Building Common Lisp Applications with Reasonable Performance*

John Boreczky
Lawrence A. Rowe

Computer Science Division-EECS
University of California
Berkeley, CA 94720

Abstract

This paper describes our experience with a number of methods that can be used to improve the performance of Common Lisp applications. These methods include vendor-independent optimizations such as recoding high traffic functions and vendor-dependent optimizations such as incremental loading. The results of these optimizations are quantified and compared.

1. Introduction

Over the past several years we have developed several applications in Common Lisp (CL) [Ste90] including a graphical user interface toolkit [RKS91], an end-user browser for a CIM database [SmR92], and a hypermedia system and courseware [BeR91,ScR92a]. It is well-known that CL is an excellent prototyping environment, however, the CPU and memory resources required by CL applications have prevented them from being used in a production environment [Gab91, LaR91]. CL vendors have invested significant resources to produce compiler improvements and tools to reduce the resources required.

This paper describes the use of tools produced by two CL vendors, namely Franz Inc. (Allegro CL [FrI91a]) and Lucid Inc. (Lucid CL [LuI90]), to productize a moderately complex semiconductor manufacturing application. A variety of different optimizations, including vendor-independent and vendor-dependent optimizations, were applied to improve resource utilization. The results of these improvements are quantified and compared.

The sample application used in this experiment is CIMTOOL [SmR92] which provides a graphical user interface (GUI) to a semiconductor manufacturing CIM database that contains data about the fabrication facility (e.g., floor plan, equipment, utility lines, etc.), equipment (e.g., status, trouble reports, processing history, etc.), and work-in-progress (e.g., lot status and history). It also displays data captured from real-time sensors and images and videos that describe the facility, equipment, and its operation. Figure 1 shows a screen dump with CIMTOOL running that displays the floorplan, an equipment image, a query selection window, and status windows for equipment and utility lines.

CIMTOOL is written almost entirely in CL using the Picasso GUI development environment also produced at Berkeley [RKS91, KoR91]. Picasso provides an application framework and an X Window System [Sch91] interface toolkit. CIMTOOL and Picasso make extensive use of the Common Lisp Object System (CLOS) [Kee88].

---

* This research was supported by the National Science Foundation (Grant MIP-90-14940) and the Semiconductor Research Corporation, Texas Instruments, and National Semiconductor, with a matching grant from the State of California's MICRO program. This material is based in part upon work supported by the National Science Foundation under Infrastructure Grant No. CDA-8722788.

