# Optimizing Toolkit-Generated Graphical Interfaces

*Bradley T. Vander Zanden*
Computer Science Department
University of Tennessee
Knoxville, TN 37996
Tel: 1-615-974-8175
E-mail: bvz@cs.utk.edu

**ABSTRACT**

Researchers have developed a variety of toolkits that support the development of highly interactive, graphical, direct manipulation applications such as animations, process monitoring tools, drawing packages, visual programming languages, games, and data and program visualization systems. These toolkits contain many useful features, such as 1) structured graphics, 2) automatic display management, 3) constraints, and 4) high-level input-handling models. Despite a number of optimizations that have been described in the literature, most toolkit-generated applications run in a predominantly interpreted mode at runtime: they dynamically determine the set of constraints and objects that must be redisplayed, which requires the use of time-consuming algorithms and data structures. The optimizations that do exist rely on semantic information that applies *globally* to all operations in an application. In this paper we identify a number of optimizations that require *local*, operation-specific semantic information about an application. For each operation, these optimizations pre-compute update plans that minimize the number of objects that are examined for redisplay, and pre-compute constraint plans that minimize the amount of dynamic scheduling and method dispatching that is performed for constraint satisfaction. We present performance measurements that suggest that these optimizations can significantly improve the performance of an application. We also discuss how a compiler might obtain from a programmer the information required to implement these optimizations.

**KEYWORDS:** Structured Graphics, Automatic Redisplay, Constraints, Development Tools, Optimization

## 1 INTRODUCTION

Graphical interface toolkits have gained popularity by providing features that conventional languages lack and which simplify the creation of highly interactive graphical applications: (a) structured graphics, (b) automatic display management, (c) dataflow-style constraints, and (d) input-handling [14, 7, 1, 3, 8]. These toolkits dramatically decrease the effort required to create applications such as animations, process monitoring tools, drawing packages, data and program visualization systems, visual programming languages, and games. However, they do so at the cost of decreased runtime performance and increased storage requirements.

This dichotomy between ease of development and runtime performance reflects the fact that toolkits have focused primarily on language design issues and have ignored optimization issues that could be addressed by compilation techniques. Toolkits currently employ interpretative mechanisms at runtime. Each time that a stimulus alters the state of an application, a toolkit dynamically devises and executes a plan that consists of (a) a set of event-handlers to which an event should be dispatched, (b) a set of constraint methods to be executed, and/or (c) a set of objects to be redrawn. However, creating plans at runtime causes applications to run substantially slower and to consume substantially more storage than comparable hand-crafted applications. For instance, an Othello board game created using the Garnet toolkit (Figure 1) runs 19 times slower and consumes 18 times more storage than a hand-crafted version of the game.

The relative inefficiency of toolkit-generated applications compared to hand-crafted applications can force a programmer to accept uncomfortable compromises, such as jerkier animations, less detail during user interaction (e.g., a dashed line rectangle rather than an actual image of the moving object), or slower response time. The larger storage requirements of toolkit-generated applications reduces the number of objects that an application may display before exceeding process limits or the limits of RAM memory

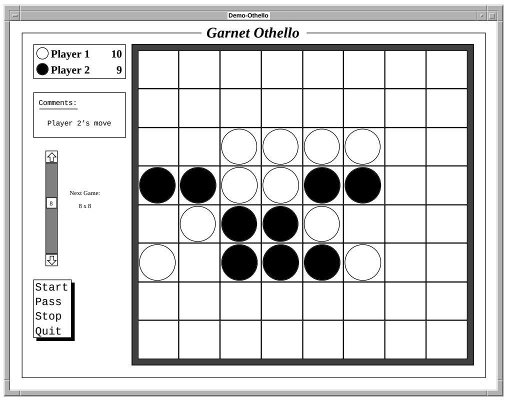



Figure 1: An Othello game that was created using the Garnet user interface development toolkit. The Garnet-generated version of the application was constructed in roughly 1/10 the time of the hand crafted version, but the handcrafted version consumed 18 times less storage and ran 19 times faster.

(exceeding the limits of RAM memory forces an application into virtual memory, thus potentially causing thrashing).

Ideally, toolkits would provide compilers that would automatically optimize toolkit-generated applications. Currently no such compilers exist. In addition, many of the optimizations that have been described in the literature for toolkits rely on global information that applies to all operations in an application. However, many profitable optimizations rely on local information that applies to only one or a few operations.

In this paper we present a number of optimizations that rely on local information about operations. These optimizations pre-compute update and constraint plans that minimize the amount of damaged object detection and constraint scheduling that must be dynamically performed. We have implemented these optimizations in a number of hand-crafted applications and compared their performance against the performance of similar toolkit-crafted applications. The results suggest that the optimizations can significantly improve the performance and storage requirements of interactive, graphical applications.

