Optimizing the Performance of Dynamically-Linked Programs

W. Wilson Ho
Wei-Chau Chang
Lilian H. Leung

Silicon Graphics, Inc.

Abstract

Dynamically-linked programs in general do not perform as well as statically-linked programs. This paper identifies three main areas that account for the performance loss. First, symbols are referenced indirectly and thus extra instructions are required. Second, the overhead in run-time symbol resolution is significant. Third, poor locality of functions in shared libraries and data structures maintained by the run-time linker may result in poor memory utilization. This paper presents new optimization techniques we developed that address these three areas and significantly improve the performance of dynamically-linked programs. Also, we provide measurements of the performance improvement achieved. Most importantly, we show that all desirable features of shared libraries can be achieved without sacrificing performance.

1. Introduction

More and more UNIX systems support dynamic linking of shared libraries [MIPS90, Arno86, Ausl90, Cout92, Ging87] because they provide many desirable features. For example, both disk space and physical memory utilization are reduced due to increased sharing, and shared libraries can be replaced transparently without re-linking all user programs. More in-depth discussions can be found in [Ging89, Saba90]. However, the use of dynamic shared libraries does incur a performance penalty. Dynamically-linked programs generally run slower than statically-linked programs because they incur extra run-time overhead. This overhead includes the execution of extra instructions resulted from indirect addressing and run-time symbol resolutions, and extra memory requirement due to poor locality of functions in shared libraries and data structures used by the run-time linker.

Several optimizations have been proposed to improve the performance of shared libraries. Run-time overhead in indirect function calls can be improved by reducing the number of instructions used in the calling sequence [Kepp93]. Symbol resolutions can be deferred and carried out on-demand to improve start-up time [Saba90]. Loading and fixing-up of shared libraries can be cached to reduce the amount of work for subsequence invocations [Nels93, Orr93]. These methods are effective in speeding up the execution of dynamically-linked programs from their corresponding initial implementations. However, there still remains significant performance degradation of these programs when compared with their statically-linked counterparts. Informal discussions on the Usenet news group *comp.arch* seem to suggest that 10-25% degradation is commonly observed.

This paper describes a number of techniques we developed that can bring the performance of dynamically-linked programs much closer to the level of that of statically-linked programs. With these optimizations, we have built the entire IRIX system (SGI's implementation of the UNIX operating system) based on shared libraries without sacrificing performance. In fact, for most libraries, only the shared versions are provided. All optimization techniques described in this paper have been implemented and are available in version 5.3 of the IRIX system.

There are three main areas that account for the performance loss of dynamically-linked programs. First, text and data symbols are referenced indirectly. As a result, more instructions need to be executed. Our approach takes advantage of the information derived from the cross-module optimization phase of the MIPS compiler [Chow86, Hime87] and converts as many indirect address references as possible to direct references.

Second, symbol resolution takes time, even with deferred binding. Furthermore, since trapping data references is expensive, binding of data symbols is carried out at start-up time. This causes large C++ applications to suffer a long start-up delay because they usually contain a large number of function pointer tables. We developed a scheme called *quick-start* that virtually eliminates any need for run-time symbol binding.

Third, despite the sharing of library text among multiple processes, the use of shared libraries can sometimes require more memory. This extra memory requirement comes from both the data structures used by the run-time linker for symbol resolution for each process and the poor access locality of functions in a shared library. We solved the first problem by pre-computing most of the information required by the run-time linker at static-link time and putting them in sharable, read-only segments of shared libraries. To improve the access locality of functions in a shared library, we developed *cord*, a tool that repositions these functions based on either profiling information or explicit user-specification.

The rest of this paper is organized as follows: Section 2 provides an overview of the first implementation of the IRIX shared libraries. Section 3, 4, and 5 present techniques we employed to improve the shared libraries performance in the three respective areas described above. Each section is followed by an analysis of the gains in performance. All measurements are taken on an Iris Indigo, which has a 100 MHz MIPS R4000 processor. Section 6 summarizes the results and offers some conclusions.

