Modifying Min-Cut for Hardware and Software Functional Partitioning

Frank Vahid
Department of Computer Science
University of California, Riverside, CA 92521
vahid@cs.ucr.edu, www.cs.ucr.edu

Abstract

The Kernighan/Lin heuristic, also known as min-cut, has been extended very successfully for circuit partitioning over several decades. Those extensions customized the heuristic and its associated data structure to rapidly compute the minimum-cut metric required during circuit partitioning; thus, those extensions are not applicable to problems requiring other metrics. In this paper, we extend the heuristic for functional partitioning in a manner applicable to the codesign problem of hardware/software partitioning as well as to hardware/hardware partitioning. The 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 results. Our experiments demonstrate extremely fast execution times (just a few seconds) with results matched only by the much slower simulated annealing heuristic, meaning that the extended Kernighan/Lin heuristic will likely prove hard to beat for hardware and software functional partitioning.

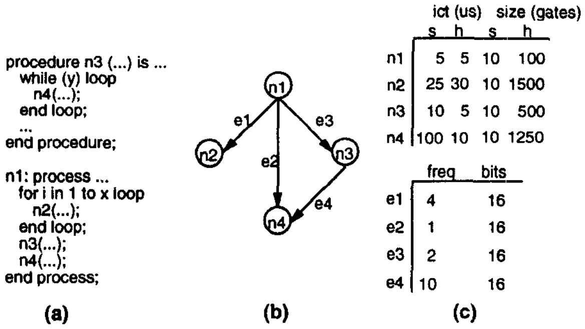
1 Introduction

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. Functional partitioners are needed to partition such programs among software and custom hardware processors, both for use in reconfigurable accelerators [1, 2], as well as embedded system design [3, 4, 5]. To fit into a compilation environment, such partitioners must be fast, executing in only seconds or minutes.

Automated hardware/software functional partitioners are presently the focus of several research efforts, using a variety of partitioning hueristics: simulated annealing in [3], custom greedy-improvement in [4], hierarchical clustering in [6] and [7], custom construction in [8], tabu-search in [9], dynamic programming in [10], and a variety of heuristics in [5]. Functional partitioning, which partitions a system's functions as opposed to its structural implementation, was also demonstrated in [11] to have numerous advantages over structural partitioning for hardware/hardware partitioning.

Our goal was to develop a heuristic that would: (1) execute in a few seconds, (2) yield excellent results, (3) be applicable to hardware/software as well as hardware/hardware partitioning, and (4) support addition of new metrics. Since no proposed heuristic satisfied all four requirements, we examined the Kernighan/Lin (KL) heuristic [12] (also known as group migration or min-cut). KL was extended by Fiduccia/Mattheyses (KLFM) [13] and others, and is said to be the most common circuit partitioning method [14].

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 hardware/software partitioning has a very different problem formulation than circuit partitioning, the extensions are non-trivial. The key part of the extension is the integration of the heuristic with a model of behavior execution-time [15], which includes a communication model that quickly but accurately computes data transfer times between hardware and software parts as well as within a single part.



```
procedure n3 (...) is ...
  while (y) loop
    n4(...);
  end loop;
  ...
end procedure;

n1: process ...
  for i in 1 to x loop
    n2(...);
  end loop;
  n3(...);
  n4(...);
end process;
```

	ict (us)		size (gates)	
	s	h	s	h
n1	5	5	10	100
n2	25	30	10	1500
n3	10	5	10	500
n4	100	10	10	1250

	freq	bits
e1	4	16
e2	1	16
e3	2	16
e4	10	16

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

2 Problem description

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 focus on partitioning a single process because this focus covers a number of problems, including speeding up a C program running on a PC or an embedded processor or microcontroller, as well as partitioning a process to meet package constraints of microcontrollers and FPGA's. We shall describe extensions for concurrent processes in Section 6. For the purpose of this paper, we assume the target architecture consists of a standard processor with memory (the software part) and a custom processor possibly with its own memory (the hardware part), although extensions can be made for multiple processors of either type. The standard processor may use a system bus, and/or may have several bidirectional data ports (such as commonly found on microcontrollers). The two processors may have different clock speeds.

The problem considered in this paper 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.

