Faster Dynamic Linking for SPARC V8 and System V.4

David Keppel and Stephen Russell

University of Washington
Technical Report 93-12-08

Abstract

Dynamic linking is again becoming common and processor pipeline depths are increasing. These factors make it worth retuning dynamic linker implementations. This note examines the current dynamic linking scheme used by System V.4 running on SPARC V8 processors and shows four alternative implementations that can improve the performance of calls to dynamically-linked functions. Improvements come from reducing the number of instructions that must be executed to perform a call; from reducing the dynamically cached size of the linkage table; and by performing control transfers using instruction immediates instead of indirecting off of a register. In the best case, the overhead of calling a dynamically linked instruction is reduced from four instructions with a register indirect to a single instruction and no register indirect. The paper also discusses issues and techniques that may be useful for other systems that perform dynamic using similar techniques. This note also briefly considers dynamic relinking.

1 Introduction

Dynamic linking [GLDW87, Sab90, See90, HO91] introduces new code in to a running program. It is used for a number of reasons: to reduce the disk space needed to store a compiled program; so that each time a program is run it will get the latest version of some library; so that an application will get a library specific to the machine [Sys90]; so that a running program can delete outdated versions of code and replace them with new versions [SF89]; and so that user commands can be translated to native code and executed efficiently [CAK+81, SR85, Hol87, NG87].

Dynamic linking is implemented many ways. Here, we are concerned with the case where a programmer writes a *caller* that makes use of a *called* routine. The compiler does not produce a direct call to the called routine. Instead, it produces a call to the dynamic linker. When the code is run, the dynamic linker is invoked to find and load the called routine, then modify some caller information so that further calls will transfer directly to the called routine.

This note focuses on the mechanism by which a call to the dynamic linker is turned in to a call to the dynamically-linked function. The most common mechanisms are:

- DIRECT: The caller is compiled so that each call goes directly to the dynamic linker. The dynamic linker patches the caller to instead call the linked function.
- ADDRESS TABLE: The caller is compiled so that each call loads the target address from a link table. Table entries initially point to the dynamic linker. The dynamic linker updates link table entries to point to dynamically-linked functions.
- CODE TABLE: The caller is compiled so that each call goes to stub code in a private segment, usually in the program's data segment. Stubs are initialized to jump to the dynamic linker. The dynamic linker patches stubs to instead jump to the dynamically-linked function.

Each mechanism has tradeoffs. For example,

Please address correspondence to the first author at the University of Washington, Department of CS&E, FR-35; Seattle, Washington 98195.

direct calls run at full speed but the caller's text is modified and thus cannot be shared.[1] Also, indirect calls through a pointer to a function are slightly trickier since the procedure address is in a register, not in an instruction immediate.

The code table approach requires two control transfers instead of one and requires dynamic code patching. However, it is useful with separate compilation because text is sharable and because calls can be compiled without distinguishing between calls to statically-linked and dynamically-linked routines. In contrast, using an address table either forces all calls to transfer via the indirect table, or requires linker support so that calls to statically-linked routines can use a fast calling sequence, while dynamically-linked calls can indirect via the address table.

System V.4 [Sys90] running on SPARC V8 [SPA91] processors uses a code table [SPA90]. The call overhead is typically low with today's applications and processors. However the overhead may grow as dynamic linking is used more widely and as processor architectures are more deeply pipelined and perform more speculative execution.

The remainder of this note is arranged as follows: Section 2 considers one widely-used implementation. Section 3 describes four alternative implementations. Section 4 considers the requirements for systems that perform dynamic relinking. Finally, Section 5 concludes.

2 Today's Implementation

Currently, application code is compiled in to a read-only text segment. Statically-linked calls are linked to jump directly within the text segment. Dynamically-linked calls are linked statically to jump to entries in a dynamic link table (DLT). The DLT is in an executable part of the data segment. Thus, the text segment is read-only and sharable, and the changes made by the dynamic linker are to private segments.

The DLT is a fixed distance from the application's `call` instruction.[2] There is one DLT entry for each function that can be dynamically linked. Logically, each dynamically-linkable function is assigned a number i and calls to routine i jump to the i'th DLT entry.

When a segment is loaded, the first N words of the DLT are initialized with code that calls the dynamic linker. The remaining words make up the DLT entries. Each DLT entry is initialized with code that loads a register with an entry number and transfers to the start of segment. A typical DLT entry is [SPA90]:

```
sethi .-DLT, %g1  // Function number
ba,a DLT          // Call dynamic linker
nop
```

Suppose that this DLT entry corresponds to the routine `foo`. The dynamic linker maps `foo` in to memory and initializes `foo`'s DLT as above. Then it patches the caller's DLT entry with code that will jump to `foo`. Finally, the dynamic linker jumps to `foo`. Further calls to `foo` jump to the DLT entry, which jumps to `foo`. Returns to the caller go directly without using the DLT entry.

