WCL: Delivering Efficient Common Lisp Applications under Unix

Wade Hennessey

Center for Design Research
Stanford University

wade@sunrise.stanford.edu

Abstract: Common Lisp implementations for Unix have traditionally provided a rich development environment at the expense of an inefficient delivery environment. The goal of WCL is to allow hundreds of Lisp applications to be realistically available at once, while allowing several of them to run concurrently. WCL accomplishes this by providing Common Lisp as a Unix shared library that can be linked with Lisp and C code to produce efficient applications. For example, the executable for a Lisp version of the canonical "Hello World!" program requires only 40k bytes under SunOS 4.1 for SPARC. WCL also supports a full development environment, including dynamic file loading and debugging. A modified version of GDB [1], the GNU Debugger, is used to debug WCL programs, providing support for mixed language debugging. The techniques used in WCL should also be applicable to other high-level languages that allow runtime mappings from names to objects.

1 Introduction

1.1 Lisp Machine Approach

Many Common Lisp implementations for Unix are modeled after the "Lisp Machine" view of computing. A Lisp development environment is often a multi-megabyte executable that contains a compiler, evaluator, dynamic linker, debugger, runtime library, threads package, editor, window system interface, etc., all running in a single address space. Support for passing data to foreign languages is typically quite good, although great effort is usually taken to distinguish Lisp data from foreign data. However, because Lisp programs cannot be provided to other programmers in a form compatible with the native system linker, Lisp must remain "in control" by linking foreign code into the Lisp image. Unfortunately, foreign debugging information is usually lost in the process, thus making debugging of mixed language programs difficult.

Standalone applications are produced by dynamically loading application code into a running Lisp and then saving a memory image to disk. Autoloading can be used to delay the loading of code until runtime, thus reducing image size, but it can substantially increase the startup time of an application. Treeshakers and selective loading of subparts of the Lisp system are also used to reduce the size of saved memory images, but the inherently intertwined nature of Common Lisp makes this difficult to use effectively. Furthermore, the read-only portions of a memory image can be shared only with other people running *exactly* the same image. Thus, every application that is even partially written in Lisp must duplicate both on disk and in memory a significant portion of the Lisp runtime library, no matter how small the executable size becomes.

1.2 Unix and WCL approach

Window oriented Unix programs encountered problems similar to Common Lisp when dealing with large libraries such as X. Every application that wanted to access the window system was statically linked to include large portions of the X library, giving each application its own copy of the library code. For example, when the executable for the X Calculator program `xcalc` is linked this way, it occupies 773k bytes of disk space under Ultrix 4.1 on a MIPS based DECStation. Dozens of other X clients have similar disk requirements, leading to a poor utilization of disk space and memory.

Shared library support was added to many versions of Unix to overcome this problem. Executables only contain the code that is unique to a specific application, while common library code is shared both on disk and in memory with other applications. For example, under SunOS 4.1 on SPARC, `xcalc` only requireds 82k bytes of disk space because it shares the X library code with other applications.

WCL takes advantage of shared libraries to produce correspondingly efficient Lisp applications. WCL emits standard ".o" files that can be linked into a sharable library using `ld`, the standard Unix linker. Applications can then be linked with shared libraries using `ld` again. Thus, Lisp libraries can be shard with other programs just as easily as a library written in C or Fortran.

Most Lisp systems also provide their own unique debugger that executes in the same address space as the Lisp system. This differs from the Unix model of debugging in which the debugger executes in a superior process that runs and controls the program being debugged in a separate, inferior process. In order to make WCL programs work well with other languages, the GNU debugger GDB has been modified to understand Lisp. GDB was chosen because it is an excellent debugger with full source code availability. The result is that Lisp and C programs can be linked together and debugged using the native tools used by other Unix programs. Similarly, native profiling and code analysis tools can also be applied to Lisp programs.

2 WCL Compiler and Runtime Model

WCL translates Common Lisp into either K+R or ANSI C. As Lisp code is translated to C, information about symbol usage, structure definitions, and package operations is also recorded. This information is currently stored as a comment at the start of the emitted C file, although it should eventually be stored in a special section of the object file. The information is later used by the linking portions of the WCL development system.

2.1 Memory Model

The runtime model for WCL is fairly conventional. All dynamically allocated storage comes from a single heap. Every object in the heap has a single word header, one byte of which is used as a type field. The least significant bit of every Lisp pointer is used as a tag bit. A tag of 0 indicates that the pointer is really a 31 bit immediate integer, while a tag of 1 signifies that the type field in the object header indicates the object's type. A single table of all possible character objects is maintained to avoid allocating characters in the heap.

