Design Automation for Embedded Systems, 2, 237–261 (1997)

Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning

FRANK VAHID vahid@cs.ucr.edu
Department of Computer Science, University of California, Riverside, CA 92521

THUY Dm LE
Department of Computer Science, University of California, Riverside, CA 92521

Abstract. The Kernighan/Lin graph partitioning heuristic, also known as min-cut or group migration, has been extended over several decades very successfully for circuit partitioning. Those extensions customized the heuristic and its associated data structure to rapidly compute the minimum-cut metric central to circuit partitioning; as such, those extensions are not directly applicable to other problems. In this paper, we extend the heuristic for functional partitioning, which in turn can solve the much investigated codesign problem of partitioning a system's coarse-grained functions among hardware and software components. The key extension customizes the heuristic and data structure to rapidly compute execution-time and communication metrics, crucial to hardware and software partitioning, and leads to near-linear time-complexity and excellent resulting quality. Another extension uses a new criteria for terminating the heuristic, eliminating time-consuming and unnecessary fine-tuning of a partition. Our experiments demonstrate extremely fast execution times (just a few seconds) with results matched only by the slower simulated annealing heuristic, meaning that the extended Kernighan/Lin heuristic will likely prove hard to beat for hardware and software functional partitioning.

1. Introduction

The maturation of high-level synthesis has created the capability to compile a single program into assembly-level software, custom digital-hardware, or combined software/hardware implementations. Such a capability may lead to: (1) reconfigurable accelerators, where field-programmable gate arrays (FPGA's) are automatically reconfigured to speedup particular software applications [1], [2], and (2) cheaper, faster, and more quickly designed embedded systems, where advanced design automation tools assist the embedded system designer in quickly exploring and generating design alternatives [3], [4], [5]. In either case, new-generation compilers will be needed that can not only compile to either software or hardware, but that can partition the program among any number of software and hardware parts. Such partitioners must be extremely fast, since they must fit into an environment in which compilations are expected to execute in just seconds or at most minutes. Hardware partitioning is currently done by partitioning structure, but results in [6] demonstrate dramatic improvements to be gained by partitioning functions instead. Hardware/software partitioning is currently done manually, but the advent of partitioning compilers will likely change matters.

Automated hardware/software partitioners are presently the focus of several research efforts. In [3], statement blocks are partitioned among a software and hardware processor using simulated annealing. In [4], statement threads are partitioned using a custom greedy-improvement heuristic. In [7], hierarchical clustering is used to merge statements

together based on their suitability for hardware of software implementation. In [8], tasks in a dataflow graph are simultaneously partitioned and scheduled using a custom constructive heuristic. In [9], basic blocks are partitioned among hardware and software using a dynamic programming algorithm that takes communication into account. In [10], processes derived from a hierarchical state-machine are partitioned among a hardware and software processor, either manually or using hierarchical clustering; several formal transformations are incorporated, such as parallelization. In [5], [11], [12], subroutines and variables are partitioned automatically among standard and custom processors, memories, and buses, using a suite of automated heuristics (e.g., greedy improvement, group migration, clustering and simulated annealing), or interactively using hints and a spreadsheet-like display of metric values and constraints. Further overviews can be found in [13], [14].

Our goal was to develop a heuristic that would : (1) run extremely quickly (executing in just a few seconds), (2) consistently yield excellent results, (3) be applicable to hardware/software partitioning as well as to hardware/hardware functional partitioning, and (4) not restrict the inclusion of new metrics. Since no proposed hardware/software heuristic satisfied all four requirements, we examined the Kernighan/Lin (KL) heuristic [15] (also known as group migration or min-cut). According to Lengauer [16], KL-based heuristics are the main methods for hypergraph partitioning, generating robust and satisfactory solutions for many applications. KL was extended by Fiduccia/Mattheyses (KLFM) [17] to run in linear time, and has since been improved to yield even better results.

With the great success and maturity of KLFM in circuit partitioning, we examined the possibility of re-extending KL for functional partitioning. This paper represents the first work in extending the highly-successful KL heuristic to the new problem of hardware/software partitioning. Because of the very different problem formulation from that of circuit partitioning, creating a set of extensions achieving linear-time complexity is non-trivial. In fact, a straightforward implementation of KL would result instead in $O(n^2)$ complexity. The most important and difficult part of the extension is the integration of the heuristic with a powerful model of behavior execution-time [18]. This model includes a communication model that quickly but accurately computes data transfer times between hardware and software parts as well as within a single part. By incorporating the heuristic with the model, excellent results can be obtained extremely quickly.

The paper is organized as follows. In Section 2, we provide a hardware/software partitioning problem definition. In Section 3, we provide background on KL and its extensions for circuit partitioning. In Section 4, we describe extensions for hardware/software partitioning, which include using an entirely different partitioning metric combined with an efficient data structure, leading to near-linear time-complexity. In Section 5, we describe a technique for obtaining practical reductions in the heuristic's runtime for the case of dealing with coarse estimates, as is the case during hardware/software partitioning. In Section 6, we provide results of experiments showing extremely fast and high-quality results of the heuristic as compared to several standard heuristics. In Section 7, we describe future extensions that might prove useful, similar to extensions made for circuit partitioning. Finally, in Section 8, we provide conclusions.

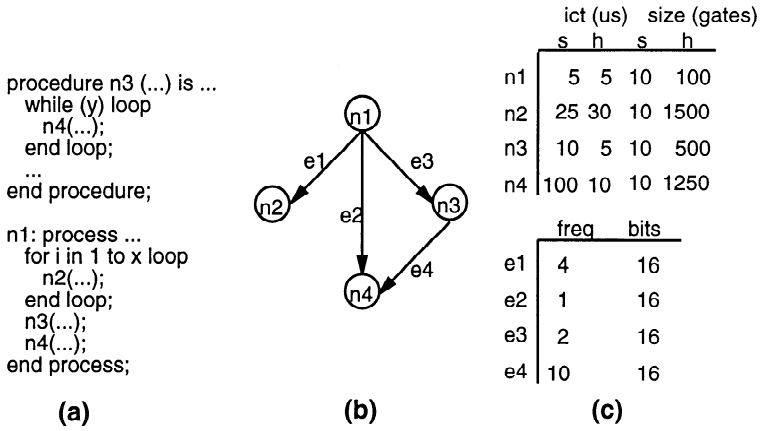

Figure 1. Example: (a) specification, (b) access graph, (c) annotations.

2. Problem Definition

We are given a single sequential process, such as that shown in Figure 1(a), which can be written in C, VHDL or some similar procedural language; we shall describe possible extensions for concurrent processes in Section 7. The program is to be implemented on a target architecture consisting of a standard processor with memory (the software part) and a custom processor possibly with its own memory (the hardware part), the most general version of which is shown in Figure 2. The standard processor may use a system bus, and/or may have several bidirectional data ports (such as commonly found on microcontrollers). The custom processor can access the software memory, as well as its own local memory. The two processors may have different clock speeds.

Our problem is to assign every piece of the program to either the software or hardware part, such that we minimize execution time while satisfying any size and I/O constraints.