2. Background

In this paper, we concentrate on the performance of SVR4-style shared libraries [Syst90], i.e., implementations that support dynamic symbol binding, sharing of text and read-only data among multiple processes, and mapping of the same library to different address space in different processes. In other words, less flexible implementations such as those that require fixed address assignment [Arno86, Hobb87] are not considered.

A full description of the basic implementation of IRIX's shared libraries can be found in [MIPS90]. This section describes only those implementation details that are relevant to the optimizations presented in this paper.

Under the IRIX implementation of shared libraries, the compiler always generates position-independent code (PIC). Thus, any object file created by the compiler can be part of a shared library. PIC is implemented by turning all address references into indirect references through a Global Offset Table (GOT). A GOT is created at static-link time and there is one for each shared library as well as the main executable itself. It is a table of addresses of all symbols that are referenced, and the content of this table is updated by the run-time linker on demand, i.e., deferred binding. Function calls are implemented by first loading the address of the callee into a register from the corresponding entry in the GOT, followed by a jump-and-link-register instruction. By updating its GOT with correct values, the run-time linker can relocate a shared library to any virtual address.

Furthermore, a dedicated register is used to hold the address pointing to the GOT corresponding to the function being executed. When control is transferred between different shared libraries, the address of the GOT is computed by the corresponding callee and the new value is put in this GOT register. However, when control is passed between functions within the same shared library, the content of the GOT register needs not be changed and can be used immediately.

Using PIC allows the text segment to be shared by all processes because it never needs to be modified. The GOT approach localizes all the run-time fix-up of addresses and cuts down the number of copy-on-write memory pages. When compared to the jump table approach in the SunOS compiler [Ging87], our implementation of function calls requires only one control transfer instead of two. This difference is of particular importance in modern computer architecture [Hein93, Hsu94], where jumps are in general expensive. Furthermore, the instruction to load addresses from the GOT can often be scheduled to be executed earlier so that other useful work can be done while the address is being fetched from memory.

3. Eliminating Indirect Addressing

As stated in Section 2, all object files generated by our compiler are PIC. This feature allows any object file created by the compiler to be put into a shared library. It also relieves programmers from the burden of deciding at compile-time whether a particular object should become part of a shared library or an executable.

However, if the main executable is never relocated at run-time, we can assign absolute addresses to it and replace all indirect references for symbols defined in the main executable by direct references. This can be achieved if the system guarantees that no shared library is mapped to the address space already used by the executable itself.

Unfortunately, under the traditional separate compilation model, the compiler cannot determine if references of undefined global symbols will be resolved with a definition from a shared library or from the main executable itself. Hence, it has to assume the worst case and generate indirect references for such symbols.

Our goal is to derive a scheme that can automatically determine if direct addressing is possible for any given symbol, without requiring any specification from the programmer. The approach is to extends the cross-module optimization phase of the compiler to collect information about where symbols are defined.

program	statically linked	dynamically linked	
		direct addressing	PIC
espresso	43.2s	45.2s (4.6%)	47.4s (9.7%)
lisp	89.0s	95.9s (7.8%)	114.8s (29.0%)
eqntott	13.9s	13.8s (-0.7%)	14.0s (0.7%)
compress	64.3s	67.0s (4.2%)	71.8s (11.7%)
sc	95.8s	102.7s (7.2%)	107.3s (12.0%)
gcc	120.9s	121.8s (0.7%)	138.8s (14.8%)
average		**3.97%**	**12.98%**

Table 1. Performance Degradation of Dynamically-Linked Programs

Under the cross-module optimization model, the MIPS compiler combines separately-compiled intermediate-code objects and performs function inlining, interprocedural register allocation, dead function removal, and many other optimizations that are not supported in the traditional model [Hime87]. All these optimizations are done before the final machine code generation. Note that during this cross-module optimization phase, the compiler collects information from all the files that compose the program, including the shared libraries that it links with, and thus has complete knowledge of where each symbol is defined. By simply recording this information in the symbol table and passing it down to the code generator, direct references can be generated for symbols defined in the executable module and indirect references can be generated for symbols defined in the shared libraries.