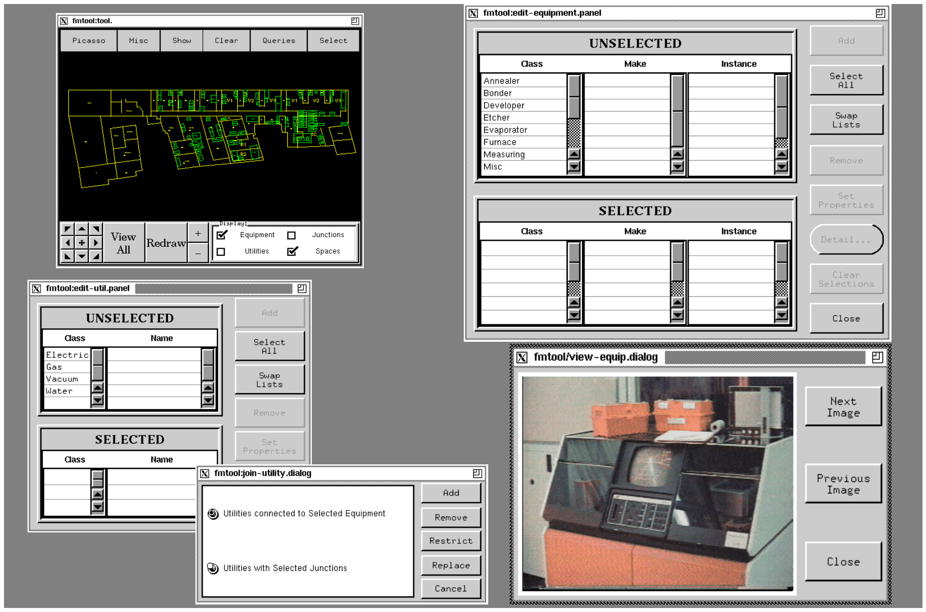

Figure 1: CIMTOOL screendump

The vendor-independent optimizations that we applied to the code included: 1) recoding high traffic functions, 2) compiling the CLOS method dispatch code, and 3) implementing a dump system to reduce interface and application object creation. The vendor-specific optimizations that we applied included: 1) dynamic loading of compiled functions supported by Allegro CL and 2) reducing the working set size supported by Lucid CL. Applying these optimizations reduced startup time by 50% and execution time by 25% on a representative task. Memory requirements increased slightly for Lucid CL because we were not able to use their Treeshaker technology to remove unused code for reasons discussed below. Memory requirements were reduced 20% for Allegro CL. However, in the best case, CIMTOOL required 19 MB of memory which is impractical for deployment to a user community that might run 10 concurrent copies of the application on X terminals connected to a shared server.

The remainder of this paper describes the experiment in more detail. It is organized as follows. Section 2 describes CIMTOOL, the Lisp systems, and a baseline benchmark before optimization. Section 3 describes the vendor independent optimizations, and Section 4 describes the vendor specific optimizations. Sections 3 and 4 also describe the effects of performing these optimizations. Section 5 discusses our experiences performing this experiment. Section 6 concludes the paper.

2. Baseline Application

Picasso is a GUI development system that provides an application framework and an interface toolkit. The application framework provides high-level abstractions such as dialog boxes, panels for interacting with data, frames for presenting the top level application interface, and tools for collecting the various interface abstractions into a coherent application. The interface toolkit provides an extensive collection of widgets including: buttons, scrollbars, text editors, graphic browsers, tables and video widgets. Picasso includes a simple constraint system to bind variables to other variables or interface objects, several parameter passing/variable passing options, and a large assortment of geometry managers.

The design and implementation of Picasso required over 10 man-years of work over a 5 year period by experienced programmers who had limited experience with Lisp when they started. The framework and toolkit contain roughly 35000 lines of code, while a typical Picasso application requires a few thousand lines of code (e.g., CIMTOOL is about 3000). While writing Picasso, the emphasis was on rapid development and high functionality with a hope that execution speed would be reasonable. The desire to use Picasso to deploy multimedia and semiconductor manufacturing applications in the Berkeley Microfabrication Laboratory

[VoK89] required us to improve performance which led to the work on attach/detach and the other optimizations described in this paper.

A typical Picasso application is composed of several Lisp files that describe the interface objects (e.g., non-modal panels and modal dialog boxes) and files that contain support functions for the application. The support functions are usually compiled and loaded with the application. Each interface description file contains one or more macro calls that create Picasso interface objects which are represented by CLOS objects. The constraint mechanism is used to propagate changes to a program variable to other variables or to the display. The Lisp compiler is used to compile the constraint code.

CIMTOOL provides a GUI interface to specify ad hoc queries to a database. The main memory data structure for the facility floorplan including walls and icons to represent equipment requires about 40 KB of memory. Picasso interface objects are created as needed for windows to display information.

The benchmark task used for performance measurement consists of 35 actions that involved selecting data items using a mouse, performing queries on those items, and viewing data graphically and textually. This task was a representative usage of the application, although images and video were not displayed.