To achieve such a partition, we first decompose the program into *functional objects*, where each object represents a subroutine or a variable. (We can always treat certain statement blocks as subroutines to achieve finer granularity [19]). We use the SLIF AG (System-Level Intermediate Format Access Graph) representation for this purpose [18]. The AG represents each object as a node, and each access as an edge, as illustrated in Figure 1(b).

We annotate each node and edge, as shown in Figure 1(c). Each node is annotated with estimates of internal computation times and sizes for each possible part to which it could be assigned; in the example, there are just two parts, software (*s*) and hardware (*h*) (actually, each part would be identified more precisely, such as *Intel 8051* or *Xilinx XC4010*, along with technology files, but we omit such details in this paper). The internal computation time *ict* represents a nodes's execution time on a given part, ignoring any communication time with external nodes. In the example, $n4$ requires 100 clocks for computation in software but only 10 in hardware. Each edge is annotated with the number of bits transferred during

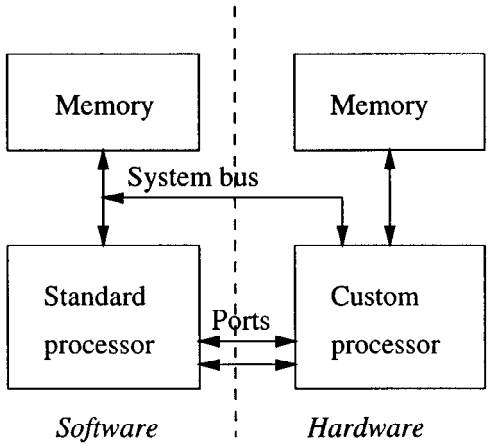

Figure 2. Target architecture.

the access, and the frequency with which the access occurs. The actual time to transfer data during an access over the edge depends on the bus to which the edge is assigned. The bus will have a specified width and protocol time; we need to multiply this time by the number of transfers required to transfer the edge's bits over the bus width. For example, transfering 16 bits over an 8-bit bus with a protocol time of 5 clocks will require $5 * 16/8 = 10$ clocks. Note that all annotation values may represent minimums, maximums, or averages.

Determining the annotations is just part of the metric evaluation process. We use a two-phase approach to metric evaluation. During the "pre-estimation" phase described above, the SLIF nodes and edges are annotated; since only performed once, before partitioning, this phase can take many minutes. During "online-estimation," the annotations are quickly combined using possibly complex equations to obtain metric values for every examined partition during partitioning; since thousands of partitions might be examined, such estimation must be extremely fast. Online-estimation is the focus of Section 4. Pre-estimation is a hard problem, requiring a combination of profilers, estimators, and synthesis tools, but is beyond the scope of this paper. Discussions regarding estimation techniques and accuracies can be found in [5], [20], [12]. For a discussion on a more complex method for hardware size estimation, which considers hardware sharing among functional objects, see [21].

The main difference between an AG and a dataflow graph (DFG) is that an AG usually uses one node for each procedure, and each call to that procedure translates to an edge pointing to that node; in contrast, a DFG may use distinct nodes for each call, resulting in a large number of nodes for examples having a deep, convergent procedure-call hierarchy. We can, of course, clone an AG node whose calls from different locations result in very different procedure executions. We assume there are no recursive procedures, meaning that the AG contains no cycles.

Given the annotated SLIF AG, we can develop an equation for estimating a node's execution time for any given partition. The equation is based on a model in which a node's

execution-time is computed as the sum of its internal computation time (on the current part to which it is assigned) and its communication time. This model will be discussed further in Section 4.

3. Kernighan/Lin Heuristic Background

3.1. Improvement Heuristic Elements

An improvement heuristic is one that, given an initial partition, moves nodes among parts seeking a lower-cost partition. Cost is measured using a cost function. A move is a displacement of a node from one part (e.g., a chip) to another part. We view such heuristics as consisting of two closely-related but distinct elements:

1. **Control strategy**: Includes three key activities—

   (a) *SelectNextMove*: chooses the next move to make.

   (b) *ModifySelCrit*: modifies the selection criteria, usually by reducing the possible moves.

   (c) *Terminate*: returns if some condition is met.

   A control strategy's goal is to overcome local cost minima while making the fewest moves. A local cost minimum is a partition for which no single move improves the cost, but for which a sequence of moves improves the cost; the goal is to find such sequences without examining all possible sequences.

2. **Cost information**: Includes—

   (a) *DS*: the data structure used to model the nodes and their partition, from which cost is computed.

   (b) *UpdateDS*: initializes DS, and modifies DS after a move, ideally in constant time.

   (c) *CostFct*: a function that, given a partition, combines various metric values into a number called cost, representing a partition's quality. Ideally, the cost function requires only constant time. We use the convention that lower cost is better.

(The term "heuristic" often refers to both elements, but can also refer to just the control strategy; in this paper we use it to refer to both elements).

For example, in simulated annealing [22], *SelectNextMove* chooses moves at random, rejecting more cost-increasing moves as the "temperature" is lowered. *Terminate* exits if the temperature indicates a frozen status. *ModifySelCrit* lowers the temperature if no lower-

cost partition has been found for some time (representing attainment of an equilibrium state at the current temperature). *Cost information* is completely user-defined.

3.2. The Kernighan/Lin Heuristic

The KL heuristic seeks to improve a given two-way graph partition by reducing the edges crossing between parts, known as the *cut*. KL's *CostFct* measures the cut size. The essence of the heuristic is its simple yet powerful control strategy, which overcomes many local minima without using excessive moves.

The control strategy can be summarized as follows. *SelectNextMove* measures the cost of all possible swaps of unlocked nodes in opposite parts, and then swaps the nodes with the best gain (greatest cost decrease or least cost increase). If all nodes are locked, the heuristic goes back to the lowest-cost partition seen, thus completing one iteration. *ModifySelCrit* locks the swapped nodes, removing them from consideration during future moves; if all nodes are locked and the iteration decreased the cost, then it unlocks all nodes and repeats. *Terminate* returns if the iteration did not decrease the cost.

We write the strategy in algorithmic form in Algorithm 3.1. Assume N is the set of nodes $n_1, n_2, \ldots, n_n$ to be partitioned, and the two-way partition is given as $P = \langle p_1, p_2, DS \rangle$, where $p_1 \bigcap p_2 = \emptyset$ and $p_1 \bigcup p_2 = N$.

Algorithm 3.1. KLControlStrategy(P)

```
IterationLoop: loop // Usually < 5 passes
   currP = bestP = P
   UnlockedLoop: while (UnlockedNodesExist(currP)) loop
      swap = SelectNextMove(currP)
      currP = MoveAndLockNodes(currP, swap)
      bestP = GetBetterPartition(bestP, currP)
   end loop

   if not (CostFct(bestP) < CostFct(P)) then
      return P // Terminate; no improvement this pass
   else // Do another iteration
      P = bestP, UnlockAllNodes(P)
   end if
end loop

// Find best (or least worst) swap by trying all
procedure SelectNextMove (P)
   SwapLoop: for each (unlocked n_i ∈ p_1, n_j ∈ p_2) loop
      Append(costlog, CostFct(Swap(P, n_i, n_j))
   end loop
   return (n_i, n_j swap in costlog with lowest cost)
```

Note that the innermost loop *SwapLoop* has a time complexity of n^2, where n is the number of nodes in N. This loop is called within the *UnlockedLoop* loop, which itself has complexity n. Both are enclosed within *IterationLoop*, which experimentally has been found to have a small constant complexity, say c_1. Hence, the runtime complexity of KL is $c_1 \times n \times n^2$, or $O(n^3)$, though certain modifications can reduce it to $O(n^2 log(n))$ [16].

