Hints on Making Garnet Programs Run Faster

Brad A. Myers

October 1993

Abstract

This chapter discusses some hints about how to make Garnet programs run faster. Most of these techniques should not be used until your programs are fully debugged and ready to be shipped.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Additional support for Garnet has been provided by NEC, Apple, Adobe, and General Electric.

1. Introduction

An important goal of Garnet has been to create a system that is as efficient as possible. For example, users should notice that version 2.2 is about two or three times faster than 2.1. Now that people are writing large-scale systems using Garnet, a number of things have been learned about how to make Garnet programs run faster. This chapter collects a number of hints about how to write efficient Garnet code. If you have ideas about how to make the underlying Garnet system run faster, or new hints to add to this section, please let us know.

The ideas in this chapter are aimed at producing the final production version of your system. Therefore, we feel that you should not worry about the comments here during early development. For example, turning off the debugging and testing information is likely to make your development more difficult. Also, declaring constants makes changing code more difficult. Generally, you should get your system to a fairly well-debugged state before applying these ideas.

Of course, the easiest way to make Garnet run faster is to get a faster machine and/or more physical memory. With SPARC IIs and HP Snakes becoming more prevalent, and 100 mip machines like the DEC Alpha around the corner, we see expect that the next generation of applications will have much less of a problem with achieving adequate performance.

2. General

Ideas in this section are relevant to any code written in Lisp, not just Garnet code. Some of these may seem obvious, but we have seen code that violates many of them.

- Be sure to compile all your files.
- The variable `user::*default-garnet-proclaim*`, which is defined in `garnet-loader.lisp`, provides some default compiler optimization values for Allegro, Lucid, CMU, and LispWorks lisp implementations. The default gives you fast compiled code with verbose debugging help. You can `setf` this variable before loading (and compiling) Garnet to override the default proclamations, if you want to sacrifice debugging help for speed:

```
(PROCLAIM '(OPTIMIZE (SPEED 3) (SAFETY 0) (SPACE 0)
                     (COMPILATION-SPEED 0)))
```

- Fundamental changes in underlying algorithms will often overcome any local tweaking of code. For example, changing an algorithm that searches all the objects to one that has a pointer or a hash table to the specific object can make an application practical for large numbers of objects.
- Use a fast Lisp system. We have found that Allegro Version 4.0 is much faster than Allegro V3.x. Also, Allegro and Lucid are much faster than KCL and AKCL on Unix machines.
- Most systems have specialized commands and features for making smaller and faster systems. For example, if you are using Allegro, check out PRESTO, which tries to make the run-time image smaller. One user reported that the "reorganizer" supplied with Lucid, the CPU time used decreased about 10-20%, and the overall time for execution dropped by about 30%. We have found that the tracing tools supplied by vendors to find where code is spending its time are mostly worthless, however.
- Beware of Lisp code which causes CONS'ing. Quite often, the most natural way to write Lisp code is the one that creates a lot of intermediate storage. Unfortunately, this may result in severe performance problems, as allocating and garbage-collecting storage is among the slowest operations in Lisp. The recommendations below apply to all of your code in general,

but in particular to code that may be executed often (such as the code in certain formulas which need to be recomputed many times).

- As a rule, mapping operations (like `mapcar`) generate garbage in most Lisp implementations, because they create temporary (or permanent) lists of results. Most mapping operations can be rewritten easily in terms of DO, DOLIST, or DOTIMES.
- Handling large numbers of objects with lists is generally expensive. If you have lists of more than a few tens of objects, you should consider using arrays instead. Arrays are just as convenient as lists, and they require much less storage. If your application needs variable numbers of objects, consider using variable-length arrays (possibly with fill pointers).
- Declare the types of your variables and functions (using DECLARE and PROCLAIM).
- Some Lisp applications will give you warnings or notes about Lisp constructs that are potentially inefficient. In CMU Common Lisp, for example, setting SPEED to 3 and COMPILATION-SPEED to 0 generates a number of messages about potentially inefficient constructs. Many such inefficiencies can be eliminated easily, for example by adding declarations to your code.
- Wrap all lambdas in #' rather than just ' (in CLtL2 the # is no longer optional). This comes up in Garnet a lot in final-functions for interactors and selection-functions for gadgets. Note, in the `:parts` or `:interactors` parts of aggregadgets or aggrelists, use `,#'` (comma-number-quote) before lambdas and functions.
- You can save an enormous amount of time loading software if you make images of lisp with the software already loaded. For example, if you start lisp and load Garnet, you can save an image of lisp that can be restarted later with Garnet already loaded. We have simplified this procedure by providing the function `opal:make-image`. If you want to make images by hand, you will have to use `opal:disconnect-garnet` and `opal:reconnect-garnet` to sever and restore lisp's connection with the X server. All of these functions are documented in the Opal manual.
- It may help to reboot your workstation every now and then. This will reset the swap file so that large applications (like Garnet) run faster.