This optimization improves the performance in several ways. First, function call overhead within the main executable is reduced. There is no more memory reference for loading the address of the callee. Also, with the absolute address embedded as an immediate value in the jump-and-link instruction, the processor can potentially pre-fetch the next instruction without delay. Second, overhead for indirect references of data symbols is also eliminated. Very few other implementations, if any, can support dynamically-linked programs with direct data references. This optimization can have a significant effect on performance, especially when the symbols are referenced within a loop. Third, all symbols defined inside the main executable can be removed from the GOT, and thus the amount of work performed by the run-time linker is reduced. Furthermore, since the size of the GOT is reduced, the compiler can put more global variables in the data area close to the GOT and access these variables through a 16-bit immediate offset from the GOT register [Chow87]. Under the MIPS architecture, accessing data via an immediate offset from a register takes only one instruction, and is even faster than the normal direct addressing, which takes two instructions instead of one [Hein93].

Table 1 compares the performance of several dynamically-linked programs against the statically-linked versions. These programs are from the SPECint benchmark suite [SPEC91]. Three versions of each programs were compared: the statically-linked version, the dynamically-linked version with the direct-addressing optimization described in this section, and the default PIC version. The performance of each dynamically-linked version is compared to that of the statically-linked counterpart. Numbers in parenthesis are the percentage degradation.

By eliminating indirect addressing, the performance of these programs is consistently better than the PIC version. The most significant improvement is found in *lisp* and *gcc*, both of which contains a large number of function calls. When compared with the statically-linked version, the worst performance degradation is cut down to only 7.8%, while *eqntott* and *gcc* show almost no degradation. On average, dynamically-linked programs are now only 3.97% slower than statically-linked programs, which is a great improvement over what is commonly observed in other implementations.

Note that only the performance of single-process runs are compared. In general, the use of shared libraries provides a better system throughput because their text pages are shared among different processes, resulting in less memory consumption system-wide. Our results show that such an improved system throughput can be achieved with minimal penalty to the performance of individual process. We discuss the memory requirement of shared libraries in more detail in Section 5.1.

4. Quickstart

One of the major performance degradation in executing dynamically-linked programs is due to run-time symbol resolutions. The performance is affected in two ways. First, more instructions are executed because of the work performed by the run-time linker. Second, if the run-time linker needs to update the GOT's, more copy-on-write pages are needed and thus the residence size of the program is also increased. Since it is too expensive to trap data references, data symbols, including function pointers, are resolved by the run-time linker at program start-up time. As a result, in cases where function pointers are frequently used, as in many C++ programs, deferred binding cannot alleviate the run-time linker's work to a large extent. Furthermore, in some cases, deferred binding of symbols can cause the program to run even slower than immediate binding of all symbols at start-up time. These suggests that minimizing symbol resolution by the run-time linker is crucial in achieving better run-time performance.

A new mechanism called *quickstart* is developed to address this problem. Under quickstart, the static linker attempts to resolve all symbol references at static link-time, and if none of these references need to be updated at run-time, the run-time linker needs to perform virtually nothing other than the mapping of the required shared libraries. However, to achieve quickstart, the run-time linker must verify that the following two conditions are satisfied before running the user program [Hime90]. First, the shared libraries used at run-time must be the same as those used at static link--time, and second, all libraries are mapped into their pre-assigned default locations and none of the libraries in the list overlaps in virtual addresses.

To assist the run-time linker in verifying the first condition, each shared library is uniquely identified with a name, a time-stamp (the time when it was built), a checksum (the sum of the external interfaces and sizes of all common symbols), and a version string. At static link-time, all the information about the shared libraries used in building a certain binary are recorded in the binary itself, regardless of whether the binary is an executable or another shared library. In addition, symbols that have multiple definitions are recorded in the binary to enable the run-time linker to correctly resolve references to these symbols. To satisfy the second condition, we support a registry file that the static linker uses to determine the default virtual addresses for each individual shared library, which ensures that the shared libraries are assigned non-overlapping addresses.

As soon as the run-time linker detects that any of those two conditions is not met, it has to perform all the necessary symbol resolutions and relocations before running the program. This is known as *non-quickstart*.