3.3. Fiduccia/Mattheyses Extensions

Fiduccia/Mattheyses [17] made three key extensions to KL:

1. They redefined the cut metric for hypergraphs, rather than graphs.
2. They redefined a move as a move of a single object from one part to another, rather than as a swap of two objects.
3. They described a data structure that permitted *SelectNextMove* to find the best next move in constant time.

We refer to KL with these extensions as KLFM.

Regarding the first extension, the difference between a hypergraph and graph is that the former's edges, or hyperedges, may connect more than just two nodes. Hypergraphs more closely model real circuits.

The second extension reduces the complexity of *SelectNextMove* from $O(n^2)$ down to $O(n)$, since we now consider an average of $n/2$ moves during each call to the procedure. The redefinition has the disadvantage of slightly increasing the final partition's cost, since fewer partitions are being examined, but it has the added advantage of permitting unbalanced parts (though KLFM still requires a minimal balancing).

Regarding the third extension, the data structure maintains possible moves in an array. Each node is stored in the array at an index corresponding to the gain achieved when moving the node. Because several nodes could have the same gain, each array item is actually a list. In Algorithm 3.2, we see that *SelectNextMove* now performs two operations: *PopBestMove* and *UpdateDS*. *PopBestMove* must remove the first object in the array. *UpdateDS* must update gains of neighboring objects and then reposition those objects in the array. These operations are performed in constant time as described in [17], so the entire procedure requires only constant time, say c_2. *IterationLoop* has been found to loop a constant number of times, say c_1, and *UnlockedLoop* still requires n iterations, so the time complexity is $c_1 \times n \times c_2$, or $O(n)$.

Algorithm 3.2. KLFM's SelectNextMove(P)
```
best_move = PopBestMove(P)
UpdateDS(P, best_move)
return (best_move)
```

c_1 of KLFM has been found to be slightly greater than c_1 of KL, because there are fewer single object moves than swaps, so KLFM examines fewer moves per iteration than KL, thus requiring more iterations [16].

3.4. Lookahead and Multiway Extensions

Krishnamurthy [23] sought to decrease KLFM's final partition cost by replacing the arbitrary choice of equal-gain moves by a more intelligent choice. He introduced the concept of

lookahead, by extending the gain to a sequence of numbers. The first number in the sequence is the gain as defined above. The subsequent numbers in the sequence are other metrics used to distinguish between objects with the same gain. Sanchis [24] extended Krishnamurthy's technique for multiway partitioning.

4. Extensions for Hardware/Software Partitioning

We now describe three extensions to KL for hardware/software partitioning, similar in idea but very different in detail from the KLFM extensions for hypergraph partitioning:

1. We replace the cut metric by an execution-time metric.
2. We redefine a move as a single object move, rather than a swap.
3. We describe a data structure that permits *SelectNextMove* to find the best next move in constant time.

We now describe the details of each extension.

4.1. Replacing Cut by Execution-Time Including Communication

This modification is the basis of the extensions for hardware/software partitioning. We first noted that minimizing wires between parts (i.e., the cut) is not a particularly relevant goal during hardware/software partitioning, since the wires between parts are fixed, or can be easily reduced by time-multiplexing data transfers between parts. (Note that such time multiplexing is not usually possible during circuit partitioning, since the scheduling of transfers to clock cycles is done before partitioning—this is one reason we might perform functional partitioning rather than circuit partitioning even when partitioning among hardware blocks only). We might consider replacing the goal of minimizing cut by the goal of minimizing bits communicated between parts. However, minimizing bits is merely an indirect measure of our real goal, which is to minimize execution time.

There are other metrics that must also be considered during hardware/software partitioning, such as hardware and software size and I/O. Since the techniques for incorporating those metrics are straightforward, we discuss them later Section 7, and focus in this paper on the hardest technique of minimizing execution time.

A node's execution time can be computed using a model in which the execution-time equals the sum of the node's internal computation time and the time spent accessing other nodes. This model can be captured as an equation as described in [18]; a simplified form of the equation is:

$$n.et = n.ict + n.ct \tag{1}$$

$$\begin{aligned} n.ict &= n.ict_p,\ p \text{ is } n\text{'s current part} \\ n.ct &= \textstyle\sum_{e_k \in n.outedges} e_k.freq \times (e_k.tt + (e_k.dst).et) \\ e_k.tt &= \lceil bus\_delay \times (e_k.bits \div bus\_width) \rceil \end{aligned}$$

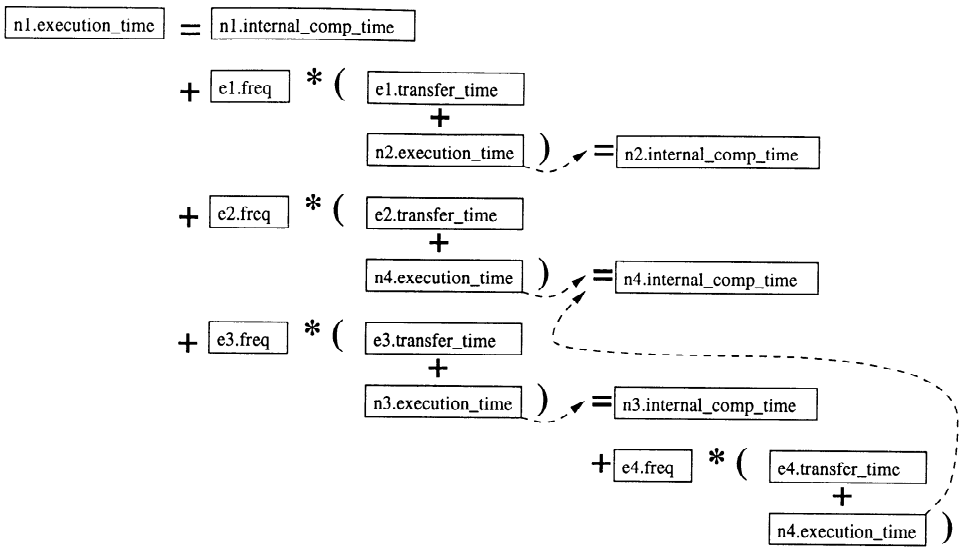

Figure 3. Execution-time model.

