

Prelink

Jakub Jelínek
Red Hat, Inc.
jakub@redhat.com

March 4, 2004

Abstract

Prelink is a tool designed to speed up dynamic linking of ELF programs on various Linux architectures. It speeds up start up of OpenOffice.org 1.1 by 1.8s from 5.5s on 651MHz Pentium III.

1 Preface

In 1995, Linux changed its binary format from a .out to ELF. The a.out binary format was very inflexible and shared libraries were pretty hard to build. Linux's shared libraries in a.out are position dependent and each had to be given a unique virtual address space slot at link time. Maintaining these assignments was pretty hard even when there were just a few shared libraries, there used to be a central address registry maintained by humans in form of a text file, but it is certainly impossible to do these days when there are thousands of different shared libraries and their size, version and exported symbols are constantly changing. On the other side, there was just minimum amount of work the dynamic linker had to do in order to load these shared libraries, as relocation handling and symbol lookup was only done at link time. The dynamic linker used the `uselib` system call which just mapped the named library into the address space (with no segment or section protection differences, the whole mapping was writable and executable).

The ELF¹ binary format is one of the most flexible binary formats, its shared libraries are easy to build and there is no need for a central assignment of virtual address space slots. Shared libraries are position independent and relocation handling and symbol lookup are done partly at the time the executable is created and partly at runtime. Symbols in shared libraries can be overridden at runtime by preloading a new shared library defining those symbols or without relinking an executable by adding symbols to a shared library which is searched up earlier during symbol lookup or by adding new dependent shared libraries to a library used by the program. All these improvements have their price, which is a slower program startup, more non-shareable memory per process and runtime cost associated with position independent code in shared libraries.

Program startup of ELF programs is slower than startup of a.out programs with shared libraries, because the dynamic linker has much more work to do before calling program's entry point. The cost of loading libraries is just slightly bigger, as ELF shared libraries have typically separate read-only and writable segments, so the dynamic linker has to use different memory protection for each segment. The main difference is in relocation handling and associated symbol lookup. In the a.out format there was no relocation handling or symbol lookup at runtime. In ELF, this cost is much more important today than it used to be during a.out to ELF transition in Linux, as especially GUI programs keep constantly growing and start to use more and more shared libraries. 5 years ago programs using more than 10 shared libraries were very rare, these days most of the GUI programs link against around 40 or more shared and in extreme cases programs use even more than 90 shared libraries. Every shared library adds its set of dynamic relocations to the cost and enlarges symbol search scope, so in addition to doing more symbol lookups, each symbol lookup the application has to perform is on average more expensive. Another factor increasing the cost is the length of symbol names which have to be compared when finding symbol in the symbol hash table of a shared library. C++ libraries tend to have extremely long symbol names and unfortunately the new C++ ABI puts namespaces and class names first and method names last in the mangled names, so often symbol names differ only in last few bytes of very long names.

Every time a relocation is applied the entire memory page containing the address which is written to must be loaded into memory. The operating system does a copy-on-write operation which also has the consequence that the physical

¹As described in generic ABI document [1] and various processor specific ABI supplements [2], [3], [4], [5], [6], [7], [8].

memory of the memory page cannot anymore be shared with other processes. With ELF, typically all of program's Global Offset Table, constants and variables containing pointers to objects in shared libraries, etc. are written into before the dynamic linker passes control over to the program.

On most architectures (with some exceptions like AMD64 architecture) position independent code requires that one register needs to be dedicated as PIC register and thus cannot be used in the functions for other purposes. This especially degrades performance on register-starved architectures like IA-32. Also, there needs to be some code to set up the PIC register, either invoked as part of function prologues, or when using function descriptors in the calling sequence.

Prelink is a tool which (together with corresponding dynamic linker and linker changes) attempts to bring back some of the a.out advantages (such as the speed and less COW'd pages) to the ELF binary format while retaining all of its flexibility. In a limited way it also attempts to decrease number of non-shareable pages created by relocations. Prelink works closely with the dynamic linker in the GNU C library, but probably it wouldn't be too hard to port it to some other ELF using platforms where the dynamic linker can be modified in similar ways.

2 Caching of symbol lookup results

Program startup can be speeded up by caching of symbol lookup results². Many shared libraries need more than one lookup of a particular symbol. This is especially true for C++ shared libraries, where e.g. the same method is present in multiple virtual tables or *RTTI* data structures. Traditionally, each ELF section which needs dynamic relocations has an associated `.rela*` or `.rel*` section (depending on whether the architecture is defined to use `RELA` or `REL` relocations). The relocations in those sections are typically sorted by ascending `r_offset` values. Symbol lookups are usually the most expensive operation during program startup, so caching the symbol lookups has potential to decrease time spent in the dynamic linker. One way to decrease the cost of symbol lookups is to create a table with the size equal to number of entries in dynamic symbol table (`.dynsym`) in the dynamic linker when resolving a particular shared library, but that would in some cases need a lot of memory and some time spent in initializing the table. Another option would be to use a hash table with chained lists, but that needs both extra memory and would also take extra time for computation of the hash value and walking up the chains when doing new lookups. Fortunately, neither of this is really necessary if we modify the linker to sort relocations so that relocations against the same symbol are adjacent. This has been done first in the Sun linker and dynamic linker, so the GNU linker and dynamic linker use the same ELF extensions and linker flags. Particularly, the following new ELF dynamic tags have been introduced:

```
#define DT_RELACOUNT 0x6ffffff9
#define DT_RELCOUNT 0x6ffffffa
```

New options `-z combrelloc` and `-z nocombrelloc` have been added to the linker. The latter causes the previous linker behavior, i.e. each section requiring relocations has a corresponding relocation section, which is sorted by ascending `r_offset`. `-z combrelloc`³ instructs the linker to create just one relocation section for dynamic relocations other than symbol jump table (PLT) relocations. This single relocation section (either `.rela.dyn` or `.rel.dyn`) is sorted, so that relative relocations come first (sorted by ascending `r_offset`), followed by other relocations, sorted again by ascending `r_offset`. If more relocations are against the same symbol, they immediately follow the first relocation against that symbol with lowest `r_offset`.⁴ The number of relative relocations at the beginning of the section is stored in the `DT_RELACOUNT` resp. `DT_RELCOUNT` dynamic tag.

The dynamic linker can use the new dynamic tag for two purposes. If the shared library is successfully mapped at the same address as the first `PT_LOAD` segment's virtual address, the load offset is zero and the dynamic linker can avoid all the relative relocations which would just add zero to various memory locations. Normally shared libraries are linked with first `PT_LOAD` segment's virtual address set to zero, so the load offset is non-zero. This can be changed through a linker script or by using a special prelink option `--reloc-only` to change the base address of a shared library. All prelinked shared libraries have non-zero base address as well. If the load offset is non-zero, the dynamic linker can still make use of this dynamic tag, as relative relocation handling is typically way simpler than handling other

²Initially, this has been implemented in the `prelink` tool and `glibc` dynamic linker, where `prelink` was sorting relocation sections of existing executables and shared libraries. When this has been implemented in the linker as well and most executables and shared libraries are already built with `-z combrelloc`, the code from `prelink` has been removed, as it was no longer needed for most objects and just increasing the tool's complexity.

³`-z combrelloc` is the default in GNU linker versions 2.13 and later.

⁴In fact the sorting needs to take into account also the type of lookup. Most of the relocations will resolve to a PLT slot in the executable if there is one for the lookup symbol, because the executable might have a pointer against that symbol without any dynamic relocations. But e.g. relocations used for the PLT slots must avoid these.

relocations (since symbol lookup is not necessary) and thus it can handle all relative relocations in a tight loop in one place and then handle the remaining relocations with the fully featured relocation handling routine. The second and more important point is that if relocations against the same symbol are adjacent, the dynamic linker can use a cache with single entry.

The dynamic linker in glibc, if it sees statistics as part of the LD_DEBUG environment variable, displays statistics which can show how useful this optimization is. Let's look at some big C++ application, e.g. konqueror. If not using the cache, the statistics looks like this:

```
18000: runtime linker statistics:
18000: total startup time in dynamic loader: 270886059 clock cycles
18000: time needed for relocation: 266364927 clock cycles (98.3%)
18000: number of relocations: 79067
18000: number of relocations from cache: 0
18000: number of relative relocations: 31169
18000: time needed to load objects: 4203631 clock cycles (1.5%)
```

This program run is with hot caches, on non-prelinked system, with lazy binding. The numbers show that the dynamic linker spent most of its time in relocation handling and especially symbol lookups. If using symbol lookup cache, the numbers look different:

```
18013: total startup time in dynamic loader: 132922001 clock cycles
18013: time needed for relocation: 128399659 clock cycles (96.5%)
18013: number of relocations: 25473
18013: number of relocations from cache: 53594
18013: number of relative relocations: 31169
18013: time needed to load objects: 4202394 clock cycles (3.1%)
```

On average, for one real symbol lookup there were two cache hits and total time spent in the dynamic linker decreased by 50%.

3 Prelink design

Prelink was designed, so that it requires as few ELF extensions as possible. It should not be tied to a particular architecture, but should work on all ELF architectures. During program startup it should avoid all symbol lookups which, as has been shown above, are very expensive. It needs to work in an environment where shared libraries and executables are changing from time to time, whether it is because of security updates or feature enhancements. It should avoid big code duplication between the dynamic linker and the tool. And prelinked shared libraries need to be usable even in non-prelinked executables, or when one of the shared libraries is upgraded and the prelinking of the executable has not been updated.

To minimize the number of performed relocations during startup, the shared libraries (and executables) need to be relocated already as much as possible. For relative relocations this means the library needs to be loaded always at the same base address, for other relocations this means all shared libraries with definitions those relocations resolve to (often this includes all shared libraries the library or executable depends on) must always be loaded at the same addresses. ELF executables (with the exception of *Position Independent Executables*) have their load address fixed already during linking. For shared libraries, prelink needs something similar to a.out registry of virtual address space slots. Maintaining such registry across all installations wouldn't scale well, so prelink instead assigns these virtual address space slots on the fly after looking at all executables it is supposed to speed up and all their dependent shared libraries. The next step is to actually relocate shared libraries to the assigned base address.