3. Making your Garnet Code Faster

This section contains hints specifically about how to make Garnet code faster.

- The global switch `:garnet-debug` can be removed from the `*features*` list to cause all the debugging and demo code in Garnet to be ignored during compiling and loading. This will make Garnet slightly smaller and faster. The `:garnet-debug` keyword is pushed onto the `*features*` list by default in `garnet-loader.lisp`, but you can prevent this by setting `user::Garnet-Garnet-Debug` to NIL before compiling and loading Garnet. Garnet will need to be recompiled with the new `*features*` list, so that the extra code will not even get into the compiled binaries. Of course, you will lose functions like `inter:trace-inter`.
- Turn off KR's type-checking by setting the variable `kr::*types-enabled*` to NIL. Note: the speed difference may be imperceptible, since the type system has been implemented very efficiently (operations are only about 2% slower with type-checking).
- If you have many objects in a window, and an interactor only works on a small set of those objects, then the small set of objects should be in their own aggregate or subwindow. This will cause Opal's `point-in-gob` methods run faster, which identify the object that you clicked on. When objects are arranged in an orderly aggregate hierarchy, then the

`point-in-gob` methods can reject entire groups of objects, without checking each one separately, by checking whether a point is inside their *aggregate*'s bounding box. For example, in `demo-motif` the scroll bars are in their own aggregate. Putting objects in a seperate subwindow is even faster, since the coordinates of the click will only be checked against objects in the same window as the click.

- Use `o-formula`s instead of `formula`s. O-formulas are compiled along with the rest of the file, whereas formulas are compiled at load- or run-time, which is much slower.
- Try not to use formulas where not really needed. For example, if the positions of objects won't change, use expressions or numbers instead of formulas to calculate them.
- Try to eliminate as many interactors as possible. Garnet must linearly search through all interactors in each window. To see how many interactors are on your window, you can use `(inter:print-inter-levels)`. If this is a long list, then try to use one global interactor with a start-where that includes lots of objects, rather than having each object have its own interactor. This can even work if you have a lot of scattered gadgets. For example, if you have a lot of buttons, you can use a button-panel and override the default layout to individually place each button.
- The `fast-redraw` property of graphical objects can be set to make objects move and draw faster. This can be used in more cases than with previous versions of Garnet, but it is still restricted. See the fast-redraw section of the Opal manual.
- Aggrelists are quite general, and have a lot of flexibility. If you don't need this flexibility, for example, if your objects will always be in a simple left-aligned column, it will be more efficient to place the objects yourself, or create custom formulas.
- If you are frequently destroying and creating new objects of the same type, it is more efficient to just keep a list of objects around, and re-using them. Allocating memory in Lisp is fairly expensive.
- If you are deleting a number of objects at the same time, first set the window's `:aggregate` slot to NIL and update the window. Then, when you are done destroying, set the aggregate back and update again. For example, to destroy 220 rectangles on a Sparc, removing the aggregate reduced the time from 11.8 to 2.4 seconds (80%)! So your new code should be:

```
;; Code fragment to quickly destroy all the objects within an aggregate.
(let ((temp-agg (kr:gv my-window :aggregate)))
  (when temp-agg
    ;; First, temporarily remove the aggregate:
    (kr:s-value my-window :aggregate NIL)
    (opal:update my-window)
    ;; Now do the actual destroying:
    (dolist (object (kr:get-values temp-agg :components))
      (opal:destroy object))
    ;; Finally, restore the aggregate:
    (kr:s-value my-window :aggregate temp-agg)
    (opal:update my-window)))
```