Two commercial Common Lisp implementations were used: Allegro CL 4.0.1 and Lucid CL 4.0.1+. All tests were run on a SPARCstation 1+ with 40 megabytes of physical memory and 55 megabytes of swap space running Sun OS 4.1.1. Relatively little paging was done because of the physical memory. Attempts to run any version of CIMTOOL, including those heavily optimized, on machines with less physical memory (e.g., 24 MB) were much too slow. Since CLX 11R5 fixed a number of bugs under Lucid CL but did not change performance, our tests were run with CLX 11R4 in Allegro CL and CLX 11R5 in Lucid CL. All Lisp source files were compiled with optimization settings of safety 1, space 1, debug 1, speed 2, and compilation-speed 0. All executable images and the CIMTOOL files were NFS mounted across a local Ethernet.

Allegro CL partitions dynamic memory into old space and new space. Text space is a third area of memory used to hold static data. New space is divided into two equal-sized pieces, called *semi-spaces*, to allow stop-and-copy garbage collection. New space is used to store newly created objects. When an object in new space has survived a fixed number of garbage collections, called *generations*, it is placed in old space (i.e., tenured). We created our Allegro images with 14 MB of old space and 11 MB of new space (5.5 MB per semi-space). This space is in the virtual address space for the image but not necessarily in the working set when an application is run. This amount of memory is more than is necessary, but it reduces the number of garbage collections significantly. Since an object in new space is processed multiple times before it is tenured, reducing the number of garbage collections improves performance, even though each garbage collection takes longer.

The Lucid memory organization is more complicated and more flexible. Generational garbage collection is implemented with a hierarchy of dynamic spaces called *ephemeral levels*. Objects that survive a garbage collection are moved into the next ephemeral level. There are separate areas for static data, temporary data, and foreign data in addition to the ephemeral levels and the two semi-spaces for dynamic objects. Ephemeral level 0 is where new objects are created. When it is filled, a garbage collection is done and any surviving objects are moved to ephemeral level 1. In the default setup, level 0 is 512 KB, and levels 1 and 2 are 640 KB. Objects that survive level 2 move to the current dynamic semi-space, which is expanded as needed. Consequently, Lucid CL garbage collection results in shorter but more frequent garbage collections than Allegro CL. It is possible to select the number of ephemeral levels and the size of the memory areas to tune the memory setup for the needs of an application. We used an image with 24 MB of total space for Lucid CL.

Given the substantial differences between the memory organization of these two implementations and the ability to add memory to an image, it is difficult to determine the size of a running program. The method used in this paper is to determine the process size from the UNIX process status command (*ps*) and to subtract the unused portion of old space for Allegro CL and dynamic space for Lucid CL. The new/ephemeral spaces were untouched because changing their sizes would affect performance. Thus, our reported execution size is the run time size of the smallest images that could run with the reported performance. It would be unrealistic to run these images because there is no room to tenure objects, but they serve as a useful means of comparison.

We rejected other possible measures of process size, such as number of bytes consed, number of page faults, and resident set size, because they did not reflect the real memory required to run the application.

The load times reported are the elapsed time required to load the application from disk and initialize it. Execution time is the time elapsed from invoking CIMTOOL until the Lisp prompt returns after completing the test and quitting the CIMTOOL application. All times are reported to the nearest second. We report elapsed times rather than CPU times because we want to measure user's perceived performance of the application. In addition, the application is not CPU bound, it is memory and I/O bound, and CPUs are getting faster more rapidly than memory and I/O bandwidth are increasing.

To provide a sense of perspective and establish a baseline performance, Tables 1 and 2 show performance measurements of the base CL implementations and the effect of adding Picasso and CIMTOOL code. The load times for CIMTOOL are large because of the time needed to build the main memory database. We report both disk size and execution size of the application although users typically care more about execution size. Although specific details are different between the two implementations, overall performance is quite similar.