When this is done, the actual prelinking of shared libraries can be done. First, all dependent shared libraries need to be prelinked (prelink doesn't support circular dependencies between shared libraries, will just warn about them instead of prelinking the libraries in the cycle), then for each relocation in the shared library prelink needs to look up the symbol in natural symbol search scope of the shared library (the shared library itself first, then breadth first search of all dependent shared libraries) and apply the relocation to the symbol's target section. The symbol lookup code in the

dynamic linker is quite complex and big, so to avoid duplicating all this, `prelink` has chosen to use dynamic linker to do the symbol lookups. Dynamic linker is told via a special environment variable it should print all performed symbol lookups and their type and `prelink` reads this output through a pipe. As one of the requirements was that prelinked shared libraries must be usable even for non-prelinked executables (duplicating all shared libraries so that there are pristine and prelinked copies would be very unfriendly to RAM usage), `prelink` has to ensure that by applying the relocation no information is lost and thus relocation processing can be cheaply done at startup time of non-prelinked executables. For RELA architectures this is easier, because the content of the relocation's target memory is not needed when processing the relocation.⁵ For REL architectures this is not the case. `prelink` attempts some tricks described later and if they fail, needs to convert the REL relocation section to RELA format where addend is stored in the relocation section instead of relocation target's memory.

When all shared libraries an executable (directly or indirectly) depends on are prelinked, relocations in the executable are handled similarly to relocations in shared libraries. Unfortunately, not all symbols resolve the same when looked up in a shared library's natural symbol search scope (i.e. as it is done at the time the shared library is prelinked) and when looked up in application's global symbol search scope. Such symbols are herein called *conflicts* and the relocations against those symbols *conflicting relocations*. Conflicts depend on the executable, all its shared libraries and their respective order. They are only computable for the shared libraries linked to the executable (libraries mentioned in DT_NEEDED dynamic tags and shared libraries they transitively need). The set of shared libraries loaded via `dlopen(3)` cannot be predicted by `prelink`, neither can the order in which this happened, nor the time when they are unloaded. When the dynamic linker prints symbol lookups done in the executable, it also prints conflicts. `Prelink` then takes all relocations against those symbols and builds a special RELA section with conflict fixups and stores it into the prelinked executable. Also a list of all dependent shared libraries in the order they appear in the symbol search scope, together with their checksums and times of prelinking is stored in another special section.

The dynamic linker first checks if it is itself prelinked. If yes, it can avoid its preliminary relocation processing (this one is done with just the dynamic linker itself in the search scope, so that all routines in the dynamic linker can be used easily without too many limitations). When it is about to start a program, it first looks at the library list section created by `prelink` (if any) and checks whether they are present in symbol search scope in the same order, none have been modified since prelinking and that there aren't any new shared libraries loaded either. If all these conditions are satisfied, prelinking can be used. In that case the dynamic linker processes the fixup section and skips all normal relocation handling. If one or more of the conditions are not met, the dynamic linker continues with normal relocation processing in the executable and all shared libraries.

4 Collecting executables and libraries which should be prelinked

Before the actual work can start the `prelink` tool needs to collect the filenames of executables and libraries it is supposed to prelink. It doesn't make any sense to prelink a shared library if no executable is linked against it because the prelinking information will not be used anyway. Furthermore, when `prelink` needs to do a REL to RELA conversion of relocation sections in the shared library (see later) or when it needs to convert SHT_NOBITS PLT section to SHT_PROGBITS, a prelinked shared library might grow in size and so prelinking is only desirable if it will speed up startup of some program. The only change which might be useful even for shared libraries which are never linked against, only loaded using `dlopen`, is relocating to a unique address. This is useful if there are many relative relocations and there are pages in the shared library's writable segment which are never written into with the exception of those relative relocations. Such shared libraries are rare, so `prelink` doesn't handle these automatically, instead the administrator or developer can use `prelink --reloc-only=ADDRESS` to relocate it manually. Prelinking an executable requires all shared libraries it is linked against to be prelinked already.

`Prelink` has two main modes in which it collects filenames. One is *incremental prelinking*, where `prelink` is invoked without the `-a` option. In this mode, `prelink` queues for prelinking all executables and shared libraries given on the command line, all executables in directory trees specified on the command line, and all shared libraries those executables and shared libraries are linked against. For the reasons mentioned earlier a shared library is queued only if a program is linked with it or the user tells the tool to do it anyway by explicitly mentioning it on the command line. The second mode is *full prelinking*, where the `-a` option is given on the command line. This in addition to incremental prelinking queues all executables found in directory trees specified in `prelink.conf` (which typically includes all or most directories where system executables are found). For each directory subtree in the config file the user can specify whether symbolic links to places outside of the tree are to be followed or not and whether searching should continue even across filesystem boundaries.

⁵Relative relocations on certain RELA architectures use relocation target's memory, either alone or together with `r_addend` field.

175 There is also an option to blacklist some executables or directory trees so that the executables or anything in the
176 directory trees will not be prelinked. This can be specified either on the command line or in the config file.

177 Prelink will not attempt to change executables which use a non-standard dynamic linker ⁶ for security reasons,
178 because it actually needs to execute the dynamic linker for symbol lookup and it needs to avoid executing some random
179 unknown executable with the permissions with which prelink is run (typically root, with the permissions at least
180 for changing all executables and shared libraries in the system). The administrator should ensure that `prelink.conf`
181 doesn't contain world-writable directories and such directories are not given to the tool on the command line either, but
182 the tool should be distrustful of the objects nevertheless.

183 Also, prelink will not change shared libraries which are not specified directly on the command line or located in the
184 directory trees specified on the command line or in the config file. This is so that e.g. prelink doesn't try to change
185 shared libraries on shared networked filesystems, or at least it is possible to configure the tool so that it doesn't do it.

186 For each executable and shared library it collects, prelink executes the dynamic linker to list all shared libraries it
187 depends on, checks if it is already prelinked and whether any of its dependencies changed. Objects which are already
188 prelinked and have no dependencies which changed don't have to be prelinked again (with the exception when e.g.
189 virtual address space layout code finds out it needs to assign new virtual address space slots for the shared library or
190 one of its dependencies). Running the dynamic linker to get the symbol lookup information is a quite costly operation
191 especially on systems with many executables and shared libraries installed, so prelink offers a faster `-q` mode. In
192 all modes, prelink stores modification and change times of each shared library and executable together with all
193 object dependencies and other information into `prelink.cache` file. When prelinking in `-q` mode, it just compares
194 modification and change times of the executables and shared libraries (and all their dependencies). Change time is
195 needed because prelink preserves modification time when prelinking (as well as permissions, owner and group). If
196 the times match, it assumes the file has not changed since last prelinking. Therefore the file can be skipped if it
197 already prelinked and none of the dependencies changed. If any time changed or one of the dependencies changed, it
198 invokes the dynamic linker the same way as in normal mode to find out real dependencies, whether it has been prelinked
199 or not etc. The collecting phase in normal mode can take a few minutes, while in quick mode usually takes just a few
200 seconds, as the only operation it does is it calls just lots of `stat` system calls.

5 Assigning virtual address space slots

201 Prelink has to ensure at least that for all successfully prelinked executables all shared libraries they are (transitively)
202 linked against have non-overlapping virtual address space slots (furthermore they cannot overlap with the virtual ad-
203 dress space range used by the executable itself, its `brk` area, typical stack location and `ld.so.cache` and other files
204 mmaped by the dynamic linker in early stages of dynamic linking (before all dependencies are mmaped). If there were
205 any overlaps, the dynamic linker (which mmaps the shared libraries at the desired location without `MAP_FIXED` mmap
206 flag so that it is only soft requirement) would not manage to mmap them at the assigned locations and the prelinking
207 information would be invalidated (the dynamic linker would have to do all normal relocation handling and symbol
208 lookups). Executables are linked against very wide variety of shared library combinations and that has to be taken into
209 account.

210 The simplest approach is to sort shared libraries by descending usage count (so that most often used shared libraries
211 like the dynamic linker, `libc.so` etc. are close to each other) and assign them consecutive slots starting at some
212 architecture specific base address (with a page or two in between the shared libraries to allow for a limited growth of
213 shared libraries without having to reposition them). Prelink has to find out which shared libraries will need a REL to
214 RELA conversion of relocation sections and for those which will need the conversion count with the increased size of
215 the library's loadable segments. This is prelink behavior without `-m` and `-R` options.

216 The architecture specific base address is best located a few megabytes above the location where `mmap` with `NULL` first
217 argument and without `MAP_FIXED` starts allocating memory areas (in Linux this is the value of `TASK_UNMAPPED.BASE`
218 macro). ⁷ The reason for not starting to assign addresses in prelink immediately at `TASK_UNMAPPED.BASE` is that
219 `ld.so.cache` and other mappings by the dynamic linker will end up in the same range and could overlap with the
220 shared libraries. Also, if some application uses `dlopen` to load a shared library which has been prelinked, ⁸ those

⁶Standard dynamic linker path is hardcoded in the executable for each architecture. It can be overridden from the command line, but only with one dynamic linker name (normally, multiple standard dynamic linkers are used when prelinking mixed architecture systems).

⁷`TASK_UNMAPPED.BASE` has been chosen on each platform so that there is enough virtual memory for both the `brk` area (between executable's end and this memory address) and `mmap` area (between this address and bottom of stack).

⁸Typically this is because some other executable is linked against that shared library directly.

few megabytes above `TASK_UNMAPPED.BASE` increase the probability that the stack slot will be still unused (it can clash with e.g. non-prelinked shared libraries loaded by `dlopen` earlier⁹ or other kinds of `mmap` calls with `NULL` first argument like `malloc` allocating big chunks of memory, `mmaping` of locale database, etc.).

This simplest approach is unfortunately problematic on 32-bit (or 31-bit) architectures where the total virtual address space for a process is somewhere between 2GB (S/390) and almost 4GB (Linux IA-32 4GB/4GB kernel split, AMD64 running 32-bit processes, etc.). Typical installations these days contain thousands of shared libraries and if each of them is given a unique address space slot, on average executables will have pretty sparse mapping of its shared libraries and there will be less contiguous virtual memory for application's own use¹⁰.

