UC Irvine
ICS Technical Reports

Title
Layout placement for sliced architecture

Permalink
https://escholarship.org/uc/item/0x97m5w1

Authors
Larmore, Lawrence L.
Gajski, Daniel D.

Publication Date
1989-10-19

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Z
699
C3
no. 89-36

Layout Placement for Sliced Architecture

Lawrence L. Larmore
Dept. of Mathematics and Computer Science
University of California at Riverside
Riverside, CA 92521

Daniel D. Gajski
Dept. of Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Technical Report 89-36

October 19, 1989

Layout Placement for Sliced Architecture

Lawrence L. Larmore
Dept. of Mathematics and Computer Science
University of California at Riverside
Riverside, CA 92521

Daniel D. Gajski
Dept. of Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Abstract

This paper defines a new sliced layout architecture for compilation of arbitrary schematics (netlists) into layout for CMOS technology. This sliced architecture uses over-the-cell routing on the second metal layer. We define three different architectures with simple folding, interleaved folding and unrestricted folding. We present a linear time algorithm for placement of components in architectures with simple folding. We prove interleaved folding is *NP*-hard and give an algorithm of complexity $O(nbH/\delta)$ for approximating an optimal module, where n is the number of components, b is the width of the least-area module, H is the total height of the components, and $\delta > 0$ is arbitrarily chosen. The error of this algorithm (*i.e.*, the difference between the area of the resulting module and the optimal one) is $O(nb\delta)$. We conclude the paper with a proof that the architecture with interleaved folding is as good as the architecture with unrestricted folding with respect to area minimization of the total layout.

1 Introduction

Complexities of VLSI designs have reached the level of a million transistors on one VLSI chip. At this level the use of analysis and synthesis CAD tools becomes necessary. They improve the productivity, shorten the design time and thus shorten the time to market the new product. The basic requirements of all CAD tools are to cover wide variety of designs and offer good performance. The main problem in the past was to keep good performance with increasing complexities of design. The development of high-performance engineering workstations and specialized engines for a small number of analysis tasks has been the favorite solution to this problem. In addition to coverage and performance CAD synthesis tools are required to produce high quality design, possibly outperforming human designers. That requires different algorithms for different design styles, and usually, several different algorithms to optimize different goals, such as cost, area, delay and testability, for each design style.

In this paper we define a new sliced layout architecture for arbitrary microarchitectural schematics and give several algorithms for optimizing the layout area for several variants of the sliced architecture.

In the rest of this paper we will describe the problem and its solutions. In section 2 we will give overview of the placement and routing problems and basic algorithms for solving them. In section 3 we will describe the reasons for sliced architecture while section 4 will introduce the placement problem for the sliced architecture. In sections 5, 6, and 7 we will give placement algorithms for minimizing layout area for simple folding, interleaved folding and unrestricted folding. Section 8 contains the results of our experiments while section 9 concludes the paper.

2 Background

The transformation of a design consisting of a set of components and their interconnections consists of three tasks: generation of layout for each component, placement of components and routing of placed components. Layout generation consists of placing transistors on silicon in diffusion and polysilicon layers and interconnecting them with two additional metal layers. Thus, each component may be represented by a rectangle with inputs and outputs on its periphery. The placement task determines the position of each component (rectangle) in the design schematic, usually defined with a netlist.

Component placement methods fall into two groups: constructive and iterative. Constructive placement methods produce a complete placement from the netlist or partial placement while iterative methods improve a complete placement by modifying it. Improvement is measured by some metric such as area, total wire length, wire density, *etc.* The routing task determines the placement of connections between components. It consists of three subtasks. First, the routing surface must be divided into rectangular routing areas which meet the restrictions of the routing algorithm to be used. Second, a global router assigns each net to a subset of routing areas,*i.e.*, the global router defines areas through which the net will be placed. Third, a detailed router calculates the exact wiring path in each routing area.

3 Motivation

In order to simplify the general placement and routing problem several layout architectures have been developed. They restrict the freedom in placing components and wires and thus simplify the placement and routing algorithms. The best known and the most popular one is the standard cell architecture. The general microarchitectural components such as ALUs, registers, counters, encoders and shifters are further decomposed into gates, such as NAND, NOR and EXOR; and storage elements, such as latches and flip-flops. Each of these elements is laid out by hand as a cell. All cells are of the same height but of different width with inputs and outputs on top and the bottom of the cell. The cells are placed in rows which are stacked in several levels. Every two rows are separated by the routing area called routing channels (Figure 1). Several dummy cells used only for routing cells in non-adjacent rows are inserted where needed.

One of the weaknesses of the standard cell architecture is that routing occupies more than 50% of the total area. It should be noted that cells use basically diffusion and polysilicon layers with some routing in the first metal layer while channels are routed in two metal layers. Thus, it would be beneficial to route over the cells and minimize routing area. Another weakness is that standard cell architecture does not take advantage of replicability of microarchitectural components which consists of many identical bit slices. Those slices can be laid out as one cell instead of as several standard cells, and connected through diffusion and polysilicon layers, thus drastically reducing number of wires in the channel.

We now introduce the sliced architecture that avoids the above mentioned weaknesses of standard cell architecture. For all microarchitectural components in the library there are cells implementing functionality of one-bit slices of each component. All cells are of the same width. Obviously, each microarchitectural component has cells of different height h (Figure 2). Each cell has n second-metal tracks used for input, output and over-the-cell routing among different components. Each input or output to the cell may use any of the tracks entering from the top or bottom or passing through.

All components in a design are stacked one on top of the other and routed in second metal through the available tracks over the cells (Figure 3). The weakness of such a placement is that different components have different numbers of bits and thus components with fewer bits waste area. Thus the main problem is to fold the sliced architecture into itself in a way that minimizes wasted area. By folding, we mean here, pairing of different components so that all the pairs are of approximately the same width.

4 Problem definition

The problem of sliced architecture placement can be defined as follows: "Given a set of microarchitectural components and their connections, place components in such a way that the area of the bounding rectangle including components and routing is minimal."

We assume in this definition that each component C_i has height h_i and is b_i slices wide. We also assume that all the routing among components and to and from the outside world can be accommodated through over the cell routing, that is, the track density of each cell is narrower or equal to the density of all interconnections in the particular placement.

The first solution to the above problem we call simple folded sliced architecture. It is obtained by sorting the components by their bit widths and folding the ordered stack of components approximately in the middle (Figure 4). The routing tracks are also folded so that they make a 180 degree turn as shown in Figure 4. The inputs and outputs of the unfolded part are on the top while the folded part may have inputs and outputs on the top or bottom. This strategy imposes a constraint on the bit width of the bottom components. The bottom component of the folded part must not have more bit slices than half the width of the module.

This restriction can be removed using the interleaved folding as shown in Figure 5. The inputs to the unfolded part are at the top while inputs and outputs to the folded part are on the bottom. Both parts are connected together through routing around the module. In this second solution to the sliced architecture placement the possibly smaller component area is traded off for the larger routing area in comparison to the simply folded architecture.

The third solution to the above problem is the unrestricted architecture in which components are not sorted or folded. Components are stacked with respect to the left and the right edge of the module in such a way that wasted area is minimal as shown in Figure 6. This architecture does not have any restriction on ordering or size of particular components. It assumes, however, that track densities of the left and the right parts together are less than or equal to the track density of the bit slice.

5 Simple-folded sliced architecture

In this section we give a fast algorithm for minimizing the area of the module, using the simple-folded sliced architecture.

Sort the components by width. Let b_i and h_i be the width and height of the i^{th} widest component. Let the width of the module be b; for any fixed b, our algorithm minimizes the required height of the module. The area of the module can be minimized by running the algorithm once for each choice of b, and comparing those best answers.

We define the component C_i to be *wide* if $b_i > b/2$, and *narrow* otherwise. In the simple-folded sliced architecture, components C_1 through C_k (for some k) will be stacked on the left in sorted order, while components C_{k+1} to C_n will be stacked on the right. The routing strategy requires that all the components on the right be narrow.

We say that components C_i and C_j are *compatible* if they can be laid side by side in the module, *i.e.*, if $b_i + b_j \leq b$. Otherwise, we say they are *incompatible.* For any narrow component C_i, define the *critical obstructing component* of C_i to be that C_j of maximum index which is incompatible with C_i, and we say that j is the *critical index* of i, and write $crit(i) = j$. If there is no incompatible component, we say that $crit(i) = 0$.

The algorithm needs only find the correct value of the parameter k. For each k, define y_k to be the minimum distance possible from the top of the module to the bottom of C_{k+1} if the folding occurs at k. Note that $y_k \geq \sum_{i=k+1}^{n} h_i$. Strict equality may not occur since obstructions caused by wide components on the right may prevent the folded portion from abutting the top of the module. All values of y_k can be found in linear time by reverse iteration, starting with $k = n$. Since there are no components on the right side, $y_n = 0$. For $k < n$, we need to compute y_k, knowing y_{k+1}. Let $j = crit(k)$, and let $x_j = \sum_{i=1}^{j} h_i$.

(Note that $x_0 = 0$.) If $y_{k+1} \geq x_j$, then the top of C_{k+1} can be placed at depth y_{k+1} without encountering any obstruction from components on the left side, so $y_k = y_{k+1} + h_{k+1}$ in that case. On the other hand, if $y_{k+1} < x_j$, the top of C_{k+1} abuts the bottom edge of its critical obstructing component C_j, so $y_k = x_j + h_{k+1}$. The optimal choice of k is that for which $max(x_k, y_k)$ is minimized, subject to the condition that $b_{k+1} \leq b/2$.

Algorithm 1

```
$x_0 \leftarrow 0$
for $j \leftarrow 1$ to $n$ do
    $x_j \leftarrow x_{j-1} + h_j$
end for
$y_n \leftarrow 0$
$crit \leftarrow 0$
for $k \leftarrow n-1$ downto 1 do
    if $b_k > b/2$ exit
    while $b_{crit+1} + b_k \leq b$ do
        $crit \leftarrow crit + 1$
    end while
    $y_k \leftarrow h_k + \max\{y_{k+1}, x_{crit}\}$
end for
$Best\_k \leftarrow$ that $k$ for which $\max\{x_k, y_k\}$ is minimum
```

Figure 7(a) illustrates the geometry behind the definition of the y_k. Figure 7(b) shows the optimal folding in that example, together with the routing.

Analysis. For fixed b, it takes $O(n)$ time to compute all x_j, then $O(n)$ time to compute all y_k in reverse order. Finally, it takes $O(n)$ time to find the optimal folding point. The entire algorithm thus takes linear time.