A conservative GC based on the work done in Scheme->C [2] is used to garbage collect the dynamic heap. A Common Lisp conservative GC is actually a bit simpler than a conservative GC for Scheme because Common Lisp does not support heap allocated upward continuations. A static heap that is not subject to garbage collection is also provided for permanent objects that should never be copied.

2.2 Runtime Model

Every Lisp function is translated into a corresponding C function, using name mangling to convert Lisp names into legal C names. Depending upon the compiler's optimization settings, Lisp functions may be called directly, just as they would normally be called in C, or they may be called indirectly through the function cell of the symbol that names the function. The first argument to every Lisp function is an argument count, followed by the required arguments. The varargs/stdarg support in C is used to support optional, keyword and rest arguments. Both heap allocated and stack allocated rest argument lists are supported, as well as stack allocated rest argument vectors.

Multiple values are supported by overloading the purpose of the argument count. If a caller wants to receive multiple values, a pointer to a stack allocated structure is passed as the argument count. The actual argument count is stored inside of this structure, along with sufficient space for receiving the values. The structure pointer is distinguishable from an actual argument count because argument counts are small integers, while stack pointers appear as large integers greater than `call-arguments-limit`.

If a callee wants to return multiple values, the first value is returned as the function's normal return value, while the remaining values are stored into the values structure along with a return value count. This system imposes no overhead on the normal single value expected, single value returned, function call. The only complication involves passing a multiple value holder on to other functions when making tail calls. For example, if A calls B expecting multiple values, and B does a tail call to C, B must pass along the multiple value holder from A to C. However, it must determine at runtime if A passed a multiple value holder. This check adds a few instructions to tail calls. However, this approach has the advantage of using standard C while still being efficient.

`Catch`, `throw`, dynamic `return-from`, dynamic `go`, `unwind-protect`, and special variable binding are all supported by `setjmp` and `longjmp`, and a stack allocated "dynamic chain". For example, a call to `throw` first searches the dynamic chain for a corresponding `catch` tag. If found, then the chain is unwound, undoing special bindings and using `longjmp` to hop up the stack and execute cleanup code in any `unwind-protects`. Once the stack is unwound, `longjmp` transfers control to the catch point.

`Setjmp` and `longjmp` present a few problems for Lisp. Implementations of `setjmp` and `longjmp` that do not preserve all program state at a setjmp site require volatile declarations, decreasing the efficiency of the code. Also, because C and Lisp can freely call each other, C code can do a `longjmp` through several dynamic chain frames to a `setjmp` site. However, `longjmp` has no idea that the dynamic chain exists, and thus no unwinding will occur and the top-of-chain pointer will be incorrect. For these two reasons, it may be preferable to write a custom version of `setjmp` and `longjmp` that saves all state and has builtin support for Lisp's dynamic chain, although this is not currently done.

Closures present another interesting problem given that all Lisp functions should look like C functions. A closure consists of an environment vector and some code that expects to use that environment. These two components are glued together by a small piece of heap allocated machine code that stores the address of the environment vector into a global variable and then jumps to the closure's code. The code then stores the contents of the global variable into a local environment pointer variable, effectively using the global variable as an extra argument register. The garbage collector has special knowledge about closures and understands how to load and store the environment pointer that has been encoded into the instruction stream of the closure.

Foreign functions must be declared before being called. For example, the C function `ldexp` demonstrates how WCL shares numbers with C:

```
(defforeign ldexp ((significand double)
                   (exponent int) =>
                   (result double)))
```

The call `(ldexp sig exp)` compiles into the following C code:

```
NEW_FLOAT(ldexp(RAW_FLOAT(v_SIG_0),
                FX_TO_INT(v_EXP_1)));
```

WCL provides various C macros which are used to pass data between Lisp and C. Similarly, WCL also provides support for sharing other data types such as strings and arrays. All Lisp strings are null terminated so they can be passed directly to C. Returned strings are represented in the same way that a complex vector would be represented. A two word complex string object is allocated and initialized with a pointer to the returned string. The length field of the complex string object is also initialized by calling `strlen`. Thus, native C strings appear to be full fledged Lisp objects without any copying of data, allowing destructive operations to work on returned data as expected. This same technique can also be used to pass and receive arrays of any rank and element type.

3 Linking Shared Libraries and Applications

3.1 Linking Shared Libraries

Anyone may build a shared library of Lisp code with the functions `define-library` and `link-library`. However, before `ld` can link a library of Lisp files, the extra linking information that was written during the compilation of the library is used to create additional code and data.