We convert the program into a SLIF AG (System-Level

0-8186-7895-X/97 $10.00 © 1997 IEEE

Intermediate Format Access Graph) [15], as illustrated in Figure 1(b). Each node represents a *functional object*, such as a subroutine or global variable, and each edge represents an access by one object to another, representing a functional call or variable read/write. Statement blocks can always be exlined into subroutines to achieve finer granularity [16]). We assume there are no recursive procedures, meaning that the AG contains no cycles.

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 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; 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, 17]. For a discussion on a more complex method for hardware size estimation, which considers hardware sharing among functional objects, see [18].

3 Kernighan/Lin heuristic background

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. Such heuristics consist of two elements. The **control strategy** includes three key activities – *SelectNextMove*, which chooses the next move to make, *ModifySelCrit*, which modifies the selection criteria, usually by reducing the possible moves, and *Terminate*, which 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. The **cost information** includes *DS*, which is the data structure used to model the nodes and their partition, from which cost is computed, *UpdateDS*, which initializes DS, and modifies DS after a move, ideally in constant time, and *CostFct*, which is a function that, given a partition, combines various metric values into a number called cost, representing a partition's quality. Ideally the cost function takes constant time. We use the convention that lower cost is better.

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.

We write the strategy in algorithmic form in Algorithm 3.1. Assume N is the set of nodes $n_1, n_2, ..., n_n$ to be partitioned, and the two-way partition is given as $P =< p_1, p_2, DS >$, 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($P$)) loop
        $swap$ = SelectNextMove($currP$)
        $currP$ = MoveAndLockNodes($P, swap$)
        $bestP$ = 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 $n_i \in p_1, n_j \in p_2$) loop
        Append(costlog, CostFct(Swap($P, n_i, n_j$)),$n_i, n_j$)
    end loop
    return ($n_i, n_j$ swap in costlog with lowest cost)
```

In other words, *SelectNextMove* trys all possible swaps of unlocked nodes, and swaps the best or least worst. Once all nodes are locked (already swapped), the heuristic reverts to the lowest-cost partition seen so far, completing one iteration, terminating if no improvement was found during the iteration. 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 $O(n^2 log(n))$ [14].

Fiduccia/Mattheyses [13] made three key extensions to KL: (1) They redefined the cut metric for hypergraphs (where edges may connect more than two nodes, more closely modeling real circuits), (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, and (3) They described a data structure enabling *SelectNextMove* to find the best next move in constant time. We refer to KL with these extensions as KLFM.

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. 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 constant time as described in [13], 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)
```

4 Extensions for hw/sw partitioning

We now describe three extensions to KL for hardware/software functional 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 describe each extension in detail.

4.1 Replacing cut by execution-time

This modification is the basis of the extensions for hardware/software partitioning. We first noted that minimizing execution time, not cut, is the main goal of hardware/software partitioning (a simple technique for incorporating other metrics, such as size and cut, is discussed in Section 6. A node's execution time can be modeled as as the sum of the node's internal computation time and the time spent accessing other nodes [15]; a simplified form of the equation is:

$$n.et = n.ict + n.ct \qquad (1)$$

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

In other words, a node'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.

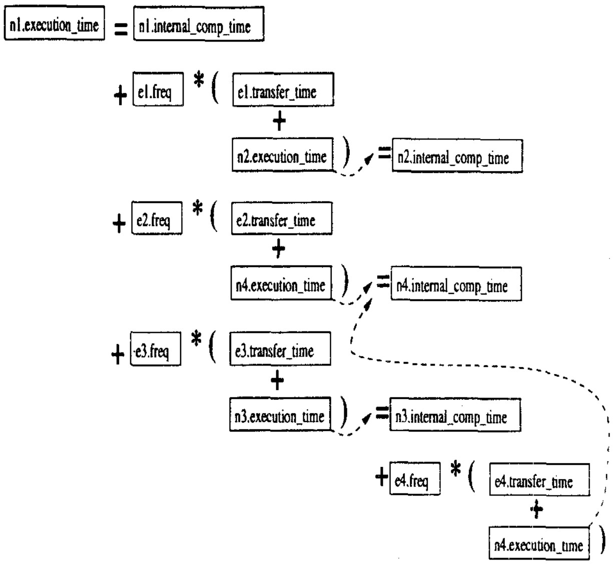


Figure 2: Execution-time model.