The rest of this paper is organized as follows. Section 2 describes a number of common high-level features provided by toolkits and summarizes optimizations for these features that have been discussed in the literature. Section 3 describes the applications we employed in our experiments. Section 4 describes the optimizations we have identified and Section 5 presents performance measurements for these optimizations. Section 6 describes the types of information that a compiler must have in order to make these optimizations, and how a compiler might obtain this information (e.g., by statically analyzing code, by employing hints provided by a programmer, or by profiling the application). Finally Section 7 presents our conclusions.

## 2 DEFINING THE PROBLEM: CURRENT STATUS

In this section we briefly describe three common high-level features of toolkits that we would like to optimize: 1) one-way constraints, 2) structured graphics, and 3) automatic display management. Studies indicate that input-handling typically consumes relatively few resources relative to these three features and thus optimizations for input-handling will not be considered in this paper [18, 16, 9].

### 2.1 Constraints

A one-way constraint is a relationship defined by a programmer and then automatically maintained by a system. For example, a programmer could use the equation `rect2.top = rect1.bottom + 10` to constrain the top of one rectangle to remain 10 pixels below the bottom of another rectangle. Whenever `rect1` moves vertically, the system automatically satisfies the equation, causing `rect2` to move correspondingly. Constraints can be generalized to handle linked data structures involving pointers by allowing constraints to include pointer variables [18, 9, 10]. For example, the above constraint can be rewritten as:

```
rect2.top = self.previous.bottom + 10
rect2.previous = rect1
```

where `self` refers to the object containing the variable `top`, in this case, `rect2`. The constraint will be satisfied by following the link in `previous` to `rect1`, retrieving the value of `rect1`'s `bottom` attribute, and adding 10 to it. By changing the value of the variable `previous`, `rect2` can be positioned under any object.

One-way constraint systems are fine-grained and dynamically scheduled. This combination affords a major advantage: flexibility—any part of the system can be related to any other part by constraints created at runtime. But it also results in two disadvantages—high time and space overhead at runtime. Fine-grained evaluation contributes to overhead since it requires a method to be called to evaluate each constraint. A method typically consists of only one or two statements, and thus the cost of the invocation exceeds the cost of the evaluation. The high time and space overhead associated with fine-grained, dynamically scheduled evaluation limits the number of objects and constraints which can be used.

Traditional methods for reducing overhead involve (1) compiling constraints into plans [6, 17] and (2) using constant propagation [12, 15]. Constraints are compiled into a plan by determining the order in which they will be scheduled, placing them in a single procedure, and removing their individual method wrappers. This compilation eliminates dynamic scheduling and individual method calls. In the past, constraint plans have been dynamically compiled for operations [17, 2, 13] and statically pre-compiled for groups of objects [6]. However, they have not been statically pre-compiled for operations, which is one of the optimizations examined in this paper. A constraint solver implements constant propagation by determining whether all the arguments to a constraint are constant, and if they are, evaluating the constraint for its result and then eliminating it. By eliminating constraints, constant propagation reduces the number of equations evaluated at runtime and the number of constraints stored at runtime. Constant propagation has been reported on elsewhere [12, 15], and thus will not be addressed further in this paper.

### 2.2 Structured Graphics and Automatic Redisplay

Structured graphics maintain the state of every graphical object in the system—a feature allowing a toolkit automatically to detect and redisplay damaged areas of the screen. This automatic redisplay relieves the application programmer of the responsibility for implementing potentially complicated data structures and algorithms for display maintenance. There are also attendant costs. Structured graphic systems typically have high time and space overhead because 1) every object displayed must have a corresponding object in memory; and 2) they often lack the semantic information required to speed up damage detection and redisplay. Lacking semantic information, they must always perform expensive intersections tests and maintain potentially expensive data structures at runtime.

Toolkits have typically tried to reduce these costs in several ways. Glyphs are extremely lightweight objects that only store information which cannot be computed by other means [5]. Virtual aggregates attempt to eliminate objects completely by constructing objects only when they are demanded by an operation, and then reclaiming the objects once the operation finishes [15]. However, these approaches only partially solve the problem: They reduce the storage cost, but do not eliminate the expensive intersection tests that must be performed at runtime to redraw the individual objects. Display lists compile draw-methods down to lists of graphics display instructions. However, this strategy only partially remedies the problem because it is still necessary to 1) dynamically detect damaged areas of the display, 2) find and redisplay the objects that intersect the damaged areas, and 3) update the display list as objects are modified, created, or deleted. Finally, some toolkits, such as Garnet [11, 15], allow the programmer to explicitly specify semantic information about an application. For example, a programmer might specify that a menu cannot overlap any other object or might provide an estimate of the number of objects that will be used at runtime. This information allows the system to select a data structure that reduces the number of intersection tests required to detect damaged objects.

## 3 EXAMPLES

This section describes the three graphical applications that were used in our experiments. The applications were implemented in Lucid Common Lisp using the Garnet user interface development environment [14] and then reimplemented in Common Lisp and CLX (Common Lisp X) without the aid of a toolkit. The algorithms that Garnet uses for automatic redisplay and constraint satisfaction are representative of algorithms used in other toolkits.