Even with quickstart, the run-time linker still needs to perform symbol resolutions on those multiply-defined symbols, if they exist. In order to reduce the amount of work, we implemented *hidden* symbols. Hidden symbols can *only* be referenced within the shared library that they are defined in. Since these symbols cannot be referenced by other binaries, they directly contribute to making the list of multiply-defined symbols smaller, effectively alleviating the work of the run-time linker.

To handle those cases when the quickstart conditions are violated, e.g., when some of the shared libraries used in an executable are replaced, we developed a system that traverses the entire list of installed system binaries, performs necessary symbol resolutions and relocations, and at the end rewrites those modified binaries in place. All of these are typically done transparently during new software installation. We call this system *re-quickstart.* This scheme turns out to be very successful in maintaining quickstart status without re-linking any executable or shared library.

It should be pointed out that quickstart is *only* an optimization feature, and has no bearing on the correct execution of a program. Without quickstart, the run-time linker performs deferred binding by default, while immediate binding of symbols is also provided as an option.

Table 2 shows the performance improvement of quickstart and deferred binding over immediate binding. For each program, the total time spent in the run-time linker is measured. The percentage improvement is given in parenthesis. These five programs are chosen mainly because of their popularity of use by programmers. The other reason is their differences in characteristics such as the number of shared libraries used and the number of data symbols that needs to be resolved versus that of text symbols. *DiskView,* a huge X-window application written in C++, is a graphical interface for disk and file system manipulation used frequently by system administrators. *Insight,* a large multi-media application, is used for searching and browsing through on-line documents. *Gdiff,* a medium-sized X-window program, is a graphical differential file comparator. *Dbx,* a small C++ program, is SGI's source-level debugger. Last but not least, *ls,* listing the contents of a directory, is a widely used UNIX command.

program	**# of libs.**	**non-quickstart**		**quickstart**
		immed. binding	**deferred binding**	
diskview	17	3.66s	3.43s (6%)	2.06s (44%)
insight	14	3.36s	1.59s (53%)	0.13s (96%)
gdiff	6	0.28s	0.31s (-11%)	0.04s (84%)
dbx	3	0.32s	0.32s (0%)	0.02s (94%)
ls	1	0.04s	0.04s (0%)	0.00s (99%)
average			**10%**	**83%**

Table 2. Performance Improvement of Quickstart and Deferred Binding over Immediate Binding.

Overall, programs that are quickstarted spend at least 40% less time in the run-time linker than those with immediate binding. On average, there is an impressive 83% improvement. Most programs show that deferred binding of symbols has advantage over immediate binding [Saba90], and it is more apparent in cases where the number of shared libraries used is larger. This is probably because in general most library functions in a shared library are not used and thus need not be resolved at all. In the case of *gdiff,* however, the run-time linker was invoked many times over the course of execution, owing to many function calls that required symbol resolution, resulting in longer time spent in the run-time linker. In conclusion, whether deferred binding is used or not, it is far better to maintain the quickstart status, requiring the run-time linker to perform the least amount of work. It should be mentioned that even though only five programs are shown here, we have done substantial testing on hundreds of applications and the results are unanimously in favor of quickstart.

5. Reducing Memory Requirements

One of the goals of shared libraries is to reduce the system-wide physical memory requirement through increased sharing of text and read-only data of shared libraries among multiple processes. By reducing the system-wide memory requirement, the number of page faults is generally reduced, thus improving the overall performance.

While the use of shared library does reduce memory requirement, this reduction is sometimes offset by other overhead of dynamic linking. The first problem is that locality of functions within a shared library might be poor. A shared library is created by combining all functions it provides into a single module. When a shared library is linked with a process, the entire module is mapped into the process' address space and every function defined in the library is available to the process. Only those memory pages corresponding to functions that are actually invoked are loaded on demand. Typically, a process uses only a subset of the functions provided by a shared library. Unless these functions are all packed closely together within the shared library, many memory pages that are loaded might be partially filled with functions that are never or rarely invoked. As a result, memory utilization is sub-optimal.