Every time a Lisp file contains a reference to a Lisp symbol, a reference to the mangled name of the symbol is emitted. However, at some point exactly one correctly initialized instance of the symbol must be created. The Unix linker's concept of a common area provides part of the support necessary to create exactly one symbol, but it lacks support for properly initializing this common area so that the symbol's name, function cell, etc. all contain appropriate values.

To overcome this problem, information about symbol usage in each file is stored in the file's linking information, and exactly one instance of every referenced symbol is written out as a C structure. The slots of this structure are also initialized according to the linking information. For example, if one file defines a function named `FOO` while another file contains the form `(DEFVAR FOO 3)`, the compiler records these facts in the linkage information for each file. At link time, the function cell of `FOO` will be initialized to point to the corresponding C code, while the value cell will contain the fixnum 3, thus avoiding any runtime work to initialize these slots. The linker does not currently know how to initialize the package system at link time. Thus, every symbol is interned at runtime in its corresponding package. The garbage collector is also informed about all symbols so that they can be traced.

Linking information about structure definitions is also recorded and used to define efficient structure predicates. In a fluid development environment, a structure predicate may potentially have to examine the structure hierarchy at runtime to determine the type of an object. However, by freezing the structure hierarchy at link time, WCL can create structure predicates that can perform type checks more quickly.

While compiling a Lisp file, every top-level form is collected into an initialization thunk that must be called at runtime to initialize the file, and the name of this

Library	Text	Data	File Size
Common Lisp	1351k	311k	2113k
Compiler	868k	221k	1425k
CLXR5	1548k	352k	2400k
Development executable	41k	8k	49k

Table 1: WCL Library and Executable Sizes

thunk is recorded in the linking information. When linking a group of files, a single initialization thunk is defined for all the files, sequentially calling each file's initialization code. In order to "start" the whole collection of files, only this single initialization thunk must be called.

Once these extra pieces of code and data have been compiled, everything is linked by `ld` into a sharable library. Additional information about the procedures and symbols defined in the library is also recorded.

3.2 Linking an Application

A list of files may also be linked into an executable application. Custom code and data are created as before, and a small main program is written to start the application. All of these files and any required shared libraries are then linked together to form an executable.

4 WCL Status

4.1 Functionality

WCL is currently organized into three shared libraries:

- Common Lisp - implements approximately 80% of the functions, macros, etc. in the Common Lisp manual, as well as the `loop` macro, the condition system, a foreign function interface, and some popular extensions to Common Lisp.
- Compiler - implements the compiler and linker functions
- CLXR5 - implements an interface to the X window systems.

The sizes of these libraries are shown in table 1. All libraries were compiled with GCC 2.1 [3] into position-independent code. Unfortunately, this tends to make them a bit slower than position-dependent code, but it is necessary for code sharing.

While some of the more obscure Common Lisp functions and features are not implemented yet, enough of Common Lisp has been implemented to allow WCL to completely compile, link and debug itself, thus supporting its own development. Because all of the functionality provided by WCL is provided in the form of shared libraries, the executable for the development environment is only 49kbytes.

The following packages also compile and run, but are not provided as shared libraries yet:

- May Day version of PCL [4]
- XP Pretty Printer
- Logical Pathnames

All three of these systems should eventually be integrated into the Common Lisp library. Of these systems, PCL presents the most integration work because it contains significant boot strapping and initialization code that would make starting the library too slow. More work must be done to eliminate these initialization steps or perform them at compile or link time. Together, the libraries and systems described above implement most of the functionality described in the second edition of the Common Lisp manual [5].

4.2 Porting

At one time WCL ran on a MIPS based DECStation under Ultrix, but that port has not been maintained and needs work. However, running WCL on a little-endian MIPS processor and a big-endian SPARC processor was useful for removing byte order dependencies in the code. Table 2 shows some statistics about the source code for WCL 2.0 running on SPARC.

The dynamic loader is the largest port specific piece of WCL. It consists of 1200 lines of C code, and significant changes are required to port it to other operating systems. A small amount of assembly code is also required for garbage collection, closures and fixnum to bignum overflow handling. However, because WCL uses C as an intermediate language, it is much more operating system dependent than it is machine dependent.