In other words, a node n's execution time $n.et$ equals its internal computation time $n.ict$ plus its communication time $n.ct$. Note that a node's ict may differ on different parts; for example, a node might have $n.ict_{sw} = 100us$ and $n.ict_{hw} = 5us$. A node's ct equals the transfer time $e_k.tt$ over each outgoing edge e_k, plus the execution time of each accessed object $(e_k.dst).et$, times the number of such accesses $e_k.freq$. The transfer time equals the bus delay, times a factor denoting the number of transfers required to transfer the edge's bits over the given bus width. The bus delay is the time required by the bus' protocol to achieve a single data transfer. We associate two such delay's with each bus, one for when the edge is contained within a single part, and another when the edge crosses between parts; the latter is usually larger.

The equation for *n1* from the earlier example is shown graphically in Figure 3. Details of the transfer times have been omitted for simplicity.

Though the model yields some inaccuracies since some computation and communication could actually occur in parallel, it provides a powerful means for obtaining quick yet fairly accurate execution-time estimates. The communication model is quite general since each edge can be associated with buses of various delays and widths, since different communication times can be used for inter-part and intra-part communication, and since more sophisticated transfer-time equations could be used—for example, one could include a factor to reduce actual transfer-time based on bus load [25]. The actual implementation scheme can be determined during an interface synthesis step that follows partitioning [5], [12].

Given the above definition of a node's execution time, we obtain the equations shown in Figure 4(a) for the execution times of the nodes in Figure 1. Suppose there is a time-

```
n2.et  =  n2.ict
n4.et  =  n4.ict
n3.et  =  n3.ict + e4.freq * (e4.tt + n4.et)
n1.et  =  n1.ict + e1.freq * (e1.tt + n2.et)
                 + e2.freq * (e2.tt + n4.et)
                 + e3.freq * (e3.tt + n3.et)
```

(a)

```
n1.et  =  n1.ict
        + n2.ict * e1.freq
        + n3.ict * e3.freq
        + n4.ict * (e2.freq + e3.freq*e4.freq)
        + e1.tt * e1.freq
        + e2.tt * e2.freq
        + e3.tt * e3.freq
        + e4.tt * e3.freq * e4.freq
```

(b)

Figure 4. Execution-time: (a) for all nodes, (b) for n1, after inlining.

constraint on $n1$, meaning we are most concerned with $n1$'s execution time equation. Figure 4(b) shows $n1$'s equation after inlining all other nodes' execution-time equations. (Note that such inlining would not have been possible if the original program was recursive).

4.2. Redefining a Move as a Single Object Move

Swaps were used in KL to maintain balanced part sizes. Although we have two parts (software and hardware), we assume that the part sizes having different units (i.e., instructions versus gates) invalidates the need for balanced part sizes. Therefore, we redefine a move as a single object move, since single object moves permit unbalanced numbers of objects in each part.

4.3. Data Structure

Ideally, we would build a data structure DS such that *SelectNextMove* executes in constant-time. In a straightforward approach, we would build a *SelectNextMove* that tried all possible moves and picked the best (as in Algorithm 3.1, but replacing swaps by single-object moves), but this approach would yield a linear-time *SelectNextMove* and thus a KL of

$O(n^2)$, assuming that the cost function and DS updates are designed to execute in constant time. Instead, we divide *SelectNextMove* into two parts, *PopNextMove* and *UpdateDS*, as in Algorithm 3.2, and we try to build these to execute in constant time, as in KLFM. However, because of the very different characteristic of the execution-time metric from the cut metric, we will not be able to achieve constant time, but instead logarithmic time.

First, we note that moving a node affects the execution time of that node and its ancestors, and not of all nodes. In particular, given the equation of Figure 4(a), we observe that when an AG node n is moved from one part to another, the node's execution time $n.et$ may change due to a change in $n.ict$, and due to a change in the transfer time of any adjacent edge (i.e., a change in $e.tt$ of any e connecting n). We also observe that any node whose communication time changes (due either to a change in an $e.tt$ or an $n.et$) will have a changed execution time. Therefore, moving a node changes the execution time of the node itself and of accessing nodes. Changes in an accessing node m's execution time further changes the execution time of m's accessing nodes. Not all nodes are changed, but rather only those that are ancestors of the moved node in the AG. There is thus a local effect on execution time, though not as localized as the effect on cut, where only immediate neighbors are affected. This localized effect means that we may need only update a small portion of DS when a node is moved.

Next, we devise a technique to rapidly compute the change in execution-time when moving a node in a partition, so that we can determine the best node to move. Since the execution time is computed by an equation, as the one in Figure 4(b), we must compute how each possible node move changes the result of this equation. The equation terms that may change are the node ict values and the edge tt values; the change of $n1.ict$ is written as $Dn1.ict$, of $e1.tt$ as $De1.tt$, and so on, as illustrated in Figure 5. Note that the $e.freq$ values are constants. Figure 5 shows the terms that change when a particular object is moved. For example, moving $n2$ changes $n2.ict$ and $e1.tt$. Based on these relationships, we collect all terms from Figure 4(b) that change for a given move, and create a *change equation* that computes the change in $n1.et$ for that move, as shown in Figure 6(a).

We are now ready to build the DS. Given the above change equations, we build a *change list*, which is an array where nodes are inserted at the index corresponding to their change values. Because multiple nodes could have the same change value, each array item is actually a list. To build the change list, we evaluate each change equation to compute $Dn1.et$ for each node. We store each node at $ChangeList(Dn1.et)$, as shown in Figure 6(b). The array has a minimum index of $-$*MaxIncrease* and a maximum index of *MaxIncrease*, where *MaxIncrease* can be conservatively chosen as the worst-case execution time of $n1$. As we insert nodes into the array, we compare the current index with *BestChange*, which is updated if the current index is closer to the front of the array. Thus, *BestChange* will be the index of the best node.

Now we must implement *SelectNextMove* (see Algorithm 3.2). The first part, *PopNextMove*, consists of deleting the best node from the change list, and updating *BestChange*. Finding and deleting the best node clearly can be done in constant time. However, updating *BestChange* is not as easy, because we do not know where the next node is in the array. We use the approach in [17] of decrementing *BestChange* until we find a non-empty array item. Such a decrement approach was proven in [17] to be constant time for the cut metric.

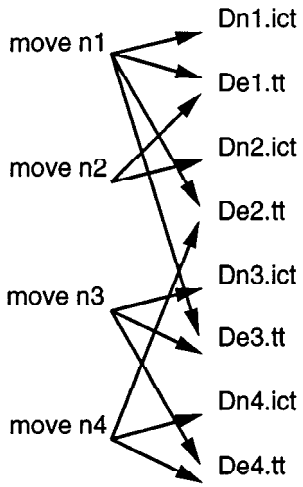

Figure 5. Terms that change during moves.