During dynamic linking, the dynamic linker must patch the DLT entry in a way that is immune to race conditions. Suppose that while the dynamic linker is patching the DLT entry for `foo`, a signal arrives and the signal handler needs to call `foo`. The code in the DLT entry could then be executed while the dynamic linker is patching the DLT entry. Simply turning off interrupts does not solve the problem, as the same condition arises when multiple threads of control are executing in the same address space.

The potential race is avoided by ensuring that each DLT entry is updated so that every partially-patched sequence is legal. First, the `nop` is replaced with a `jmpl` instruction, and the instruction cache is flushed or updated. Although branches in branch delay slots are normally invalid, this sequence is valid, because the delay slot instruction after a `ba,a` is never executed.

```
sethi .-DLT, %g1
ba,a DLT
jmpl %g1+%lo(foo), %g0
```

If the DLT entry is executed at this point, it will transfer control to the dynamic linker, which will link `foo` and proceed normally. Note that as long as `foo` is always loaded at the same address, overwriting is *idempotent*: that is, the DLT entry can be written an arbitrary number of times with the same effect as if it were written just once.

[1] Systems can sometimes arrange to have a dynamically-linked image shared among processes that use the same linkages [KK92, NH93].

[2] Some other systems use a fixed offset from a base register that points to the data segment.

After the dynamic linker writes the `jmpl`, the `ba,a` is replaced with an instruction that sets `%g1` to have the high address bits of `foo`. This, in combination with the `jmpl`, is the final code that transfers control to `foo`:

```
sethi .-DLT, %g1
sethi %hi(foo), %g1
jmpl %lo(foo)+%g1, %g0
```

There are several things to note about this code fragment. First, it starts with an unused `sethi`, left over from the call to the dynamic linker. Thus, the sequence could, in theory, be one instruction shorter. However, any shorter sequence must retain the property of idempotent updates. A second observation is that the `jmpl` always executes the following (delay slot) instruction. However, by construction, the following instruction is always the first `sethi` of the following DLT entry (or `nop` for the last DLT entry), and is thus safe to execute.

Typically, only the first call through a DLT entry invokes the dynamic linker. In the normal case of calling an already-linked function, the above implementation executes four instructions. Cache locality is slightly better than that, because each DLT entry is three words; the `jmpl`'s delay slot executes the first instruction of the following DLT entry. The current implementation also performs an indirect branch off a register, which may interfere with branch prediction on some machines.

3 Other Implementations

3.1 Variation 1 – Branch Ordering

Here, we propose an implementation that takes fewer instructions once the called routine has been linked. Initially, the DLT entry contains a branch to the stub that calls the dynamic linker, plus a `sethi` to describe which DLT entry was called.

```
nop
ba DLT            // Call dynamic linker
sethi .-DLT, %g1  // Function number
```

The dynamic linker first replaces the `nop` with a `sethi` that sets the high bits of the function address. The DLT entry can still be executed safely, because the `sethi` in the branch delay slot will correctly overwrite `%g1` to tell the dynamic linker which function was originally called.

```
sethi %hi(foo), %g1
ba DLT
sethi .-DLT, %g1
```

Next, the `ba` is replaced with a `jmpl` to `foo`. The sequence of instructions is still correct because the value of `%g1` is not needed after the `jmpl` and may, therefore, be clobbered safely.

```
sethi %hi(foo), %g1
jmpl %lo(foo)+%g1, %g0
sethi .-DLT, %g1
```

This sequence is faster than the original because it executes fewer instructions. However, it still occupies three words in the cache and performs an indirect branch off of a register.

3.2 Variation 2 – Immediate Branches

High-performance SPARC implementations will run faster if the branch target can be determined soley by looking at the opcode instead of needing to read a register value. Since the called function may be linked at an arbitrary place in the address space, the only SPARC immediate branch instruction that will do is the `call` instruction: `call` can jump to any part of a 32-bit address space. All other immediate branch instructions branch to targets that must be within 8Mb of the branch instruction. Since `call` clobbers the return program counter (noted `%rpc`), it is necessary to save and restore the return program counter around the call. Thus, the final code should look like:

```
movl %rpc, %g1
call foo
movl %g1, %rpc
```

We build the unlinked DLT entry using the same sequence of instructions, but the `call` initially goes to the dynamic linker. Unfortunately, the sequence of three instructions above does not uniquely describe to the dynamic linker which function was called (or, equivalently, which DLT entry to update).