5 Development Environment

The WCL development environment is really just a Lisp application that uses the Compiler, Common Lisp, and

CL Library Lisp code	15300 lines	3% Unix dependent
CL Library C code	8000 lines	18% SunOS dependent
CL Library assembly code	90 lines	100% SPARC dependent
Compiler/Linker Lisp code	5700 lines	1% SunOS dependent code
CLX R5 Lisp Code	18900 lines	3% Machine and SunOS dependent.

Table 2: WCL Source Code Information

CLX libraries. GNU Emacs [6] and GDB also serve as useful development tools, although they are not required. When using Emacs, GDB mode is usually used to start an inferior GDB, which in turn starts an inferior WCL process. Unix profiling tools are also useful for analyzing performance.

5.1 Dynamic loading

The Common Lisp `load` function provides an interface to the dynamic code loader. The same object files that are linked into an executable image can also be dynamically loaded into a running Lisp. The loader first reads the various sections of the object file into memory and then performs all necessary relocations. In order to perform these relocations, the loader's symbol table is initialized with all the external symbols defined by the executable as well all the symbols defined by any shared libraries with which the executable is linked. After this initialization, new external symbols are added as files are loaded.

The loader described above is sufficient for loading and relocating normal C files. However, the files produced by WCL contain references to Lisp symbols even though these symbols may not exist. Normally these symbols are defined through the application linking process described earlier. However, in the dynamic linker they may appear to be undefined symbol references. If the loader calls the undefined symbol handler, the symbol name is first checked to see if it is the name of a Lisp symbol. If so, then the name is demangled, and the Lisp function `intern` is called with the appropriate symbol and package name, or `make-symbol` is called if the symbol is not in a package. Thus, loading an object file can cause new Lisp symbols to be created.

After the file is loaded and correctly relocated, the Lisp linking information associated with the file is examined. This information is used to correctly initialize symbol function and value cells, and to call the initialization thunk for the file. A "fluid" `defstruct` predicate is also defined for any structures that are defined in the file. This predicate differs from the one created during application linking by examining the structure inheritance hierarchy at runtime. Although this approach is slower, it is more useful for program development since it tracks `defstruct` inheritance changes correctly.

5.2 Debugging with GDB

GDB is a multi-language debugger that already supports C, C++, Modula-2 and Fortran, and several changes have been made to support Lisp debugging. Some of these changes are unusual because they rely on the underlying Lisp process to actively cooperate with GDB, which is quite different from the normally passive role the inferior process plays.

A Lisp name demangler has been added to the parts of GDB that print frame names. Thus, backtraces now show mangled Lisp names as they originally appeared, while still displaying regular C function names correctly. The frame handling code in GDB also has special knowledge about the names of functions in the Lisp evaluator so that frames that are really being used to interpret Lisp code are either hidden or are displayed as the name of an interpreted function. When examining an interpreted stack frame, it is also possible to recursively enter the Lisp evaluator with the current lexical environment using a new GDB command called `eval`.

The `eval` command uses GDB's ability to call functions in the process being debugged. When `eval` is called, the name of the current frame is examined. If GDB recognizes that the frame is part of the evaluator, then a new read-eval-print loop is started in the underlying process using the current lexical environment. A small portion of the evaluator is written in C to allow convenient access to this environment. This loop allows access by name to local variables, and thus any Lisp expression may be interactively evaluated in the lexical environment of an interpreted function. If the current frame is not an interpreted function, but is instead a C function or a compiled Lisp function, then the read-eval-print loop is run in the null lexical environment.

GDB is also integrated with the Common Lisp condition system by using GDB's ability to call functions in the inferior Lisp process. The new command `info restarts` shows all available restart options, while the command `restart` allows the user to select an option. The new `abort` command is also available as a convenient short cut for aborting to top-level by selecting the outermost abort restart. This command is especially useful when Lisp is in an infinite loop and control-c is used to interrupt the computation. The standard GDB `continue` command can also be used to continue an in-