Two of the applications, the arithmetic editor and Othello, are provided as demos in the Garnet release and were slightly modified for the experiments. The selection sort animation and all three hand-crafted applications were implemented by the author. The author is one of the Garnet developers and thus is an expert Garnet programmer. The applications are representative of the class of applications that we are interested in optimizing. The three applications are 1) the game of Othello, 2) a visual programming language for arithmetic expressions, and 3) an animation for selection sort.

### 3.1 Othello Game

Othello is played by two players on an arbitrarily sized square board with $n \times n$ squares. The winner is the player with the most game pieces on the board when the board is either completely filled or neither player has a legal move. A player takes a turn by placing a game piece adjacent to one of the opposing player's game pieces (all eight neighbors of a square are considered adjacent to a square). The placement is legal if by moving to one of the adjacent squares that contains one of the opposing player's game pieces, and then continuing in the same direction, a piece that belongs to the player who moved is eventually encountered. All of the opposing player's pieces between the two ‘‘bookend’’ pieces are replaced by the pieces of the player who moved. The search is repeated in each of the seven other directions. Players alternate turns. If a player has no legal move, the player's turn is skipped.

The interface for the Othello game is shown in Figure 1. The interface includes a score box that displays the current score, a message box that provides instructions, a scroll bar that controls the size of the gameboard, a menu of options, and the gameboard itself. Players press on an appropriate square to make a move.

### 3.2 Arithmetic Editor

The arithmetic editor allows a user to construct arithmetic expressions involving addition, subtraction, multiplication, and division. The interface for the editor is shown in Figure 2. The user creates arithmetic expressions by drawing data and operator nodes, and then drawing directed edges from inputs to operators, and from operators to outputs. The user can edit the data nodes, move the data and operator nodes, delete the data and operator nodes, and delete the edges connecting nodes. When nodes are either moved or resized, the arrows automatically reposition themselves so that their endpoints lie on the imaginary line that runs through the centers of the two nodes they connect.

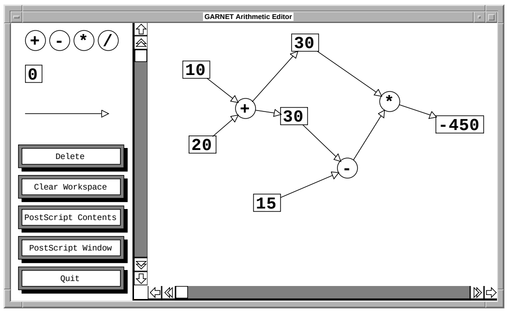



Figure 2: An interface for a visual, arithmetic expression language. The user builds expressions by wiring together operators and operands to create dataflow diagrams. An evaluator automatically computes the values of the expressions and updates the appropriate data nodes.

### 3.3 Selection Sort

Selection sort orders a list of *n* elements by making *n-1* passes through the list. On the *i*th pass, it locates the *i*th smallest element in the list, and swaps this element with the element at the *i*th position in the list. Thus after the *i*th pass, the *i* smallest elements in the list have been found and properly ordered.

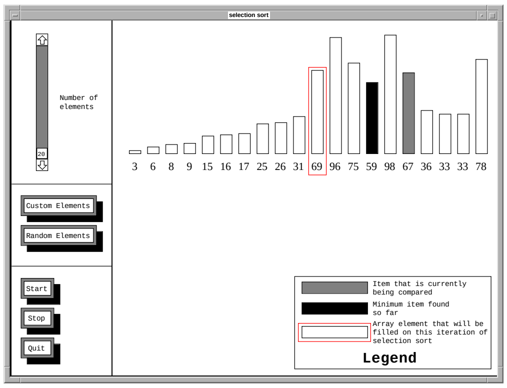
Figure 3 shows the interface for the selection sort animation. The animation adopts the convention of using rectangular bars [4] to represent the size of each element. The *i*th element in the list, the smallest element found thus far in the *i*th pass, and the current element being examined are highlighted with different textures (Figure 3.a). When the list has been completely examined, the two objects to be swapped are animatedly moved along separate paths (Figure 3.b).

## 4 OPTIMIZATIONS

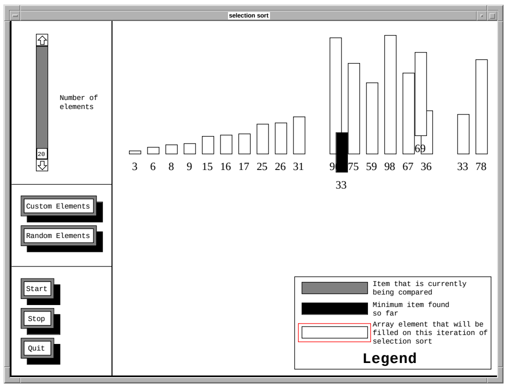
After the hand-crafted applications had been written, they were examined for optimizations that were relevant to structured graphics, redisplay, and constraint management. In the following section, these optimizations are grouped according to whether they could be applied to the structured graphics, redisplay, or constraint portion of a toolkit.