The second problem is that data structures used by the run-time linker for symbol resolutions may sometimes be large. For example, the IRIX implementation of *xterm*, a typical X application, links with seven shared libraries containing a total of 5856 symbols. Note that these data structures are private to each process and are not shared. The effect of this hidden cost of dynamic linking has rarely been studied.

Section 5.1 describes *cord*, a tool we developed to improve function locality within a shared library. Sections 5.2 presents an optimization to reduce the amount of memory required by the run-time linker.

5.1. Procedure Repositioning

To reduce system-wide memory usage of instruction text space, a post-linking binary optimization tool called cord is implemented to reposition procedures in executables and shared libraries. Cord attempts to reduce run-time memory usage and achieve better instruction cache mapping by grouping frequently used procedures on the same page or adjacent pages, thus reducing the total number of memory pages needed during program execution. The procedure repositioning is controlled by a feedback file generated by profiling tools. The feedback file can also be hand-tuned to directly control the procedure placement.

Cord and the profiling system can be used to reposition procedures based on three different criteria: procedure invocation count, procedure cycle count and procedure density (procedure cycle count divided by procedure size). Using procedure invocation count to guide procedure repositioning improves memory usage by grouping frequently invoked procedures into adjacent pages. Procedure repositioning based on procedure density is intended to give smaller procedures a heavier weight in deciding the order of repositioning. Earlier experiments show that the effects on performance and memory usage of repositioning based on these three factors varies, but show very little difference. Our results indicate that cycle count yields a small advantage in the overall performance. Therefore, it is chosen as the default and is used in the measurement presented in this section.

The fact that libraries may be shared by different processes makes the performance modeling and measurement of shared libraries more difficult. In contrast, the benefit of procedure repositioning is more apparent for executables, because their execution pattern is more tractable. As a result, the benefit of procedure repositioning for shared libraries is an interesting issue worth investigating.

There has been a number of researches on binary repositioning [Hwu89, McFa89, Pett90], including procedure and basic block repositioning. However, none of those addresses memory usage and shared libraries performance, which is one of the important benefits of using cord.

To study the effect of procedure repositioning on memory usage, several key applications, including *Xsgi, 4Dwm* and *xwsh,* are selected as targets for memory usage optimization. Xsgi is an X window system server on IRIX. 4Dwm is the IRIX extended *Motif* window manager. Xwsh is an IRIX terminal emulator. Four libraries are chosen for this measurement because they tend to use more memory during most X sessions. The libraries are *libXm, libX11, libc,* and *libXt.*

The executables are profiled by a typical run. For libraries, the repositioning is based on profile data from 4Dwm and xwsh. The details of the profile run for each library are listed in Table 3. 4Dwm uses libXm, libX11, libc, libXt and others. Libraries used in xwsh include libX11, libc, and others.

The measurement is done after re-booting IRIX followed by a login session with 4Dwm, clock, two copies of xwsh, and a number of other X applications. It is important to note that this experiment is carried out on a loaded system. Many other processes are sharing those four libraries in addition to 4Dwm and xwsh. For example, libX11 and libXt is used by *xdm*, the X display manager, and libc is used

library	profiled with	
	4Dwm	xwsh
libXm	X	
libX11	X	X
libc	X	X
libXt	X	

Table 3. How Libraries are Profiled.

by all dynamically-linked processes.

The improvement of instruction memory usage for "cord'ed" executables is listed in Table 4. The figures show the total amount of memory used by each program. The results show that the instruction memory usage has been reduced by an average of 20.1%. These programs typically have long life spans and are frequently run throughout the entire X session. Therefore, the amount of memory usage reduction is quite significant.

The improvement of instruction memory usage for cord'ed shared libraries is listed in Table 5. Taken into account that those libraries are shared by other processes during the measurement, the average instruction memory reduction of 13.35% is quite impressive.