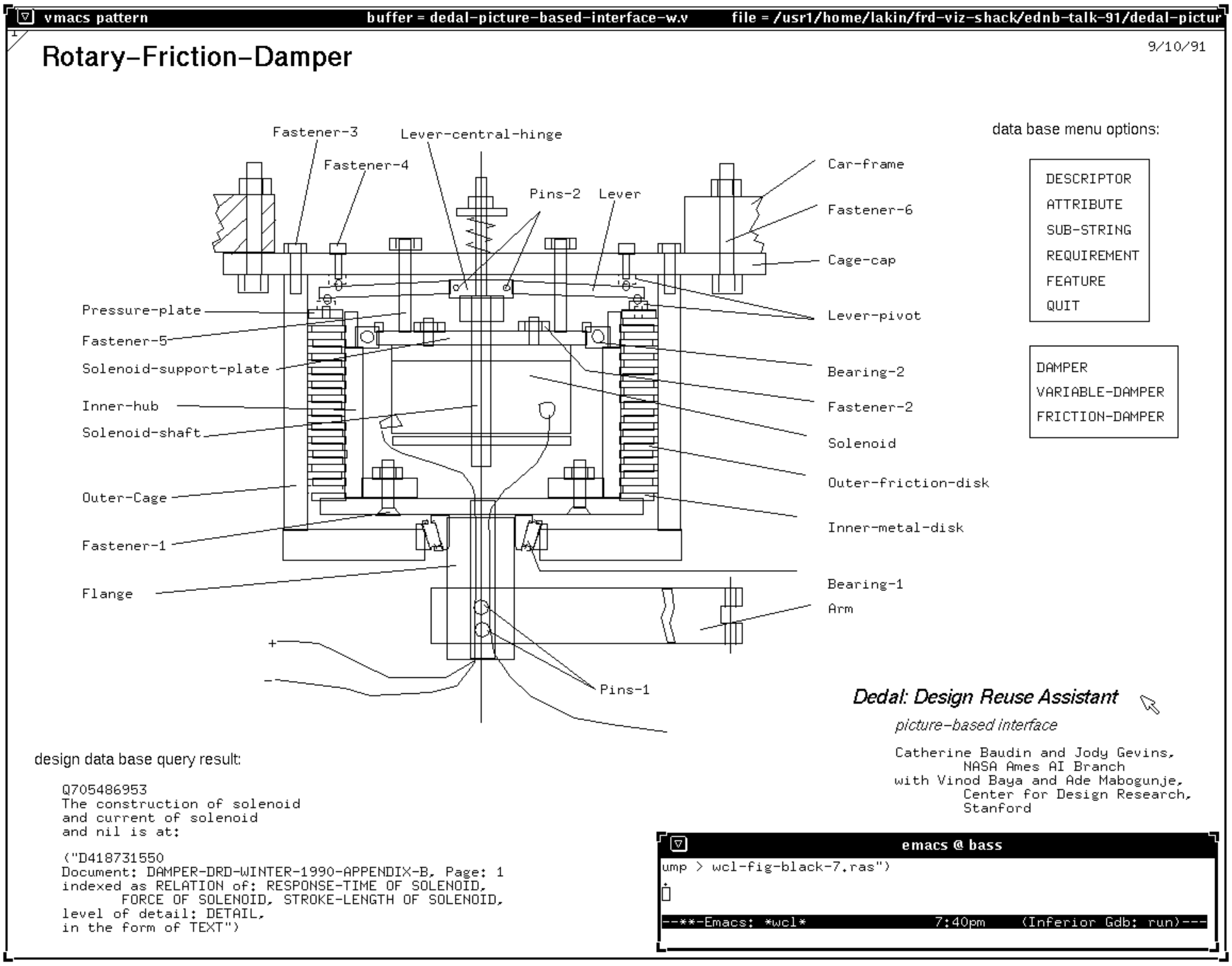

Figure 1: WCL Running the VMACS Electronic Design Notebook

terrupted computation, or to select a restart option.

Source level debugging of compiled Lisp functions is not yet supported. Instead, the translated C code must be debugged. Although source level debugging of Lisp using GDB is the ultimate goal of WCL, debugging the C code is still feasible. Variables in compiled Lisp functions can be examined with the new GDB `lprint` command. `Lprint` accepts the mangled name of a Lisp variable, and calls the printer in the underlying Lisp process with the value of that variable.

The GDB `add-symbol-file` command already supports adding the symbol table and debugging information of a file that has been dynamically loaded at a known address in the inferior process. However, debugging dynamically loaded Lisp code also required a few changes to GDB. After Lisp loads a compiled file, a temporary file is written containing the name of the loaded file and the address at which it was loaded. WCL then uses the Unix `kill` system call to send `SIGUSR1`, a user defined Unix signal, to itself. Because GDB is debugging WCL, it intercepts this signal and understands that it should update its symbol table using the information in the temporary file. Execution of WCL is then resumed normally, and the dynamically loaded file can be fully debugged without any intervention from the user.

6 Application Performance

6.1 Real Applications

Figure 1 shows WCL running the VMACS Electronic Design Notebook [7]. VMACS is a visual editor implemented in 87000 lines of Common Lisp code, and is the largest real application that WCL has compiled and run.