The equation for *n1* from the earlier example is shown graphically in Figure 2. 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.

4.2 Redefining a move as a single object move

In hardware/software partitioning, there is not a concept of maintaining balanced part sizes, especially since the parts have different units (i.e., instructions versus gates). Therefore, we redefine a move as a single object move, since such 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. We divide *SelectNextMove* into two parts, *PopNextMove* and *UpdateDS*, as in Algorithm 3.2, and we try to build these to execute constant time, as in KLFM. We'll see that we can't achieve constant time for the execution-time metric, but instead an average time of $\log n$.

First, we note that moving a node affects the execution time of that node and its ancestors. In particular, in Figure 2 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

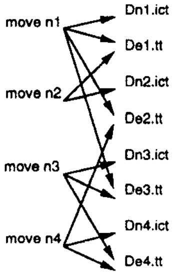


Figure 3: Terms that change during moves.

time of any adjacent edge (i.e., a change in *e.tt* of any *e* connecting *n*). Also, 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 (ancestor) nodes. Since only ancestors are affected, there is a local effect on execution time, though not as local as the cut metric which only had immediate neighbors affected. This localized effect means that we only need to update a small portion of DS when a node is moved.

Next, we must rapidly compute how each possible node move changes the execution-time, as computed by an equation like that in Figure 2, so we can pick the best node to move. The equation terms that may change are the node ict values and the edge tt values; the change of *nl.ict* is written as *Dnl.ict*, of *el.tt* as *Del.tt*, and so on, as illustrated in Figure 3. Note that the *e.freq* values are constants. Figure 3 shows the terms that change when a particular object is moved. For example, moving *n2* changes *n2.ict* and *el.tt*. Based on these relationships, we collect all terms from Figure 2 that change for a given move, and create a *change equation* that computes the change in *nl.et* for that move, as shown in Figure 4(a).

We now 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 *Dnl.et* for each node. We store each node at *Array(Dnl.et)*, as shown in Figure 4(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 *nl*. 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 best node's index.

We now implement *SelectNextMove* (see Algorithm 3.2). The first part, *PopNextMove*, deletes the best node from the change list, which can easily be done in constant time, and updates *BestChange*, which is a bit harder since we don't know where the next node is in the array. We use the approach in [13] of decrementing *BestChange* until we find a non-empty array item; the approach was shown to be constant time for the cut metric. The second part, *UpdateDS*, recomputes change equations that contain a term that changed during the move (see Figure 3 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.

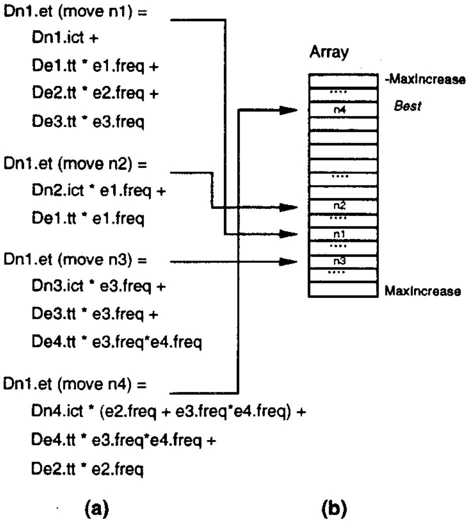


Figure 4: Data structure: (a) change equations, (b) change list.

The average complexity of updating the data structure is $\log n$, not a constant c as in KLFM. In KLFM, a node move affects the gain values of the node's neighbors, which in the worst case could be all n nodes, but instead is usually a small number c. In our problem, a node move affects the gain values of the node's ancestors, which in the worst case could be all n nodes, corresponding to a program where every procedure calls and is called by exactly one procedure, but instead is usually a smaller number $\log n$, as verified in [19].

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 an average of $\log n$, as described above. Thus, the average time complexity is $c_1 \times n \times (c_2 + \log n)$. If we know the ancestor hierarchy is $\log n$, then the heuristic's complexity is $O(n \log n)$.

