6. Running time vs. result cost tradeoff of the heuristics (CCR=high, T_0=low).

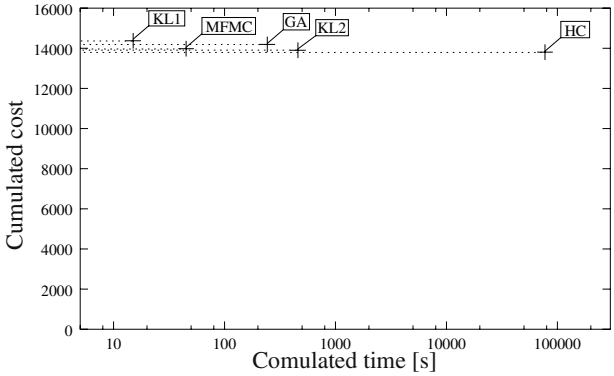

Fig. 7. Running time vs. result cost tradeoff of the heuristics (CCR=high, T_0=high).

The figures clearly show the different running time vs. result quality characteristics of the algorithms. The following observations can be made:

- The best results are achieved in all cases by the HC algorithm. However, its running time is very high, even higher than indicated in the figures. This is because we used a timeout of 10 hours for each run, and for the biggest benchmarks the HC algorithm did not finish within this time limit. In such cases, we calculated with a running time of 10 hours. Actually, it is not surprising that the running time of the HC algorithm is very high on large benchmarks, because it makes use of an exact partitioning algorithm.

- The MFMC algorithm and the GA achieved somewhat worse results, but produced running times that are orders of magnitudes lower.

- The results found by the KL1 algorithm were the poorest. However, they were only about 15% worse on average than those of the HC algorithm. Conversely, KL1 was extremely fast: it solved even the largest problems in a couple of seconds. This makes it very well suited both for quick design space exploration (e.g. in an interactive design environment) and as a pre-optimization step in a hybrid partitioning algorithm.

- The results of the KL2 algorithm were comparable to those of the HC algorithm: they were only about 1% worse on average and 3% worse in the worst case. This disadvantage is actually negligible when considering that partitioning usually works with estimated and thus unprecise cost values. On the other hand, the KL2 algorithm was much faster than HC: it solved even the biggest benchmarks, for which the HC algorithm is clearly not practical anymore, in a couple of minutes. Thus it can be stated that from the practical algorithms it produced consistently the best results.

- The choice of CCR and T_0 did not fundamentally affect the running time of the algorithms, nor their order concerning result quality. Conversely, it did have clear impact on the difference between the result quality of the algorithms. In particular, the difference was smaller in the CCR=high cases. We believe that this can be attributed to the easier nature of these problems: when communication is the dominant cost factor, the partitioning problem essentially becomes a minimum cut problem, which can be solved optimally in polynomial time.

Finally it should be noted that the KL2 algorithm can be effectively parallelized since it consists of independent KL runs. Thus, if sufficient computational resources are available, it can be approximately as fast as the KL1 algorithm.

8. Conclusions

In this paper, we investigated the applicability of KL-type algorithms for the problem of hardware/software partitioning. We have seen that, although the original problem formulation for which the KL algorithm was developed is in several aspects simpler than hardware/software partitioning, the basic idea can be used in this context as well.

We investigated different choices of the gain function, the initial partition, as well as the possibility of transferring improvement suggestions (concerning tie-breaking strategies and locking schemes) for the KL algorithm to the domain of hardware/software partitioning. We also showed that, with the help of a suitable data structure, the algorithm can be very efficiently implemented for the hardware/software partitioning problem as well.

Our empirical results on real and randomly generated benchmarks provided practical evidence for the applicability of the presented approach. In particular, the best configurations of the algorithm used permissive gain functions, LIFO tie-breaking, and dynamic locking. This shows that the presented improvement possibilities really enhance the effectiveness of the algorithm.

We compared the two most promising variants—KL1 and KL2—with some other state-of-the-art partitioning algorithms. We found that KL1 is extremely fast with acceptable result quality, and thus it is well suited as a pre-optimization step or for quick design space exploration. KL2 produced excellent results with acceptable running time even for the largest benchmarks. Thus it is a high-quality partitioning algorithm on its own.

Acknowledgements

This work was supported by the Hungarian National Science Fund (grants OTKA T043329 and T042559).

References

Abdelzaher T.F. and Shin K.G. (2000): *Period-based load partitioning and assignment for large real-time applications*. — IEEE Trans. Comput., Vol. 49, No. 1, pp. 81–87.

Alpert C.J. and Kahng A.B. (1995): *Recent developments in netlist partitioning: A survey*. — VLSI J., Vol. 19, No. 1–2, pp. 1–81.

Arató P., Juhász S., Mann Z.Á., Orbán A. and Papp D. (2003a): *Hardware/software partitioning in embedded system design*. — Proc. IEEE Int. Symp. *Intelligent Signal Processing*, Budapest, Hungary, pp. 197–202.