6 Sliced architecture with interleaved folding

In this section we give an algorithm which minimizes the area of the module for sliced architecture with interleaved folding. The algorithm takes exponential time in theory, but in fact is quite fast in practical examples.

NP-*hardness.* We first show that the problem is NP-hard, by reducing the classical partioning problem to it. Given a set of n items, each of which has a weight w_i, we may wish to divide the set into two subsets which have as close to equal weight as possible. This problem is known to be NP-hard [2], meaning that existence of a polynomial time algorithm for it is equivalent to the statement that $P = NP$.

We reduce the above partitioning problem to our architecture problem as follows: assume that we have n components of equal width (say $b_i = 1$) and heights w_i, as well as a single component of width 2 and any height. The minimum area module has width 2, and is obtained by partitioning the components into left and right subsets of as nearly equal total height as possible; the one long component goes at the top or bottom. This partitioning yields an optimal solution to the classic problem. It follows that if we could find a polynomial time algorithm for optimizing sliced architecture with interleaved folding, we would have proved that $P = NP$.

The situation is very far from hopeless, however. In this section we give an algorithm for minimizing the area of the module which runs in exponential time in the worst case. This algorithm, which we call the *List-Merge* algorithm, actually takes very little time to execute in practical cases, as we will show below. Even in the worst case, the List-Merge algorithm can be used to find an approximately optimal solution very fast, which differs from the optimal solution by a provably narrow amount.

Assumptions. We will fix a module width b. The List-Merge algorithm will find that placement of the components which minimizes the module height. The algorithm can then be run once for each choice of b.

We sort the components in the order in which they will be processed by the algorithm. The longest and shortest components come first, and those whose widths are closest to $b/2$ come last. This sorting is accomplished in two steps. In the first step, we sort the wide components (those of width greater than $b/2$) by width, widest first, and we sort the narrow components (those of width less than or equal to $b/2$) by width, narrowest first. These two lists are then merged, with the rule that if a wide and a narrow component are compatible (*i.e.*, they can be placed side-by-side in the module) the narrow one is first, while if they are incompatible, the wide one is first. For example, if there are 9 components of widths $1, 2, 4, 5, 6, 7, 8, 9, 10$ and if $b = 12$, the sorted list of wide components will have widths $(10, 9, 8, 7)$ and the sorted list of narrow components will have widths $(1, 2, 4, 5, 6)$. The merged list of components will have widths $(1, 2, 10, 9, 4, 8, 5, 7, 6)$. Let b_i, h_i be the width and height of the i^{th} component using this ordering.

Partial solutions and signatures. The List-Merge algorithm is a dynamic programming algorithm which successively builds up lists of partial solutions "up to" k, for k from 0 to n. We define a *partial solution up to k* to be a placement of $C_1, ... C_k$ in the module which satisfies the following conditions:

1. Components on the left are sorted in decreasing width going down.
2. Components on the right are sorted in decreasing width going up.
3. There is a "force" which pulls all wide left components and all narrow right components as far up as possible.
4. There is a "force" which pulls all wide right components and all narrow left components as far down as possible.

A partial solution up to n is a sosolution, , *i.e.*, a placement of all components.

We define the *top excess*, x, of a partial solution up to k as follows. Let u be the distance from the top of the module to the bottom of the lowest wide component on the left, and let v be the distance from the top of the module to the bottom of the lowest narrow component on the right. We define $x = max\{0, v - u\}$ The *bottom excess*, y, is defined similarly. The ordered pair (x, y) we call the *signature* of the partial solution.

For any given signature, there can be many partial solutions up to k. Given any partial solution P, a partial solution which has the same signature as P and which has height minimal for that signature can be obtained by pushing the bottom and top components together. We call this process *height-compression.* (See Figure 8.) Two partial solutions which can be height-compressed to the same partial solution are said to be *compression equivalent.*

Lemma 6.1 *Let (x, y) be the signature of any partial solution P up to k. Then the height of the module for P is greater than or equal to the sum of the heights of all wide components among $C_1, ...C_k$ plus* $\max\{x, y\}$. *Furthermore, if P is height-compressed, its height is equal to that quantity.*

Proof: Without loss of generality, P is height-compressed. Let M, M' be the midpoints of the top and bottom edges of the module. The length of MM' is the height of the module. We consider first the case that $x \geq y$ and $x > 0$. The lowest narrow component on the right, say C, extends a distance of x below the lowest wide component on the left, say C'. C also abuts the highest large component on the right. (See Figure 9(a).) The line MM' crosses each wide component with just one gap of length x. The case where $y \geq x$ and $y > 0$ is similar.

If $x = y = 0$, the line MM' crosses each wide component with no gaps. (See Figure 9(b).) □

Parents and minimal partial solutions. Any partial solution P up to k has a unique *parent*, a partial solution up to $k-1$, obtained by simply removing C_k. A solution up to 0 is just an empty module, and has no parent. Figure 10 shows a portion of a binary tree of partial solutions, illustrating the parent-child relationship, with an empty solution at the root. We define a partial ordering on signatures as follows: $(x_1, y_1) \leq (x_2, y_2)$ if $x_1 \leq x_2$ and $y_1 \leq y_2$. We say that a signature of a partial solution up to k is *minimal* if there is no signature of any other partial solution up to k which is strictly less according to that partial ordering, and we refer to a partial solution up to k as *minimal* if its signature is minimal. We now give a series of easy remarks, which form the basis of the list-merge algorithm.

Remark 6.1 *If two partial solutions are compression-equivalent, their parents are compression-equivalent.*

Remark 6.2 *The set of all compression-equivalence classes forms a complete binary tree of height n, whose root is the class containing the empty partial solutions, and whose leaves are all classes of solutions.*

Remark 6.3 *Let (x, y) be the signature of any node in the binary tree at level $k < n$. Then if C_k is narrow, the two children of that node have signatures $(x + h_k, y)$ and $(x, y + h_k)$. If C_k is wide, the two children have signatures* $(\max\{x - h_k, 0\}, y)$ *and* $(x, \max\{0, y - h_k\})$

Remark 6.4 *If a partial solution up to $k > 0$ is minimal, then it has the same signature as another partial solution whose parent is minimal.*

Remark 6.5 *If P is a partial solution up to n,* i.e., *a solution, we say that it has* optimal signature *if it has minimal signature (x, y) and if* $\max\{x, y\}$ *is minimized over all such minimal signatures. Then P is an optimal solution if and only if it has optimal signature and is height-compressed.*

Proof: Remark 6.1 is trivial. Remark 6.2 follows from the fact that if P is a partial solution up to $k-1$, we can (by reversing the height-compression process if necessary) insert C_k in just two ways. (see Figure 10). If C_k is wide, it is inserted either as far up as possible on the left side or as far down as possible on the right side. If C_k is narrow, it is inserted either as far up as possible on the right side or as far down as possible on the left side. Remark 6.3 merely gives the correct formula for computing the signature of the child given the signature of the parent: there are four cases to consider for inserting C_k on the left and four similar cases for the right. For the left, the cases are

1. C_k is narrow and $y > 0$. Then C_k slides down until it abuts the narrow component below it on the left side. The bottom excess is increased to $y + h_k$.

2. C_k is narrow and $y = 0$. Then C_k slides down until it abuts the topmost wide component on the right, or the bottom of the module if there is no such component. The bottom excess changes to h_k.

3. C_k is wide and $x \leq h_k$. Then C_k slides up until it abuts the wide component above it or the top of the module if there is no such component. The top excess becomes 0.

4. C_k is wide and $x > h_k$. Then C_k slides up until it abuts the wide component above it or the top of the module if there is no such component. The top excess is decremented to $x - h_k$.

The other four cases are similar.

We now consider Remark 6.4. Suppose that P is a partial solution up to k, and $P' = parent(P)$ is not minimal, *i.e.*, there is a partial solution Q' up to $k-1$ whose signature is strictly less than that of $parent(P)$ in the partial ordering of signatures. Let Q be the child of Q' obtained from Q' the same way P is obtained from P', *i.e.*, if P is the left child of P' then Q' is the left child of Q, similarly right. Since $signature(Q') \leq signature(P')$, we have by Remark 6.3 that $signature(Q) \leq signature(P)$. (Just check the cases.) But since P is minimal, that inequality must be an equality. This proves Remark 6.4.

Remark 6.5 follows trivially from Lemma 6.1. This concludes the proofs of the Remarks. □

The algorithm. We can now describe an algorithm for finding an optimal solution. Simply compute the entire binary tree of partial solutions, and examine the signatures of the leaves to find one for which $max\{x, y\}$ is minimized. The height-compression of that solution will be optimal by Remark 6.5. The time for this algorithm is $O(2^n)$.

Pruning. The List-Merge algorithm is really the same as the full exponential time algorithm, but with a pruning step added at each level. We compute the binary tree of partial solutions one level at a time, *i.e.*, as k iterates from 0 to n. After computing each level, we prune the tree by deleting all nodes whose signatures are not minimal for that level. If nodes have duplicate minimal signatures, we delete all but one of them. The remaining nodes at a level all have distinct signatures which are minimal, and only the children of these nodes are considered for constructing the next level. Induction on k, using Remark 6.4, guarantees that no minimal signature will be lost. Since there is a minimal signature which is optimal, there will be at least one optimal solution constructed at the bottom level.

Symmetry. Since any partial solution can be rotated 180 degrees, if (x, y) is a minimal signature so is (y, x). A further pruning by eliminating one of every such pair of minimal signatures saves almost another factor of 2 in the number of partial solutions that need to be computed. If this symmetry pruning is used, we eliminate any partial solution whose signature is (x', y') if we keep any other partial solution of signature (x, y) such that $(x, y) \leq (y', x')$.

Implementation. Let L_k be the set of partial solutions to k which survive the pruning process. We can assume that they are maintained in a list with strictly increasing values of x and strictly decreasing values of y. The algorithm can be expressed in terms of list operations as follows:

Algorithm 2

1. Let L_0 be the list consisting of just the empty solution.

2. For each k from 1 to n, construct the list of left children and the list of right children of L_{k-1}. Merge these two lists and prune, eliminating items with non-minimal or duplicate signatures. Call the resulting list L_k.

3. Search L_n for that solution for which $max\{x, y\}$ is minimized. The height-compression of that solution is optimal, and its height is $max\{x, y\}$ plus the sum of the heights of all wide components.