`Prelink` has a special mode, turned on with `-m` option, in which it computes what shared libraries are ever loaded together in some executable (not considering `dlopen`). If two shared libraries are ever loaded together, `prelink` assigns them different virtual address space slots, but if they never appear together, it can give them overlapping addresses. For example applications using KDE toolkit link typically against many KDE shared libraries, programs written using the `Gtk+` toolkit link typically against many `Gtk+` shared libraries, but there are just very few programs which link against both KDE and `Gtk+` shared libraries, and even if they do, they link against very small subset of those shared libraries. So all KDE shared libraries not in that subset can use overlapping addresses with all `Gtk+` shared libraries but the few exceptions. This leads to considerably smaller virtual address space range used by all prelinked shared libraries, but it has its own disadvantages too. It doesn't work too well with incremental prelinking, because then not all executables are investigated, just those which are given on `prelink`'s command line. `Prelink` also considers executables in `prelink.cache`, but it has no information about executables which have not been prelinked yet. If a new executable, which links against some shared libraries which never appeared together before, is prelinked later, `prelink` has to assign them new, non-overlapping addresses. This means that any executables, which linked against the library that has been moved and re-prelinked, need to be prelinked again. If this happened during incremental prelinking, `prelink` will fix up only the executables given on the command line, leaving other executables untouched. The untouched executables would not be able to benefit from prelinking anymore.

Although with the above two layout schemes shared library addresses can vary slightly between different hosts running the same distribution (depending on the exact set of installed executables and libraries), especially the most often used shared libraries will have identical base addresses on different computers. This is often not desirable for security reasons, because it makes it slightly easier for various exploits to jump to routines they want. Standard Linux kernels assign always the same addresses to shared libraries loaded by the application at each run, so with these kernels `prelink` doesn't make things worse. But there are kernel patches, such as Red Hat's `Exec-Shield`, which randomize memory mappings on each run. If shared libraries are prelinked, they cannot be assigned different addresses on each run (prelinking information can be only used to speed up startup if they are mapped at the base addresses which was used during prelinking), which means prelinking might not be desirable on some edge servers. `Prelink` can assign different addresses on different hosts though, which is almost the same as assigning random addresses on each run for long running processes such as daemons. Furthermore, the administrator can force full prelinking and assignment of new random addresses every few days (if he is also willing to restart the services, so that the old shared libraries and executables don't have to be kept in memory).

To assign random addresses `prelink` has the `-R` option. This causes a random starting address somewhere in the architecture specific range in which shared libraries are assigned, and minor random reshuffling in the queue of shared libraries which need address assignment (normally it is sorted by descending usage count, with randomization shared libraries which are not very far away from each other in the sorted list can be swapped). The `-R` option should work orthogonally to the `-m` option.

Some architectures have special further requirements on shared library address assignment. On 32-bit PowerPC, if shared libraries are located close to the executable, so that everything fits into 32MB area, PLT slots resolving to those shared libraries can use the branch relative instruction instead of more expensive sequences involving memory load and indirect branch. If shared libraries are located in the first 32MB of address space, PLT slots resolving to those shared libraries can use the branch absolute instruction (but already PLT slots in those shared libraries resolving to addresses in the executable cannot be done cheaply). This means for optimization `prelink` should assign addresses from a 24MB region below the executable first, assuming most of the executables are smaller than those remaining 8MB. `prelink` assigns these from higher to lower addresses. When this region is full, `prelink` starts from address `0x40000`¹¹ up

⁹If shared libraries have first `PT_LOAD` segment's virtual address zero, the kernel typically picks first empty slot above `TASK_UNMAPPED.BASE` big enough for the mapping.

¹⁰Especially databases look these days for every byte of virtual address space on 32-bit architectures.

¹¹To leave some pages unmapped to catch `NULL` pointer dereferences.

271 till the bottom of the first area. Only when all these areas are full, `prelink` starts picking addresses high above the
272 executable, so that sufficient space is left in between to leave room for `brk`. When `-R` option is specified, `prelink`
273 needs to honor it, but in a way which doesn't totally kill this optimization. So it picks up a random start base within
274 each of the 3 regions separately, splitting them into 6 regions.

275 Another architecture which needs to be handled specially is IA-32 when using `Exec-Shield`. The IA-32 architecture
276 doesn't have an bit to disable execution for each page, only for each segment. All readable pages are normally exe-
277 cutable. This means the stack is usually executable, as is memory allocated by `malloc`. This is undesirable for security
278 reasons, exploits can then overflow a buffer on the stack to transfer control to code it creates on the stack. Only very
279 few programs actually need an executable stack. For example programs using GCC trampolines for nested functions
280 need it or when an application itself creates executable code on the stack and calls it. `Exec-Shield` works around this
281 IA-32 architecture deficiency by using a separate code segment, which starts at address 0 and spans address space until
282 its limit, highest page which needs to be executable. This is dynamically changed when some page with higher address
283 than the limit needs to be executable (either because of `mmap` with `PROT_EXEC` bit set, or `mprotect` with `PROT_EXEC`
284 of an existing mapping). This kind of protection is of course only effective if the limit is as low as possible. The
285 kernel tries to put all new mappings with `PROT_EXEC` set and `NULL` address low. If possible into *ASCII Shield area*
286 (first 16MB of address space), if not, at least below the executable. If `prelink` detects `Exec-Shield`, it tries to do
287 the same as kernel when assigning addresses, i.e. prefers to assign addresses in *ASCII Shield area* and continues with
288 other addresses below the program. It needs to leave first 1MB plus 4KB of address space unallocated though, because
289 that range is often used by programs using `vm86` system call.

6 Relocation of libraries

290 When a shared library has a base address assigned, it needs to be relocated so that the base address is equal to the first
291 `PT_LOAD` segment's `p_vaddr`. The effect of this operation should be bitwise identical as if the library were linked with
292 that base address originally. That is, the following scripts should produce identical output:

```
293 $ gcc -g -shared -o libfoo.so.1.0.0 -Wl, -h, libfoo.so.1 \  
294     input1.o input2.o somelib.a  
295 $ prelink --reloc-only=0x54321000 libfoo.so.1.0.0
```

Listing 0: Script to relocate a shared library after linking using `prelink`

296 and:

```
297 $ gcc -shared -Wl,--verbose 2>&1 > /dev/null \  
298 | sed -e '/^=====/,/^=====/{d' \  
299     -e '/^=====/{d;s/0\(+ SIZEOF_HEADERS\)/0x54321000\1/' \  
300     > libfoo.so.lds  
301 $ gcc -Wl,-T,libfoo.so.lds -g -shared -o libfoo.so.1.0.0 \  
302     -Wl,-h,libfoo.so.1 input1.o input2.o somelib.a
```

Listing 1: Script to link a shared library at non-standard base

303 The first script creates a normal shared library with the default base address 0 and then uses `prelink`'s special mode
304 when it just relocates a library to a given address. The second script first modifies a built-in GNU linker script for
305 linking of shared libraries, so that the base address is the one given instead of zero and stores it into a temporary file.
306 Then it creates a shared library using that linker script.

307 The relocation operation involves mostly adding the difference between old and new base address to all ELF fields
308 which contain values representing virtual addresses of the shared library (or in the program header table also represent-
309 ing physical addresses). File offsets need to be unmodified. Most places where the adjustments need to be done are
310 clear, `prelink` just has to watch ELF spec to see which fields contain virtual addresses.

311 One problem is with absolute symbols. `Prelink` has no way to find out if an absolute symbol in a shared library is
312 really meant as absolute and thus not changing during relocation, or if it is an address of some place in the shared

library outside of any section or on their edge. For instance symbols created in the GNU linker's script outside of section directives have all SHN_ABS section, yet they can be location in the library (e.g. `symbolfoo = .`) or they can be absolute (e.g. `symbolbar = 0x12345000`). This distinction is lost at link time. But the dynamic linker when looking up symbols doesn't make any distinction between them, all addresses during dynamic lookup have the load offset added to it. `prelink` chooses to relocate any absolute symbols with value bigger than zero, that way `prelink --reloc-only` gets bitwise identical output with linking directly at the different base in almost all real-world cases. Thread Local Storage symbols (those with STT_TLS type) are never relocated, as their values are relative to start of shared library's thread local area.

When relocating the dynamic section there are no bits which tell if a particular dynamic tag uses `d_un.d_ptr` (which needs to be adjusted) or `d_un.d_val` (which needs to be left as is). So `prelink` has to hardcode a list of well known architecture independent dynamic tags which need adjusting and have a hook for architecture specific dynamic tag adjustment. Sun came up with `DT_ADDRRNGLO` to `DT_ADDRRNGHI` and `DT_VALRNGLO` to `DT_VALRNGHI` dynamic tag number ranges, so at least as long as these ranges are used for new dynamic tags `prelink` can relocate correctly even without listing them all explicitly.

When relocating `.rela.*` or `.rel.*` sections, which is done in architecture specific code, relative relocations and on `.got.plt` using architectures also PLT relocations typically need an adjustment. The adjustment needs to be done in either `r_addend` field of the `ElfNN_Rela` structure, in the memory pointed by `r_offset`, or in both locations. On some architectures what needs adjusting is not even the same for all relative relocations. Relative relocations against some sections need to have `r_addend` adjusted while others need to have memory adjusted. On many architectures, first few words in GOT are special and some of them need adjustment.

The hardest part of the adjustment is handling the debugging sections. These are non-allocated sections which typically have no corresponding relocation section associated with them. `Prelink` has to match the various debuggers in what fields it adjusts and what are skipped. As of this writing `prelink` should handle DWARF 2 [15] standard as corrected (and extended) by DWARF 3 draft [16], Stabs [17] with GCC extensions and Alpha or MIPS Mdebug.

DWARF 2 debugging information involves many separate sections, each of them with a unique format which needs to be relocated differently. For relocation of the `.debug_info` section compilation units `prelink` has to parse the corresponding part of the `.debug_abbrev` section, adjust all values of attributes that are using the `DW_FORM_addr` form and adjust embedded location lists. `.debug_ranges` and `.debug_loc` section portions depend on the exact place in `.debug_info` section from which they are referenced, so that `prelink` can keep track of their base address. DWARF debugging format is very extendable, so `prelink` needs to be very conservative when it sees unknown extensions. It needs to fail prelinking instead of silently break debugging information if it sees an unknown `.debug_*` section, unknown attribute form or unknown attribute with one of the `DW_FORM_block*` forms, as they can potentially embed addresses which would need adjustment.

For stabs `prelink` tried to match GDB behavior. For `N_FUN`, it needs to differentiate between function start and function address which are both encoded with this type, the rest of types either always need relocating or never. And similarly to DWARF 2 handling, it needs to reject unknown types.