Optimization	Load Time (sec)	Disk Size (KB)	Execute Size (KB)
Allegro CL + CLX + CLOS	16	5672	16347
+ Picasso	49	11616	22349
+ CIMTOOL	158	11602	24132

Table 1: Allegro Common Lisp Baseline Performance

Optimization	Load Time (sec)	Disk Size (KB)	Execute Size (KB)
Lucid CL + CLX + CLOS	19	10048	15432
+ Picasso	40	14656	22096
+ CIMTOOL	136	14648	26340

Table 2: Lucid Common Lisp Baseline Performance

3. Vendor Independent Optimizations

Many optimizations can be applied to improve the performance of a CL application. Some optimizations involve making improvements to the source code of an application and are thus implementation independent, even though vendors might provide tools to aid in the process. This section describes three such optimizations and the resulting performance improvements.

3.1 Time and Space Profiling and Type Declarations

Profiling is used to provide a rough estimate of the time and space used by individual functions. Functions that require more time and space than seem necessary or functions that are called very often can be rewritten using a better implementation or algorithm. Both Lisp implementations provide facilities to count function calls and estimate time and space usage.

The Lucid profiler provides data in a number of formats. We examined profile data for both initialization and execution. General internal functions such as LRUN were the largest time consumers, but some standard functions were also large time consumers. We then used the Performance Monitor to gather more detailed and accurate statistics on these standard functions.

We examined all functions that accounted for more than 0.5% of the initialization time or more than 0.5% of the execution time, which included more than 50 functions. The percentage of time spent in the standard functions during loading/initialization and execution are shown in Tables 3 and 4, respectively. The sequence and arithmetic functions indicate that type declarations should improve performance. For the other functions, it is easy to see and accept why a large amount of time is spent there. The information did tell us that we might achieve a 5 to 10% performance improvement from careful type declarations, but it did not suggest a specific course of action.

Function	% of load time
NTHCDR	3.88
GETHASH	3.52
READ-CHAR	1.70
MEMBER	1.21
LIST-NREVERSE	1.17
DIRECTORY-NAMESTRING	1.14
SUBTYPEP	0.82
TYPEP	0.60
STRING=	0.55
ASSOC	0.51

Table 3: Percentage of Load Time Spent in Standard CL Functions

Function	% of execution time
GETHASH	3.96
READ-CHAR	1.14
LIST-NREVERSE	1.06
EQUAL	0.89
DIRECTORY-NAMESTRING	0.81
TYPEP	0.71
LISTEN	0.63
SUBTYPEP	0.60
POSITION	0.60
STRING=	0.57

Table 4: Percentage of Execution Time Spent in Standard CL Functions

We did not collect time and space profiles using Allegro CL. The profilers in version 4.0.1 and 4.1 Beta didn't work with our code, although the official 4.1 release fixed many of the problems. We have not pursued this analysis further, mainly because we doubt that significant improvement is possible since we already profiled with the Lucid CL tools.

Type declarations allow a Lisp compiler to produce more efficient code by reducing the need for run-time type checks and by reducing the overhead to dispatch functions. Both Lisp compilers can report the type information that is assumed or inferred while optimizing. When these assumptions are insufficient, type declarations allow the compiler to do a much better job of code optimization.

We began using declarations early in Picasso development expecting that the compilers and run-time environments would report type errors. Unfortunately, Lisp compilers at the time (late '86 and early '87) did not do type checking or make much use of type declarations. This deficiency discouraged us from using declarations since there was no immediate benefit. In retrospect we should have disciplined ourselves to use type declarations but we naively believed the Lisp religion that faster machines would solve all performance problems. The only type declarations currently in Picasso are `ignore` declarations in methods and `integer` declarations in a few time-critical methods.

We asked both compilers to provide suggestions for type declarations, and they produced a lot of output. Trimming away the useless information left us with two main types of declarations: declare numerical arguments and/or results to be fixnums and declare sequences to be simple arrays (the same recommendations we inferred from the profiling). The results of adding type declarations to the code are shown in Section 3.4.