In Figure 11, we show the steps of a single example. Figure 11(a) shows the list of components, sorted in the order required by Algorithm 2. Figure 11(b) shows the binary tree of signatures, where all non-minimal and duplicate nodes are pruned. Symmetry pruning is also used, as this cuts the size of the tree roughly in half. (The lists L_k are not shown in the figure, since the sorting required by the algorithm is not closely related to the binary tree structure and a figure showing both would look very tangled.) The path to the optimal signature (which is a leaf at depth n) is indicated. Figure 11(c) shows the configuration of minimal height, together with routing. The routing is also shown.

Time complexity. The complexity of the List-Merge algorithm is $O(\sum_{i=1}^{n} |L_i|)$, which is exponential in the worst case. But, in a practical sense, the algorithm takes very little time, as indicated by the Lemmas 6.2 and 6.3 and Theorem 6.1, below.

Lemma 6.2 *Suppose that there is some positive real number δ such that h_k is an integral multiple of δ for all k. Let $M = (\sum_{k=1}^{n} h_k)/\delta$. Then $|L_k| \leq M + 1$ for all k, and the List-Merge algorithm takes $O(nM)$ time.*

Proof: If (x, y) is a minimal signature of a partial solution up to k, then x must be an integral multiple of δ. Since all such pairs in L_k must have distinct values of x, and none can exceed $M\delta$ (since that is the sum of all heights), there can be at most $M + 1$ items in L_k. The stated time complexity follows.

In many practical situations, the heights of all components are integral multiples of some unit, such as 1 micron. Lemma 6.2 then guarantees fast execution for the List-Merge algorithm, provided that unit is not extremely narrow.

But what about the worst case, namely where the heights of the components are arbitrary real numbers? We can approximate all heights to the nearest integral multiple of some unit, and the solution obtained may not be optimal, but will be provably close. Lemma 6.3 and Theorem 6.1 below summarize the result.

Lemma 6.3 *Suppose that the height of the i^{th} component is approximated to be h'_i, and that an approximately optimal solution is computed by Algorithm 2 using the actual widths and approximated heights. The components are then placed according to the instructions given by this approximated solution. The height of the module given by this solution is then greater than the height of the module of the optimal solution by at most $\sum_{i=1}^{n} |h'_i - h_i|$.*

Proof: There are two kinds of errors: overestimates and underestimates. Let E^+ be the sum of the overestimates, that is, the sum of all $h'_i - h_i$ for all i for which that quantity is positive, and let E^- be the sum of the underestimates, that is, the sum of all $h_i - h'_i$ for all i for which that quantity is positive. We call $E = E^+ + E^- = \sum_{i=1}^{n} |h'_i - h_i|$ the total error.

Let h be the height of the truly optimal module. Let h' be the height of the optimal module obtained using the estimated inputs. It is easy to see that $h' \leq h + E^+$, for if we use the truly optimal arrangement, and increase any given component's height using the same arrangement, the height of the module can be increased by at most that same amount. A decrease of the height of one component may or may not result in an decrease of the height of the module, but will never cause an increase.

Now use the arrangement that is optimal for the estimated heights. We can increase every underestimated height to its true value, this causes an increase of the height of the module by at most E^-, by the same argument. □

Theorem 6.1 *For any given positive δ, there exists an algorithm for the sliced architecture with interleaved folding which takes $O(n\sum_{i=1}^{n} h_i/\delta)$ time and produces a solution which is within $n\delta/2$ of optimal.*

Proof: Approximate each h_k by letting h'_k be the nearest integral multiple of δ. Then apply Lemmas 6.2 and 6.3. □

Practical cases. In practice, we expect that n will be approximately 30, and that the sum of the heights of all components will be approximately 15000μ. If the heights of all components are rounded to the nearest micron, the total error $\sum |h'_i - h_i|$, in that case, cannot exceed 15μ and is expected to be only 7.5μ. In that situation, Algorithm 2 will execute on the order of a million instructions, and will produce a solution whose deviation from optimal is on the order of 1%.

7 Sliced architecture with unrestricted folding

We now consider a still more unrestricted architecture. We allow any component to be located anywhere in the module, as long as its base end abuts either the right or left edge of the module. Eliminating the restriction on sorting makes routing around the module impossible, so we must assume that there are two wires available for each cell. (See Figure 6.)

It might be expected that removing the sorting restriction will allow a decrease in the area of the module in some cases. We shall see that this is false, in fact the unrestricted folding architecture always yields the same optimal sized modules as the interleaved folding architecture in the previous section. There can be savings in the routing area, though.

Theorem 7.1 *For any given b, the minimum possible height of any module of width b using the sliced architecture with unrestricted folding is equal to the minimum possible height of any module of width b using the sliced architecture with interleaved folding.*

Proof: It is clear that allowing the additional freedom of unrestricted folding cannot increase the height of the module, since any placement that fulfills the conditions of the previous section still fulfills the new conditions, since they are weaker. We thus only need show that the module cannot be made shorter.

Consider a set of components, and consider an optimal placement using the unrestricted folding condition. Let h be the height of the module for this placement. We define the "left moiety" to be the set of components which abut the left side of the module, and the "right moiety" similarly. For any real number $c \in [0, b]$, define $f(c)$ to be the sum of the heights of all left moiety components whose widths are at least equal to c, and let $g(c)$ be the sum of the heights of all right moiety components whose widths exceed $b - c$.

Lemma 7.1 *For any c, $f(c) + g(c) \leq h$.*

Proof: Slice the module horizontally, using every line which is an extension of either a top or a bottom edge of any component. (There will be at most $2n$ such lines.) This divides the module into a finite number of pieces. Each piece intersects at most one left component and at most one right component. Furthermore, if any component meets one of these pieces, it completely covers it vertically, though not horizontally. (See Figure 13(a).)

We classify the pieces into three groups:

1. those pieces which meet a left moiety component of width at least c
2. those pieces which meet a right moiety component of width greater than $b - c$
3. the other pieces

It is clear that groups 1 and 2 cannot have any common members, since otherwise the components would overlap. It is furthermore clear that $f(c)$ is the sum of the heights of the pieces in group 1, while $g(c)$ is the sum of the heights of the pieces in group 2. Since the sum of the heights of all pieces is h, the lemma follows. □

We now return to the proof of Theorem 7.1. We will remove all components from the module and replace them in sorted order, preserving the moieties. The left moiety will be sorted in order of decreasing width from the top, and will be top-justified. The right moiety will be sorted in order of increasing width from the top, and will be bottom-justified. Figure 12 shows a module before and after this sorting. The module still has height h, and the only thing that could go wrong is that some left and some right component overlap.

Suppose that a left component C_i of width c overlaps a right component C_j, which must then have width greater than $b - c$. Let x be the distance from the lower right corner of C_i to the top of the module, and let y be the distance from the upper left corner of C_j to the bottom of the module. Since all left moiety components above C_i have width at least c, $x \leq f(c)$. Since all right moiety components below C_j have width greater than $b - c$, $y \leq g(c)$. By Lemma 7.1, $x + y \leq f(c) + g(c) \leq h$, which means that the lower right corner of C_i is above the upper left corner of C_j, contradicting the hypothesis that they overlap. $\square$

8 Experiments

Since it is very difficult to obtain real-life examples on the microarchitectural level of design we generated more than ten examples by random. We set the number of components to be a random number between 10 and 50. This is approximately number of components on a controller, I/O interface chip or a medium size processor. The number of bits per component is a random number between 1 and 32, which covers most of the design with exception of floating-point arithmetic and number crunching high-performance processors. Similarly, the height of each component is a random number between 100 and 600 microns. We developed most of the microarchitectural components using Silicon Compiler System's Generator Development Tools and found that most components fall into that range. We used independent uniform distributions in the above mentioned ranges. We approximated the quality of our algorithm by wasted area in the layout, *i.e.*, the difference between the area of the module and the sum of all component areas. Table 1 shows for each example the number of components, areas of bounding rectangles for unfolded and folded archirecture, as well as wasted areas for both cases. On average our algorithm produced placements with less than 8.3% wasted area for folded architecture in comparison with 53.4% for unfolded architecture.

9 Conclusion

We have presented a new sliced architecture and an algorithm for placement of arbitrary microarchitectural schematics. We gave an algorithm for simple folding and interleaved folding. We also gave a bound on the algorithm quality. Our experiments show that wasted area is very small. Since this new architecture uses second metal for routing between components and since each component is hand laid it is expected that it will outperform the standard-cell architecture in area and performance.

Our algorithm did not take into account routing area around the module needed to connect two folded parts. It is an open problem to modify the placement algorithm to minimize the sum of the component and the routing areas.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms,* Addison-Wesley (1974).

[2] M.J.Garey, David Johnson.

[3] B. T. Preas, M. J. Lorenzetti (eds), *Physical Design Automation of VLSI Systems,* Benjamin Cummings, (1988).

[4] D. D. Gajski (ed), *Silicon Compilation*, Addison Wesley, (1988).

[5] M. J. Trick, S. W. Director, "LASSIE: Structure to Layout for Behavioral Synthesis," Proc. 26th DAC, June 25-29, (1989).

[6] N. Karmarkar, R. M. Karp, "An Efficient Approximation Scheme for the One-Dimensional Bin-Packing Proble," *Proceedings 23rd Annual Symposium on Foundations of Computer Science*, November 1982, pp. 312-320.