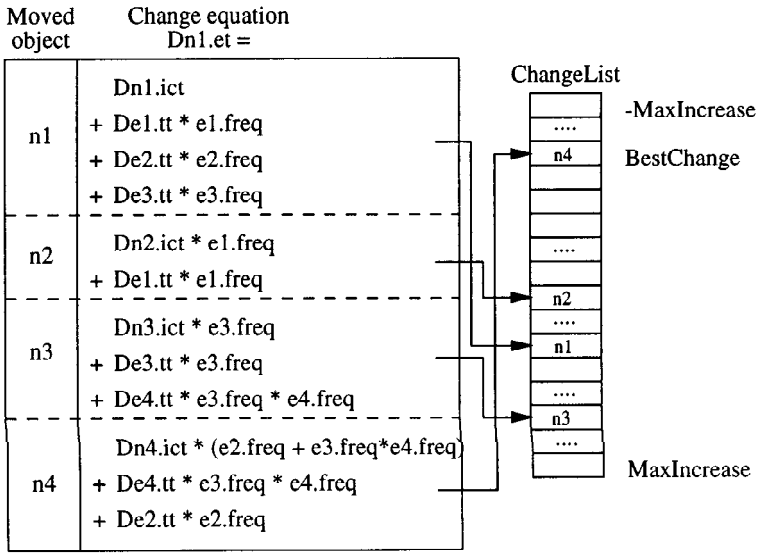

Figure 6. Data structure: (a) change equations, (b) change list.

While we have not looked into such a proof for the execution-time metric, we note that we can view the number of decrements as some constant number less than *MaxIncrease*.

The second part, *UpdateDS*, consists of recomputing change values. We need not recompute all change equations; instead, we only recompute those containing a term that changed during the move (see Figure 5 for terms that change). If a node's change equation result is updated, we delete and reinsert the node in gain list, updating *BestChange* if necessary.

The complexity of updating the data structure is $\log n$, not a constant c as in KLFM; we shall now explain why. In KLFM, a node move affects the gain values of the node's neighbors. In our problem, a node move affects the gain values of the node's *ancestors*. In the absolute worst case, a node in the KLFM problem could have n neighbors and a node in our problem could have n ancestors, but fortunately, these cases rarely or never occur. In the KLFM problem, the worst case corresponds to a graph where each node is adjacent to every other node, i.e., a fully-connected graph. However, since the graph represents a logic network, the graph will instead have nodes adjacent to a small number c of neighbors. Likewise, in our problem, the worst case corresponds to each node having indegree of 1 and outdegree of 1, i.e., a chain of nodes. (A fully-connected AG would also yield n ancestors per node, but this can't occur since we don't allow cycles). However, since the SLIF AG represents accesses among procedures and variables, the SLIF AG will instead have nodes adjacent to a small number of neighbors. In other words, people don't write programs where every procedure accesses exactly one other procedure and is accessed from exactly one procedure; instead, each procedure accesses a small number of other procedures, and may be called from more than one procedure. In a SLIF-AG derived from such a program, the depth of the directed graph will usually not exceed $\log n$ (this number was verified with several examples in [26]).

To determine the complexity of the heuristic, we note that *IterationLoop* once again loops a constant number of times, say c_1. *UnlockedLoop* still requires n iterations. *SelectNextMove* consists of *PopBestMove* and *UpdateDS*. We assume the former requires constant time, say c_2, while the latter requires $\log n$, as described above. Thus, the time complexity is $c_1 \times n \times (c_2 + \log n)$, or $O(n \log n)$. While this complexity does not meet that of KLFM, it is still quite good.

4.4. Example

Figure 7 provides an example of applying the extended KL heuristic on the example of Figure 1. We arbitrarily start with all nodes assigned to software, though any intitial partition would be possible. The $n1.et$ equation of Figure 4(b) evaluates to 2205 for such a partition. We obtain values for the nodes' change equations of Figure 6(a), and insert each node into the change list corresponding to its change value. In Figure 7, we pop the best node $n4$ and move it to hardware, update change equations for $n3$ and $n1$ ($n2$ is unaffected, and $n4$ is locked), update $n1.et$ by the change value $(2205 - 1680)$, and reinsert $n3$ and $n1$ into the change list. We continue such popping and updating until all nodes are moved exactly once. Finally, we return to the partition with the lowest $n1.et$, which in this case happens to be the last partition. Note that the local minimum of 335 was overcome by the KL control strategy. Also note that we only considered execution time in the example for

simplicity, causing all nodes to go to hardware. When other metrics are also considered, as to be discussed in Section 4.5, the final outcome will be different.

4.5. Incorporating Multiple Metrics

For simplicity, the technique until now has focused on minimizing the execution-time of a single behavior. We now describe incorporation of other metrics, such as hardware size, software size, hardware I/O, software I/O, and multiple behaviors' execution-time.

First, let us consider multiple metrics m_i whose values should be minimized. We can use the following cost function to combine the values into one number (an improved function will be discussed shortly):

$$CostFct = m_1 + m_2 + \cdots m_l \quad (2)$$

We maintain different information for each type of metric:

- Behavior execution-time—For each such metric, we maintain a unique set of change equations as defined earlier.
- Hardware size—When hardware size is computed as the sum of node sizes, then the node's hardware size annotation indicates the change value for this metric.
- Software size—Since software size is computed as the sum of node sizes, the node's software size annotation indicates the change value for this metric.
- Hardware and software I/O—This metric is the same as the cut metric in KLFM. We treat it in a similar manner as in KLFM. Note that software I/O is relevant when the standard processor has multiple data ports (commonly the case for a microcontroller), which act as general I/O pins rather than a system bus, and we wish to use as few of those pins as possible.

We compute initial change values for each node for each metric. These change values are summed to give the total change in cost that would be obtained when moving a node. The total change is then used as the index into the change array.

Figure 8 provides an example of incorporating four metrics: *n1*'s execution time, *n4*'s execution time, hardware size, and software size. Initially, we compute change values for each metric independently, combine those values into a single *Change* value, and insert nodes into the change list. We pop the best node *n4* and then update any change values; only the *n1* and *n3* execution-time values must be updated. We recombine change values and repeat until all nodes have been moved. In this example, the best overall cost occurs when only *n4* is moved to hardware. The hardware size metric thus prevented all nodes from moving to hardware, in contrast with the previous example.

The above cost function can be further improved as follows:

$$CostFct = k_1 \cdot F(m_1) + k_2 \cdot F(m_2) + \cdots k_l \cdot F(m_l) \quad (3)$$

The k's are user-defined constants indicating the relative importance of each metric. The function F normalizes each value to a number between 0 and 1000. Such normalization