3.2 Attach/Detach

Attaching is the process of creating the CLX data structures that correspond to a Picasso interface object, such as a window, and establishing a link between the object and the data structures. This ensures that the X server knows about and allocates the proper resources for the object and its children. Normally, Picasso objects are created and attached when they are needed, and they are implicitly detached when an application is exited. Detach methods allow Picasso objects (and their children) to be severed from the X server once they are initialized which means that Picasso applications can be initialized, detached, and dumped to disk for later use. Upon loading, these images require the Picasso objects to be attached, and the application is ready to run.

Using attach/detach resulted in substantial savings in startup time because the creation and initialization of Picasso objects is quite slow. This slowness is due to the large number of slots that need to be initialized for some objects, the large number of "call-next-method" calls to support inheritance for Picasso objects, and the naive use of CLOS by inexperienced programmers. Because the top-level Picasso tool has all of the other Picasso objects in an application as descendants, it is possible to perform attaches or detaches with one top-level method call.

3.3 CLOS method dispatch compilation

The final vendor independent optimization that we investigated was compilation of CLOS method dispatching. When a generic function is invoked for the first time or after a new method is defined, method dispatch code is created and compiled. This compilation requires the compiler to be present in the image, makes loading of files with many method definitions slow, and causes poor performance for the first call to a generic function. Both Lucid CL and Allegro CL provide facilities to compile the dispatch code from a running application into a file so that it can be loaded at startup the next time that application is run.

3.4 Optimization performance tests

The results of applying the optimizations described above alone and in combination to both Allegro CL and Lucid CL are shown in Tables 5 and 6, respectively.

Optimization	Time (% of unoptimized)		Disk Size (KB)	Execute Size (KB)
	Load	Execute		
No Optimizations	158 (100)	171 (100)	11602	24132
Profiling/Declarations	159 (101)	146 (85)	11570	24102
Attach/Detach	82 (52)	135 (79)	14026	24720
Method Compilation	153 (97)	153 (89)	11610	24028
Method + Attach	86 (54)	130 (76)	14034	24720
All 3 Combined	92 (58)	127 (74)	14242	24571

Table 5: Effect of Vendor Independent Optimizations on Allegro CL Application Performance

Optimization	Time (% of unoptimized)		Disk Size (KB)	Execute Size (KB)
	Load	Execute		
No Optimizations	136 (100)	157 (100)	14648	26340
Profiling/Declarations	137 (101)	143 (91)	14520	25956
Attach/Detach	74 (54)	126 (80)	16888	28324
Method Compilation	133 (98)	136 (87)	14714	25060
Method + Attach	81 (60)	116 (74)	16954	28324
All 3 Combined	83 (61)	129 (82)	16696	28260

Table 6: Effect of Vendor Independent Optimizations on Lucid CL Application Performance

As discussed above, we expected only a small performance improvement from the use of type declarations. Most sequences in Picasso are lists, and the few arrays we use are of variable size, so the sequence declarations provide no benefit. The fixnum declarations do help, but many functions where such declarations would have the greatest effect, such as geometry managers, use ratios or floats. We made the recommended declarations when possible, erring on the side of caution to avoid the introduction of new bugs. In general, type declarations resulted in a 15 to 25% reduction in size in the compiled code for functions that had a large number of declarations. Averaged across all of Picasso, however, the space reduction was small. The effect on execution time was also small. Elapsed time was reduced by 10 to 15%.

Attach/detach provided tremendous improvements in load time because most of the slow initialization of Picasso objects is precomputed. Much of the load time is due to disk I/O. The effect is especially dramatic for CIMTOOL because the facility database is loaded when the program is executed. Using attach/detach eliminates data loading. Attach/detach allows a user to stop a Picasso application in progress and restart it at a later time. When using Lucid Common Lisp, it is a good idea to garbage collect dynamic space before doing a dump, because the image dump process places all tenured objects into static space.

The effect of loading the compiled dispatch code upon startup is small. The extra overhead of loading the binary containing the compiled dispatch code is repaid by slightly faster initialization and execution, except when used with attach/detach, where there is not enough initialization performed to recover the cost.

