Automatic Strategies in the Siemens RTL Tiled Window Manager

Ellis S. Cohen
A. Michael Berman
Mark R. Biggers
Joseph C. Camaratta
Kevin M. Kelly

Siemens Research & Technology Laboratories
105 College Road East
Princeton NJ 08540-6668
(609) 734-6524
siemens!ellis (uucp)
ellis.cohen@agp.cs.cmu.edu (arpa)

The Siemens RTL Tiled Window Manager automatically adjusts the size and position of tiled windows to balance competing demands for screen space. The degree of automation can be set by the user and controls strategies which range from strictly local to those which can affect all the windows on the screen. This paper discusses these strategies, indicates the tradeoffs involved, and briefly describes the algorithms used in their implementation.

At the 1st IEEE Conference on Computer Workstations, we described our plans for a constraint-based tiled window manager [Cohe85] (republished as [Cohe86]). Window sizes were to be determined automatically through a prorating algorithm which fairly allocated screen space. During prorating, window placements were to be adjusted automatically by moving windows out of the way of a window which was being placed or resized. A variety of stability constraints could be used to limit the extent of side effects.

Our model drew some criticism at that conference -- namely, that users would not be pleased by the extensive automation we proposed. It was argued that users would want windows to be placed and sized exactly as specified by the user. In order to explore this issue, and because we wanted a working system before we fully designed the complete prorating and adjustment model, we built a sequence of window managers with varying degrees of automation. The automatic strategies we have explored fall into four categories, divided by proximity of effect.
1. Local strategies: Windows are placed or resized exactly how the user specifies, and adjacent windows are affected only to the extent necessary to maintain a tiled window manager. When space later becomes available, affected windows are restored to their original size and location.

2. Neighborhood strategies: When a window is placed or resized, its size and location, as well as that of its immediate neighbors, may be adjusted based on properties of the windows involved.

3. Regional strategies: The window manager may adjust the size and location of other windows on the screen in order to avoid having to close windows automatically. The goal is to keep the size of the affected region as small as possible.

4. Global strategies: The goal of global automation is to fairly allocate space on a screen-wide basis. Changing the size or location of a single window is likely to affect all other windows on the screen.

Our experience indicates that local automatic strategies alone produce a pleasing and quite workable tiled window manager. On the other hand, we have found that some users do want to have other automatic strategies available. Though the effects of window operations may sometimes be less predictable, these strategies prevent windows from being automatically closed, and provide fairer allocation of screen space to windows.

The remainder of this paper discusses these strategies by category, indicates the tradeoffs involved, and briefly describes the algorithms used in their implementation.

2. Local Strategies

With purely local strategies, windows are placed or resized exactly where the user specifies, and the window manager only shrinks or closes windows in order to maintain a tiled desktop. In addition, when space later becomes available, the window manager can enlarge a window to its desired size, and can reopen, at its previous location, a window that was automatically closed.
2.1. Automatically Shrinking and Closing Windows

When a user attempts to place or resize a window, neighbor windows may be occupying the space needed. This space is made available by shrinking or closing the affected neighbors.

There is typically more than one way to shrink a neighbor. In Figure 2-1, the window B may either be shrunk towards the top or towards the right in order to make space available for resizing window A.

The decision is first based upon the neighbor's minimum size. The minimum size may be set by the client, by the user, or by a defaults file (where an entry can indicate the minimum size of all windows of a class - e.g. the minimum size of all EMACS windows). If both alternate ways of shrinking the neighbor would cause the neighbor to shrink below its minimum size (in either height or width), the neighbor is automatically closed. If only one of the alternatives leaves the neighbor above the minimum size, then that alternative is chosen.

Where there are multiple alternatives, we calculate the goodness of each alternative and choose the one with the highest goodness. The goodness is calculated based on properties obtained from a defaults file. The properties indicate the importance of reaching the desired height and the desired width, which may differ. For example, it is more important that text windows reach their desired width than their desired height. Narrow text windows may cause lines to wrap, which reduces the effective window height as well as making the contents more difficult to read. Based on the properties, then, goodness is calculated separately for height and width, and the minimum of the two is taken as the overall goodness of the window.