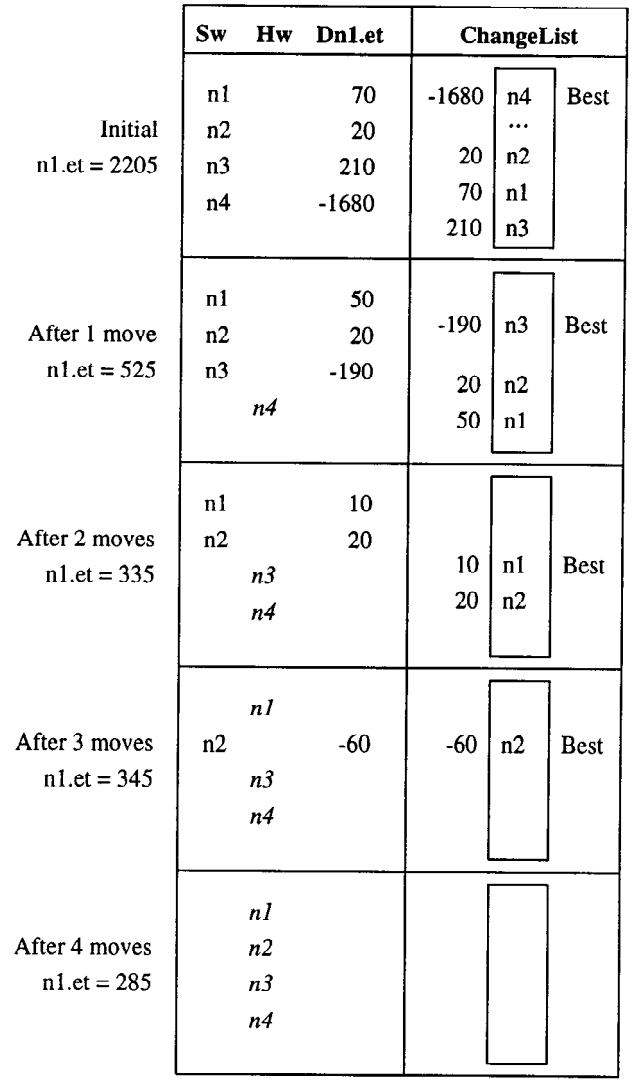

Figure 7. Extended KL example using the change list.

	Sw	Hw	Dn1.et	Dn4.et	hw.size	sw.size	Change	ChangeList		
Initial	n1		70	0	100	-10	160	-530	n4	Best
	n2		20	0	1500	-10	1510	160	n1	
	n3		210	0	500	-10	700	700	n3	
	n4		-1680	-90	1250	-10	-530	1510	n2	
After 1 move	n1		50	0	100	-10	140			
	n2		20	0	1500	-10	1510	140	n1	Best
	n3		-190	0	500	-10	300	300	n3	
		n4						1510	n2	
After 2 moves		*n1*								
	n2		-60	0	1500	-10	1430	260	n3	Best
	n3		-230	0	500	-10	260	1430	n2	
		n4								
After 3 moves		*n1*								
	n2		-60	0	1500	-10	1430			
		n3								
		n4						1430	n2	Best

Figure 8. Example of change list use with multiple metrics.

can be achieved by dividing the value by the largest possible value for that metric, and multiplying by 1000. For example, a node's execution-time value can be divided by the largest possible change in execution time, which is determined by finding the largest possible change for each node. Likewise, a hardware size value can be divided by the largest possible hardware size change, which simply equals the hardware size of the largest node. We multiply by 1000 so that the cost can still serve as an index into the change list. Note that such normalizing actually solves the problem of requiring a huge change list to account for the maximum possible change; the unnormalized maximum change in execution time exceeded 1,000,000 in some examples. (KLFM does not encounter this problem since the maximum change is just the most that cutsize could change when moving a single node). For further details on normalizing metrics in a cost function, see [11], [5].

We have assumed that each metric is an optimization metric, i.e., that we seek to minimize the value of the metric. However, some metrics are merely constraint metrics. For example, we may have 10,000 gates available in hardware, and we might only be concerned with not violating that constraint. In this case, we need not minimize hardware size below 10,000 gates. Such constraint metrics can be handled with a simple modification of F. Any change that happens below the constraint value is ignored and thus forced to 0. For example, if

the hardware size constraint is 10,000 gates, the current metric value is 5,000 gates, and moving $n1$ would change this by 2,000 gates while moving $n2$ would change this by 6,000 gates, then the hardware size change value for $n1$ is 0, and the change value for $n2$ is $(5,000 + 6,000) - 10,000$, or 1,000.

5. Extension for Faster Termination

There are two observations that led us to find a KL termination criteria that led to substantial practical reductions in KL runtimes. First, there is a well-known notion that we should not record a numerical value to a degree of precision that exceeds the accuracy of the measuring instrument. For example, if an instrument measures mass to 0.01 milligrams of accuracy, we should not record measurements to 0.001 degree of precision. A similar notion applies to hardware/software partitioning. Specifically, we should not partition to reduce cost to a degree of precision that exceeds the accuracy of our estimates. For example, if our execution-time estimate is accurate to within 10%, then we need not spend time reducing cost by amounts less than 10%.

Second, we observed after numerous KL trials that the amount by which each iteration reduced the cost usually decreased after each iteration. In particular, if an iteration decreased the cost by X%, then it rarely occurred that a subsequent iteration decreased cost by a Y% such that $Y > X$; instead, Y was usually less than X, eventually becoming 0 after which KL terminates. In other words, the *change* in cost as a function of iteration number usually monotonically decreased.

Based on the above observations, we developed a new *Terminate* function for KL. Rather than exiting when an iteration's initial and final costs are equal (i.e., 0% improvement), we exit when those costs are within some degree of precision. In other words, when the difference between one iterations's cost and the next is less than some percentage of the original cost, we terminate. We experimented with several precisions, and present results in Section 6.

6. Experiments

We performed three types of experiments. First, we compared the quality of results obtained by KL with other common heuristics. Second, we compared the runtimes of our version of KL with a straightforward implementation of KL. Finally, we compared results using various termination precisions.

6.1. Quality of Results

Table 1 provides a comparison of the quality of results on several examples for the following heuristics: Random assignment (*Ra*), Greedy improvement (*Gd*), KL for functional partitioning (*KL*), hierarchical clustering (*Cl*), clustering followed by greedy improvement (*Cg*), and simulated annealing (*SA*). *Gr* moves nodes from one part to another as long as

Table 1. Cost comparison for functional partitioning heuristics.

Ex	P	Ra	Gd	KL	Cl	Cg	SA
1	2	314	68	40	85	59	15
	3	443	50	0	168	96	22
	4	428	88	29	218	15	16
	hs	576	61	16	88	66	18
2	2	236	69	43	141	34	47
	3	256	25	7	244	16	0
	4	234	0	2	339	15	0
	hs	160	0	0	0	0	0
3	2	893	90	68	111	78	30
	3	1081	115	71	154	142	63
	4	1220	141	100	141	137	94
	hs	2115	83	20	147	144	20
4	2	960	105	60	109	62	7
	3	1206	114	114	155	5	97
	4	1338	66	39	193	37	72
	hs	660	102	23	102	76	0
	Avg	758	74	40	150	57	31

cost reductions are obtained, having a computational complexity of $O(n)$. *KL* was described in this paper, with complexity $O(n \log n)$. *Cl* computes closeness between all pairs of nodes, using the closeness metrics of communication, connectivity, common accessors, and balanced size, and applies pairwise merging, having complexity $O(n^2)$. *Cg* is *Cl* followed by *Gd*, thus having complexity $O(n^2)$. *Sa* uses random moves to escape even more local minima, at the expense of generally long runtimes. Annealing parameters included a temperature range of 50 down to 1, a temperature reduction factor of 0.93, an equilibrium condition of 200 moves with no change, and an acceptance function as defined in [5]. The complexity of *Sa* is generally unknown, but its CPU times with the above parameters usually exceed those of $O(n^2)$ heuristics. *Gd*, *KL*, and *Sa* all use the output of *Ra* as their initial partition.