4. Vendor Specific Optimizations

This section describes the tools provided by Franz and Lucid to productize CL applications.

4.1 Allegro Common Lisp

Allegro Presto [FrI91b] allows a Lisp image to be built that contains only the functions necessary for an application with the option of loading in other functions if needed. It does this dynamic loading by extending the concept of autoloading, which allows an entire compiled file to be loaded if one of its functions is called. This approach allows individual functions to be loaded as needed.

Allegro Presto allows a compiled binary to be libfasl loaded by creating a stub function in the image for each function in the compiled file. The code vectors and other information, such as argument lists, are loaded from the files on disk only when those functions are called. Thus, the compiled files serve as libraries of functions that can be loaded if and when they are needed. Consequently, it is possible to create a small Lisp system that has the standard CL functions libfasl loaded and build your application on this foundation.

Once an image is built with Allegro Presto, libfasl loaded binary files cannot by moved or modified, since the stubs refer to the code vectors by position within an absolute filename. Allowing these files to be referenced by relative pathname is necessary for delivering applications and it is reportedly under development.

Allegro Presto also allows you to specify the functions that will always be needed in an application and to place them in text space. A process similar to performance monitoring is used to record the functions that are loaded from stubs while the user exercises the application. A new Lisp image can then be built that contains these functions in text space. Placing the required functions in text space helps to improve locality of reference and allows code to be shared by multiple running images.

The results of applying Allegro Presto alone and in combination with some vendor independent optimizations to CIMTOOL are shown in Table 7. The "All Optimizations" case is a libfasl loaded image with code in text space, attach/detach, method compilation, and type declarations. Loading functions dynamically, done with libfasl and code in text space, reduces the memory required at the expense of longer load time. The disk size for these images is small because we dynamically load the 7 MB Lisp bundle file that contains the compiled code for the standard CL functions and optional packages.

Optimization	Time (% of unoptimized)		Disk Size (KB)	Execute Size (KB)
	Load	Execute		
No optimizations	158 (100)	171 (100)	11602	24132
Libfasl	162 (103)	154 (90)	6483	20547
Code in text space	186 (118)	147 (86)	8681	19225
Libfasl + Text + Attach	76 (48)	123 (72)	11289	19440
All Optimizations	77 (49)	126 (74)	11434	19230

Table 7: Effect of Allegro Presto on Common Lisp Application Performance

Many Lisp images must be created when using Allegro Presto. Starting from scratch, placing code in text space requires three images to be built: one in which to compile your application, the second to make a libfasl image of your application to determine which functions are loaded, and the third to place those functions in text space. Allegro Presto is also of limited use during application development, since it requires rebuilding the image when any of the libfasl loaded binaries change. The interface to Allegro Presto in Allegro CL 4.0.1 requires the user to be involved with the low level details of how images are built. The interface is slightly improved in Allegro CL 4.1.

4.2 Lucid Common Lisp

The Lucid Delivery Toolkit [LuI91] includes the Backtrace Logging Facility and the Performance Monitor. In addition, it includes the code Reorganizer and Treeshaker. We used the Beta version of the Treeshaker provided with Lucid CL 4.0.1+.

The Reorganizer rearranges the memory space of a Lucid CL application image to reduce disk paging. If items that are used together are located near each other, locality of reference is increased and paging is reduced. The Reorganizer operates like a profiler in that monitoring code is added to the image that records which functions and data items are accessed during execution. You can then dump a fresh image of your application using the data produced by the Reorganizer.

The Treeshaker removes unused functions and other data (e.g., data types and printing formats) from an application to reduce the image size. Unlike Allegro Presto, items that are removed from an image are no longer available, so the Treeshaker is really creating a subset of Lucid Common Lisp that is sufficient to execute the application. The Treeshaker is not intended to be used on applications that use the compiler while running because including the compiler and all code it needs severely limits the amount of code and data that can be removed from the image.