Metric	WCL-2.0	Lucid 4.0	AKCL 1-530
executable size	40k	1564k	2440k
text size	32k	1040k	1290k
data size	8k	475k	1150k
minimum start+exit time	1.2s	.3s	.01s

Table 3: "Hello World!" Statistics

It uses the Compiler, Common Lisp and CLX libraries as well as the logical pathname system, thus serving as a broad test of WCL features.

6.2 Code size

In order to compare minimum application sizes, an executable that prints the string "Hello World!" and then exits was created using three different Lisps. The resulting executable sizes are listed in table 3.

AKCL [8] is a modified version of KCL [9]. The ACKL executable was produced with the `save-system` command. AKCL simply dumps a memory image of the running Lisp, and thus the executable's size is essentially the same size as the Lisp development environment's size.

The Lucid [10] executable was produced using Lucid's delivery toolkit. This toolkit is based on the concept of treeshaking. Starting from a known root set in a running Lisp, all functions that are not required in the final application are "shaken" away, leaving only the required portions of the runtime system. Because Lisp's inherently intertwined nature makes it difficult to determine at link time which functions will really be needed at runtime, the toolkit provides a fairly complex set of options for manually declaring what will be needed in the final application.

The WCL executable was linked with the shared Common Lisp library described earlier. The use of a shared library has important implications for delivering and running many Lisp based applications on a single machine. Using the Lucid or AKCL "disksave" approach, ten other applications that are similar to the "Hello World!" program will not only require the same amount of disk space as "Hello World!", but they will also contain ten almost identical copies of the Common Lisp runtime library that will not be shared on disk or in memory. This lack of sharing dramatically increases the amount of memory required to run several Lisp based applications at once. A shared library can also be replaced at anytime by a newer version of itself without recompiling or relinking any of the applications which use the library, thus making it easy for all applications to benefit from bug fixes or performance improvements.

6.3 Startup time

Table 3 also shows the minimum amount of time required to start and exit the "Hello World!" program. These times were obtained by repeatedly running the application until a consistent, minimal time was obtained. These numbers are meant to avoid paging time since that can vary in response to a variety of factors. However, in realistic day to day usage, the actually startup time for an application is heavily dependent on its size and the resulting paging time. WCL actually benefits from having many Lisp applications running at once, because each application increases the likelyhood that a shared WCL library will be resident in main memory. The exact opposite is true of statically linked applications because they do not share code and thus fight with each other for available memory.

A WCL applications spends its startup time setting up shared libraries and running initialization code. Ideally, a shared library consists of only position-independent code that can be mapped anywhere into the address space of a process and can start executing without excessive runtime relocation and without having to make private copies of the library. Unfortunately, the Lisp library also contains a significant amount of initialized pointer data such as lists, arrays, and symbols. The correct value for these pointers is position-dependent, and thus they must be relocated when the library starts, slowing the start of an application.

Under SunOS, these symbol relocations are not resolved incrementally, but are instead completely resolved *before* the program starts executing, while procedure relocations are done incrementally as they are needed during execution. SunOS supports linking with ".sa" files containing exported, initialized shared library data to avoid this problem. Although WCL does not currently support this option, including all of the initialized pointer data in each application should improve the startup time at the cost of an increased executable size. One compromise is to support two versions of the library, one of which contains the initialized data for programs that must be small at the expense of startup time, and another library that does not contain the data for applications that don't mind the space penalty.

Initialization code is another contributor to startup time. Lisp has traditionally supported a "load-and-

Benchmark	WCL 2.0	Lucid 4.0	AKCL 1-530
Boyer	3.54	2.94	2.70
Browse	5.75	2.81	4.50
CTak	7.25	1.09	7.35
Dderiv	1.60	.82	1.48
Deriv	1.20	.65	1.05
Destructive	.67	.26	.45
Iterative div2	.58	.32	.55
Recursive div2	2.07	1.88	2.00
FFT	8.20	.34	9.21
FPrint	.31	.49	.25
FRead	.69	.29	.23
Frpoly10r	.01	.01	.02
Frpoly10r2	.10	.05	.10
Frpoly10r3	1.21	.65	1.25
Frpoly15r	9.45	5.19	13.03
Puzzle	.83	.95	1.72
STak	.59	.67	.90
Tak	.05	.05	.87
Mas	.35	.32	.30
Takr	.08	.10	.10
TPrint	1.55	.56	.58
Traverse	5.76	4.28	7.40
Triangle	15.42	12.22	19.217

Table 4: Gabriel Benchmark Times

disksave" model of linking. Thus, initialization actions such as computing a dispatch table are done at load time and do not cost anything when the resulting memory image is dumped and later run as an executable. Programs such as PCL make extensive use of this feature. Unfortunately, linking under Unix does not support this, and thus initialization functions must be run *every* time an executable is started. Because of this, WCL defers some initialization actions until they are actually required, rather than unconditionally doing them at startup time. However, more work can be done to reduce initialization time.