4.4 Example

Figure 5 provides an example of applying the extended KL heuristic on the example of Figure 1. We start with all nodes assigned to software, though any intitial partition would be possible. The *nl.et* equation of Figure 2 evaluates to 2225 for such a partition. We obtain values for the nodes' change equations of Figure 4(a), and insert each node into the change list corresponding to its change value. In Figure 5(b), we pop the best node *n4* and move it to hardware, update change equations for *n3* and *nl* (*n2* is unaffected, and *n4* is locked), update *nl.et* by the change value (2225 - 1680), and reinsert *n3* and *nl* 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 *nl.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, the final outcome will be different.

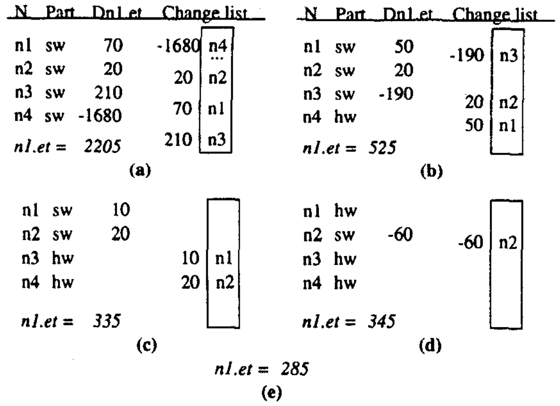


Figure 5: KL example using the change list: (a) initial, (b) after 1 move, (c) 2 moves, (d) 3 moves, (e) 4 moves.

5 Experiments

Figure 6(a) 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 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 $n \log n$ is slightly greater than *Gr*, yet, as the table demonstrates, *KL* outperformed all heuristics except simulated annealing.

We compared the runtime of KL extended for functional partitioning (*KL'*) with the runtime of a straightforward KL implementation (*KL*) that checks all possible moves to select the best next move. Results are summarized in Figure 6(b)

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

(a)

Nodes	KL'	KL
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

(b)

Figure 6: Results: (a) comparison of KL with other heuristics, (b) comparison of extended KL with KL.

for 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 [19] for information on generated examples. Because the number of iterations of KL usually varies from 2 to 6, there can be some time fluctuation among different examples not related to the size of those examples. Because our goal here is simply to observe the *difference* in runtimes, we ran each KL version for exactly one iteration. The results show that the extended KL is far faster than 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 are in seconds on a 166Mhz Pentium.

The improvement in speed is gained with no loss in quality. The sequences of moves made by the extended KL and KL are identical; only the time required to determine the next best move is changed.

6 Extensions and future work

Several additional extensions are straightforward. (1) We can incorporate additional metrics, such size and I/O, by using a weighted-sum cost function and maintaining change equations for each metric; (2) We can perform multiway partitioning by maintaining change equations for each metric, and having each entry in the change list represents a 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). (3) We can partition multiple processes easily: 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 need to develop techniques to account for multi-tasking overhead. Future extensions include adding lookahead and multiway lookahead, as done for circuit partitioning in [20, 21]. Finally, performing transformations during partitioning, would likely lead to much improved results.

7 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 $O(n \log n)$ for most problems, 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*, vol. 26, pp. 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*. New Jersey: Prentice Hall, 1994.

[6] 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.

[7] 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.

[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. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Hardware-software partitioning with iterative improvement heuristics," in *International Symposium on System Synthesis*, pp. 71–76, 1996.

[10] 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.

[11] F. Vahid, T. Le, and Y. Hsu, "A comparison of functional and structural partitioning," in *International Symposium on System Synthesis*, pp. 121–126, 1996.

[12] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, February 1970.

[13] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the Design Automation Conference*, 1982.

[14] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. England: John Wiley and Sons, 1990.

[15] 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.

[16] F. Vahid, "Procedure exlining: A transformation for improved system and behavioral synthesis," in *International Symposium on System Synthesis*, pp. 84–89, 1995.

[17] 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.

[18] F. Vahid and D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 3, no. 3, pp. 459–464, 1995.

[19] 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.

[20] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks," *IEEE Transactions on Computers*, May 1984.

[21] L. Sanchis, "Multiple-way network partitioning," *IEEE Transactions on Computer-Aided Design*, January 1989.