- If you have objects in different parts of the same window changing at the same time, it is often faster to call update explicitly after one is changed and before the other. (This is only true if neither of the objects is a fast-redraw object. Many of the built-in gadgets are fast redraw objects for this reason, so this usually is not necessary for built-in gadgets.) The reason for this problem is that Garnet will redraw everything in a bounding box which includes all the changed objects. If the changed objects are in different parts of a window, then everything in between will be redrawn also. Ways around this problem include calling update explicitly after one of the objects changes, making one of the objects be a fast redraw object if possible, moving the objects closer together if possible (so there aren't objects in between), or putting the objects in separate subwindows if possible (subwindows are updated independently).

- Conventional object-oriented programming relies heavily on message sending. In Garnet, however, this technique is often less efficient than the preferred Garnet programming style, which relies on slots and constraints. Rather than writing methods to get values from certain slots in an object, for example, consider accessing those slots directly and having a formula compute their value. The Garnet style is more efficient, since it avoid the message-sending overhead. Because Garnet provides a powerful constraint mechanism, the functionality that would normally be associated with a method can typically be implemented in a formula.

- If you use the same formula in multiple places, it is more efficient to declare a formula prototype, and create instances of it. For example:

```
(defparameter leftform (o-formula (+ 10 (first (gvl :box)))))
;;for every object
(create-instance NIL <whatever>
                 ...
                 (:left (formula leftform)))
```

- If many objects in your scene have their own feedback objects, maybe you can replace these with one global feedback object instead. The button and menu interactors can take a `:final-feedback-obj` parameter and will duplicate the feedback object if necessary.

- If you have a lot of objects that become invisible and stay invisible for a reasonable period if time, it might be better to remove them from their aggregate rather than just setting their `:visible` slot. There are many linear searches in Garnet that process all objects in an aggregate, and each time it must check to see if the objects are invisible.

- It is slightly more efficient when you are creating a window at startup, if you add all the objects to the top level aggregate *before* you add the aggregate to the window.

- The use of double-buffering doesn't make your applications run faster (they actually run a little slower), but it usually *appears* faster due to the lack of flicker. See the section in the Opal manual on how to make a window be double-buffered.

4. Making your Binaries Smaller

This section discusses ways to make the run-time size of your application smaller. This is important because when your system gets big, it can start to swap, which significantly degrades performance. We have found that many applications would be fast enough if they all fit into physical memory, whereas when they begin swapping virtual memory, they are not fast enough.

- Don't load the PostScript module or debugging code unless you need to. Change the values of the appropriate variables in `garnet-loader`, or set the variables before loading Garnet. The values will not be overridden, since they are defined with `defvar` in `garnet-loader`.

- Declare constants where possible. This allows Garnet to throw away formulas, which saves a lot of run-time space. All the built-in objects and gadgets provide a `:maybe-constant` slot, which means that you can use `(:constant T)` to make all the slots constant. The `:maybe-constant` will contain all of the slots discussed in the manual as parameters to the object or gadget. Of course, the slots that allow the widget to operate (e.g., the buttons to be pressed or the scroll-bar-indicator to move) are not declared constant. Remember that only slots that don't change can be declared constant. Therefore, if your gadget changes position or items or active or font after creation, then you should `:except` the appropriate slots. For example:

```
(create-instance NIL gg:motif-radio-button-panel
  ;; only the :active slot will change
  (:constant '(T :except :active))
  (:left 10)(:top 30)
  (:items '("Start" "Pause" "Quit")))
```

Several functions are discussed in the Debugging Manual (starting on page 469) that are very helpful in determining which slots should be declared constant. The KR Manual describes the fundamentals of constant declarations in detail.

- Don't load gadget files you don't need. Most Garnet applications (like the demos), load only the gadgets they need, if they haven't been loaded already. This approach means that lots of gadgets you never use won't take up memory.
- Consider using `virtual-aggregates` if you have a lot of similar objects in an interface, such as lines in a map or dots on a graph. This will decrease storage requirements significantly.
- The variable `kr::store-lambdas` can be set to NIL to remove the storage of the lambda expressions for compiled formulas. This will save some storage, but it prevents objects from being stored to files.

Table of Contents