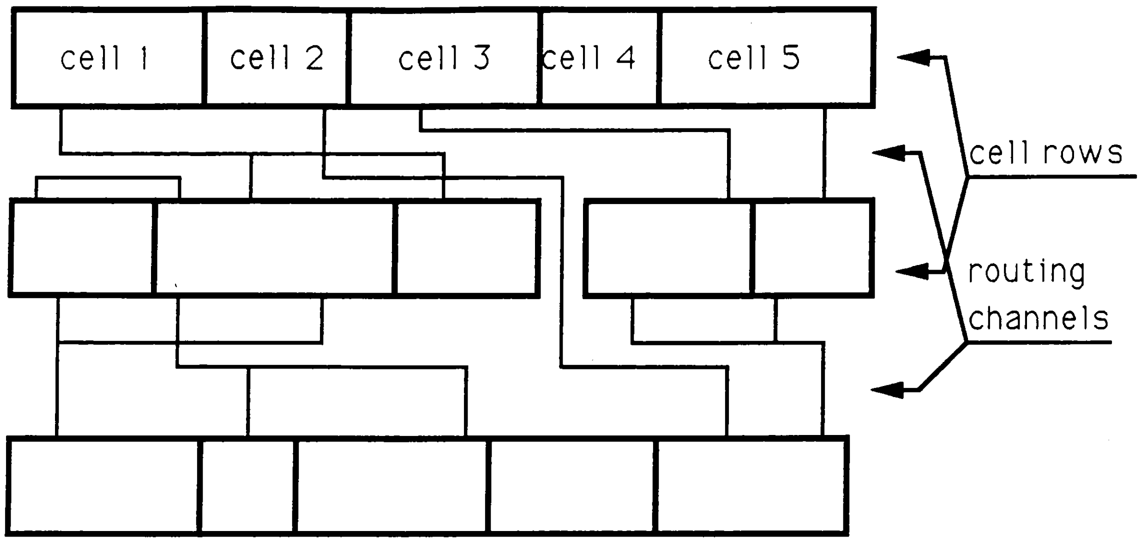

Figure 1. Standard-cell architecture

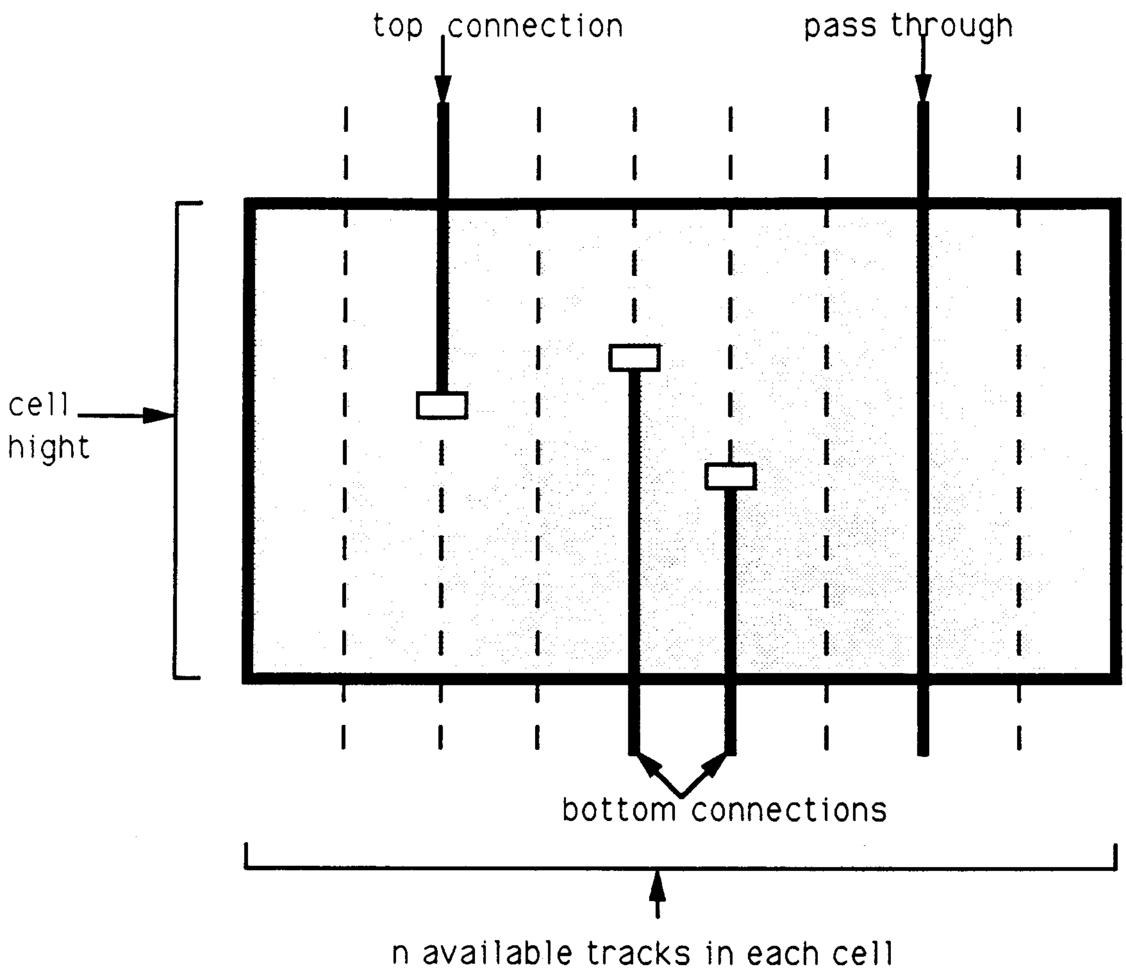

Figure 2. Over-the-cell routing

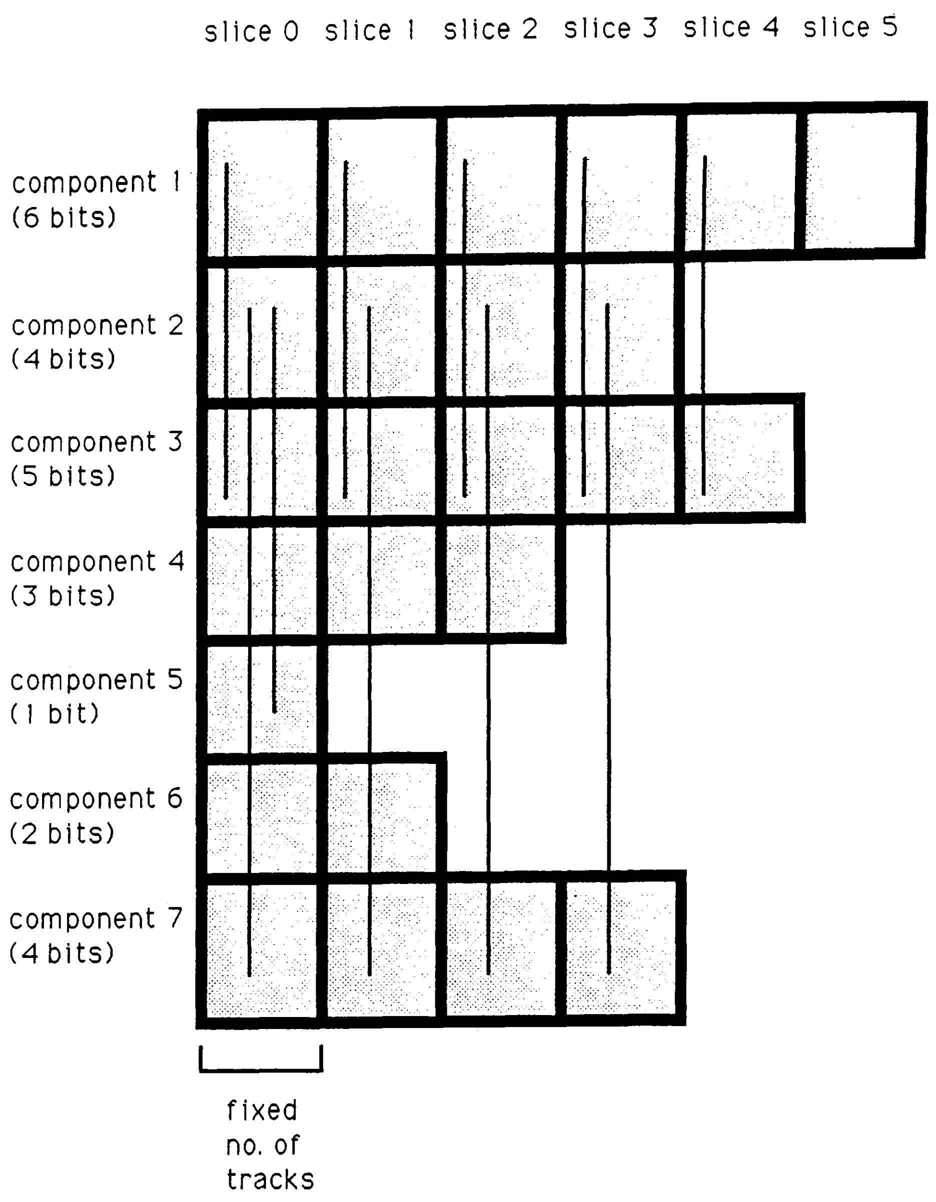

Figure 3. Sliced architecture

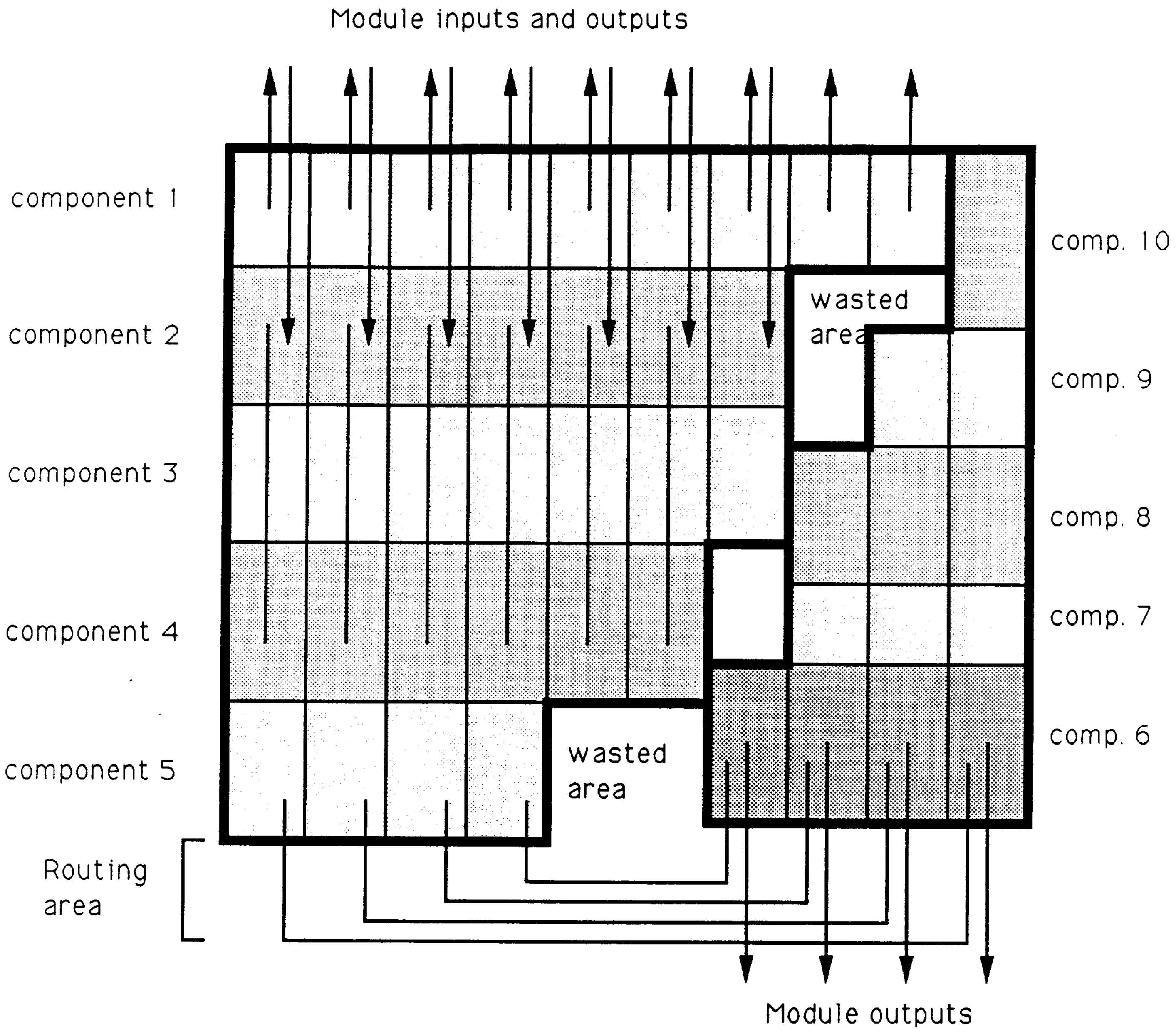