### 4.1 Structured Graphics

The optimizations for the structured graphics systems were storage optimizations that allowed the creation of lightweight rather than heavyweight objects.




(a)




(b)

Figure 3: An animation of selection sort. (a) The element of the array that will be filled by this iteration is outlined by a rectangle (the entry occupied by 69), the minimum entry found so far is blackened, and the current entry being examined is filled with gray. (b) A snapshot taken as two elements are animatedly swapped.

*Composite Object Folding*. Many of the objects used in the toolkit-generated applications were aggregate objects that consisted of one or more child objects. For example, the labeled rectangles in the selection sort animation were composed from three objects: an aggregate object, a rectangle object that represented the frame, and a text object that represented the label (the frame and label were children of the aggregate object). Similarly, the arrows in the arithmetic editor consisted of three objects: the aggregate arrowline object, a line object, and a polyline object representing an arrow. In the hand-crafted applications, the children were folded into the aggregate object so that only one object was required (i.e., the children objects were eliminated and only essential information, such as a text object's string, was stored in the composite object).

*Glyph Optimization*. The objects in the handcoded applications were lightweight objects that averaged less than 40 bytes per object, as opposed to several hundred to several thousand bytes per object in the toolkit-generated applications. This compactness was achieved by factoring out information that was common to objects and placing the information either in ‘‘container’’ objects that laid out the objects or as class variables in a class. For example, the width and height of the game pieces was stored in the game board rather than the individual pieces. Information that could be computed by inexpensive functions was also factored out of the objects. For example, text objects did not store their width or height, since these values could be inexpensively obtained by calling an appropriate function and passing the string and font as parameters. When text objects were folded into aggregate objects, as in the labeled rectangle object, their position information was also eliminated and computed on demand by the draw method.

### 4.2 Automatic Redisplay

The automatic redisplay algorithm for Garnet constructs two bounding boxes as objects are modified by an application. One bounding box contains the old positions of the objects and one bounding box contains the new positions. When the application asks Garnet to update the display, Garnet’s redisplay algorithm finds all objects that intersect the two bounding boxes and redraws them, clipped to the bounding boxes. Garnet examines all the objects that are registered with the window that is being redisplayed. When the redisplay algorithm encounters aggregate objects, it determines whether the aggregate’s bounding box intersects the damaged regions of the display, and, if it does, recursively applies the redisplay algorithm to the aggregate’s components. Branches of the aggregate hierarchy whose bounding boxes do not intersect the damaged areas of the screen are eliminated from the search without examining their children [11].

The following optimizations from the hand-crafted applications were able to perform faster than this algorithm because they used application-specific information that was not available to Garnet.

*Static Update Plans*. In many cases, we knew in advance the set of objects that an operation would modify or potentially damage. Thus we could precompute update plans that 1) redrew the changed objects and 2) checked each of the objects in the potentially damaged set and redrew them only if they were damaged. For example, when the user selects an item from the object pallette in the arithmetic editor, only the objects in the pallette are damaged. Similarly, when a player makes a move in Othello, the score strings are the only objects redrawn in the score box and the message string is the only object redrawn in the message box. Consequently, we created plans that redrew or checked these objects, and which ignored all other objects in the interface (a separate dynamic update plan was devised to handle the updating of the Othello game board).

*Dynamic Update Plans*. Sometimes the objects that will be modified or damaged by an operation cannot be determined until the operation is initiated at runtime. However, we could still employ information that was available statically to create dynamic update plans. A number of these optimizations included:

1. Redrawing only Altered Objects: Many operations require that only the objects that the operations modify be redrawn. For example, when one of the Othello players makes a move, the only dynamically created objects that must be redrawn are the game pieces that change. The programmer can optimize these operations by generating code which redraws only the changed objects.

2. Geographic Locality: Operations that repetitively execute until an event such as a mouse-up or a carriage return are received often execute over a restricted portion of the display. Resizing, repositioning, and text editing operations are representative of this class of operation. For example, as the labels in the arithmetic editor are edited, the damaged area is restricted to the area around the label and the arrows attached to the data node. In the selection sort animation, the damaged area on each incremental move as the two elements are swapped is restricted to a narrow region about each of the two moving rectangles. The programmer can optimize these operations by generating code that dynamically computes a bounding box that is somewhat larger than the initial damaged area and that caches all objects that intersect this bounding box. As long as the incremental changes stay within the bounding box, only the cached objects have to be checked for damage (or alternatively, simply redrawn).