The Reorganizer requires a lot of memory to run. It requires roughly 3 times the amount used when running the application normally. In addition, the image runs very slowly due to the code that is tracking data and code references. The results of applying the Reorganizer alone and in combination with the vendor independent optimizations to CIMTOOL are shown in Table 8. As promised, the Reorganized image is slightly larger and about 18% faster. The “All Optimizations” image includes applying the Reorganizer with attach/detach, method compilation and type declarations.

Optimization	Time (% of unoptimized)		Disk Size (KB)	Execute Size (KB)
	Load	Execute		
No Optimizations	136 (100)	157 (100)	14648	26340
Reorganizer	132 (97)	128 (82)	14872	24420
Reorg. + Attach	64 (47)	137 (87)	18048	27428
All Optimizations	68 (50)	121 (77)	17856	27364

Table 8: Effect of Lucid Delivery Toolkit on Common Lisp Application Performance

The Treeshaker requires even more memory to run - closer to 4 times the application’s normal usage. We tried to use Treeshaker even though Picasso does use the compiler in certain situations. Unfortunately, the Beta version of the Treeshaker was not able to handle the dynamic generation and loading of code performed by Picasso. The Lucid support staff was helpful in our getting Treeshaker to run on small Lisp test cases, but we never managed to get it to run on a Picasso image, even with a simpler Picasso application or no application at all. When Treeshaker is officially released, it can be used to reduce Lucid CL image sizes. However, in our experience it will have a minimal effect on execution time unless you are running on a severely memory-limited system.

5. Discussion

This section discusses the overall effect of the optimizations and the tradeoff between the effort needed and the benefit gained.

Using a combination of Lucid Reorganizer, attach/detach, method compilation and type declarations on the original CIMTOOL application, we realized a 50% reduction in load time and a 23% reduction in execution time at the expense of a 22% increase in disk image size and a 4% increase in run-time image size.

The best results with Allegro Common Lisp came from using a combination of dynamic loading, placing code in text space and attach/detach on the original application. This resulted in a 52% reduction in load time and a 28% reduction in execution time while also providing a 3% decrease in disk image size and a 19% decrease in run-time image size.

Looking over the effects of the optimizations, it is clear that attach/detach is the big winner by providing a 20% execution time reduction and a 50% reduction in load time. The load time improvement is not surprising, because attach/detach allows most initialization to be done before the image is made. The increased performance during execution occurs because some initialization is done when a Picasso object is first displayed, and attach/detach reduces or eliminates this time.

The other optimizations produced 15% or less reductions in elapsed execution time, although the Reorganizer did a bit better. Unfortunately, many of the optimizations conflict, that is, when two or more are combined, performance is often worse than the best of the included optimizations. This result means that we might be able do a few percent better than our best for Lucid CL by not using method compilation. We certainly do not want to try every combination to see which is best, but the given results provide some ideas as to what might work well together.

Attach/detach was not difficult to implement because much of Picasso was designed with the feature in mind. Although the exact mechanism does not translate to other applications, the idea of doing as much computation as possible in advance can be useful.

CLOS method compilation is easy to apply to an application, unfortunately it tends to cause worse performance when combined with other optimizations. The overhead of loading the compiled dispatch code must be offset by enough initialization and execution to avoid a net loss. Given the utility of attach/detach, which removes most of the initialization, it does not seem appropriate to use method compilation.

Allegro Presto provided substantial reductions in the image size on disk, given that the bundle file needs to be present on disk anyway. For a small increase in load time we get a 10 to 15% reduction in elapsed execution time. It took considerable effort and understanding to produce Lisp images using all the features of Allegro Presto, and there are limitations that make it less useful during development. Nevertheless, it produced good results.

The Lucid Reorganizer provided a reduction in execution speed at the expense of a slight increase in disk image size. Creating the Reorganizer data is slow but not very complicated. And, using the Reorganizer seems to produce few conflicts with the other optimizations. When the Treeshaker is available it should help to reduce load time and disk image size as well, although permanently removing code from an image can cause problems.