Later on, we shall see that window goodnesses are also used for strategies that balance the size of windows that compete for screen space. In this context, a window's overall goodness is adjusted by a factor reflecting the relative priority of the window, yielding an adjusted goodness. For example, clock windows ordinarily have lower priority than editing windows, and so will be less likely to reach their desired size in the case of contention.

Figure 2-1: Shrinking A Window

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before</strong></td>
<td><strong>Vertical Shrink</strong></td>
</tr>
</tbody>
</table>

2.2. Desired Sizes

When a user explicitly shrinks, moves, or closes a window, the available space may be desired by an adjacent window.

Each window has a desired size - the size at which the user last explicitly resized it. A window may be shrunk below its desired size as a side effect of placing or resizing another window, but when the space that it previously occupied becomes available, the window manager can enlarge the window up to its desired size.

In our experience, this has been the single most important strategy in providing a comfortable user interface for tiled windows.

The window manager may also (optionally) grow a window to its desired size when that window becomes the listener (that is, after a user has pressed a mouse button or typed a key with the mouse cursor in the window). This closely corresponds to overlapping window managers that expose an obscured window when it becomes the listener.

Even when this option is selected, the user may want to bypass it. For example, imagine that a user is editing in one window (the listener) and a compilation has just finished in another window. The user may temporarily want to type into the other window to start another compilation. However, there is almost certainly no reason to enlarge (or in an overlapping system, expose) that other window just to type those few characters. In our window manager, when the user moves the cursor to the header of the other window and types there, the input is directed to that window; however, the window is not enlarged, nor is the old listener shrunk.

2.3. Automatic Repopulation

A window may be automatically closed as a side effect of placing or resizing another window, but when the space that it previously occupied becomes available, the window manager can reopen the window at its original location.

As a general strategy, we have found this useful, but not vital. However, in conjunction with two features we shall describe below - zooming and listener enlargement - automatic repopulation is essential.

By selecting a menu entry (or clicking in a gadget next to the window header), the user can zoom a window. This enlarges it to a pre-specified size. The user can then unzoom the window, which allows it to shrink to its former size. If the user explicitly resized the window while it was zoomed, then a subsequent zoom will enlarge it to that size.¹

¹Note that there are two common, but different, meanings of zoom. The other meaning, typically used in CAD systems, keeps the window size the same, but enlarges the contents.
In an overlapping window system, when a window is zoomed, the size and location of the remaining windows are unaffected. In a tiled window system, zooming a window can cause major changes by forcing the shrinking and closing of adjacent windows.

The problem (which we have previously called the Temporary Enlargement Problem [Cohe86]) of returning the screen to its pre-zoom configuration is solved by automatically reopening those windows closed by the zoom operation, along with enlarging those windows that were shrunk by it.

Similarly, if we have enabled the option that enlarges a window to its desired size when it becomes the listener, that may force adjacent windows to close. When the window is no longer the listener, those previously closed windows are automatically reopened, shrinking the window that was the listener in order to provide space for them.

### 3. Neighborhood Strategies

Using local strategies only, a window is always set to the location and size specified when it is explicitly resized or placed by the user. The window manager may later shrink it, but not move it, to make space for another window. With neighborhood strategies, the location and size of both the window being acted upon, as well as its neighbors, may be adjusted during placement or resizing.

#### 3.1. Avoiding Closing

We believe that most users want to avoid having windows automatically closed. We provide a number of neighborhood strategies to support this.

##### 3.1.1. Adjusting the Specified Window Size

A user, in enlarging a window, or while specifying its size while placing it, may force its neighbor to shrink just a little bit too much, forcing the neighbor to close. The window manager may adjust the newly specified window size slightly (limited to a percent set in a defaults file) to avoid closing the neighbor. The size specified by the user is still remembered (i.e. the desired size), and the window will grow back to that size when possible.