3. Nonoverlapping Bounding Boxes: If the operation damages a number of non-overlapping regions of the display, it may be possible to pre-compute the number of bounding boxes that should be used (e.g., one bounding box per damaged area). For example, when the selection sort animation swaps the two labeled rectangles by moving them along two separate paths, only the two regions containing the two labeled rectangles must be updated. Consequently we created two bounding boxes to handle the swapping operation.

4. Erasing: Display managers typically erase an object by drawing a rectangle in the current background color of the window, and then redrawing the objects that intersect the erased region. However, there may be more efficient ways to erase objects, for example, by simply redrawing them if their new size is at least as large as their old size or by redrawing them in the background color if they do not overlap other objects. For example, as rectangles are highlighted and dehighlighted in the selection sort animation, they are simply redrawn, rather than erased and then redrawn. The rectangle that surrounds the *i*th element in the

selection sort list is erased by redrawing the rectangle using the current background color.

5. Special Data Structures: Although update plans can cache objects as an operation is executed, the update plan needs to efficiently find these objects at the start of an operation. For example, when a data node or operator node is moved in the arithmetic editor, the display manager must determine which objects are initially affected by this move. Frequently, an additional set of objects will also have to be redrawn when the operation terminates. For example, once the user finishes editing a data node in the arithmetic editor, the data nodes whose values depend on the edited data node must be redrawn, as well as the arrows attached to these data nodes. When the number of objects in the interface is relatively small, maintaining the objects in a linked list and serially searching them as Garnet does works well. However, when there are potentially hundreds or thousands of objects in an application, as in the arithmetic editor, a more complicated data structure may provide better performance. In the arithmetic editor, the display was divided into an *n* by *n* grid and the objects that intersected each grid were stored in a linked list (an object could be on multiple lists). When objects were modified, a bounding box for the modified objects was constructed, and only the grids intersecting the bounding box were searched.

### 4.3 Constraints

Each time a variable is changed in Garnet, all constraints that depend directly or indirectly on this variable are marked out-of-date. When the application requests the value of a variable defined by one of these constraints, the constraint is reevaluated, as well as any out-of-date constraints that this constraint directly or indirectly depends on [18]. The constraint optimizations described in this section reduce the space overhead of constraints by eliminating the number of constraints that must be maintained at runtime. They also improve the performance of an application by reducing the amount of dynamic scheduling and method dispatching that must be performed for constraint satisfaction.

*Constraint Plans*. We found that if we were given semantic information about an operation, we could statically pre-compute a plan for that operation which consisted of an ordered list of constraints affected by that operation. This constraint plan was executed each time the operation was invoked at runtime. For example, constraints were used to right justify the scores in the Othello score box and center the messages in the message box. A constraint plan that included these constraints was constructed and executed each time a player made a move. The constraint plans eliminated the dynamic cost of scheduling the constraints and the function overhead of calling the individual constraint methods. In addition, since all the constraints in each of the applications was compiled into a plan for each operation that affected them, the constraints did not have to be created at runtime, thus saving storage.

*Iteration*. In a number of instances, an object in the toolkit-generated application contained several constraints that were repetitively cloned, and that expressed a relationship that could be converted to iteration. For example, the `x` position of each rectangular bar in the selection sort animation was related to the origin of the bars list by the constraint:

```
self.left = bar_list.left +
 (self.rank * (self.width + h_space));
```

`self` is a pointer to the object containing the formula and `rank` is the bar's position in the list. In the handcoded version of the selection sort animation, the same layout was achieved using iteration:

```
for each self ∈ bar_list.components do
self.left = bar_list.left +
   (self.rank * (self.width + h_space));
```

As another example, the toolkit-generated arithmetic editor connected the arrows to nodes using the constraints:

```
endpoint1 =
   clip_from_center(self.from_obj,self.to_obj)
endpoint2 =
   clip_from_center(self.to_obj,self.from_obj))
```

`from_obj` and `to_obj` are the objects that the arrow originates from and points to, and `clip_from_center(x,y)` is a function that returns the point on object `x` which lies on the line that connects `x`'s center to `y`'s center. Whenever a node moves, the constraints of all the arrows attached to the moved node are executed. In the handcoded arithmetic editor, the same behavior was achieved using iteration:

```
for each arrow in node.in_arrows
    arrow.endpoint2 =
      clip_from_center(self.to_obj,self.from_obj)
for each arrow in node.out_arrows
    arrow.endpoint1 =
      clip_from_center(self.from_obj,self.to_obj)
```

`in_arrows` and `out_arrows` are lists of arrows that enter and leave a node.

By using iteration, the handcoded applications were able to use only one copy of the constraints, eliminate dynamic scheduling and function calls, and decrease storage.

## 5 ANALYSIS

The hand-crafted and toolkit-crafted applications described in Section 3 were benchmarked on an HP720 that contained 32 megabytes of memory. Tables 1-3 show representative results of these benchmarks. These benchmarks were performed on 1) an 8x8 Othello gameboard, 2) an arithmetic editor consisting of approximately 100 data nodes and 100 operators, and 3) a selection sort animation involving 10 elements. It is clear from these tables that significant improvements in performance can be obtained by applying the optimizations described in the previous section.