The relocation code in `prelink` is a little bit more generic than what is described above, as it is used also by other parts of `prelink`, when growing sections in a middle of the shared library during REL to RELA conversion. All adjustment functions get passed both the offset it should add to virtual addresses and a start address. Adjustment is only done if the old virtual address was bigger or equal than the start address.

7 REL to RELA conversion

On architectures which normally use the REL format for relocations instead of RELA (IA-32, ARM and MIPS), if certain relocation types use the memory `r_offset` points to during relocation, `prelink` has to either convert them to a different relocation type which doesn't use the memory value, or the whole `.rel.dyn` section needs to be converted to RELA format. Let's describe it on an example on IA-32 architecture:

```
$ cat > test1.c <<EOF
extern int i[4];
int *j = i + 2;
EOF
```

```

361 $ cat > test2.c <<EOF
362 int i[4];
363 EOF
364 $ gcc -nostdlib -shared -fpic -s -o test2.so test2.c
365 $ gcc -nostdlib -shared -fpic -o test1.so test1.c ./test2.so
366 $ readelf -l test1.so | grep LOAD | head -1
367 LOAD 0x000000 0x00000000 0x00000000 0x002b8 0x002b8 R E 0x1000
368 $ readelf -l test2.so | grep LOAD | head -1
369 LOAD 0x000000 0x00000000 0x00000000 0x00244 0x00244 R E 0x1000
370 $ readelf -r test1.so
371
372 Relocation section '.rel.dyn' at offset 0x2b0 contains 1 entries:
373 Offset      Info      Type           Sym.Value  Sym. Name
374 000012b8    00000d01 R_386_32      00000000    i
375 $ objdump -s -j .data test1.so
376
377 test1.so:      file format elf32-i386
378
379 Contents of section .data:
380 12b8 08000000      ....
381 $ readelf -s test2.so | grep i\$
382 11: 000012a8    16 OBJECT GLOBAL DEFAULT    8 i
383 $ prelink -N ./test1.so ./test2.so
384 $ readelf -l test1.so | grep LOAD | head -1
385 LOAD 0x000000 0x04dba000 0x04dba000 0x002bc 0x002bc R E 0x1000
386 $ readelf -l test2.so | grep LOAD | head -1
387 LOAD 0x000000 0x04db6000 0x04db6000 0x00244 0x00244 R E 0x1000
388 $ readelf -r test1.so
389
390 Relocation section '.rel.dyn' at offset 0x2b0 contains 1 entries:
391 Offset      Info      Type           Sym.Value  Sym. Name + Addend
392 04dbb2bc    00000d01 R_386_32      00000000    i + 8
393 $ objdump -s -j .data test1.so
394
395 test1.so:      file format elf32-i386
396
397 Contents of section .data:
398 4dbb2bc b072db04      .r..
399 $ readelf -s test2.so | grep i\$
400 11: 04db72a8    16 OBJECT GLOBAL DEFAULT    8 i

```

Listing 2: REL to RELA conversion example

401 This relocation is against $i + 8$, where the addend is stored at the memory location pointed by `r_offset`. Prelink
 402 assigned base address `0x4dba000` to `test1.so` and `0x4db6000` to `test2.so`. Prelink above converted the REL
 403 section in `test1.so` to RELA, but let's assume it did not. All output containing `2bc` above would change to `2b8`
 404 (that changed above only because `.rel.dyn` section grew up by 4 bytes during the conversion to RELA format), the
 405 rest would stay unchanged. When some program linked against `test1.so` was prelinked, the (only) relocation in
 406 `test1.so` would not be used and `j` would contain the right value, `0x4db72b0` (address of $i + 8$; note that IA-32 is little
 407 endian, so the values in `.data` section are harder to read for a human). Now, let's assume one of the shared libraries
 408 the executable is linked against is upgraded. This means prelink information cannot be used, as it is out of date. Let's
 409 assume it was a library other than `test2.so`. Normal relocation processing for `test1.so` needs to happen. Standard
 410 `R_386_32` calculation is $S + A$, in this case $0x4db72a8 + 0x4db72b0 = 0x9b6e558$ and `j` contains wrong value. Either
 411 `test2.so` could change and now the `i` variable would have different address, or some other shared library linked to
 412 the executable could overload symbol `i`. Without additional information the dynamic linker cannot find out the addend
 413 is 8.

414 The original value of a symbol could perhaps be stored in some special allocated section and the dynamic linker could
 415 do some magic to locate it, but it would mean standard relocation handling code in the dynamic linker cannot be used
 416 for relocation processing of prelinked shared libraries where prelinking information cannot be used. So `prelink` in
 417 this case converts the whole `.rel.dyn` section into the RELA format, the addend is stored in `r_addend` field and when

418 doing relocation processing, it really doesn't matter what value is at the memory location pointed by `r_offset`. The
419 disadvantage of this is that the relocation section grew by 50%. If prelinking information can be used, it shouldn't
420 matter much, since the section is never loaded at runtime because it is not accessed. If prelinking cannot be used,
421 whether because it is out of date or because the shared library has been loaded by `dlopen`, it will increase memory
422 footprint, but it is read-only memory which is typically not used after startup and can be discarded as it is backed out
423 by the file containing the shared library.

424 At least on IA-32, REL to RELA conversion is not always necessary. If `R_386_32` added is originally 0, `prelink`
425 can instead change its type to `R_386_GLOB_DAT`, which is a similar dynamic relocation, but calculated as `S` instead of
426 `S + A`. There is no similar conversion for `R_386_PC32` possible though, on the other side this relocation type should
427 never appear in position independent shared libraries, only in position dependent code. On ARM, the situation is the
428 same, just using different relocation names (`R_ARM_32`, `R_ARM_GLOB_DAT` and `R_ARM_PC24`).

429 The `.rel.plt` section doesn't have to be converted to RELA format on either of these architectures, if the conversion is
430 needed, all other `.rel.*` allocated sections, which have to be adjacent as they are pointed to by `DT_REL` and `DT_RELSZ`
431 dynamic tags, have to be converted together. The conversion itself is fairly easy, some architecture specific code just has
432 to fetch the original addend from memory pointed by the relocation and store it into `r_addend` field (or clear `r_addend`
433 if the particular relocation type never uses the addend). The main problem is that when the conversion happens, the
434 `.rel.dyn` section grows by 50% and there needs to be room for that in the read-only loadable segment of the shared
435 library.

436 In shared libraries it is always possible to grow the first read-only `PT_LOAD` segment by adding the additional data at the
437 beginning of the read-only segment, as the shared library is relocatable. `Prelink` can relocate the whole shared library
438 to a higher address than it has assigned for it. The file offsets of all sections and the section header table file offset
439 need to be increased, but the ELF header and program headers need to stay at the beginning of the file. The relocation
440 section can then be moved to the newly created space between the end of the program header table and the first section.

441 Moving the section from the old location to the newly created space would leave often very big gap in virtual address
442 space as well as in the file at the old location of the relocation section. Fortunately the linker typically puts special
443 ELF sections including allocated relocation section before the code section and other read-only sections under user's
444 control. These special sections are intended for dynamic linking only. Their addresses are stored just in the `.dynamic`
445 section and `prelink` can easily adjust them there. There is no need for a shared library to store address of one of the
446 special sections into its code or data sections and existing linkers in fact don't create such references. When growing
447 the relocation section, `prelink` checks whether all sections before the relocation section are special¹² and if they are,
448 just moves them to lower addresses, so that the newly created space is right above the relocation section. The advantage
449 is that instead of moving all sections by the size of the new relocation section they can be adjusted ideally just by the
450 difference between old and new relocation section size.

451 There are two factors which can increase the necessary adjustment of all higher sections. The first is required section
452 alignment of any allocated section above the relocation section. `Prelink` needs to find the highest section alignment
453 among those sections and increase the adjustment from the difference between old and new relocation section up to the
454 next multiple of that alignment.

455 The second factor is only relevant to shared libraries where linker optimized the data segment placement. Traditionally
456 linker assigned the end address of the read-only segment plus the architecture's maximum ELF page size as the start
457 address of the read-write segment. While this created smallest file sizes of the shared libraries, it often wasted one
458 page in the read-write segment because of partial pages. When linker optimizes such that less space is wasted in partial
459 pages, the distance between read-only and read-write segments can be smaller than architecture specific maximum ELF
460 page size. `Prelink` has to take this into account, so that when adjusting the sections the read-only and read-write
461 segment don't end up on the same page. Unfortunately `prelink` cannot increase or decrease the distance between the
462 read-only and read-write segments, since it is possible that the shared library has relative addresses of any allocated
463 code, data or `.bss` sections stored in its sections without any relocations which would allow `prelink` to change them.
464 `Prelink` has to move all sections starting with the first allocated `SHT_PROGBITS` section other than `.interp` up to the
465 last allocated `SHT_PROGBITS` or `SHT_NOBITS` section as a block and thus needs to increase the adjustment in steps of
466 the highest section alignment as many times times as needed so that the segments end up in different pages. Below are
467 3 examples:

¹²As special sections `prelink` considers sections with `SHT_NOTE`, `SHT_HASH`, `SHT_DYNSYM`, `SHT_STRTAB`, `SHT_GNU_verdef`,
`SHT_GNU_verneed`, `SHT_GNU_versym`, `SHT_REL` or `SHT_RELA` type or the `.interp` section.

```

468 $ cat > test1.c <<EOF
469 int i[2] __attribute__((aligned (32)));
470 #define J1(N) int *j##N = &i[1];
471 #define J2(N) J1(N##0) J1(N##1) J1(N##2) J1(N##3) J1(N##4)
472 #define J3(N) J2(N##0) J2(N##1) J2(N##2) J2(N##3) J2(N##4)
473 #define J4(N) J3(N##0) J3(N##1) J3(N##2) J3(N##3) J3(N##4)
474 J4(0) J4(1) J3(2) J3(3) J1(4)
475 const int l[256] = { [10] = 1 };
476 /* Put a zero sized section at the end of read-only segment,
477    so that the end address of the segment is printed. */
478 asm (".section ro_seg_end, \"a\"; .previous");
479 EOF
480 $ gcc -shared -O2 -nostdlib -fpic -o test1.so test1.c
481 $ readelf -S test1.so | grep '^ \['
482 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
483 [ 0]                               NULL                               00000000  000000  000000  00   0  0  0
484 [ 1] .hash                             HASH                               000000b4  0000b4  000930  04   A  2  0  4
485 [ 2] .dynsym                             DYNSYM                             000009e4  0009e4  001430  10   A  3  d  4
486 [ 3] .dynstr                             STRTAB                             00001e14  001e14  000735  00   A  0  0  1
487 [ 4] .rel.dyn                             REL                                0000254c  00254c  000968  08   A  2  0  4
488 [ 5] .text                               PROGBITS                           00002eb4  002eb4  000000  00  AX  0  0  4
489 [ 6] .rodata                             PROGBITS                           00002ec0  002ec0  000400  00   A  0  0 32
490 [ 7] ro_seg_end                         PROGBITS                           000032c0  0032c0  000000  00   A  0  0  1
491 [ 8] .data                               PROGBITS                           000042c0  0032c0  0004b4  00  WA  0  0  4
492 [ 9] .dynamic                             DYNAMIC                             00004774  003774  000070  08  WA  3  0  4
493 [10] .got                               PROGBITS                           000047e4  0037e4  00000c  04  WA  0  0  4
494 [11] .bss                               NOBITS                             00004800  003800  000008  00  WA  0  0 32
495 [12] .comment                             PROGBITS                           00000000  003800  000033  00   0  0  1
496 [13] .shstrtab                             STRTAB                             00000000  003833  000075  00   0  0  1
497 [14] .symtab                             SYMTAB                             00000000  003b28  001470  10   15 11  4
498 [15] .strtab                             STRTAB                             00000000  004f98  000742  00   0  0  1
499 $ readelf -l test1.so | grep LOAD
500 LOAD                     0x000000 0x00000000 0x00000000 0x032c0 0x032c0 R E 0x1000
501 LOAD                     0x0032c0 0x000042c0 0x000042c0 0x00530 0x00548 RW 0x1000
502 $ prelink -N ./test1.so
503 $ readelf -l test1.so | grep LOAD
504 LOAD                     0x000000 0x02000000 0x02000000 0x03780 0x03780 R E 0x1000
505 LOAD                     0x003780 0x02004780 0x02004780 0x00530 0x00548 RW 0x1000
506 $ readelf -S test1.so | grep '^ \['
507 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
508 [ 0]                               NULL                               00000000  000000  000000  00   0  0  0
509 [ 1] .hash                             HASH                               020000b4  0000b4  000930  04   A  2  0  4
510 [ 2] .dynsym                             DYNSYM                             020009e4  0009e4  001430  10   A  3  d  4
511 [ 3] .dynstr                             STRTAB                             02001e14  001e14  000735  00   A  0  0  1
512 [ 4] .rel.dyn                             REL                                0200254c  00254c  000e1c  0c   A  2  0  4
513 [ 5] .text                               PROGBITS                           02003374  003374  000000  00  AX  0  0  4
514 [ 6] .rodata                             PROGBITS                           02003380  003380  000400  00   A  0  0 32
515 [ 7] ro_seg_end                         PROGBITS                           02003780  003780  000000  00   A  0  0  1
516 [ 8] .data                               PROGBITS                           02004780  003780  0004b4  00  WA  0  0  4
517 [ 9] .dynamic                             DYNAMIC                             02004c34  003c34  000070  08  WA  3  0  4
518 [10] .got                               PROGBITS                           02004ca4  003ca4  00000c  04  WA  0  0  4
519 [11] .bss                               NOBITS                             02004cc0  003cc0  000008  00  WA  0  0 32
520 [12] .comment                             PROGBITS                           00000000  003cc0  000033  00   0  0  1
521 [13] .gnu.liblist                         GNU_LIBLIST                         00000000  003cf3  000000  14   14  0  4
522 [14] .gnu.libstr                          STRTAB                             00000000  003cf3  000000  00   0  0  1
523 [15] .gnu.prelink_undo                   PROGBITS                           00000000  003cf4  00030c  01   0  0  4
524 [16] .shstrtab                             STRTAB                             00000000  004003  0000a0  00   0  0  1
525 [17] .symtab                             SYMTAB                             00000000  0043a0  001470  10   18 11  4
526 [18] .strtab                             STRTAB                             00000000  005810  000742  00   0  0  1

```

Listing 3: Growing read-only segment with segment distance one page

527 In this example the read-write segment starts at address 0x42c0, which is one page above the end of read-only segment.

528 Prelink needs to grow the read-only PT_LOAD segment by 50% of .rel.dyn size, i.e. 0x4b4 bytes. Prelink just
529 needs to round that up for the highest alignment (32 bytes required by .rodata or .bss sections) and moves all
530 sections above .rel.dyn by 0x4c0 bytes.

```

531 $ cat > test2.c <<EOF
532 int i[2] __attribute__((aligned (32)));
533 #define J1(N) int *j##N = &i[1];
534 #define J2(N) J1(N##0) J1(N##1) J1(N##2) J1(N##3) J1(N##4)
535 #define J3(N) J2(N##0) J2(N##1) J2(N##2) J2(N##3) J2(N##4)
536 #define J4(N) J3(N##0) J3(N##1) J3(N##2) J3(N##3) J3(N##4)
537 J4(0) J4(1) J3(2) J3(3) J1(4)
538 const int l[256] = { [10] = 1 };
539 int k[670];
540 asm (".section ro_seg_end, \"a\"; .previous");
541 EOF
542 $ gcc -shared -O2 -nostdlib -fpic -o test2.so test2.c
543 $ readelf -S test2.so | grep '^ \['
544 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
545 [ 0]                               NULL                               00000000 000000 000000 00      0  0  0
546 [ 1] .hash                               HASH                               000000b4 0000b4 000934 04      A  2  0  4
547 [ 2] .dynsym                             DYNSYM                             000009e8 0009e8 001440 10      A  3  d  4
548 [ 3] .dynstr                             STRTAB                             00001e28 001e28 000737 00      A  0  0  1
549 [ 4] .rel.dyn                             REL                                00002560 002560 000968 08      A  2  0  4
550 [ 5] .text                               PROGBITS                           00002ec8 002ec8 000000 00     AX  0  0  4
551 [ 6] .rodata                             PROGBITS                           00002ee0 002ee0 000400 00      A  0  0 32
552 [ 7] ro_seg_end                         PROGBITS                           000032e0 0032e0 000000 00      A  0  0  1
553 [ 8] .data                               PROGBITS                           00004000 004000 0004b4 00     WA  0  0  4
554 [ 9] .dynamic                             DYNAMIC                             000044b4 0044b4 000070 08     WA  3  0  4
555 [10] .got                               PROGBITS                           00004524 004524 00000c 04     WA  0  0  4
556 [11] .bss                               NOBITS                             00004540 004540 000a88 00     WA  0  0 32
557 [12] .comment                           PROGBITS                           00000000 004540 000033 00      0  0  1
558 [13] .shstrtab                          STRTAB                             00000000 004573 000075 00      0  0  1
559 [14] .symtab                             SYMTAB                             00000000 004868 001480 10     15 11  4
560 [15] .strtab                             STRTAB                             00000000 005ce8 000744 00      0  0  1
561 $ readelf -l test2.so | grep LOAD
562 LOAD                               0x000000 0x00000000 0x00000000 0x032e0 0x032e0 R E 0x1000
563 LOAD                               0x004000 0x00004000 0x00004000 0x00530 0x00fc8 RW 0x1000
564 $ prelink -N ./test2.so
565 $ readelf -l test2.so | grep LOAD
566 LOAD                               0x000000 0x02000000 0x02000000 0x037a0 0x037a0 R E 0x1000
567 LOAD                               0x0044c0 0x020044c0 0x020044c0 0x00530 0x00fc8 RW 0x1000
568 $ readelf -S test2.so | grep '^ \['
569 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
570 [ 0]                               NULL                               00000000 000000 000000 00      0  0  0
571 [ 1] .hash                               HASH                               020000b4 0000b4 000934 04      A  2  0  4
572 [ 2] .dynsym                             DYNSYM                             020009e8 0009e8 001440 10      A  3  d  4
573 [ 3] .dynstr                             STRTAB                             02001e28 001e28 000737 00      A  0  0  1
574 [ 4] .rel.dyn                             RELA                                02002560 002560 000e1c 0c      A  2  0  4
575 [ 5] .text                               PROGBITS                           02003388 003388 000000 00     AX  0  0  4
576 [ 6] .rodata                             PROGBITS                           020033a0 0033a0 000400 00      A  0  0 32
577 [ 7] ro_seg_end                         PROGBITS                           020037a0 0037a0 000000 00      A  0  0  1
578 [ 8] .data                               PROGBITS                           020044c0 0044c0 0004b4 00     WA  0  0  4
579 [ 9] .dynamic                             DYNAMIC                             02004974 004974 000070 08     WA  3  0  4
580 [10] .got                               PROGBITS                           020049e4 0049e4 00000c 04     WA  0  0  4
581 [11] .bss                               NOBITS                             02004a00 004a00 000a88 00     WA  0  0 32
582 [12] .comment                           PROGBITS                           00000000 004a00 000033 00      0  0  1
583 [13] .gnu.liblist                        GNU_LIBLIST                        00000000 004a33 000000 14     14  0  4
584 [14] .gnu.libstr                          STRTAB                             00000000 004a33 000000 00      0  0  1
585 [15] .gnu.prelink_undo                  PROGBITS                           00000000 004a34 00030c 01      0  0  4
586 [16] .shstrtab                          STRTAB                             00000000 004d43 0000a0 00      0  0  1
587 [17] .symtab                             SYMTAB                             00000000 0050e0 001480 10     18 11  4
588 [18] .strtab                             STRTAB                             00000000 006560 000744 00      0  0  1