The four examples were VHDL descriptions of a volume-measuring medical instrument (*Ex1*), a telephone answering machine (*Ex2*), an interactive-TV processor (*Ex3*), and an Ethernet coprocessor (*Ex4*).

The partition cost *C* is a unitless number indicating the magnitude of estimated constraint violations [5]. Constraints on hardware size, software size, hardware I/O, and execution time were intentionally formulated such that there would be constraint violations (non-zero cost), so that we could compare how close each heuristic came to achieving zero cost. Each example was partitioned among *2* ASICs (VTI), *3* ASICs, *4* ASICs, and a hardware/software (*hs*) configuration of one processor (8086) and one ASIC.

KL's complexity of $O(n \log n)$ is just above that of the greedy heuristic *Gr*, yet, as the table demonstrates, *KL* outperformed all of the other heuristics except for simulated annealing.

Table 2. Runtimes of KL for f.p. vs. straightforward KL (real examples).

Ex	Nodes	KL for f.p. (s)	KL (s)
1	30	.7	1.8
2	45	1.1	1.1
3	70	1.4	21.0
4	85	1.5	19.7
5	123	2.2	58.1

6.2. Runtimes

We compared the runtime of KL extended for functional partitioning (*KL for f.p.)* with the runtime of a straightforward KL implementation (*KL*)—note that resulting costs for the two heuristics will be identical, since they make exactly the same moves. In the straightforward implementation, the next best move is determined by tentatively moving every free node, rather than using a change equations and a change list, as was done in the original KL heuristic.

Results are summarized in Table 2 and Table 3. The heuristics were run on five real examples: a microwave-transmitter controller (*Ex1*), an answering machine (*Ex2*), a fuzzy-logic controller (*Ex3*), an interactive TV processor (*Ex4*), and an Ethernet coprocessor (*Ex5*). These examples ranged in size from 30 nodes to 123 nodes. The heuristics were also run on generated examples ranging in size from 10 nodes to 200 nodes in increments of 10. Such a range enables us to see how the heuristics scale with problem size—see [26] for information on generated examples.

Because the number of iterations of KL usually varies from 2 to 6, different examples may yield runtime fluctuations not related to the size of those examples. Thus, because our goal is to observe the *difference* in runtimes between the two KL versions, we ran each version for exactly one iteration.

The results clearly show that the extended KL is far faster than the straightforward KL. The extended KL scales nearly linearly with problem size, whereas the non-extended KL grows quadratically, as expected. The extended KL handled problems of quite a large size (200 nodes) in just a few seconds. Times were measured on a 166Mhz Pentium.

One should keep in mind that the improvement in speed is gained with *absolutely no loss in quality*. The sequences of moves made by each heuristic are identical, but the time required to determine the next best move is greatly reduced in the extended KL.

6.3. Faster Termination

We have performed two experiments to demonstrate the extension for faster termination of KL. In the experiments, we examine two factors: the reduction of the partition cost in successive partitioning iterations, and the runtime of the heuristic. In the first experiment, we used KL to perform hardware/software partitioning among an Intel 8086 processor and an FPGA on six real examples and eight generated generic examples ranging from 40 nodes

Table 3. Runtimes of KL for f.p. vs. straightforward KL (generated examples).

Nodes	KL for f.p. (s)	KL (s)
10	0.8	0.8
20	0.7	0.6
30	0.7	0.7
40	1.2	3.0
50	1.0	4.2
60	1.5	7.1
70	1.6	21.1
80	1.7	16.7
90	1.8	20.9
100	2.4	56.3
110	2.3	62.8
120	3.1	49.0
130	3.2	64.5
140	3.5	84.0
150	4.0	148.0
160	4.3	102.3
170	4.5	115.1
180	4.7	157.3
190	4.8	174.0
200	5.2	200.0

to 120 nodes. Figures 9 and 10 demonstrate the number of iterations KL required on the examples, and the cost resulting after each iteration; for presentation purposes, we have normalized costs between 0 and 1. Table 4 demonstrates the time reductions achieved if we terminate when an iteration reduces cost by less than 10%, 5% or 1%. If we terminate hw/sw partitioning of the real examples when an iteration reduces cost by less than 10%, we reduce average runtime by 53%. For precisions of 5% and 1%, we reduce average runtime by 43% and 10%, respectively. The substantial reductions in runtime are paid for by only a small cost increase, as illustrated in Table 5. For the hardware/software partitioning of the six real examples at 5% precision, we can reduce average runtime by 43% at an average cost increase of only 2.4%. Because we likely do not have estimations that are accurate to that degree, we are not concerned with this small cost increase.

In the second experiment, we performed hardware/hardware partitioning among two FPGAs on six real examples and eight generated generic examples ranging from 40 nodes to 120 nodes. Figure 11 and 12 summarize costs per iteration. As with the hardware/software partitioning, the runtime reductions are nearly 50% in many cases, with cost increases of only 1 or 2%.

7. Discussion and Future Work

While we have focused on partitioning among one hardware and one software part in this paper, the technique can be extended to any number of parts. We maintain change equations for each constrained or optimization metric, and each entry in the change list represents a

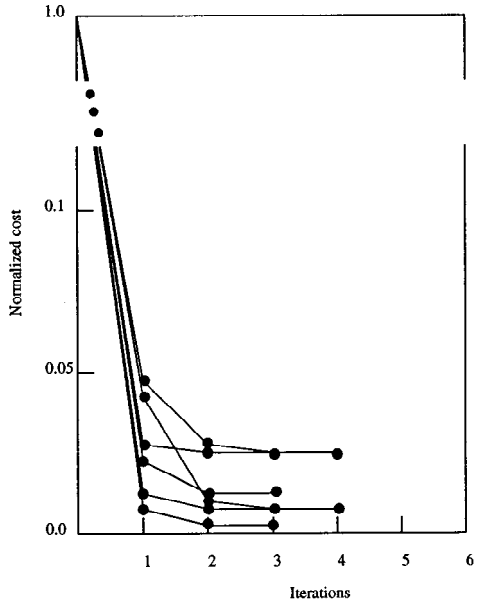

Figure 9. Iterations vs. cost (hw/sw, real examples).

Table 4. Average time reduction.

Precision	Hardware/software		Hardware/hardware	
	real	generated	real	generated
10%	53%	56%	49%	62%
5%	43%	52%	36%	47%
1%	10%	36%	27%	24%

move of an object to a particular part; thus, each node will appear in the change list more than once (M-1 times, where M is the number of parts).

The extended KL can be applied to multiple processes without modification. If there is more than one constrained node, we simply maintain unique change equations for each. When partitioning among hardware parts, we can assume that each process is implemented on its own custom processor, so there is no multi-tasking overhead. However, when partitioning onto a software part, we might want to consider multi-tasking overhead (e.g., as achieved using a real-time operating system) to obtain more accurate execution-time estimations. One method for considering this is to keep track of a processor's load, and reduce internal computation time values by some factor when the load is heavy. This remains an area for future work.