Figure 4. Simple-folded sliced architecture

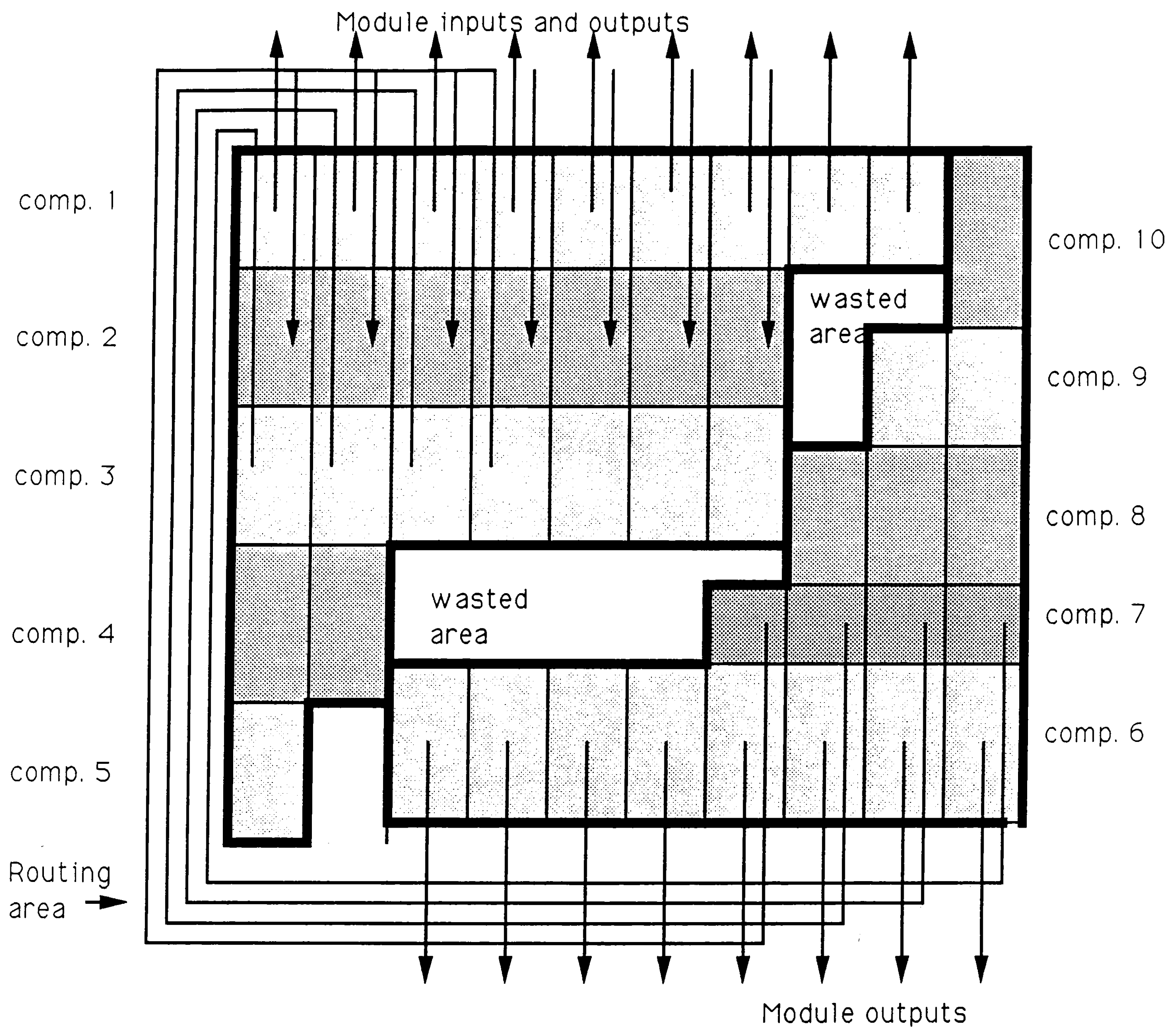

Figure 5. Sliced architecture with inter-leaved folding and I/O on top and bottom

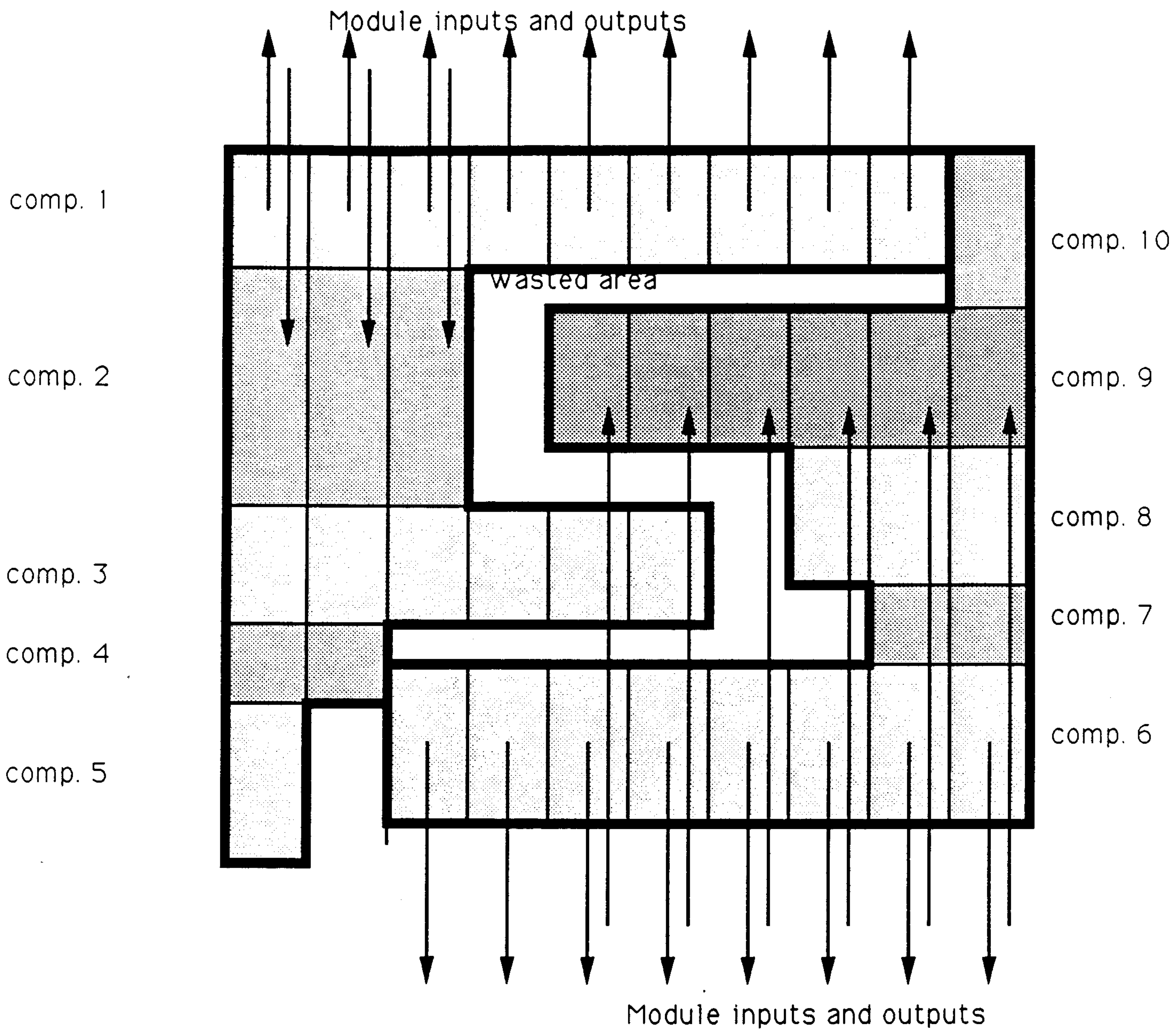

Figure 6. Sliced architecture with unrestricted folding and with I/O on the top and bottom.