5.2. Pre-compute Run-time Information

As stated in Section 4, a successfully quickstarted process virtually eliminates any processing by the run-time linker. However, when quickstart fails, the run-time linker needs to perform symbol resolution. Typically, for each unresolved symbol, the run-time linker goes through the symbol table of each shared library and searches for a definition. Defined symbols in each symbol table can be accessed via a hash table, which is pre-computed and placed in the read-only segment of a shared library.

executable	original (K bytes)	cord'ed (K bytes)	memory usage reduction
xwsh	383	312	18.5%
Xsgi	466	410	12.0%
4Dwm	308	216	29.8%
average			**20.1%**

Table 4. Total Instruction Memory Usage for Procedure Repositioned Executables.

library	original (K bytes)	cord'ed (K bytes)	memory usage reduction
libXm	1212	914	24.6%
libX11	561	503	10.3%
libc	600	526	12.3%
libXt	351	329	6.2%
average			**13.35%**

Table 5. Total Instruction Memory Usage for Procedure Repositioned Shared Libraries.

program	No. of Symbols	Size of msym tables (Kbytes)
diskview	22505	87.9
insight	19468	76.0
gdiff	5745	22.4
dbx	5324	20.8
ls	2567	10.0

Table 6. Size of Msym Tables.

To further reduce symbol search time, the run-time linker caches each hash value it computes and keeps these values in a table called the *msym* table, which has a one-to-one corresponding relationship with the symbol table itself. Hence, the hash value of each symbol needs to be computed only once. Given an undefined symbol, the run-time linker first extracts the corresponding msym table for the hash value. It then goes through the hash tables of the main executable and each shared library, and looks for a definition of the symbol using this hash value. As a result, only simple table look-up is needed and run-time symbol resolution is very efficient.

Table 6 shows the sizes of the msym tables for the same set of programs listed in Table 2. For each program, the total number of symbol table entries from the program and the shared libraries is shown. The msym tables contain a four-byte hash value for each symbol table entry. The table shows that the amount of memory used by the msym tables is very large. Furthermore, the msym tables are private to each process. Therefore, in a typical workstation environment with 50 processes, the msym tables can easily take up several megabytes of memory. For example, the libc on IRIX has 2368 symbols. Since every program links with libc, the size of the msym tables for libc alone for 50 processes is about half a megabyte. A simple X application such as xterm requires 23 kilobytes for the msym tables per process.

Fortunately, the memory requirement of the msym tables can be greatly reduced without sacrificing the speed of symbol resolution. This can be achieved by pre-computing the hash value for each symbol and putting the entire msym table into the shared library when it is built. As a result, the sizes of the shared libraries increase, but the msym tables are shared among all processes and so the system-wide memory requirement is reduced. Furthermore, these tables are only loaded on demand. If a process is quickstarted, none of the symbol tables, hash tables, and msym tables is even touched.

6. Conclusion

This paper identifies three main areas that account for the performance loss of dynamically-linked programs, and presents the optimization techniques that we developed to solve these problems. All optimizations described in this paper have been implemented and the results have been proved to be very positive. In fact, we applied these optimizations to build the entire IRIX system based on shared libraries and no noticeable performance degradation can be observed.

We show that the cross-module optimization can eliminate indirect addressing for data references and function calls, and greatly reduces the run-time overhead of dynamically-linked programs. The quickstart mechanism virtually eliminates all run-time symbol resolution. By repositioning the procedures in a shared library and pre-computing the run-time data structures for symbol resolution, the system-wide memory requirement can be greatly reduced.

Most important of all, we show that the flexibility, usability, and other advantages of shared libraries can be achieved without sacrificing performance.

Acknowledgment

We would like to thank Sun Chan, Fred Chow, Seema Hiranandani, and Raymond Lo for their valuable comments on earlier drafts of this paper.

References

[MIPS90] *MIPS Processor Supplement for the System V Application Binary Interface,* Prentice Hall, Englewood Cliffs, NJ (1990).

[Syst90] *System V Application Binary Interface,* Prentice Hall, Englewood Cliffs, NJ (1990).

[SPEC91] *SPEC Newsletter*. December 1991.

[Arno86] Arnold, J. Q., “Shared Libraries on UNIX System V,” *Proc. Summer Usenix*, pp. 1-10 (1986).

[Ausl90] Auslander, M. A., “Managing Programs and Libraries in AIX Version 3 for RISC System/6000 processors,” *IBM Journal of Research and Development* Vol. **34**(1) pp. 98-104 (January 1990).