Other future extensions include adding lookahead and multiway lookahead to the hardware/software partitioning technique in this paper; such additions might improve the quality

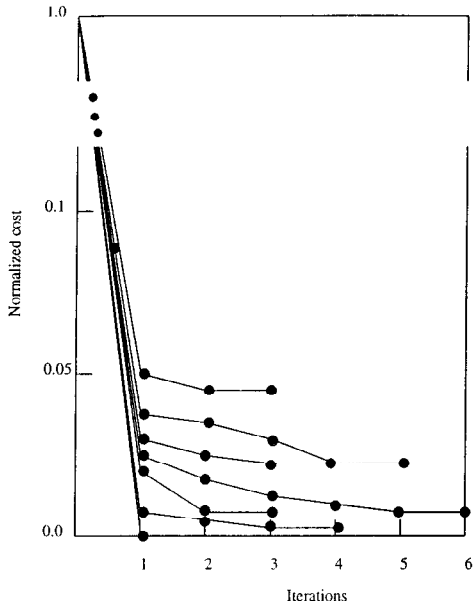

Figure 10. Iterations vs. cost (hw/sw, generated examples).

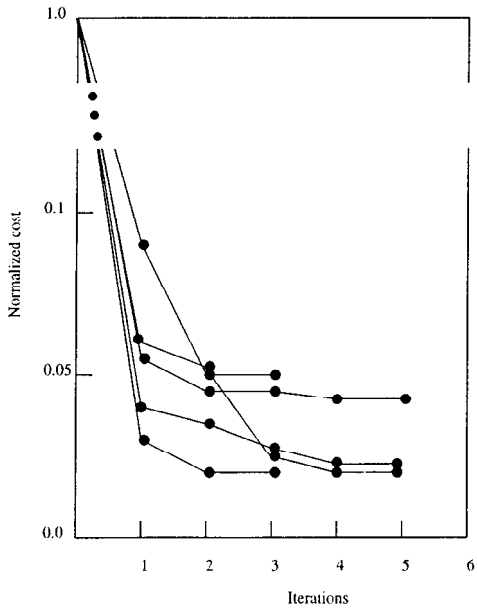

Figure 11. Iterations vs. cost (hw/hw, real examples).

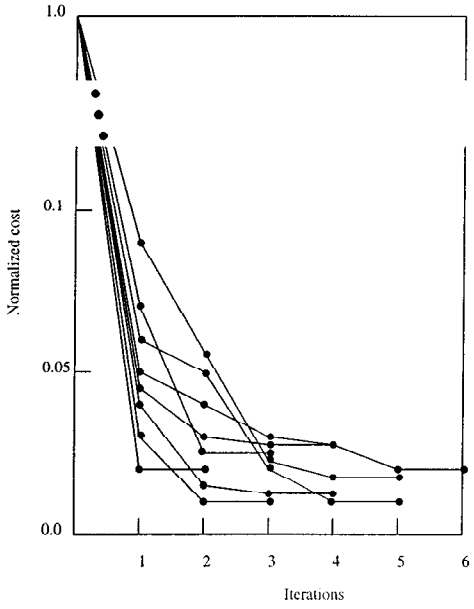

Figure 12. Iterations vs. cost (hw/hw, generated examples).

Table 5. Average cost increase.

Precision	Hardware/software		Hardware/hardware	
	real	generated	real	generated
10%	2.4%	2.1%	1.4%	3.3%
5%	2.4%	0.7%	1.4%	1.2%
1%	0.3%	0.5%	0.4%	0.4%

of results in the same way that they improved circuit partitioning. Another future area of research is to extend the technique to address the problem of combined partitioning with scheduling, a problem addressed in [8]. Finally, performing transformations during partitioning, such as cloning nodes, would likely lead to much improved results.

8. Conclusions

We have extended the successful Kernighan/Lin partitioning heuristic for functional partitioning. The new heuristic: (1) runs extremely quickly, having a time-complexity of just $O(n \log n)$, and completing in just seconds for numerous examples; (2) achieves excellent results, nearly equal to results achieved by simulated annealing running for an order of magnitude more time; (3) can be applied to hardware/software partitioning as well as

hardware/hardware functional partitioning; (4) allows addition of new metrics. With these features, especially its speed and result quality, the heuristic will likely prove hard to beat for functional partitioning, and is ideally suited for new-generation compiler/partitioners.

References

1. P. Athanas and H. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer* 26: 11–18, March 1993.
2. R. Hartenstein, J. Becker, and R. Kress. Two-level partitioning of image processing algorithms for the parallel map-oriented machine. In *International Workshop on Hardware-Software Co-Design*, pp. 77–84. 1996.
3. R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. In *IEEE Design & Test of Computers*, pp. 64–75, December 1994.
4. R. Gupta and G. DeMicheli. Hardware-software cosynthesis for digital systems. In *IEEE Design & Test of Computers*, pp. 29–41, October 1993.
5. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, New Jersey, 1994.
6. F. Vahid, T. Le, and Y. Hsu. A comparison of functional and structural partitioning. In *International Symposium on System Synthesis*, pp. 121–126, 1996.
7. X. Xiong, E. Barros, and W. Rosentiel. A method for partitioning UNITY language in hardware and software. In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.
8. A. Kalavade and E. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *International Workshop on Hardware-Software Co-Design*, pp. 42–48, 1994.
9. P. Knudsen and J. Madsen. PACE: A dynamic programming algorithm for hardware/software partitioning. In *International Workshop on Hardware-Software Co-Design*, pp. 85–92, 1996.
10. A. Balboni, W. Fornaciari, and D. Sciuto. Partitioning and exploration strategies in the tosca co-design flow. In *International Workshop on Hardware-Software Co-Design*, pp. 62–69, 1993.
11. F. Vahid and D. Gajski. Closeness metrics for system-level functional partitioning. In *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 328–333, 1995.
12. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung. System design methodologies: Aiming at the 100 h design cycle. *IEEE Transactions on Very Large Scale Integration Systems* 4(1): 70–82, 1996.
13. W. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE* 82(7): 967–989, 1994.
14. D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers* 12(1): 53–67, 1995.
15. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, February 1970.
16. T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, England, 1990.
17. C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the Design Automation Conference*, 1982.
18. F. Vahid and D. Gajski. SLIF: A specification-level intermediate format for system design. In *Proceedings of the European Design and Test Conference (EDTC)*, pp. 185–189, 1995.
19. F. Vahid. Procedure exlining: A transformation for improved system and behavioral synthesis. In *International Symposium on System Synthesis*, pp. 84–89, 1995.
20. J. Gong, D. Gajski, and S. Narayan. Software estimation using a generic processor model. In *Proceedings of the European Design and Test Conference (EDTC)*, pp. 498–502, 1995.
21. F. Vahid and D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on Very Large Scale Integration Systems* 3(3): 459–464, 1995.
22. S. Kirkpatrick, C. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science* 220(4598): 671–680, 1983.
23. B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, May 1984.

24. L. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computer-Aided Design*, January 1989.
25. S. Narayan and D. Gajski. Synthesis of system-level bus interfaces. In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.
26. F. Vahid and T. Le. Towards a model for hardware and software functional partitioning. In *International Workshop on Hardware-Software Co-Design*, pp. 116–123, 1996.