Arató P., Mann Z.Á. and Orbán A. (2003b): *Hardware-software co-design for Kohonen's self-organizing map*. — Proc. IEEE 7th Int. Conf. *Intelligent Engineering Systems*, Luxor, Egypt, pp. 173–178.

Arató P., Mann Z.Á. and Orbán A. (2005a): *Algorithmic aspects of hardware/software partitioning*. — ACM Trans. Design Autom. Electron. Syst., Vol. 10, No. 1, pp. 136–156.

Arató P., Mann Z.Á. and Orbán A. (2005b): *Time-constrained scheduling of large pipelined datapaths*. — J. Syst. Arch. Vol. 51, No. 12, pp. 665–687.

Arató P., Mann Z.Á. and Orbán A. (2007): *Finding optimal hardware/software partitions*. — Formal Meth. Syst. Design, (submitted).

Barros E., Rosenstiel W. and Xiong X. (1993): *Hardware/software partitioning with UNITY*. — Proc. 2nd Int. Workshop *Hardware-Software Codesign*, Cambridge, USA.

Binh N.N., Imai M., Shiomi A. and Hikichi N. (1996): *A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts*. — Proc. 33rd *Design Automation Conference*, Las Vegas, USA, pp. 527–532.

Chatha K.S. and Vemuri R. (2001): *MAGELLAN: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs*. — Proc. 9-th Int. Symp. *Hardware/Software Codesign*, Copenhagen, Denmark, pp. 42–47.

Cormen Th.H., Leiserson Ch.E., Rivest R.L. and Stein C. (2001): *Introduction to Algorithms, 2nd Ed.* — Cambridge: MIT Press.

Dasdan A. and Aykanat C. (1997): *Two novel multiway circuit partitioning algorithms using relaxed locking*. — IEEE Trans. Comput. Aided Des. Integ. Circ. Syst., Vol. 16, No. 2, pp. 169–177.

de Berg M., Schwarzkopf O., van Kreveld M. and Overmars M. (2000): *Computational Geometry: Algorithms and Applications, 2nd Ed*. — Berlin: Springer.

Dick R.P. and Jha N.K. (1998): *MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of hierarchical heterogeneous distributed embedded systems*. — IEEE Trans. Comput. Aided Des. Integ. Circ. Syst., Vol. 17, No. 10, pp. 920–935.

Eles P., Peng Z., Kuchcinski K. and Doboli A. (1996): *Hardware/software partitioning of VHDL system specifications*. — Proc. *European Design Automation Conference*, Geneva, Switzerland, pp. 434–439.

Eles P., Peng Z., Kuchcinski K. and Doboli A. (1997): *System level hardware/software partitioning based on simulated annealing and tabu search*. — Des. Autom. Emb. Syst., Vol. 2, No. 1, pp. 5–32.

Ernst R., Henkel J. and Benner T. (1993): *Hardware/software cosynthesis for microcontrollers*. — IEEE Des. Test Comput., Vol. 10, No. 4, pp. 64–75.

Fiduccia C.M. and Mattheyses R.M. (1982): *A linear-time heuristic for improving network partitions*. — Proc. 19th *Design Automation Conference*, Piscataway, USA, pp. 175–181.

Grode J., Knudsen P.V. and Madsen J. (1998): *Hardware resource allocation for hardware/software partitioning in the LYCOS system*. — Proc. Conf. *Design Automation and Test in Europe, DATE*, Paris, France, pp. 22–27.

Gupta R.K. and de Micheli G. (1993): *Hardware-software cosynthesis for digital systems*. — IEEE Des. Test Comput., Vol. 10, No. 3, pp. 29–41.

Guthaus M.R., Ringenberg J.S., Ernst D., Austin T.M., Mudge T. and Brown R.B. (1997): *MiBench: A free, commercially representative embedded benchmark suite*. — Proc. IEEE 4th Ann. Workshop *Workload Characterization*, Austin, USA, pp. 3–14.

Hagen L., Huang J.H. and Kahng A.B. (1997): *On implementation choices for iterative improvement partitioning algorithms*. — IEEE Trans. Comput. Aided Des. Integ. Circ. Syst., Vol. 16, No. 10, pp. 1199–1205.

Henkel J. and Ernst R. (2001): *An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques*. — IEEE Trans. VLSI Syst., Vol. 9, No. 2, pp. 273–289.

Hoffmann A.G. (1994): *The dynamic locking heuristic – a new graph partitioning algorithm*. — Proc. IEEE Int. Symp. *Circuits and Systems*, London, UK, pp. 173–176.

Kalavade A. (1995): *System-level codesign of mixed hardware-software systems*. — Ph.D. thesis, University of California, Berkeley, CA, USA.

Kalavade A. and Lee E.A. (1997): *The extended partitioning problem: hardware/software mapping, scheduling and implementation-bin selection*. — Des. Autom. Emb. Syst., Vol. 2, No. 2, pp. 125–164.