6. Conclusions

In the end, the best improvements came from modifying and tuning our code. It is good to see major Common Lisp vendors addressing the performance and resource problems of Lisp, and their tools do provide benefit. As expected, these general optimization tools are not a substitute for careful design, good coding, and a bit of after-the-fact tuning.

The Franz and Lucid products both provide excellent development environments. In comparison, each product has strengths. The Allegro dynamic loading of functions facility (Presto) does a good job of reducing image sizes. The Lucid compiler provides considerable help in detailing which declarations will best improve performance. The bottom line is that for the versions we tested, Lucid is better for development and Allegro is better for deployment. However, the differences between these systems are insignificant compared to the differences between CL and other programming languages.

The downside of this experiment was the effort required. Man-months of extra work to tune the performance of an application is excessive and rarely available. Although the situation is improving, it is still difficult to justify the use of Common Lisp for the development of a large application that needs to be delivered to users. In our case, the performance and space gains were not sufficient to justify continuing the development of Picasso in Lisp.

Acknowledgements

We want to thank Jon L. White and Sayuri Nishimura at Lucid Inc. and Charles Cox at Franz Inc. for their help.

References

[BeR90] B. S. Becker and L. A. Rowe, "A Hypermedia Extension of the PICASSO Application Framework", Proc. NIST Advanced Information Interfaces: Making Data Accessible 1991, June 1991.

[FrI91a] Franz Incorporated, Allegro Common Lisp User Guide Release 4.0, Berkeley, CA, January 1991.

[FrI91b] Franz Incorporated,"Technical Memorandum #14: Allegro Presto", Berkeley, CA, January 1991.

[Gab91] R. P. Gabriel, "Lisp: Good News, Bad News, How to Win Big", AI Expert, June 1991, pp. 31-39.

[Kee88] S. E. Keene, Object-Oriented Programming in Common Lisp, Addison-Wesley, Reading, MA, 1988.

[KoR91] J. A. Konstan, L. A. Rowe, "Developing a GUIDE Using Object-Oriented Programming", OOPSLA '91.

[LaR91] D. K. Layer and C. Richardson, "Lisp Systems in the 1990s", Communications ACM, Vol. 34, No. 9, September 1991, pp. 48-57.

[LuI90] Lucid Incorporated, Sun Common Lisp Version 4.0 User's Guide, Second Ed., Menlo Park, CA. September 1990.

[LuI91] Lucid Incorporated, Sun Common Lisp Version 4.0: The Delivery Toolkit (Beta Draft), Menlo Park, CA, March 1991.

[RKS91] L. Rowe, J. Konstan, B. Smith, S. Seitz and C. Liu, "The PICASSO Application Framework", UIST '91.

[Sch91] R. W. Scheifler, et al., CLX - Common LISP X Interface, Release 5, August 1991.

[ScR92a] P. K. Schank and L. A. Rowe, "An Introduction to Semiconductor Manufacturing and Markets", (ERL Report No. UCB/ERL M92/35), Berkeley, CA: University of California, Electrical Engineering and Computer Sciences.

[ScR92b] P. K. Schank and L. A. Rowe, "Navigation and Learning in a Multimedia Course on Semiconductor Manufacturing", (ERL Report No. UCB/ERL M92/36), Berkeley, CA: University of California, Electrical Engineering and Computer Sciences.

[SmR92] B. C. Smith, L. A. Rowe, "An Application-Specific Ad Hoc Query Interface", IEEE Transactions on Semiconductor Manuf., Vol. 5, No. 4, November 1992, pp. 281-289.

[Ste90] G. L. Steele Jr., Common Lisp, The Language, Second Ed., Digital Press, 1990.

[VoK89] K. Voros and P. K. Ko, "Evolution of the Microfabrication Facility at Berkeley", Electronics Research Lab. Memorandum M89/109, Sept. 1989