```

589 In the second example prelink can grow by just 0x4c0 bytes as well, eventhough the distance between read-write
 590 and read-only segment is just 0xd20 bytes. With this distance, hypothetical adjustment by any size less than 0xd21
 591 bytes (modulo 4096) would need just rounding up to the next multiple of 32 bytes, while adjustments from 0xd21 up
 592 to 0xfe0 would require adjustments in multiples of 4096 bytes.

```

593 $ cat > test3.c <<EOF
594 int i[2] __attribute__((aligned (32)));
595 #define J1(N) int *j##N = &i[1];
596 #define J2(N) J1(N##0) J1(N##1) J1(N##2) J1(N##3) J1(N##4)
597 #define J3(N) J2(N##0) J2(N##1) J2(N##2) J2(N##3) J2(N##4)
598 #define J4(N) J3(N##0) J3(N##1) J3(N##2) J3(N##3) J3(N##4)
599 J4(0) J4(1) J3(2) J3(3) J1(4)
600 int k[670];
601 asm (".section ro_seg_end, \"a\"; .previous");
602 EOF
603 $ gcc -shared -O2 -nostdlib -fpic -o test3.so test3.c
604 $ readelf -S test3.so | grep '^ \['
605 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
606 [ 0]                               NULL                               00000000  000000  000000  00      0  0  0
607 [ 1] .hash                               HASH                               000000b4  0000b4  00092c  04      A  2  0  4
608 [ 2] .dynsym                             DYNSYM                             000009e0  0009e0  001420  10      A  3  c  4
609 [ 3] .dynstr                             STRTAB                             00001e00  001e00  000735  00      A  0  0  1
610 [ 4] .rel.dyn                             REL                                00002538  002538  000968  08      A  2  0  4
611 [ 5] .text                               PROGBITS                           00002ea0  002ea0  000000  00     AX  0  0  4
612 [ 6] ro_seg_end                         PROGBITS                           00002ea0  002ea0  000000  00      A  0  0  1
613 [ 7] .data                               PROGBITS                           00003000  003000  0004b4  00     WA  0  0  4
614 [ 8] .dynamic                             DYNAMIC                             000034b4  0034b4  000070  08     WA  3  0  4
615 [ 9] .got                                 PROGBITS                           00003524  003524  00000c  04     WA  0  0  4
616 [10] .bss                                 NOBITS                             00003540  003540  000a88  00     WA  0  0 32
617 [11] .comment                             PROGBITS                           00000000  003540  000033  00      0  0  1
618 [12] .shstrtab                           STRTAB                             00000000  003573  00006d  00      0  0  1
619 [13] .symtab                              SYMTAB                             00000000  003838  001460  10     14 10  4
620 [14] .strtab                              STRTAB                             00000000  004c98  000742  00      0  0  1
621 $ readelf -l test3.so | grep LOAD
622 LOAD                             0x000000 0x00000000 0x00000000 0x02ea0 0x02ea0 R E 0x1000
623 LOAD                             0x003000 0x00003000 0x00003000 0x00530 0x00fc8 RW 0x1000
624 $ prelink -N ./test3.so
625 $ readelf -l test3.so | grep LOAD
626 LOAD                             0x000000 0x02000000 0x02000000 0x03ea0 0x03ea0 R E 0x1000
627 LOAD                             0x004000 0x02004000 0x02004000 0x00530 0x00fc8 RW 0x1000
628 $ readelf -S test3.so | grep '^ \['
629 [Nr] Name                               Type                               Addr      Off      Size    ES Flg Lk Inf Al
630 [ 0]                               NULL                               00000000  000000  000000  00      0  0  0
631 [ 1] .hash                               HASH                               020000b4  0000b4  00092c  04      A  2  0  4
632 [ 2] .dynsym                             DYNSYM                             020009e0  0009e0  001420  10      A  3  c  4
633 [ 3] .dynstr                             STRTAB                             02001e00  001e00  000735  00      A  0  0  1
634 [ 4] .rel.dyn                             RELA                                02002538  002538  000e1c  0c      A  2  0  4
635 [ 5] .text                               PROGBITS                           02003ea0  003ea0  000000  00     AX  0  0  4
636 [ 6] ro_seg_end                         PROGBITS                           02003ea0  003ea0  000000  00      A  0  0  1
637 [ 7] .data                               PROGBITS                           02004000  004000  0004b4  00     WA  0  0  4
638 [ 8] .dynamic                             DYNAMIC                             020044b4  0044b4  000070  08     WA  3  0  4
639 [ 9] .got                                 PROGBITS                           02004524  004524  00000c  04     WA  0  0  4
640 [10] .bss                                 NOBITS                             02004540  004540  000a88  00     WA  0  0 32
641 [11] .comment                             PROGBITS                           00000000  004540  000033  00      0  0  1
642 [12] .gnu.liblist                         GNU_LIBLIST                         00000000  004573  000000  14     13  0  4
643 [13] .gnu.libstr                          STRTAB                             00000000  004573  000000  00      0  0  1
644 [14] .gnu.prelink_undo                   PROGBITS                           00000000  004574  0002e4  01      0  0  4
645 [15] .shstrtab                           STRTAB                             00000000  00485b  000098  00      0  0  1
646 [16] .symtab                              SYMTAB                             00000000  004bc8  001460  10     17 10  4

```

Listing 5: Growing read-only segment if page padding needed

648 In the last example the distance between PT_LOAD segments is very small, just 0x160 bytes and the adjustment had to
 649 be done by 4096 bytes.

8 Conflicts

650 As said earlier, if symbol lookup of some symbol in particular shared library results in different values when that
 651 shared library's natural search scope is used and when using search scope of the application the DSO is used in, this is
 652 considered a *conflict*. Here is an example of a conflict on IA-32:

```

653 $ cat > test1.c <<EOF
654 int i;
655 int *j = &i;
656 int *foo (void) { return &i; }
657 EOF
658 $ cat > test2.c <<EOF
659 int i;
660 int *k = &i;
661 int *bar (void) { return &i; }
662 EOF
663 $ cat > test.c <<EOF
664 #include <stdio.h>
665 extern int i, *j, *k, *foo (void), bar (void);
666 int main (void)
667 {
668 #ifdef PRINT_I
669     printf ("%p\n", &i);
670 #endif
671     printf ("%p %p %p %p\n", j, k, foo (), bar ());
672 }
673 EOF
674 $ gcc -nostdlib -shared -fpic -s -o test1.so test1.c
675 $ gcc -nostdlib -shared -fpic -o test2.so test2.c ./test1.so
676 $ gcc -o test test.c ./test2.so ./test1.so
677 $ ./test
678 0x16137c 0x16137c 0x16137c 0x16137c
679 $ readelf -r ./test1.so
680
681 Relocation section '.rel.dyn' at offset 0x2bc contains 2 entries:
682   Offset      Info    Type           Sym.Value  Sym. Name
683   000012e4    00000d01 R_386_32          00001368    i
684   00001364    00000d06 R_386_GLOB_DAT    00001368    i
685 $ prelink -N ./test ./test1.so ./test2.so
686 $ LD_WARN= LD_TRACE_PRELINKING=1 LD_BIND_NOW=1 /lib/ld-linux.so.2 ./test1.so
687     ./test1.so => ./test1.so (0x04db6000, 0x00000000)
688 $ LD_WARN= LD_TRACE_PRELINKING=1 LD_BIND_NOW=1 /lib/ld-linux.so.2 ./test2.so
689     ./test2.so => ./test2.so (0x04dba000, 0x00000000)
690     ./test1.so => ./test1.so (0x04db6000, 0x00000000)
691 $ LD_WARN= LD_TRACE_PRELINKING=1 LD_BIND_NOW=1 /lib/ld-linux.so.2 ./test \
692 | sed 's/^[[:space:]]*/ /'
693     ./test => ./test (0x08048000, 0x00000000)
694     ./test2.so => ./test2.so (0x04dba000, 0x00000000)
695     ./test1.so => ./test1.so (0x04db6000, 0x00000000)
696     libc.so.6 => /lib/tls/libc.so.6 (0x00b22000, 0x00000000) TLS(0x1, 0x00000028)
697     /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00b0a000, 0x00000000)
698 $ readelf -S ./test1.so | grep '\.data\|\.got'
699 [ 6] .data                PROGBITS          04db72e4 0002e4 000004 00 WA 0 0 4

```

```

700 [ 8] .got          PROGBITS          04db7358 000358 000010 04  WA  0   0   4
701 $ readelf -r ./test1.so
702
703 Relocation section '.rel.dyn' at offset 0x2bc contains 2 entries:
704   Offset      Info      Type           Sym.Value  Sym. Name
705 04db72e4  00000d06 R_386_GLOB_DAT  04db7368   i
706 04db7364  00000d06 R_386_GLOB_DAT  04db7368   i
707 $ objdump -s -j .got -j .data test1.so
708
709 test1.so:          file format elf32-i386
710
711 Contents of section .data:
712 4db72e4 6873db04                      hs..
713 Contents of section .got:
714 4db7358 e8120000 00000000 00000000 6873db04 .....hs..
715 $ readelf -r ./test | sed '/\.\gnu\.\conflict/, $!d'
716 Relocation section '.gnu.conflict' at offset 0x7ac contains 18 entries:
717   Offset      Info      Type           Sym.Value  Sym. Name + Addend
718 04db72e4  00000001 R_386_32                      04dbb37c
719 04db7364  00000001 R_386_32                      04dbb37c
720 00c56874  00000001 R_386_32                      ffffffff0
721 00c56878  00000001 R_386_32                      00000001
722 00c568bc  00000001 R_386_32                      ffffffff4
723 00c56900  00000001 R_386_32                      ffffffffec
724 00c56948  00000001 R_386_32                      ffffffffdc
725 00c5695c  00000001 R_386_32                      ffffffff0
726 00c56980  00000001 R_386_32                      ffffffff8
727 00c56988  00000001 R_386_32                      ffffffff4
728 00c569a4  00000001 R_386_32                      ffffffff8
729 00c569c4  00000001 R_386_32                      ffffffff0
730 00c569d8  00000001 R_386_32                      080485b8
731 00b1f510  00000007 R_386_JUMP_SLOT              00b91460
732 00b1f514  00000007 R_386_JUMP_SLOT              00b91080
733 00b1f518  00000007 R_386_JUMP_SLOT              00b91750
734 00b1f51c  00000007 R_386_JUMP_SLOT              00b912c0
735 00b1f520  00000007 R_386_JUMP_SLOT              00b91200
736 $ ./test
737 0x4dbb37c 0x4dbb37c 0x4dbb37c 0x4dbb37c

```

Listing 6: Conflict example

In the example, among some conflicts caused by the dynamic linker and the C library,¹³ there is a conflict for the symbol *i* in *test1.so* shared library. *test1.so* has just itself in its natural symbol lookup scope (as proved by

```

740 LD_WARN= LD_TRACE_PRELINKING=1 LD_BIND_NOW=1 /lib/ld-linux.so.2 ./test1.so