[Chow86] Chow, F., M. Himelstein, E. Killian, and L. Weber, “Engineering a RISC Compiler System,” *Proc. Compcon.*, pp. 132-137 (March 1986).

[Chow87] Chow, F., S. Correll, M. Himelstein, E. Killian, and L. Weber, “How Many Addressing Modes are Enough?,” *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 117-121 (October 1987).

[Cout92] Coutant, Cary A. and Michelle A. Ruscetta, “Shared Libraries for HP-UX,” *Hewlett-Packard Journal*, pp. 46-53 (June 1992).

[Ging87] Gingell, R. A., M. Lee, X. T. Dang, and M. S. Weeks, “Shared Libraries in SunOS,” *Proc. Summer Usenix*, pp. 131-145 (Summer 1987).

[Ging89] Gingell, Robert A., “Shared Libraries,” *UNIX Review* Vol. **7**(8) pp. 56-66 (August 1989).

[Hein93] Heinrich, Joseph, *MIPS R4000 User’s Manual,* Prentice Hall, Englewood Cliffs, NJ (1993).

[Hime87] Himelstein, Mark I., Fred C. Chow, and Kevin Enderby, "Cross-Module Optimizations: Its Implementation and Benefits," *Proc. Summer Usenix*, pp. 347-356 (June 1987).

[Hime90] Himelstein, Mark I., "An Implementation of Dynamic Shared Objects," *Unpublished paper*, MIPS Computer Systems, Inc., (August 1990).

[Hobb87] Hobbs, J., "Installed Shareable Images," *Dec. Professional* Vol. **6**(4) pp. 78-82 (April 1987).

[Hsu94] Hsu, Peter Yan-Tek, "Designing the TFP Microprocessor," *IEEE Micro*, pp. 23-33 (April 1994).

[Hwu89] Hwu, Wen-mei and Pohua Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *16th ISCA*, pp. 242-251 (1989).

[Kepp93] Keppel, David and Stephen Russell, "Faster Dynamic Linking for SPARC V8 and System V.4," Technical Report 93-12-08, University of Washington (1993).

[McFa89] McFarling, Scott, "Program Optimization for Instruction Caches," *International Symposium on Architecture Support for Programming Languages and Operating Systems*, pp. 183-191 (April 1989).

[Nels93] Nelson, Michael. N. and Graham Hamilton, "Higher Performance Dynamic Linking Through Caching," *Proc. Summer Usenix*, pp. 253-266 (June 1993).

[Orr93] Orr, Douglas B., John Bonn, Jay Lepreau, and Robert Mecklenburg, "Fast and Flexible Shared Libraries," *Proc. Summer Usenix*, pp. 237-251 (June 1993).

[Pett90] Pettis, Karl and Robert Hansen, "Profile Guided Code Positioning," *Proceedings of the SIGPLAN Programming Language Design and Implementation*, pp. 16-27 (June 1990).

[Saba90] Sabatella, Marc, "Issues in Shared Libraries Design," *Proc. Summer Usenix*, pp. 11-23 (June 1990).

W. Wilson Ho received his M.S. and Ph.D. degree in computer science from the University of California at Davis. He has been working in the compiler group in Mips, Inc. and Silicon Graphics, Inc. since 1992. His research interests include inter-procedural analysis, dynamic linking, link-time optimization, and debugging of distributed programs. He is also the author the GNU dynamic loader "dld". His email address is ho@sgi.com.

Wei-Chau Chang has been working in the compiler group in Mips, Inc. and Silicon Graphics, Inc. since 1992. His interests include compiler optimization for instruction cache, program instrumentation, and execution-driven simulation. He received his M.S. degree in computer science from the University of Illinois at Urbana-Champaign.

Lilian H. Leung received her M.S. degree in Computer Science from Stanford University in 1986. She has been developing compiler at Mips, Inc, and Silicon Graphics, Inc. since 1988. Before that, she was at Informix Software, Inc. designing database languages. Her most recent work focus is on static and run-time linking. She was the original writer of the SGI/Mips' run-time loader.