The most impressive performance improvements are

| Optimizations | Operation | Garnet Update Time (msec) | Hand-Crafted Update Time (msec) | Performance Improvement |
|---|---|---|---|---|
| Geographic Locality, Nonoverlapping Bounding Boxes | Animated Swapping in Selection Sort | 7.9 | 0.2 | 39.5 |
| Grid | Creating Arrows in Arithmetic Editor | 55 | 2.0 | 27.5 |
| Static Update Plan, Redraw only Altered Objects | Create Othello Game Pieces | 66 | 3.3 | 20.0 |
| Redraw only Altered Objects | Create Nodes in Arithmetic Editor | 16 | 1.7 | 9.4 |
| Geographic Locality, Grid | Move Nodes in Arithmetic Editor | 27 | 4.8 | 5.6 |
| Static Update Plan | Manipulate Othello Scrollbar | 20 | 3.9 | 5.1 |
| Redraw only Altered Objects, Erasing by Redrawing | Highlight Selection Sort Elements | 5.5 | 1.6 | 3.4 |
| Grid | Edit Data Nodes in Arithmetic Editor | 198 | 60 | 3.3 |

Table 1: Benchmark times in milliseconds for updating the display after performing various operations. The update times are the average time required to update the display after one operation (e.g., after adding one data node in the arithmetic editor or incrementally moving two labeled rectangles along their respective paths in the selection sort animation). The optimizations column lists the optimizations that were used in the hand-crafted versions.

| Optimizations | Operation | Garnet Constraint Solving Time (msec) | Hand-Crafted Constraint Solving Time (msec) | Performance Improvement |
|---|---|---|---|---|
| Constraint Plan | Highlight Selection Sort Elements | 0.5 | 0.0 | - - |
| Constraint Plan | Create Nodes in Arithmetic Editor | 5.9 | 0.3 | 19.7 |
| Constraint Plan | Create Othello Game Pieces | 3.2 | 0.2 | 16.0 |
| Iteration, Constraint Plan | Create Arrows in Arithmetic Editor | 30 | 2.1 | 14.2 |
| Constraint Plan | Animated Swapping in Selection Sort | 1.0 | 0.1 | 10.0 |
| Iteration, Constraint Plan | Edit Data Nodes in Arithmetic Editor | 438 | 78 | 5.6 |
| Iteration, Constraint Plan | Move Nodes in Arithmetic Editor | 12.7 | 3.1 | 4.1 |
| Constraint Plan | Manipulate Othello Scrollbar | 3 | 0.8 | 3.8 |

Table 2: Benchmark times in milliseconds for resatisfying constraints after performing various operations. The times represent the average time required to satisfy the constraints after one operation (e.g., after adding one data node in the arithmetic editor or incrementally moving two labeled rectangles along their respective paths in the selection sort animation). The optimizations column lists the optimizations that were used in the hand-crafted versions.

storage improvements. These gains are somewhat greater than the gains that would be realized from applying glyph and composite object folding optimization to other toolkits, since Garnet maintains additional information in each object that is useful for prototyping tools. For example, each object contains a list of variables that may be set by a user and which may be used by property sheets. However, the storage improvements cannot be explained by the prototyping overhead alone. Each Garnet object contains a significant amount of information that is used by the automatic redisplay algorithm, such as a list of old values for variables that affect the object's appearance. None of this display information is maintained by the hand-crafted objects since the information can be efficiently computed given semantic information about the application's operations. Similarly, glyph optimization eliminates many of the appearance attributes of objects, such as their position and size, since these attributes can be efficiently computed immediately before redisplay. Composite object folding completely eliminates the storage overhead associated with maintaining forward pointers to each of an aggregate's children and backpointers from each of the children to the parent. It also eliminates duplicate information, such as a string or filling-style, that is copied to the children. Finally, the use of constraint plans and constant propagation completely eliminates the overhead of storing constraints and their dependencies.

| Optimizations | Object | Garnet Size (bytes) | Hand-Crafted Size (bytes) | Storage Improvement |
|---|---|---|---|---|
| Composite Object Folding, Glyph Optimization | Labeled Rectangle | 4188 | 24 | 174 |
| Glyph Optimization | Othello Menu | 23488 | 160 | 147 |
| Composite Object Folding, Glyph Optimization | Data Node in Arithmetic Editor | 4668 | 44 | 106 |
| Composite Object Folding, Glyph Optimization | Arrowline in Arithmetic Editor | 7276 | 84 | 87 |
| Glyph Optimization | Othello Scroll Bar | 19792 | 280 | 71 |
| Glyph Optimization | Othello Game Piece | 836 | 28 | 30 |
| Glyph Optimization | Othello Square | 696 | 28 | 25 |

Table 3: Number of bytes occupied by various objects in the example applications. The optimizations column lists the optimizations that were used to reduce the storage requirements of objects in the hand-crafted versions.