6.4 Benchmarks

Table 4 presents the running times of the gabriel benchmarks in three Lisps. All tests were compiled with the compiler set for maximum optimization, and were run on an idle Sun SparcServer 4/330 running SunOS 4.1.1 with 96 megabytes of physical memory. Both the WCL and the KCL benchmarks were compiled with GCC 2.1 using the `-O2` option.

Three runs of each benchmark were performed, and the best elapsed real time is listed in the table. All benchmarks were preceded by a garbage collection. Lucid was run with the ephemeral garbage collector turned off.

The two benchmarks with the widest variation in time are FFT and CTak. FFT does poorly in both WCL and KCL because these systems are storing floating point results in the heap, and thus they spend most of their time allocating memory. Lucid avoids this by only allocating floats in the heap when necessary.

CTak is also much slower in WCL and KCL because these systems implement `catch` and `throw` with `setjmp` and `longjmp`. Because SPARC uses a register window design, `setjmp` can be implemented by simply saving the stack pointer and relying on the fact that all registers are safely saved somewhere in the register file or on the stack. However, in order to retrieve the registers for a specific environment, `longjmp` flushes the *entire* register file to the stack, which is an expensive operation. Thus, both WCL and KCL spend roughly 80% of their time in `longjmp` flushing the register file! The current implementation of `setjmp` and `longjmp` is probably optimal for most real programs which dynamically execute `setjmp` much more often than `longjmp`, but CTak would benefit from an implementation which assumes that these operations dynamically occur with equal frequency.

7 Comparisons to Related Work

WCL is clearly related to other Lisp-to-C translators such as KCL, Scheme->C [11], Chestnut Software's Lisp translator, and Ibuki's CONS system. However, WCL differs from these systems not only in the runtime model that it uses for translating Lisp to C, but also in its approach to delivering applications. Of these systems, Chestnut and Ibuki seem to have done the most work directed towards producing an efficient delivery environment. However, these systems take a fundamentally different approach than WCL toward solving the delivery problem because they choose to statically link a standalone executable.

An earlier version of WCL also supported static linking with a library from which applications could extract only the code they required. However, static linking was eventually abandoned because the Lisp library is too intertwined to easily produce small binaries without excessive intervention on the part of the programmer. This intervention requires the programmer to assist the Lisp system in determining what parts of Lisp will not be required at runtime. Not only is this intervention cumbersome and error prone, but it draws a sharp distinction between the development environment and the delivery environment - a distinction C programmers do not encounter. Furthermore, no matter how small a statically linked executable becomes, it can *never* share code or data with related applications, and thus multiple applications are doomed to duplicate large pieces of common information, just as we saw in the statically linked `xcalc` example given earlier.

All of these problems with static linking led to the adoption of dynamic linking with a shared library in WCL. Using dynamic linking, there is virtually no difference between the development system and the delivery system because all language features are available to an application. Furthermore, WCL applications benefit from the sharing between applications that is inherent in shared libraries. WCL also appears to be the only Lisp system that uses the same debugger as C and other foreign languages. This combination of features allows WCL to provide a tighter and more efficient integration of Lisp with main stream computing.

8 Future Work

Improving executable startup time is an important direction for future work. As we have already seen, there are several ways to attack this problem, most of which appear to be solvable. Dealing with initialized pointer data in shared libraries is the most difficult obstacle to overcome. Ideally Unix could be made to incrementally relocate data in shared libraries just as it currently does for procedures. However, barring this sort of operating system change, the amount of initialized pointer data in the library must be reduced, or that data must be moved into each application. With a bit of assistance from the user when linking an application, it should be possible to remove almost all of the symbols from many applications.

Adding source level debugging to Lisp code is another important direction for improvement, although three problems arise when trying to do this.

First, GDB is a line oriented debugger, whereas Lisp is an expression oriented language. Thus, if several expressions occur on a line and an error occurs in one of those expressions, the best GDB can do is point at the line containing all of the expressions. This problem could be solved if Unix simply used source file character positions rather than line numbers for debugging, but for historical reasons this is unlikely to change soon.