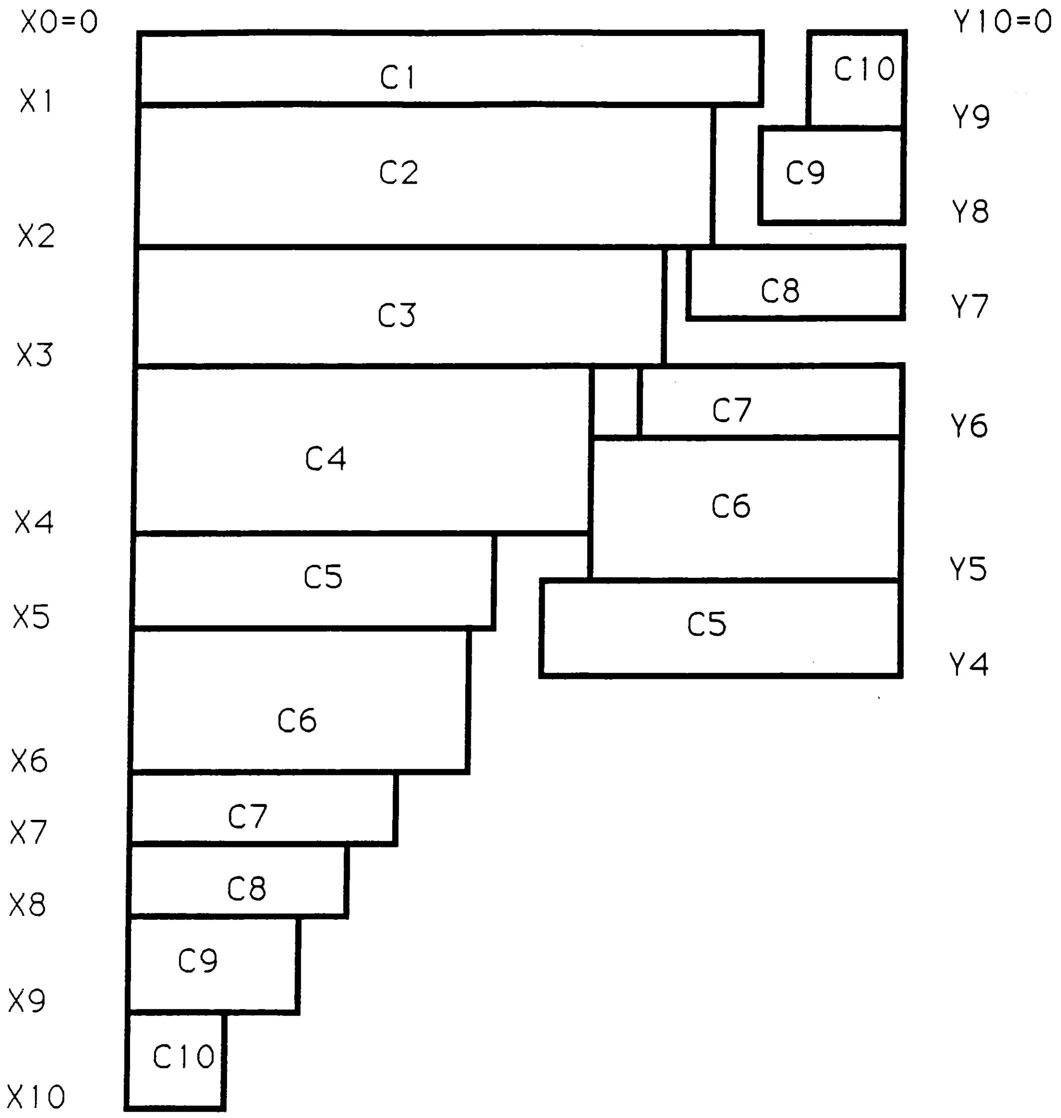

Figure 7(a) The Definitions of x i and y i

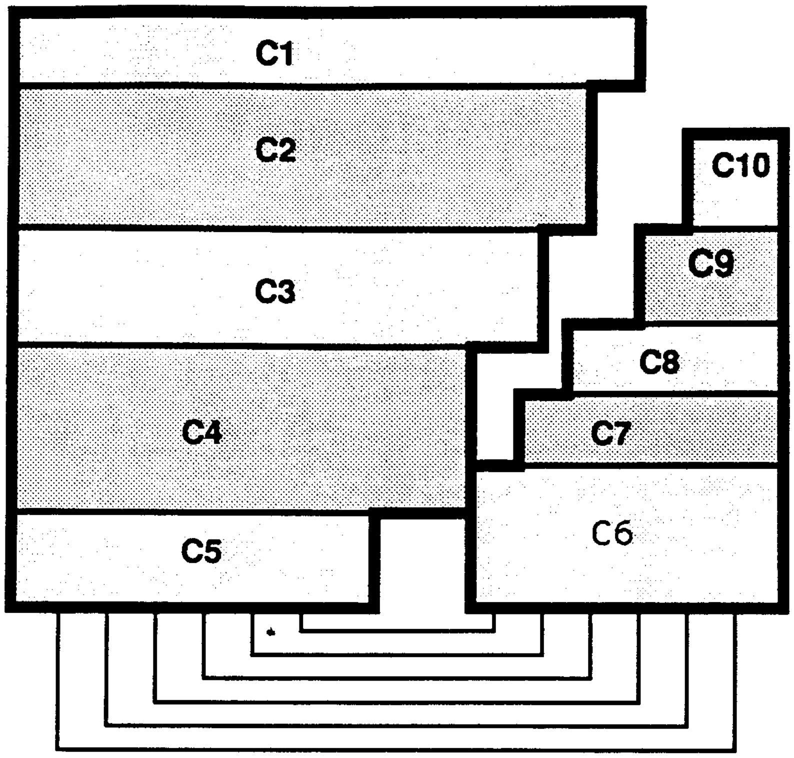

Figure 7(b) The Optimal Simple-Folded Sliced Architecture

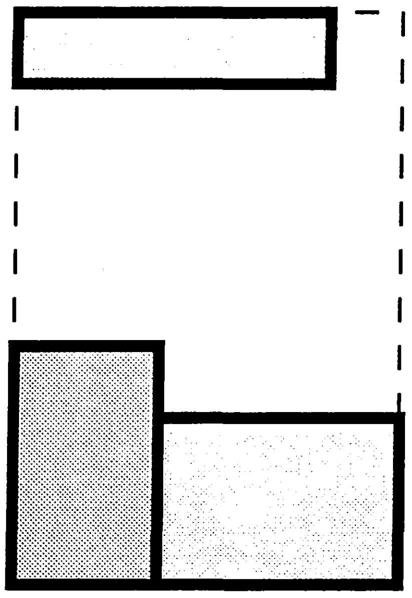

Figure 8(a) Before Height Compression

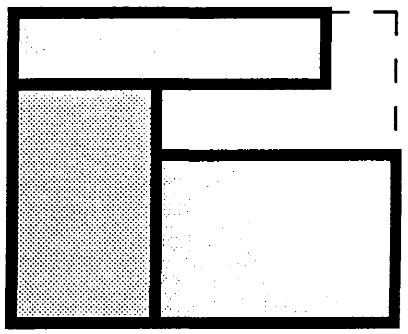

Figure 8(b) After Height Compression

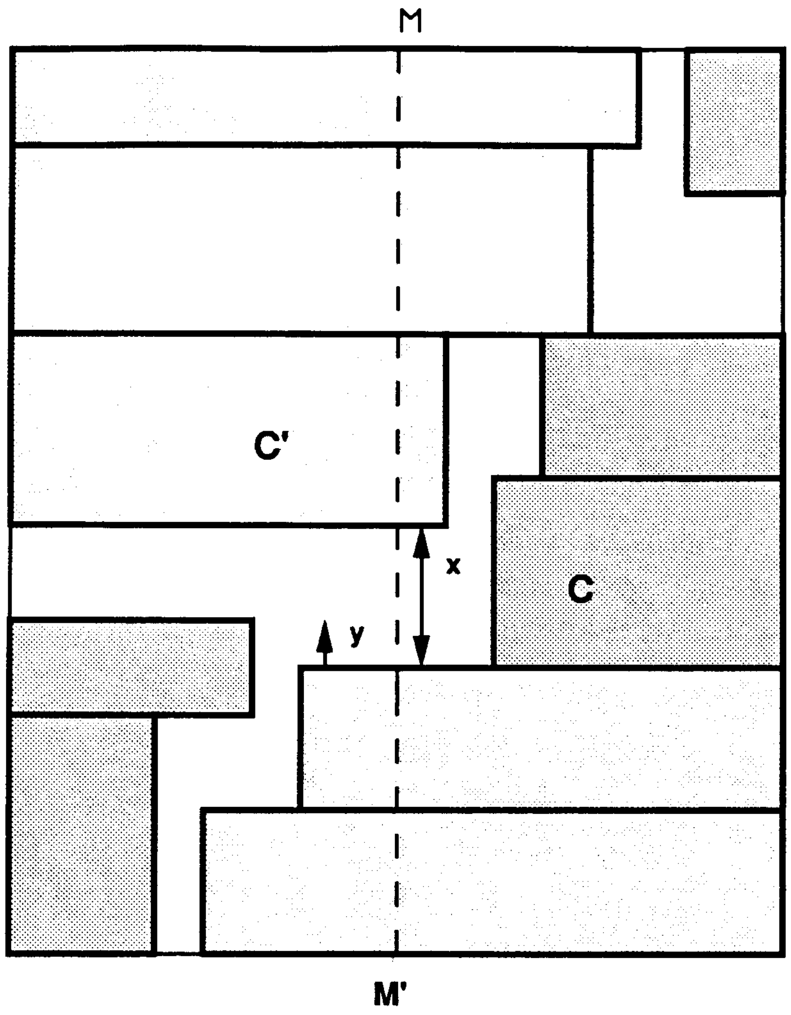

Figure 9(a) Illustrating the Proof of Lemma 6.1, Case 1

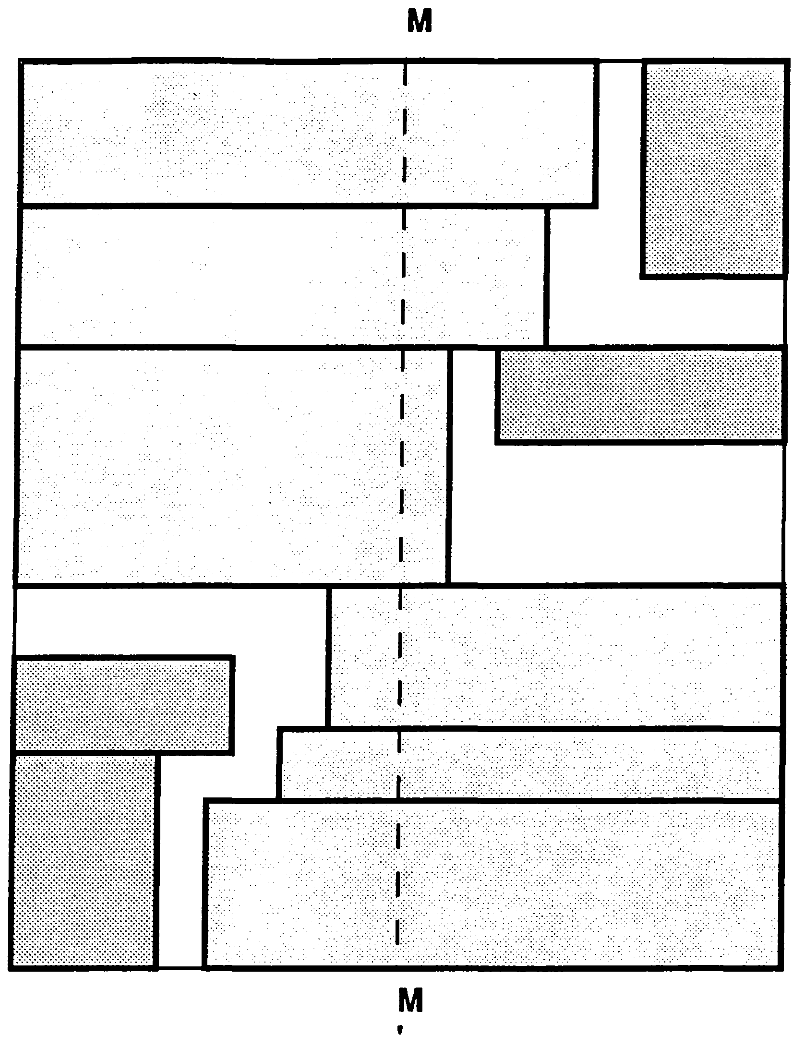