The update optimizations achieve the most significant performance improvements, since studies show that redisplay typically consumes 60-80% of an interface's time [18, 16, 9]. The performance figures in Tables 1 and 2 reinforce these findings. An interesting discovery that resulted from our implementations of hand-crafted applications was that a large number of redisplay operations can be implemented simply by redrawing only altered objects. Even when objects may overlap, as in the arithmetic editor or the Othello game, there are operations that draw objects on top of all other objects, and thus cause no damage to the display. Geographic locality was also instrumental in improving the performance of ''incremental'' operations, such as repositioning a data node or moving objects along a path.

Indeed, geographic locality and redrawing only altered objects were the dominant optimizations in two of the three applications—the selection sort animation and Othello. The operations of each of these applications was such that no complicated data structures had to be maintained to improve the performance of the display algorithm. Only the arithmetic editor required such a data structure, and even in this application, geographic locality and redrawing only altered objects were effective strategies. The grid data structure did improve the performance of the arithmetic editor, but it was the most difficult optimization to implement and debug. In contrast, the geographic locality and redraw of only altered objects optimizations were relatively easy to implement.

The constraint optimizations also provided impressive performance improvements. However, since constraint solving was typically a relatively small fraction of the update time, these optimizations were not as critical as the update optimizations. Their most important contribution was in reducing the storage size of the hand-crafted applications, as discussed above. The constraint optimizations were also fairly difficult to implement, since the constraint networks often propagated values to many different parts of the application (e.g., in the Othello application a new game piece affected constraints in the scoreboard, message box, and gameboard). Keeping track of how these constraints interacted, and in what order they should be evaluated, was a time-consuming, burdensome task.

## 6 COMPILER REQUIREMENTS

The analysis in the previous section suggests that the optimizations described in Section 4 can significantly improve the performance of an interactive, graphical interface. However, they are also very burdensome and difficult to implement, and, ideally, toolkits would provide compilers that would automatically implement them. All of these optimizations require semantic information about the application, so the compiler would have to have some means for obtaining this information. Semantic information may be either global, applying to all operations, or local, applying to only individual operations. Information about which attributes are constant in an application is global information that applies to all operations. Information that objects do not overlap may be global, but often is local, applying to only a few operations.

The optimizations that have been previously described in the literature rely on global information: constant propagation depends on knowing which attributes are constant, constraint plans for objects rely on knowing which of the objects' attributes are possible inputs and which attributes are possible outputs, and glyphs require knowledge about all operations in order to determine which information should be factored out and which information should be stored in the glyphs.

However, the optimizations that have been discussed in this paper require local information about each operation. Update plans require information about which objects an operation modifies, what regions of the screen these objects inhabit, and what kinds of objects these objects may overlap. Constraint plans require information about which attributes an operation modifies so that the constraints that depend directly or indirectly on these attributes can be scheduled. Converting constraints to iteration requires knowledge about the constraints involved in the iteration and information about how to access the objects that contain these constraints (the objects that contain these constraints are typically related to the objects modified by the operation—for example, the arrows that move in the arithmetic editor are attached to the object that is moved or resized).

A plausible explanation for the absence of these optimizations from toolkits is that local information about operations is much more difficult to obtain than global information about an application. The most desirable approach would be to statically analyze the code, as traditional compilers do, and extract the semantic information in this fashion. Unfortunately extracting information in this fashion is not practical. Much of the information needed to optimize an interactive graphical application is obscured by the conditional execution represented by event handlers and constraints. Control is distributed among various event handlers, and users can invoke the event handlers in arbitrary sequences that cannot be predicted statically. This arbitrary invocation of event handlers creates arbitrary networks of constraints, whose order of execution also cannot be predicted statically. Consequently, much of the semantic information employed in optimizing an interactive program—for instance, the attributes that cannot change value or the objects that will be modified or damaged by an operation—is not available by merely analyzing the code.

Another approach is to ask the programmer to explicitly provide semantic information on a per operation basis, much as a programmer explicitly provides information for other toolkit optimizations, such as constant propagation and constraint plans. However, this approach has a number of drawbacks, including:

1. Missed Optimizations: The amount of information that must be explicitly provided can be quite extensive and the programmer may omit important information. Indeed, our experience with Garnet optimizations shows that a significant number of opportunities for optimization are overlooked.

2. Ease of Use: The purpose of toolkits is to allow developers to think about and write code at a high-level. Programmers should be required neither to program at a low-level nor to possess familiarity with the available optimizations to obtain greater efficiency. This work should be performed by compilers. However, requiring the programmers to explicitly provide semantic information places this burden onto the programmer.