In *most* cases, the dynamic linker could figure out what routine was called, because `%rpc` points 8 bytes past the instruction that initiated the call. However, there are circumstances when it is impossible to deduce what procedure was called:

- A call indirect through `%g1`, which is clobbered by the DLT entry.

- A call indirect through a register that is clobbered in the delay slot of the call instruction.
- A call where the return pc register is set to anywhere other than 8 bytes past the call instruction.
- A delay-slot instruction that overwrites the call instruction.

Therefore, it is still necessary for the DLT entry to identify itself. This can be done, at some space cost, by having the DLT entry's `call` branch to code that identifies itself and then calls the dynamic linker. Now the DLT consists of three segments instead of two: the N words of prologue, some number of DLT entries, and an equal number of self-identifying code fragments. Suppose the DLT entry is number 17, then the entry starts off as:

```
movl %rpc, %g1
call DLT_17
movl %g1, %rpc
```

The self-identifying code, `DLT_17` calls the dynamic linker and sets the self-identification in the delay slot:

```
ba DLT
sethi .-DLT, %g1
```

When the dynamic linker wishes to update the DLT entry, it simply overwrites the `call` instruction with the call to `foo`. That is the only change to the dynamic link table entry, so the sequence of instructions is always correct.

This scheme executes three instructions on each call. Here, jumps use only immediates, and thus can have better performance for highly-pipelined SPARC implementations. However, it also requires five words per DLT entry, instead of the three used by previous schemes. Note, though, that two of the five words are used only during linking, so the normal cost is just three instructions.

Instead of putting the self-identifying code in a separate part of the link table, it is instead possible to put it directly in the DLT entry. However, idempotent sequences are four instructions. Moreover, calls to dynamically-linked code have worse cache locality because only three of the four table entries are used frequently. Larger DLT entries thus cause cache fragmentation.[3]

[3] A further problem is that the SPARC ABI allows at most 3 words per DLT entry [SPA90].

3.3 Variation 3 – Compacting

Using the cache space/fragmentation observation noted in the preceding section, we would like to make each DLT entry smaller. We can do so if the `jmpl` in each entry is followed by the `sethi` of the following dynamic link table entry. Here is the final code we would like in a linked DLT entry:

```
sethi %hi(foo), %g1
jmpl %g1+%lo(foo), %g0
```

We can achieve this by using the secondary table idea of variation 2. Each DLT entry starts out as a branch to a self-identifying fragment that branches to the dynamic linker. DLT entry number 17 starts off as:

```
ba,a DLT_17
nop
```

The self-identifying code, `DLT_17` is the same as in the previous section. The first modification replaces the `nop` with a `jmpl`. This sequence is still well-behaved:

```
ba,a DLT_17
jmpl %g1+%lo(foo), %g0
```

Next, the `ba,a` is replaced with a `sethi`.

```
sethi %hi(foo), %g1
jmpl %g1+%lo(foo), %g0
```

This variation consumes two instructions of dynamic (cache) space, three instruction times to execute, and four instructions of total memory. The principal win here is in the dynamic reduction of DLT size, which can improve cache utilization. Note that this variation suffers from the use of indirection off a register.

An alternative is to allocate three words for the DLT entry. The third instruction appears as a delay slot instruction in the final sequence. Initially, the third instruction is a `nop`. If the first word of the dynamically-linked routine is anything other than a control transfer, it can replace the `nop` and the jump can proceed directly to the second instruction of the dynamically-linked routine. This *hoisting* effectively reduces the overhead of an indirect call to just two instructions, though cache locality is worse than the form that uses just two instructions per DLT entry. Note that hoisting must be performed before the DLT entry is patched, in order to ensure that the change is idempotent.

3.4 Variation 4 – Near Branches

In some cases, the distance from the DLT entry to the called routine is within the range covered by branch instructions. In these cases, a simple branch instruction can be used; it costs just two instructions and uses an instruction immediate instead of a register indirect. This scheme avoids both the register save and restore overhead of using a `call` instruction, and also the register indirection problems of the `sethi/jmpl` pair.

A problem with this scheme is tht it only works when the distance from DLT entry to the called function is within the branch immediate distance, +/–8Mb on a SPARC. Thus, the dynamic linker must decide on an entry-by-entry basis whether the DLT entry can be implemented using near branches. However, this scheme requires only a single idempotent update to the DLT entry and can be used with any initial DLT entry format. Thus, it can be applied as an optimization to any of the preceding implementations.