Figure 9(b) Illustrating the Proof of Lemma 6.1, Case 2

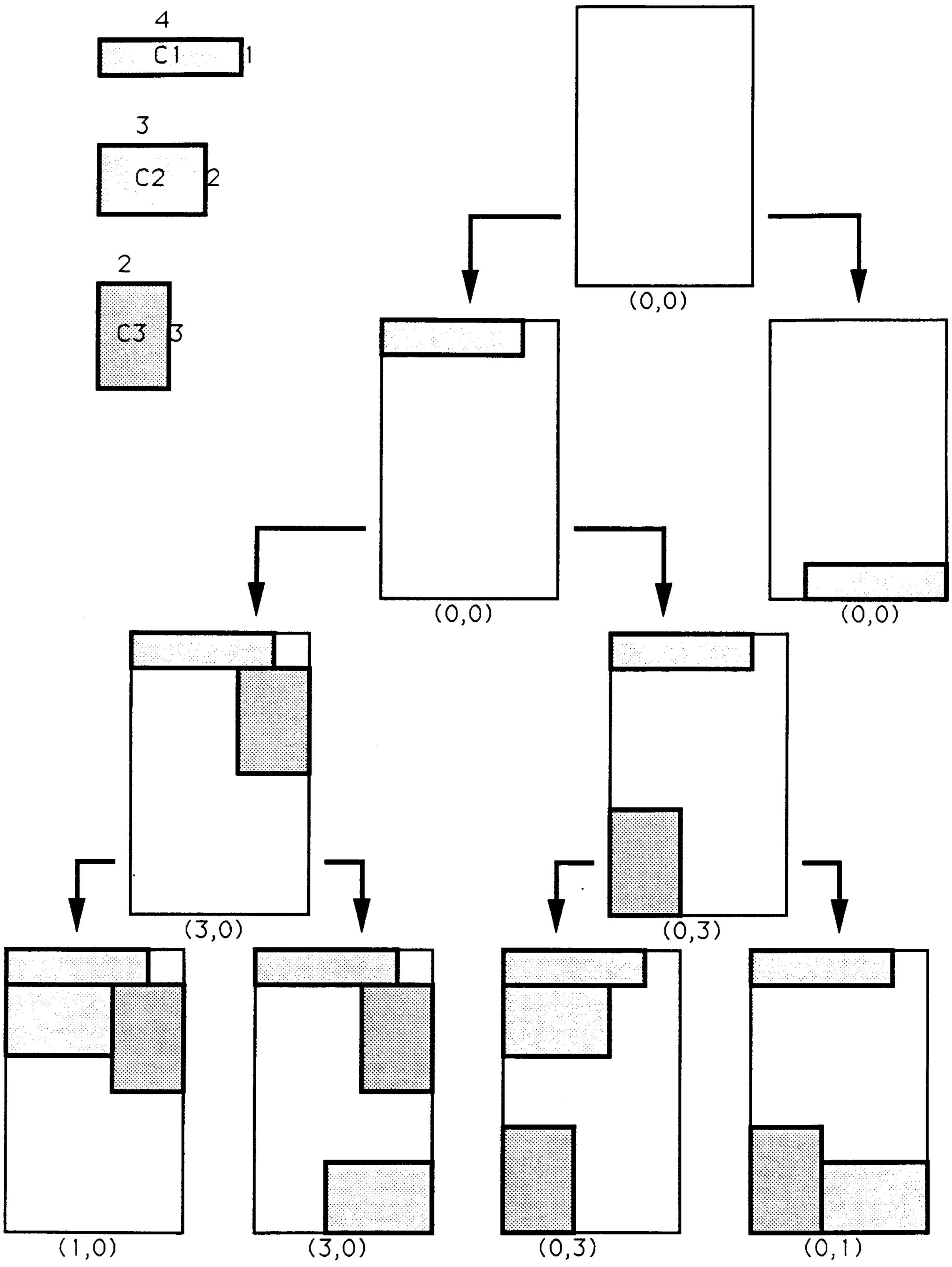

Figure 10 The Binary Tree of Partial Solutions

	Width	Height	Large or Small
C1	14	10	Large
C2	3	8	Small
C3	12	5	Large
C4	5	4	Small
C5	5	8	Small
C6	10	6	Large
C7	7	5	Small
C8	9	10	Large

Figure 11(a) The Widths and Heights of the Components

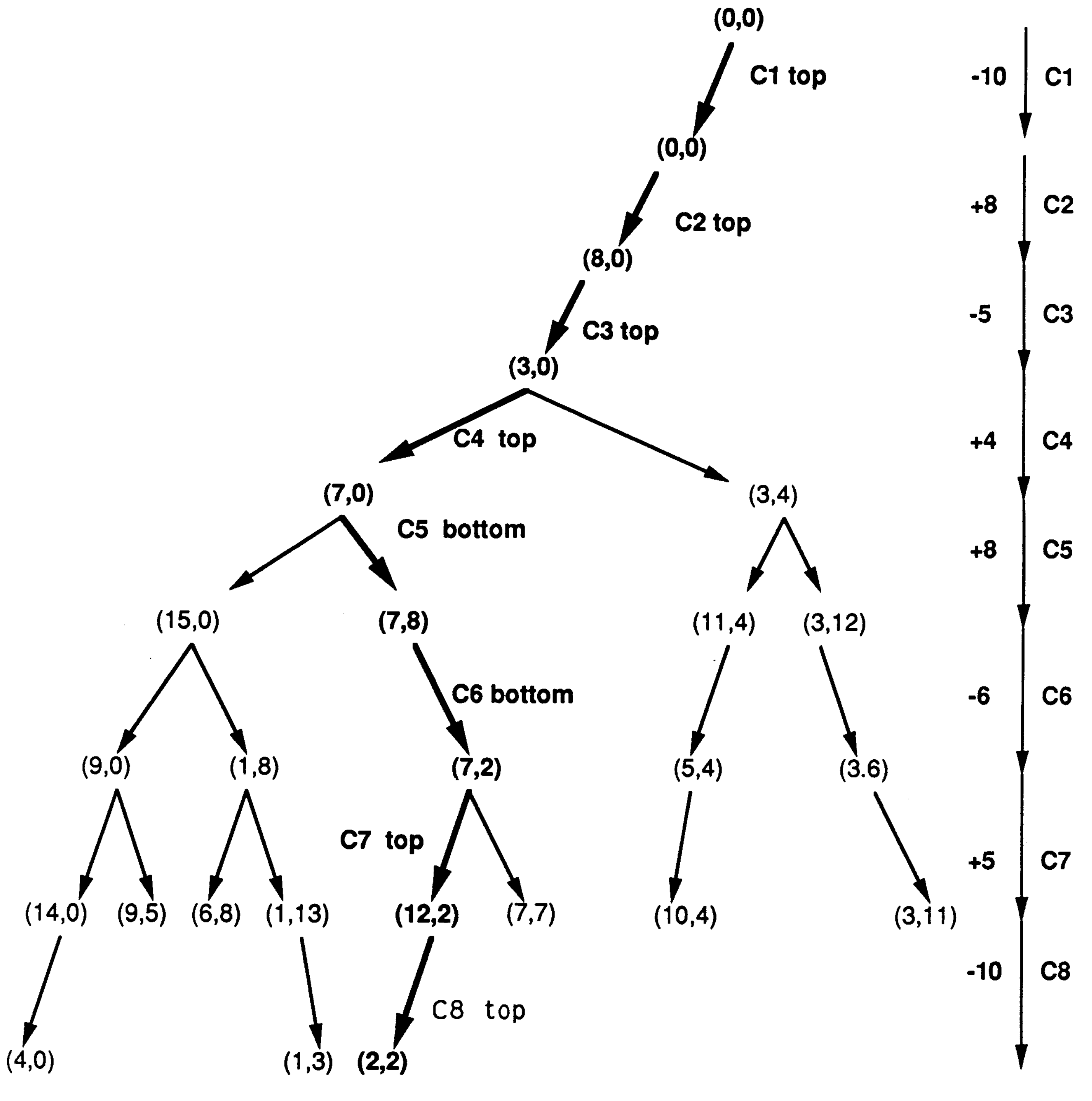

Figure 11(b) Binarty Tree of Signatures, with Duplicates and Symmetric Duplicates Deleted. Path to Optimal Signature.