A third approach is to use profile-based compilation. In this approach, a programmer operates an application while the compiler observes changes in the internal execution state of the application. As the compiler detects patterns among the changes, it will use various forms of pictorial feedback to query the programmer about the patterns it has inferred. Based on these inferences and the feedback from the programmer, the compiler will generate optimized plans for dynamically executing the application.

This approach appears promising for a number of reasons:

1. Target Applications: The applications generated by the toolkits are highly interactive. Thus it should not be a great burden on a programer to demonstrate the operation of the application on example data.

2. Recognition versus Recall: Demonstrational techniques are based on recognition memory rather than recall memory. When using demonstrational techniques, the programmer simply decides whether information is correct (recognition). When explicitly specifying the information, the programmer must generate the information before deciding whether or not the information is correct (recall).

3. Success in Other Areas: Profile-based compilation has proven successful in high performance computing. Thus there is evidence that acquiring semantic information about an application by executing the application may prove successful.

In the future, we plan to develop demonstrational, profile-based compilation techniques for interactive applications that will attempt to apply the optimizations that we have identified in the handcoded applications. We also plan to develop similar techniques for optimizations that have been described in the literature, but which currently require explicit semantic information from a programmer, or a greater effort in programming than might be desirable. For example, by observing the execution state of an application, we hope to be able to acquire enough information to perform constant propagation for constraints, and to convert heavyweight structured graphics objects to lightweight, Interviews-style glyphs.

## 7 CONCLUSIONS

In this paper we have identified a number of optimizations that are commonly found in highly interactive, handcoded graphical applications, but that are not utilized by toolkit-generated applications. These optimizations include: 1) the creation of update plans that statically predetermine the objects directly modified or damaged by an operation; 2) the use of semantic information about an operation, such as the region or regions it affects to minimize the number of objects that must be dynamically checked for damage; 3) the conversion of constraints to iteration; and 4) the creation of constraint plans that statically schedule the constraints affected by an operation. This paper has also presented experimental results that suggest that these optimizations can significantly improve the performance and storage requirements of interactive, graphical applications. Unfortunately, these optimizations require semantic information about individual operations that can be difficult to obtain from a programmer. The difficulty of obtaining this information may explain the absence of these optimizations in toolkits. However, since the evidence presented in this paper suggests that these optimizations can have a significant impact on the performance of an application, devising ways to obtain this information is an important area for future research.

## REFERENCES


**1.** Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

**2.** Alan Borning. *ThingLab--A Constraint-Oriented Simulation Laboratory*. Ph.D. Th., Department of Computer Science, Stanford University, 1979.

**3.** Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

**4.** Marc H. Brown. "Exploring algorithms using Balsa-II". *IEEE Computer 21*, 5 (May 1988), 14-36.

**5.** Paul R. Calder and Mark A. Linton. Glyphs: Flyweight Objects for User Interfaces. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 92-101.

**6.** Bjorn N. Freeman-Benson. "A Module Mechanism for Constraints in Smalltalk". *Sigplan Notices 24*, 10 (Oct. 1989), 389-396. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

**7.** Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.

**8.** Ralph D. Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In Brad A. Myers, Ed., *Languages for Developing User Interfaces*, Jones and Bartlett Publishers, Boston, MA, 1992, pp. 125-143.

**9.** Ralph D. Hill. The Rendezvous Constraint Maintenance System. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93, Atlanta, GA, Nov., 1993, pp. 225-234.

**10.** Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Tech. Rept. 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April, 1993.

**11.** David Kosbie, Brad Vander Zanden, Brad A. Myers, and Dario Giuse. Automatic Graphical Output Management. In Brad A. Myers, Ed., *The Garnet Compendium: Collected Papers, 1989-1990*, Carnegie Mellon University Computer Science Department, Technical Report CMU-CS-90-154, 1990.

**12.** John Maloney, Alan Borning, and Bjorn Freeman-Benson. "Constraint Technology for User-Interface Construction in ThingLabII". *Sigplan Notices 24*, 10 (Oct. 1989), 381-388. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

**13.** John Harold Maloney. *Using Constraints for User Interface Construction*. Ph.D. Th., Department of Computer Science & Engineering, University of Washington, Seattle, Washington 98195, 1991.

**14.** Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

**15.** Brad Myers, Dario A. Giuse, Andrew Mickish, and David Kosbie. Making Structured Graphics and Constraints Practical for Large-Scale Applications. Tech. Rept. CMU-CS-94-150, Carnegie Mellon University Computer Science Department, May, 1994.

**16.** Michael Sannella. The SkyBlue Constraint Solver. Tech. Rept. 92-07-02, Computer Science Department, University of Washington, July, 1992.

**17.** Brad T. Vander Zanden. *Incremental Constraint Satisfaction and Its Application to Graphical Interfaces*. Ph.D. Th., Cornell University, Ithaca, NY, 1988.

**18.** Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. Integrating Pointer Variables into One-Way Constraint Models. To Appear in ACM Transactions on Computer Human Interaction.