##### 3.1.2. Moving the Window

Another way to avoid closing a neighbor is by moving the window. The simplest strategy provided is sliding. For example, enlarging a window 100 pixels to the right may automatically close its right-hand neighbor. This may be avoided by sliding the window 10 pixels to the left, with the resulting effect of enlarging the window 90 pixels to the right and 10 pixels to the left.

More generally, the window may be moved anywhere within a specified neighborhood. For example, in figure 3-1, enlarging window B to the right would close window C, and sliding it to its left would close window A. By moving the window B upwards slightly, it may be enlarged by sliding it to the left, without affecting its neighbors at all.

![Figure 3-1: Moving a Window to Avoid Automatic Closing](image)

#### 3.1.3. Moving the Neighbor

A third way to avoid closing a neighbor is by moving the neighbor. There are two different ways this can be done:

- **Plowing the Neighbor**: In plowing, the neighbor is pushed out of the way of an opening or enlarging window.
- **Relocating the Neighbor**: With relocation, the neighbor is reopened as near as possible to its original position and within a limited neighborhood of it.

Figure 3-2 shows a window A at the bottom left of the screen being resized towards the right. Plowing pushes window B all the way to the right, while relocating reopens the window B just above its previous location. If no space were available to the right of the resized window A, then plowing would have failed to keep window B open. On the other hand, if no space were available above window A, then relocation would have failed (the plowed position would have been outside of the limited neighborhood of relocation).

![Figure 3-2: Moving a Neighbor to Avoid Automatic Closing](image)
Each approach has advantages and disadvantages, and different users may find one or the other more natural. We allow the user to choose either option or both.

3.2. Balancing Goodness

Using the RTL window manager, a user may open or move a window by specifying a location for it without indicating the size. In fact, once the desired size has been specified, either explicitly by the user, or from a value taken from an layout file, this is the most common way of opening or moving it.

This does not mean that the window should automatically be opened at its desired size. When there is competition for screen space, this could unduly affect its neighbors. Consequently, in our latest version of the system, we open the window as close as possible to its desired size, with the constraint that, in shrinking a neighbor as a result, the adjusted goodness of the neighbor may not be reduced below the adjusted goodness of the window being opened.

The window is not necessarily placed exactly where the user centers it. In fact, its location may be adjusted somewhat so that with its goodness balanced as described above, it can be made as large as possible.

4. Regional Strategies

Regional strategies prevent automatic closing. They are similar to corresponding neighborhood strategies except that more distant windows may be affected.

Neighborhood plowing pushes out of the way immediate neighbors of the window being placed or resized. However, they may be squeezed against their neighbors (on the other side). **Regional plowing** allows those neighbors to be moved as well (and if necessary, their neighbors, etc.)

This situation is not at all uncommon. For example, in figure 4-1, there are three windows on the screen, and the user is enlarging the top one. If both windows below it have fixed size, then the bottom window needs to be plowed downwards to prevent the middle window from being closed.

![Figure 4-1: Regional Plowing](image)

Regional relocation, like neighborhood relocation, opens a neighbor (whose space is needed) as near as possible to its original location. Neighborhood relocation requires that this location be close to the original, otherwise the neighbor will simply be closed. Regional relocation has no such limit, and in the worst case, can even move a window to the other end of the screen. Still, some users prefer this behavior to finding the window automatically closed.

5. Global Strategies

**Prorating and filling** are both global strategies aimed at fairly allocating space to windows on a screen-wide basis. Prorating balances the adjusted goodness of all windows on the screen as they compete to reach their desired sizes. Extra screen space may still be available when all windows are at their desired size. This can occur if the desired sizes are relatively small, or if there are a small number of windows. In this case, the user can choose to **fill**, which balances the goodness of the windows as they compete to reach their maximum (rather than desired) sizes and use up the extra space. Users can prorate or fill by explicitly selecting these operations from a menu, or they can set options which automatically prorate or fill after each window operation.