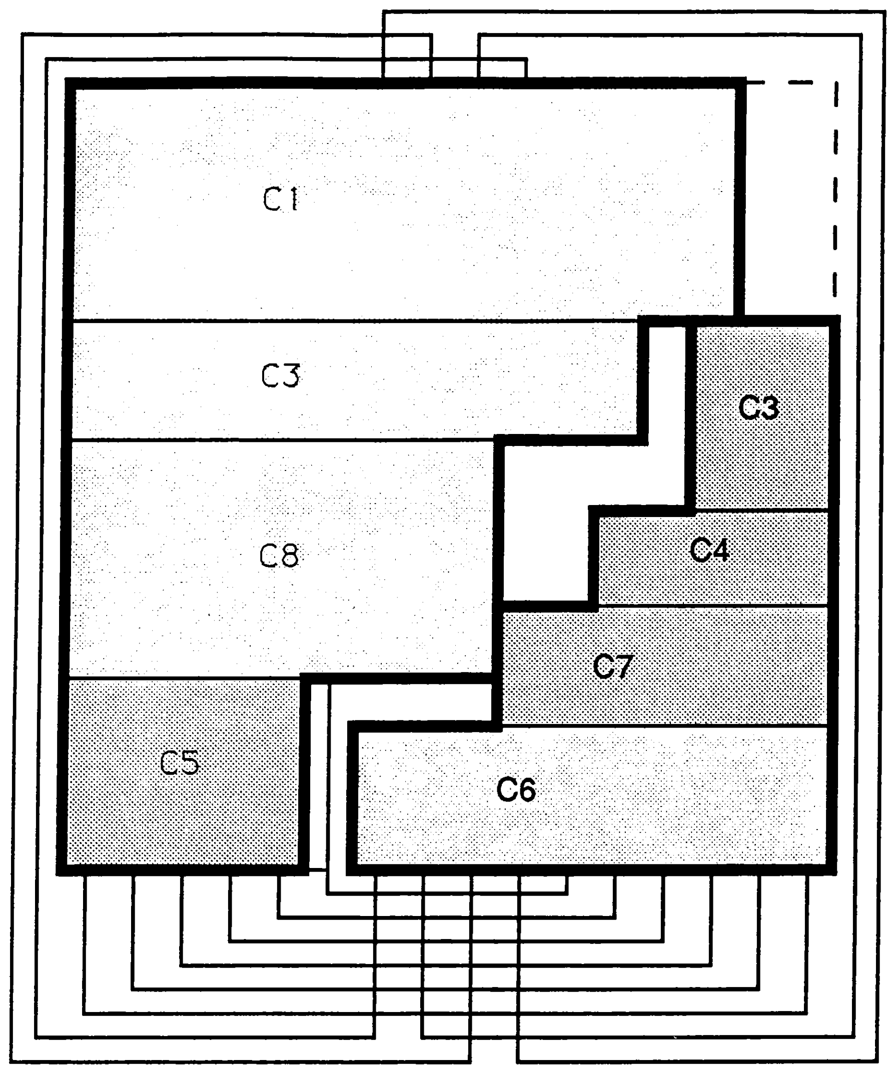

Figure 11(c) The Optimal Interleaved Architecture, Showing Routing

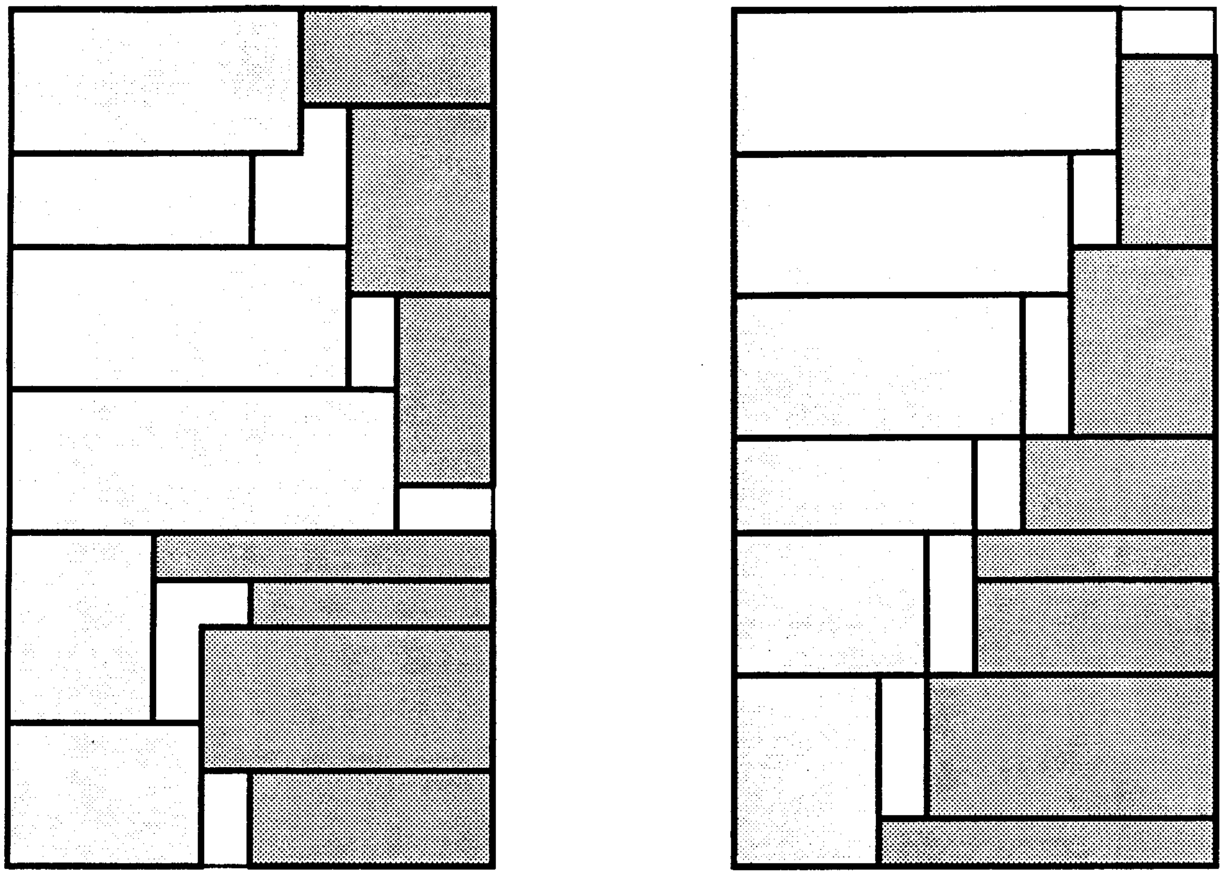

Figure 12 Illustrating Theorem 7.1

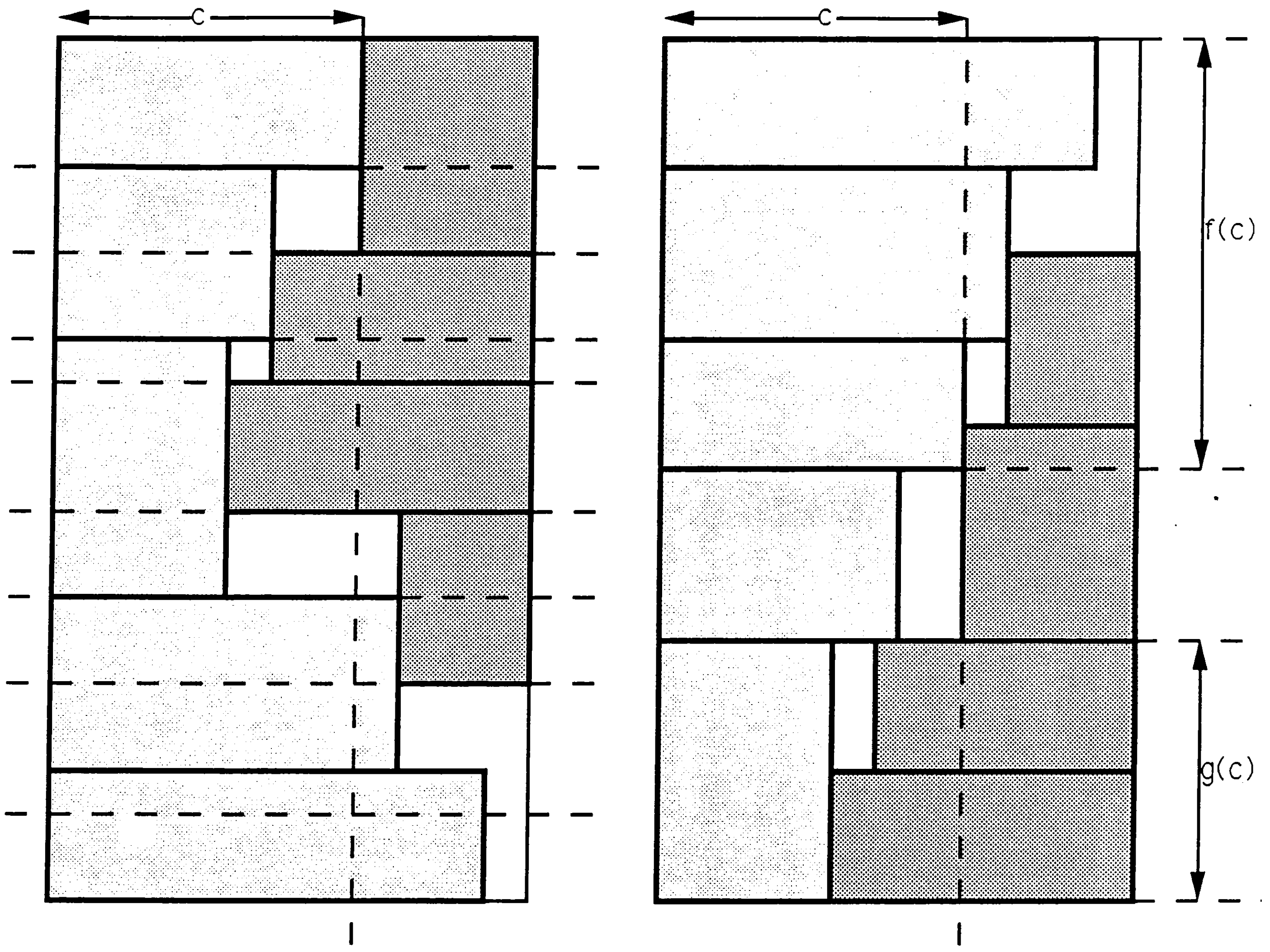

Figure 13 Illustrating Proof of Lemma 7.1

Number of Components	Area of Components in square millimeters	Maximum # of bit slices in a Component	Sum of Heights of Components in microns	Area of Unfolded Module in square millimeters	Percentage Waste of Unfolded Module	Number of Bit Slices in Optimal Interleaved Module	Height of Optimal Interleaved Module in microns	Area of Interleaved Module in square millimeters	Percentage Waste of Interleaved Module
10	7.030	29	4125	15.551	54.8%	31	2073	8.354	15.9%
10	7.863	32	2728	11.348	30.7%	32	2116	8.803	10.7%
20	15.668	32	7038	29.278	46.5%	36	3605	16.871	7.1%
20	13.648	30	6995	27.280	50.0%	31	3631	14.633	6.7%
30	21.584	32	10191	42.395	49.1%	32	5513	22.934	5.9%
30	22.532	32	9995	41.579	45.8%	33	5749	24.663	8.6%
40	24.720	32	13473	56.048	55.9%	32	6322	26.300	6.0%
40	23.395	32	14270	59.363	60.6%	32	6064	25.226	7.3%
50	23.460	32	18393	76.515	69.3%	33	5767	24.740	5.2%
50	21.137	32	17665	73.486	71.2%	31	5785	23.314	9.3%

Table 1. Comparison results