The second problem arises from Lisp macros. Because macros can perform arbitrary program transformations, it is impossible to always determine the mapping from the expansion of a macro back to the original source code. However, in practice most macros do not make copies of input expressions, but instead splice those expressions into other pieces of code, thus dissecting the original source.

The third problem arises from the uniqueness of symbols in Lisp. A single symbol may be referred to on many different source code lines, thus making it difficult to accurately associate each symbol reference with the correct line number in the presence of complex or arbitrary program transformations. The uniqueness of characters as well as the immediate representation of fixnums presents a similar problem.

The current compiler could also use numerous improvements. Adding much more sophisticated type inference and data representation analysis would yield the greatest benefits in compiled code. CMU Common Lisp [12] has already demonstrated the usefulness of these techniques. Ideally, the current compiler could also be used as the front end for an existing compiler backend such as GCC rather than emitting C code. Not only would this speed up the compiler, but it would offer new opportunities for optimization.

Other useful improvements include: changing the garbage collector to make it generational [13] and possibly incremental [14], supporting native Unix threads in operating systems that provide them, and adding efficient support for "frozen" CLOS programs. A large amount of work remains, all of which is guided by the desire to facilitate the development and delivery of Lisp applications which are as competitive as possible with C based applications.

9 Conclusion

WCL demonstrates that Common Lisp applications can be efficiently delivered under Unix by taking advantage of Unix's shared library support. This support allows hundreds of Lisp applications to realistically be available at once, while allowing several of them to run concurrently, just as C applications currently do. Furthermore, none of the unique features that make Lisp an appealing development environment need to be sacrificed. Instead, an even tighter integration between Lisp and foreign languages is possible by using a single debugger such as GDB. These features all support WCL's ultimate goal of helping real Common Lisp applications penetrate main stream computing more deeply than has previously been possible.

10 Acknowledgements

Thanks to the following people and groups who have provided software which has been modified and incorporated into WCL: Kent Pitman, Glenn Gribble, The MIT AI Lab, Texas Instruments, DEC and INRIA, CMU Spice Lisp, Mark Kantrowitz, and Guy Steele. I would also like to thank the following people for their help: Jeff Aldrich, Dave Dungan and the Center for Design Research for providing the computing resources needed to develop WCL, Joel Bartlett for his garbage collector ideas and beer and closures, Eric Benson for reviewing an early version of this paper, and Fred Lakin for his help in testing WCL, his comments on this paper, and his encouragement of this project.

References

[1] Richard M. Stallman and Rolan Pesch. *Using GDB: A Guide to the GNU Source-Level Debugger.* Free Software Foundation and Cygnus Support, Cambridge, Massachusetts, 1991.

[2] Joel F. Bartlett. Compacting Garbage Collection With Ambiguous Roots. Technical Report 88/2, DECWRL, Palo Alto, California, February 1988.

[3] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation and Cygnus Support, Cambridge, Massachusetts, 1992.

[4] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *Conference on Lisp and Functional Programming.* ACM, 1990.

[5] Guy L. Steele Jr. *Common Lisp The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[6] Richard M. Stallman. *GNU EMACS Manual.* Free Software Foundation, Cambridge, Massachusetts, 1989.

[7] The Performing Graphics Company. *VMACS Electronic Design Notebook^{TM} User's Manual.* The Performing Graphics Company, Palo Alto, California, 1991.

[8] William Schelter. AKCL is available via anonymous ftp from `rascal.ics.utexas.edu`.

[9] Taiichi Yuasa and Masami Hagiya. *Kyoto Common Lisp Report.* Kyoto University, Kyoto, Japan, 1985.

[10] SUN Microsystems. *Sun Common Lisp 4.0 User's Guide.* Sun Microsystems, Mountain View, California, 1990.

[11] Joel F. Bartlett. SCHEME->C: A Portable Scheme-to-C Compiler. Technical Report 89/1, DECWRL, Palo Alto, California, Janurary 1989.

[12] Robert A. MacLachlan. *CMU Common Lisp User's Manual.* Carnegie Mellon University, Pittsburgh, PA, 1991.

[13] Joel F. Bartlett. Mostly-Copying Garbage Collection Picks Up Generations and C++. Technical Note TN-12, DECWRL, Palo Alto, California, October 1989.

[14] G. May Yip. Incremental, Generational, Mostly-Copying Garbage Collection in Uncooperative Environments. Technical Report 91/8, DECWRL, Palo Alto, California, June 1991.