Goodnesses for filling are calculated based on different properties than for prorating, since the relative bias for height or width may change. As we noted earlier, in the case of editing windows, goodness changes more rapidly due to changes in width than in height. Beyond the desired size (at which line wrapping no longer occurs), width increases are hardly valuable at all, while increases in height remain useful.

Immediately after the user resizes a window, or after the window has been enlarged on becoming the listener, its goodness is set artificially low, so that, even with automatic prorating, it will attain its desired size at the expense of other windows. Subsequently though, it again competes normally for screen space.

In the model described in [Cohe86], prorating works hand in hand with adjustment. When a window would be closed during a prorating step due to lack of space, an attempt is made to move some window on the critical path out of the way, and prorating proceeds from the adjusted configuration.

We are continuing work on this model, but in the meantime, prorating (and filling) simply allocate space without adjusting the layout. Other local, neighborhood, or regional strategies must be enabled to avoid closing windows; prorating and filling then allocate space fairly to the ones that remain open.
6. Abandoned Strategies

This paper has presented a collection of "good" strategies, organized according to proximity of effect. Of course, this organizational model was not fully apparent when we began building our system, and strategies have been abandoned as new ones have evolved. Abandoned ideas are not often discussed, but we think the reader may be interested in three examples of how our strategies evolved.

6.1. Automatic Sizing during Placement

A user may open or move a window by specifying a location for it without indicating its size; the window manager determines the appropriate size automatically. We are now on our third method of picking this size. It is interesting to note that each step was tied to the development of a new abstraction.

In our first version, we used the previous window size. That generally worked out nicely if the window was being moved, though, the amount of space available in the new location could differ pretty dramatically from what was available at its previous location.

Before we had developed the notion of desired sizes, it was hard to do better. We hit upon the notion of desired size to allow windows to grow back to a reasonable size when space was available after they had been automatically shrunk, but this also seemed to solve the problem of automatic sizing.

So, our second version automatically chose the window's desired size. However, when there was significant contention for screen space, opening a window at its desired size would excessively shrink its neighbors. (Initially, it would even close its neighbors, but we changed the default option setting so that the newly opened window would be shrunk instead, though not below its minimum size, of course).

This seemed reasonable in a way -- a newly opened window often immediately becomes the listener, and we already optionally enlarge a window to its desired size when it becomes the listener. As a default, however, it proved too disruptive, especially since some users did not want automatic listener enlargement.

At the time we recognized this problem, we were searching for a single framework that we could use both for choosing how to shrink a window and as the basis for prorating. Once the idea of goodness had evolved, we saw that we could pick a size that would balance a newly opened window's adjusted goodness with that of its neighbors. This is used in the latest version of our system, and is described earlier in the paper.

6.2. Listener Tied To Zoom

Our second example of an abandoned strategy was due to confusion between related concepts. Even before we completely understood the notion of desired sizes, we wanted to allow a window to grow when it became the listener. Initially, we confused this idea with the idea of zooming, since both involve automatic window enlargement, and arranged for a window to optionally enlarge to its zoom size when it became the listener.

We came to realize though, that the zoom size of a window is best thought of as a pre-determined size, used only at special times. We had just developed the notion of a desired size which was the size ordinarily preferred by the user, and which changed when the user explicitly resized a window. It eventually became clear, that it is the desired size, not the zoom size, which should be used to enlarge a window when it became the listener.

6.3. Automatic Repopulation Tied To Zoom

Our third example is a strategy that was not exactly abandoned -- rather, interference with another strategy required us to modify the initial strategy.

We've described an automatic repopulation strategy which solves the Temporary Enlargement problem -- when a window is unzoomed, it automatically reopens windows that were closed during the zoom. This means that the layout after unzooming is identical to the pre-zoom configuration.

However, this solution to the problem fails because of our strategy which relocates windows rather than closing them. Since relocated windows are still open (although in a different location), they will not be placed in their original location by automatic repopulation. Moreover, they may now even be in the way of other windows which were closed that could be reopened.