```

command output), so when looking up symbol *i* in this scope the definition in *test1.so* is chosen. *test1.so* has two relocations against the symbol *i*, one *R_386_32* against *.data* section and one *R_386_GLOB_DAT* against *.got* section. When prelinking *test1.so* library, the dynamic linker stores the address of *i* (0x4db7368) into both locations (at offsets 0x4db72e4 and 0x4db7364). The global symbol search scope in *test* executable contains the executable itself, *test2.so* and *test1.so* libraries, *libc.so.6* and the dynamic linker in the listed order. When doing symbol lookup for symbol *i* in *test1.so* when doing relocation processing of the whole executable, address of *i* in *test2.so* is returned as that symbol comes earlier in the global search scope. So, when none of the libraries nor the executable is prelinked, the program prints 4 identical addresses. If prelink didn't create conflict fixups for the two relocations against the symbol *i* in *test1.so*, prelinked executable (which bypasses normal relocation processing on startup) would print instead of the desired

¹³Particularly in the example, the 5 *R_386_JUMP_SLOT* fixups are PLT slots in the dynamic linker for memory allocator functions resolving to C library functions instead of dynamic linker's own trivial implementation. First 10 *R_386_32* fixups at offsets 0xc56874 to 0xc569c4 are Thread Local Storage fixups in the C library and the fixup at 0xc569d8 is for *_IO_stdin_used* weak undefined symbol in the C library, resolving to a symbol with the same name in the executable.

```
751 0x4dbb37c 0x4dbb37c 0x4dbb37c 0x4dbb37c
```

752 different addresses,

```
753 0x4db7368 0x4dbb37c 0x4db7368 0x4dbb37c
```

754 That is a functionality change that prelink cannot be permitted to make, so instead it fixes up the two locations by
755 storing the desired value in there. In this case prelink really cannot avoid that - test1.so shared library could
756 be also used without test2.so in some other executable's symbol search scope. Or there could be some executable
757 linked with:

```
758 $ gcc -o test2 test.c ./test1.so ./test2.so
```

Listing 7: Conflict example with swapped order of libraries

759 where *i* lookup in test1.so and test2.so is supposed to resolve to *i* in test1.so.

760 Now consider what happens if the executable is linked with -DPRINT_I:

```
761 $ gcc -DPRINT_I -o test3 test.c ./test2.so ./test1.so
762 $ ./test3
763 0x804972c
764 0x804972c 0x804972c 0x804972c 0x804972c
765 $ prelink -N ./test3 ./test1.so ./test2.so
766 $ readelf -S ./test2.so | grep '\.data\|\.got'
767 [ 6] .data          PROGBITS          04dbb2f0 0002f0 000004 00  WA  0  0  4
768 [ 8] .got           PROGBITS          04dbb36c 00036c 000010 04  WA  0  0  4
769 $ readelf -r ./test2.so
770
771 Relocation section '.rel.dyn' at offset 0x2c8 contains 2 entries:
772 Offset      Info      Type           Sym.Value    Sym. Name
773 04dbb2f0    00000d06  R_386_GLOB_DAT 04dbb37c     i
774 04dbb378    00000d06  R_386_GLOB_DAT 04dbb37c     i
775 $ objdump -s -j .got -j .data test2.so
776
777 test2.so:      file format elf32-i386
778
779 Contents of section .data:
780 4dbb2f0 7cb3db04                                     |...
781 Contents of section .got:
782 4dbb36c f4120000 00000000 00000000 7cb3db04 .....|...
783 $ readelf -r ./test3
784
785 Relocation section '.rel.dyn' at offset 0x370 contains 4 entries:
786 Offset      Info      Type           Sym.Value    Sym. Name
787 08049720    00000e06  R_386_GLOB_DAT 00000000     __gmon_start__
788 08049724    00000105  R_386_COPY      08049724     j
789 08049728    00000305  R_386_COPY      08049728     k
790 0804972c    00000405  R_386_COPY      0804972c     i
791
792 Relocation section '.rel.plt' at offset 0x390 contains 4 entries:
793 Offset      Info      Type           Sym.Value    Sym. Name
794 08049710    00000607  R_386_JUMP_SLOT 080483d8     __libc_start_main
795 08049714    00000707  R_386_JUMP_SLOT 080483e8     printf
796 08049718    00000807  R_386_JUMP_SLOT 080483f8     foo
797 0804971c    00000c07  R_386_JUMP_SLOT 08048408     bar
798
799 Relocation section '.gnu.conflict' at offset 0x7f0 contains 20 entries:
```

Offset	Info	Type	Sym.Value	Sym. Name	+ Addend
04dbb2f0	00000001	R_386_32			0804972c
04dbb378	00000001	R_386_32			0804972c
04db72e4	00000001	R_386_32			0804972c
04db7364	00000001	R_386_32			0804972c
00c56874	00000001	R_386_32			ffffffff0
00c56878	00000001	R_386_32			00000001
00c568bc	00000001	R_386_32			fffffff4
00c56900	00000001	R_386_32			fffffec
00c56948	00000001	R_386_32			fffffddc
00c5695c	00000001	R_386_32			ffffffe0
00c56980	00000001	R_386_32			fffffff8
00c56988	00000001	R_386_32			ffffffe4
00c569a4	00000001	R_386_32			ffffffd8
00c569c4	00000001	R_386_32			ffffffe8
00c569d8	00000001	R_386_32			080485f0
00b1f510	00000007	R_386_JUMP_SLOT			00b91460
00b1f514	00000007	R_386_JUMP_SLOT			00b91080
00b1f518	00000007	R_386_JUMP_SLOT			00b91750
00b1f51c	00000007	R_386_JUMP_SLOT			00b912c0
00b1f520	00000007	R_386_JUMP_SLOT			00b91200
\$./test3					
0x804972c					
0x804972c	0x804972c	0x804972c	0x804972c	0x804972c	

Listing 8: Conflict example with COPY relocation for conflicting symbol

Because the executable is not compiled as position independent code and main function takes address of *i* variable, the object file for test3.c contains a R_386_32 relocation against *i*. The linker cannot make dynamic relocations against read-only segment in the executable, so the address of *i* must be constant. This is accomplished by creating a new object *i* in the executable's .dynbss section and creating a dynamic R_386_COPY relocation for it. The relocation ensures that during startup the content of *i* object earliest in the search scope without the executable is copied to this *i* object in executable. Now, unlike test executable, in test3 executable *i* lookups in both test1.so and test2.so libraries result in address of *i* in the executable (instead of test2.so). This means that two conflict fixups are needed again for test1.so (but storing 0x804972c instead of 0x4dbb37c) and two new fixups are needed for test2.so.

If the executable is compiled as position independent code,

```
$ gcc -fpic -DPRINT_I -o test4 test.c ./test2.so ./test1.so
$ ./test4
0x4dbb37c
0x4dbb37c 0x4dbb37c 0x4dbb37c 0x4dbb37c
```

Listing 9: Conflict example with position independent code in the executable

the address of *i* is stored in executable's .got section, which is writable and thus can have dynamic relocation against it. So the linker creates a R_386_GLOB_DAT relocation against the .got section, the symbol *i* is undefined in the executable and no copy relocations are needed. In this case, only test1.so will need 2 fixups, test2.so will not need any.

There are various reasons for conflicts:

- Improperly linked shared libraries. If a shared library always needs symbols from some particular shared library, it should be linked against that library, usually by adding -llibname to gcc -shared command line used during linking of the shared library. This both reduces conflict fixups in prelink and makes the library easier to load using dlopen, because applications don't have to remember that they have to load some other library first. The best place to record the dependency is in the shared library itself. Another reason is if the needed library uses symbol versioning for its symbols. Not linking against that library can result in malfunctioning

shared library. Prelink issues a warning for such libraries - Warning: *library* has undefined non-weak symbols. When linking a shared library, the `-Wl, -z, defs` option can be used to ensure there are no such undefined non-weak symbols. There are exceptions, when undefined non-weak symbols in shared libraries are desirable. One exception is when there are multiple shared libraries providing the same functionality, and a shared library doesn't care which one is used. An example can be e.g. `libreadline.so.4`, which needs some terminal handling functions, which are provided by either `libtermcap.so.2`, or `libncurses.so.5`. Another exception is with plugins or other shared libraries which expect some symbols to be resolved to symbols defined in the executable.

- A library overriding functionality of some other library. One example is e.g. C library and POSIX thread library. Older versions of the GNU C library did not provide cancelable entry points required by the standard. This is not needed for non-threaded applications. So only the `libpthread.so.0` shared library which provides POSIX threading support then overrode the cancellation entry points required by the standard by wrapper functions which provided the required functionality. Although most recent versions of the GNU C library handle cancellation even in entry points in `libc.so.6` (this was needed for cases when `libc.so.6` comes earlier before `libpthread.so.0` in symbol search scope and used to be worked around by non-standard handling of weak symbols in the dynamic linker), because of symbol versioning the symbols had to stay in `libpthread.so.0` as well as in `libc.so.6`. This means every program using POSIX threads on Linux will have a couple of conflict fixups because of this.
- Programs which need copy relocations. Although prelink will resolve the copy relocations at prelinking time, if any shared library has relocations against the symbol which needed copy relocation, all such relocations will need conflict fixups. Generally, it is better to not export variables from shared libraries in their APIs, instead provide accessor functions.
- Function pointer equality requirement for functions called from executables. When address of some global function is taken, at least C and C++ require that this pointer is the same in the whole program. Executables typically contain position dependent code, so when code in the executable takes address of some function not defined in the executable itself, that address must be link time constant. Linker accomplishes this by creating a PLT slot for the function unless there was one already and resolving to the address of PLT slot. The symbol for the function is created with `st_value` equal to address of the PLT slot, but `st_shndx` set to `SHN_UNDEF`. Such symbols are treated specially by the dynamic linker, in that PLT relocations resolve to first symbol in the global search scope after the executable, while symbol lookups for all other relocation types return the address of the symbol in the executable. Unfortunately, GNU linker doesn't differentiate between taking address of a function in an executable (especially one for which no dynamic relocation is possible in case it is in read-only segment) and just calling the function, but never taking its address. If it cleared the `st_value` field of the `SHN_UNDEF` function symbols in case nothing in the executable takes the function's address, several prelink conflict could disappear (`SHN_UNDEF` symbols with `st_value` set to 0 are treated always as real undefined symbols by the dynamic linker).
- COMDAT code and data in C++. C++ language has several places where it may need to emit some code or data without a clear unique compilation unit owning it. Examples include taking address of an inline function, local static variable in inline functions, virtual tables for some classes (this depends on `#pragma interface` or `#pragma implementation` presence, presence of non-inline non-pure-virtual member function in the class, etc.), *RTTI* info for them. Compilers and linkers handle these using various COMDAT schemes, e.g. GNU linker's `.gnu.linkonce*` special sections or using `SHT_GROUP`. Unfortunately, all these duplicate merging schemes work only during linking of shared libraries or executables, no duplicate removal is done across shared libraries. Shared libraries typically have relocations against their COMDAT code or data objects (otherwise they wouldn't be at least in most cases emitted at all), so if there are COMDAT duplicates across shared libraries or the executable, they lead to conflict fixups. The linker theoretically could try to merge COMDAT duplicates across shared libraries if specifically requested by the user (if a COMDAT symbol is already present in one of the dependent shared libraries and is `STB_WEAK`, the linker could skip it). Unfortunately, this only works as long as the user has full control over the dependent shared libraries, because the COMDAT symbol could be exported from them just as a side effect of their implementation (e.g. they use some class internally). When such libraries are rebuilt even with minor changes in their implementation (unfortunately with C++ shared libraries it is usually not very clear what part is exported ABI and what is not), some of those COMDAT symbols in them could go away (e.g. because suddenly they use a different class internally and the previously used class is not referenced anywhere). When COMDAT objects are not merged across shared libraries, this makes no problems, as each library which needs the COMDAT has its own copy. But with COMDAT duplicate removal between shared libraries there could suddenly be unresolved references and the shared libraries would need to be relinked. The only place where this could work