Kalavade A. and Subrahmanyam P.A. (1998): *Hardware/software partitioning for multifunction systems*. — IEEE Trans. Comput. Aided Des. Integ. Circ. Syst., Vol. 17, No. 9, pp. 819–837.

Kernighan B.W. and Lin S. (1970): *An efficient heuristic procedure for partitioning graphs*. — Bell Syst. Techn. J., Vol. 49, No. 2, pp. 291–307.

Krishnamurthy B. (1984): *An improved min-cut algorithm for partitioning VLSI networks*. — IEEE Trans. Comput., Vol. 33, No. 5, pp. 438–446.

Lopez-Vallejo M., Grajal J. and Lopez J.C. (2000): *Constraint-driven system partitioning*. — Proc. *Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, pp. 411–416.

Lopez-Vallejo M. and Lopez J.C. (1998): *A knowledge based system for hardware-software partitioning*. — Proc. *Design Automation and Test in Europe, DATE*, Paris, France, pp. 914–915.

Lopez-Vallejo M. and Lopez J.C. (2003): *On the hardware-software partitioning problem: system modeling and partitioning techniques*. — ACM Trans. Des. Autom. Electron. Syst., Vol. 8, No. 3, pp. 269–297.

Madsen J., Grode J., Knudsen P.V., Petersen M.E. and Haxthausen A. (1997): *LYCOS: The Lyngby co-synthesis system*. — Des. Autom. Emb. Syst., Vol. 2, No. 2, pp. 195–236.

Mann Z.Á. and Orbán A. (2003): *Optimization problems in system-level synthesis*. — Proc. 3rd Hungarian-Japanese Symp. *Discrete Mathematics and Its Applications*, Tokyo, Japan, pp. 222–231.

Mei B., Schaumont P. and Vernalde S. (2000): *A hardware/software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems*. — Proc. 11-th ProRISC Workshop *Circuits, Systems and Signal Processing*, Veldhoven, Netherlands, pp. 405–411.

Niemann R. (1998): *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. — Norwell: Kluwer.

Niemann R. and Marwedel P. (1997): *An algorithm for hardware/software partitioning using mixed integer linear programming*. — Des. Autom. Emb. Syst., Vol. 2, No. 2, pp. 165–193.

O'Nils M., Jantsch A., Hemani A. and Tenhunen H. (1995): *Interactive hardware-software partitioning and memory allocation based on data transfer profiling*. — Proc. Int. Conf. *Recent Advances in Mechatronics*, Istanbul, Turkey, pp. 447–452.

Qin S. and He J. (2000): *An algebraic approach to hardware/software partitioning*. — Tech. Rep., No. 206, The United Nations University, International Institute for Software Technology.

Quan G., Hu X. and Greenwood G. (1999): *Preference-driven hierarchical hardware/software partitioning*. — Proc. IEEE/ACM Int. Conf. *Computer Design*, Austin, USA, pp. 652–657.

Sanchis L.A. (1989): *Multiple-way network partitioning*. — IEEE Trans. Comp. Vol. 38, No. 1, pp. 62–81.

Srinivasan V., Radhakrishnan S. and Vemuri R. (1998): *Hardware software partitioning with integrated hardware design space exploration*. — Proc. *Design Automation and Test in Europe, DATE*, Paris, France, pp. 28–35.

Stitt G., Lysecky R. and Vahid F. (2003): *Dynamic hardware/software partitioning: A first approach*. — Proc. IEEE/ACM 40th *Design Automation Conference*, Anaheim, USA, pp. 250–255.

Vahid F. (1997): *Modifying min-cut for hardware and software functional partitioning*. — Proc. Int. Workshop *Hardware-Software Codesign*, Braunschweig, Germany, pp. 43–48.

Vahid F. (2002): *Partitioning sequential programs for CAD using a three-step approach*. — ACM Trans. Des. Autom. Electron. Syst., Vol. 7, No. 3, pp. 413–429.

Vahid F. and Gajski D. (1995): *Clustering for improved system-level functional partitioning*. — Proc. 8th Int. Symp. *System Synthesis*, Cannes, France, pp. 28–33.

Vahid F. and Le T.D. (1997): *Extending the Kernighan/Lin heuristic for hardware and software functional partitioning*. — Des. Autom. Emb. Syst., Vol. 2, No. 2, pp. 237–261.

Wolf W.H. (1997): *An architectural co-synthesis algorithm for distributed embedded computing systems*. — IEEE Trans. VLSI Syst., Vol. 5, No. 2, pp. 218–229.

Wolf W. (2003): *A decade of hardware/software codesign*. — IEEE Comp., Vol. 36, No. 4, pp. 38–43.

Yeh C.W., Cheng C.-K. and Lin T.-T.Y. (1994): *A general purpose, multiple-way partitioning algorithm*. — IEEE Trans. Comput. Aided Des. Integ. Circ. Syst., Vol. 13, No. 12, pp. 1480–1487.

Received: 30 October 2006
Revised: 12 March 2007