We felt that restoring the pre-zoom configuration was important, and so our first reaction was to disable relocation strategies on a zoom operation. However, users rarely change the screen layout between zooming and unzooming a window. We had already implemented a one step Undo operation, and realized that we could use it to restore the previous layout, as long as no window operations were performed between the zoom and unzoom. When other operations were performed, exactly restoring the pre-zoom configuration was no longer necessarily the correct action to take anyway.

Our latest strategy, then, is to allow relocation during zooming. On unzooming, if no other operations have intervened, we simply undo, exactly yielding the pre-zoom configuration. If there have been other operations, we try automatic repopulation.
7. Algorithms for Implementing
Automatic Strategies

The realization of the strategies described here has required some new and innovative algorithms for managing the tiled desktop. These algorithms will be described in some detail in a forthcoming paper [Berm87]. In this section we provide a brief overview of some of the algorithmic issues for tiled windows.

7.1. Data Structures

The layout of the desktop is represented internally using a version of the corner-stitching data structure discovered by Ousterhout [Oust84], originally developed for computer-aided VLSI design. In corner-stitching, the desktop subplane is tiled into solid tiles, which correspond to the windows on the desktop, and space tiles, corresponding to unused space. As we shall see below, the ability to explicitly represent empty space is useful for our algorithms. All tiles, space and solid, are linked by four pointers -- two each at top left and the bottom right corners. This allows excellent efficiency for operations such as finding all the neighbors of a window. Inserting a new tile into a desktop of roughly equal-sized tiles requires time independent of the number of tiles on the desktop. Finding the tile that contains an arbitrary point on the desktop can be done, on average, in time proportional to the square root of the number of windows. While the number of tiles on the desktop may be quite small, say, always less than 100, the efficiency of such primitive operations is critical to the performance of an interactive tiling system.

7.2. Automatic Window Placement

When automatically placing a window on the screen, other windows may be required to shrink or close. In placing the window, the manager attempts to balance the space and location requirements of the newly opened window with the costs of disrupting windows in the current configuration. The automatic window placement algorithm is required to balance these considerations, so that the needs of the new window are met with the smallest possible disruption to the user.

There are two distinct versions of the placement algorithm. In one, the algorithm is given a preferred point on the desktop; given two equally "non-disruptive" positions to open the window, the position closest to the preferred point is selected. In the second version, the position to open will be based on a "goodness of fit" criterion. In this version, the window can be opened either where the fit is "loosest", i.e., the empty space around the new window is maximized; or, it can be opened where the fit is tightest. We are experimenting with different fits in order to find the method(s) that provide the most attractive interface.

The key issue in finding the optimal position to open the window is assuring that all "candidate" configurations are evaluated. The desktop can be viewed as a set of x-y grid points at some resolution. One simple, but inefficient, solution is to try every grid point. Not only will this require examining a large number (perhaps several hundred thousand) of positions; but in each position, there are myriad possibilities for shrinking the adjacent windows, as well as shrinking the newly opened window (which is optionally allowed). The key observations to limiting the enumeration complexity are:

- Only positions adjacent to the edges of existing windows, and the edge of the desktop, need be considered.
- Of these positions, only positions where the corner of the new window is aligned with the corner of other windows on the desktop need be considered.
- Once these positions are examined, a sliding operation can be used to find the best position that is closest to the preferred point.

Finding the corner positions, and sliding, are quite efficient with the corner-stitching data structure. During the enumeration process, the examination of a position can often be terminated quickly by using the best solution found so far as a bound.

The problem of finding the optimal configuration while allowing shrinking can be cast as maximizing goodness. The goodness equations for a group of windows can be pre-solved, yielding a set of simple equations to evaluate. Given a new window which is surrounded by several windows on the screen, there is a set of at most three windows in each direction which, when balanced, yield the smallest size for the new window. Thus, the evaluation of many possible configurations is unnecessary. The entire placement algorithm requires, in the worst case, time proportional to the square of the number of windows on the desktop. However, the worst case estimate is unduly pessimistic; we can expect that most configurations will require time linear in the number of windows.