Hoisting can reduce the effective cost of this scheme to just one instruction and can potentially use as little as two instructions of cache space. Here, hoisting is done *after* the DLT entry is patched. At first, the DLT entry jumps to the first instruction of the dynamically-linked routine:

```
ba,a foo  // Call first instruction
...       // Squashed
```

Next, the `nop` is overwritten with the hoisted instruction:

```
ba,a foo  // Call first instruction
hoisted   // Squashed
```

Finally, the `ba,a` is replaced with a `ba` to the second instruction:

```
ba foo+4  // Call second instruction
hoisted
```

Care is needed, however, since the call to the first instruction can be in range of the branch, while the second instruction might be out of range.

4 Relinking

In some circumstances, it is useful to be able to unlink dynamically-linked code and link in a new version of that code [SF89], or move the existing code to e.g., reduce cache conflicts. This dynamic change of the linkage bindings is called *relinking*.

An "obvious" way to implement relinking is to first unlink the old version, then use the existing dynamic linker mechanism to link to the new code. Patching the DLT entry for dynamic unlinking is straightforward: an instruction in the DLT entry is updated to jump to the dynamic linker. The patching is race-free because unlinking needs to modify only one instruction.

Relinking, however, can be tricky. First, the application will fail if the library is unmapped when there are threads in either the library or the DLT entries that jump to the library. Second, even if the old library is left mapped, updating DLT entries can create races. For example, a thread may execute the `sethi` from a DLT entry, get suspended, a second thread or a signal handler can rewrite the DLT entry, and then the first thread can execute a `jmpl` from the new sequence. Avoiding this race requires simultaneous update of both the `sethi` and the `jmpl`. The designs presented in Sections 3.2 and 3.4 are race-free because only a single instruction is changed to effect relinking.

With multi-instruction sequences, relinking is still sometimes possible, but it is necessary to ensure that all threads are outside of the corresponding DLT entries when the relinking is performed. If all threads are outside of the DLT entries, the effect of simultaneous update can be achieved by first unlinking (replacing one of the instructions with a `ba,a`), then using the normal dynamic linking procedure to link to the new code.

5 Summary

The current dynamic link table entry layout requires four instruction times and three instructions of cache space for each call to a dynamically-linked function. In addition, one of the instructions is an indirect jump off of a register, which hurts performance in high-performance SPARC implementations.

This paper describes four alternative implementations. The implementations improve over the current implementation by reducing the number of instructions that must be executed to perform a call, by reducing the the dynamic cache size of the dynamic link table, and by performing control transfers using instruction immediates instead of indirecting off of register values. In the best case, the ef-

fective overhead of dynamic linking can be reduced from four instructions and a register indirect to a single instruction and no register indirect.

Although the examples are presented for dynamic linking on a SPARC, the techniques presented here are often applicable to other architectures.

6 Acknowledgements

Thanks to David Poole for discussing these ideas with us.

References

[CAK+81] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. Alorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Transactions on Databse Systems*, 6(1):70–94, March 1981.

[GLDW87] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared Libraries in SunOS. *Summer USENIX*, pages 131–145, 1987.

[HO91] W. Wilson Ho and Rondald A Olsson. An Approach to Genuine Dynamic Linking. *Software-Practice and Experience*, 21(4):375–390, April 1991.

[Hol87] Gerard J. Holzmann. PICO - A Picture Editor. *AT&T Technical Journal*, 66(2):2–13, March 1987.

[KK92] James Kempf and Peter B. Kessler. Cross-Address Space Dynamic Linking. *IEEE Proceedings of the International Workshop on Objeect-Orientation in Operating Systems (IWOOOS)*, September 1992.

[NG87] David Notkin and William G. Griswold. Enhancement through Extension: The Extension Interpreter. *Procedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 45–55, June 1987.

[NH93] Michael N. Nelson and Graham Hamilton. Higher Performance Dynamic Linking Through Caching. Technical Report TR-93-15, Sun Micorsystems Laboratories Inc., April 1993.

[Sab90] Marc Sabatella. Issues in Shared Libraries Design. *USENIX Summer Conference Proceedings*, pages 11–23, June 1990.

[See90] Donn Seeley. Shared Libraries as Objects. *USENIX Summer Conference Proceedings*, pages 25–37, June 1990.

[SF89] M. E. Segal and O. Frieder. Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime. *Software Maintenance: Research and Practice*, 1:59–79, 1989.

[SPA90] *System V Application Binary Interface SPARC Processor Supplement.* Prentice-Hall, 1990.

[SPA91] The SPARC Architecture Manual Version 8. Technical Report Sun Microsystems Part Number 800-1399-12, Sun Microsystems, January 1991.

[SR85] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. Technical Report Memoradum No. UCB/ERL 85/95, Electronics Research Laboratory, University of California, Berkeley, 15 November 1985.

[Sys90] *System V Application Binary Interface*. Prentice-Hall, 1990.