7.3. Prorating

As space on the desktop becomes available due to the closing and shrinking of windows, prorating allocates it among the open windows. As described in Section 5, prorating balances the goodness of all windows on the screen in an attempt to produce the most desirable overall allocation.
Prorating has some similarity to the problem of compaction that arises in VLSI CAD (see, e.g., [Cho85]). As with the compaction problem, we have a two-dimensional problem that we wish to solve — the allocation of space in the vertical and horizontal directions is not independent, and an optimal solution must consider both directions simultaneously. However, no efficient algorithm for two-dimensional compaction, or two-dimensional prorating, is known. Thus we solve the prorating problem in one direction, e.g., the horizontal, and feed that result to a prorating computation in the orthogonal direction. This process repeats until there is no change in size. In practice, iterated one-dimensional prorating halts very quickly and generally yields excellent results.

To solve a one-dimensional prorating problem, we begin by constructing a constraint graph. Suppose we are looking in the horizontal direction first. Then, the left edge of the desktop will become a source in the graph, and the right edge a sink. Nodes represent tile edges that are perpendicular to the direction of the prorate, and edges represent constraints. Suppose, for example, that tiles A, B, and C are aligned such that the sum of their horizontal sizes cannot exceed the entire desktop. Then there will be a path from the left edge, to A, to B, to C, and to the right edge. In addition to these relationship constraints, the algorithm considers location constraints (a tile is in a fixed location) and size constraints (a tile can have a minimum and maximum size). These constraints generate a set of constraint equations, which model the prorating problem.

The objective is to minimize the cost of allocating the space in the plane by determining the (discrete) size of tiles in the plane with respect to topological and tile-specific constraints. Initially, all tiles are (internally) shrunk to their minimum sizes, leaving some amount of slack space. We use a heuristic which attempts to minimize a non-linear objective function. This is accomplished by propagating slack space through the graph, and using this space to determine the size of the tiles. The time used by this method is a function of the configuration of tiles on the desktop, and the amount of slack. Prorating a typical layout of about 10 windows and icons takes less than 1/4 second, even with a version of the algorithm that has not been particularly well tuned.

As of Fall 1987, just a few people outside of our group have used our window manager extensively, and we have not yet had the opportunity to perform the kinds of human-factors experiments that would evaluate our user interface or compare various strategies. It would be especially interesting to repeat the experiment described in [Bly86] that compares tiled and overlapping windows. The tiled window system used in that experiment is missing certain key strategies — in particular, automatic listener enlargement. We hypothesize, that due to our strategies, our system should uniformly yield faster response times than overlapping windows, at least on the tasks they studied.

The proper choice of strategies is certainly crucial; making strategies optional allows each user to choose the degree of automation that makes them comfortable.

Local strategies are the most easily accepted, especially by skeptical users. With these strategies alone, a pleasing tiled window system can be provided which performs well on screens ranging in size from PCs to workstations. In addition, many users take advantage of neighborhood strategies to avoid automatic closing and to better balance space allocation.

With the screen sizes available to us, regional and global strategies have not assumed major importance. However, we believe these will become more important as large and very large (desk and blackboard size) screens become available.

Meanwhile, we are continuing our research into window systems on currently available screens. We are now working on multiple tiled desktops, extending the models used in Smalltalk-80 for projects [Gold84] and in the Rooms system [Card87]. The use of desktops means that fewer windows will be visible at a time, creating, in our view, even fewer reasons to overlap (although whether desktops themselves should be tiled or overlaid is still open).

And finally, although our insistence on tiling as a basic model still seems right, we are not rigidly bound to a pure tiling system. Recently, we have been finding more and more examples in which it is convenient to allow at most one window (not including transient pop-up windows) to be lifted out of the tiled desktop and temporarily overlay it. This will be supported in a forthcoming version of the system.

8. Conclusion

We started development of a tiled window system based on the belief that overlapping is a primitive solution to contention for screen space, and that tiling, if done right, could provide a superior interface. After more than a year of experimentation with a working tiled window manager, we see no reason to change that view.
References