903 safely is when a single package includes several C++ shared libraries which depend on each other. They are then
904 shipped always together and when one changes, all others need changing too.

9 Prelink optimizations to reduce number of conflict fixups

905 Prelink can optimize out some conflict fixups if it can prove that the changes are not observable by the application
906 at runtime (opening its executable and reading it doesn't count). If there is a data object in some shared library with
907 a symbol that is overridden by a symbol in a different shared library earlier in global symbol lookup scope or in
908 the executable, then that data object is likely never referenced and it shouldn't matter what it contains. Examine the
909 following example:

```
910 $ cat > test1.c <<EOF
911 int i, j, k;
912 struct A { int *a; int *b; int *c; } x = { &i, &j, &k };
913 struct A *y = &x;
914 EOF
915 $ cat > test2.c <<EOF
916 int i, j, k;
917 struct A { int *a; int *b; int *c; } x = { &i, &j, &k };
918 struct A *z = &x;
919 EOF
920 $ cat > test.c <<EOF
921 #include <stdio.h>
922 extern struct A { int *a; int *b; int *c; } *y, *z;
923 int main (void)
924 {
925     printf ("%p: %p %p %p\n", y, y->a, y->b, y->c);
926     printf ("%p: %p %p %p\n", z, z->a, z->b, z->c);
927 }
928 EOF
929 $ gcc -nostdlib -shared -fpic -s -o test1.so test1.c
930 $ gcc -nostdlib -shared -fpic -o test2.so test2.c ./test1.so
931 $ gcc -o test test.c ./test2.so ./test1.so
932 $ ./test
933 0xaf3314: 0xaf33b0 0xaf33a8 0xaf33ac
934 0xaf3314: 0xaf33b0 0xaf33a8 0xaf33ac
```

Listing 10: C example where conflict fixups could be optimized out

935 In this example there are 3 conflict fixups pointing into the 12 byte long `x` object in `test1.so` shared library (among
936 other conflicts). And nothing in the program can poke at `x` content in `test1.so`, simply because it has to look at it
937 through `x` symbol which resolves to `test2.so`. So in this case prelink could skip those 3 conflicts. Unfortunately it
938 is not that easy:

```
939 $ cat > test3.c <<EOF
940 int i, j, k;
941 static struct A { int *a; int *b; int *c; } local = { &i, &j, &k };
942 extern struct A x;
943 struct A *y = &x;
944 struct A *y2 = &local;
945 extern struct A x __attribute__((alias ("local")));
946 EOF
947 $ cat > test4.c <<EOF
948 #include <stdio.h>
949 extern struct A { int *a; int *b; int *c; } *y, *y2, *z;
950 int main (void)
951 {
```

```

952 printf ("%p: %p %p %p\n", y, y->a, y->b, y->c);
953 printf ("%p: %p %p %p\n", y2, y2->a, y2->b, y2->c);
954 printf ("%p: %p %p %p\n", z, z->a, z->b, z->c);
955 }
956 EOF
957 $ gcc -nostdlib -shared -fpic -s -o test3.so test3.c
958 $ gcc -nostdlib -shared -fpic -o test4.so test2.c ./test3.so
959 $ gcc -o test4 test4.c ./test4.so ./test3.so
960 $ ./test4
961 0x65a314: 0x65a3b0 0x65a3a8 0x65a3ac
962 0xbd1328: 0x65a3b0 0x65a3a8 0x65a3ac
963 0x65a314: 0x65a3b0 0x65a3a8 0x65a3ac

```

Listing 11: Modified C example where conflict fixups cannot be removed

964 In this example, there are again 3 conflict fixups pointing into the 12 byte long `x` object in `test3.so` shared library.
965 The fact that variable `local` is located at the same 12 bytes is totally invisible to `prelink`, as `local` is a `STB_LOCAL` symbol
966 which doesn't show up in `.dynsym` section. But if those 3 conflict fixups are removed, then suddenly program's
967 observable behavior changes (the last 3 addresses on second line would be different than those on first or third line).

968 Fortunately, there are at least some objects where `prelink` can be reasonably sure they will never be referenced
969 through some local alias. Those are various compiler generated objects with well defined meaning which is `prelink`
970 able to identify in shared libraries. The most important ones are C++ virtual tables and `RTTI` data. They are emitted
971 as `COMDAT` data by the compiler, in GCC into `.gnu.linkonce.d.*` sections. Data or code in these sections can
972 be accessed only through global symbols, otherwise linker might create unexpected results when two or more of these
973 sections are merged together (all but one deleted). When `prelink` is checking for such data, it first checks whether the
974 shared library in question is linked against `libstdc++.so`. If not, it is not a C++ library (or incorrectly built one) and
975 thus it makes no sense to search any further. It looks only in `.data` section, for `STB_WEAK STT_OBJECT` symbols whose
976 names start with certain prefixes¹⁴ and where no other symbols (in dynamic symbol table) point into the objects. If
977 these objects are unused because there is a conflict on their symbol, all conflict fixups pointing into the virtual table or
978 `RTTI` structure can be discarded.

979 Another possible optimization is again related to C++ virtual tables. Function addresses in them are not intended for
980 pointer comparisons. C++ code only loads them from the virtual tables and calls through the pointer. Pointers to
981 member functions are handled differently. As pointer equivalence is the only reason why all function pointers resolve
982 to PLT slots in the executable even when the executable doesn't include implementation of the function (i.e. has
983 `SHN_UNDEF` symbol with non-zero `st_value` pointing at the PLT slot in the executable), `prelink` can resolve method
984 addresses in virtual tables to the actual method implementation. In many cases this is in the same library as the virtual
985 table (or in one of libraries in its natural symbol lookup scope), so a conflict fixup is unnecessary. This optimization
986 speeds up programs also after control is transferred to the application and not just the time to start up the application,
987 although just a few cycles per method call.

988 The conflict fixup reduction is quite big on some programs. Below is statistics for `kmail` program on completely
989 unprelinked box:

```

990 $ LD_DEBUG=statistics /usr/bin/kmail 2>&1 | sed '2,8!d;s/^ *///'
991 10621:      total startup time in dynamic loader: 240724867 clock cycles
992 10621:      time needed for relocation: 234049636 clock cycles (97.2%)
993 10621:      number of relocations: 34854
994 10621:      number of relocations from cache: 74364
995 10621:      number of relative relocations: 35351
996 10621:      time needed to load objects: 6241678 clock cycles (2.5%)
997 $ ls -l /usr/bin/kmail
998 -rwxr-xr-x  1 root  root    2149084 Oct  2 12:05 /usr/bin/kmail
999 $ ( Xvfb :3 & ) >/dev/null 2>&1 </dev/null; sleep 20
1000 $ ( DISPLAY=:3 kmail& ) >/dev/null 2>&1 </dev/null; sleep 10; killall kmail
1001 $ ( DISPLAY=:3 kmail& ) >/dev/null 2>&1 </dev/null; sleep 10

```

¹⁴ `__vt_` for GCC 2.95.x and 2.96-RH virtual tables, `_ZTV` for GCC 3.x virtual tables and `_ZTI` for GCC 3.x `RTTI` data.

```

1002 $ cat /proc//sbin/pidof kmail'/statm
1003 4164 4164 3509 224 33 3907 655
1004 $ killall Xvfb kdeinit kmail

```

Listing 12: Statistics for unprelinked kmail

statm special file for a process contains its memory statistics. The numbers in it mean in order total number of used pages (on IA-32 Linux a page is 4KB), number of resident pages (i.e. not swapped out), number of shared pages, number of text pages, number of library pages, number of stack and other pages and number of dirty pages used by the process. Distinction between text and library pages is very rough, so those numbers aren't that much useful. Of interest are mainly first number, third number and last number.

Statistics for kmail on completely prelinked box:

```

1011 $ LD_DEBUG=statistics /usr/bin/kmail 2>&1 | sed '2,8!d;s/^ *///'
1012 14864:      total startup time in dynamic loader: 8409504 clock cycles
1013 14864:      time needed for relocation: 3024720 clock cycles (35.9%)
1014 14864:      number of relocations: 0
1015 14864:      number of relocations from cache: 8961
1016 14864:      number of relative relocations: 0
1017 14864:      time needed to load objects: 4897336 clock cycles (58.2%)
1018 $ ls -l /usr/bin/kmail
1019 -rwxr-xr-x  1 root    root      2269500 Oct  2 12:05 /usr/bin/kmail
1020 $ ( Xvfb :3 & ) >/dev/null 2>&1 </dev/null; sleep 20
1021 $ ( DISPLAY=:3 kmail & ) >/dev/null 2>&1 </dev/null; sleep 10; killall kmail
1022 $ ( DISPLAY=:3 kmail & ) >/dev/null 2>&1 </dev/null; sleep 10
1023 $ cat /proc//sbin/pidof kmail'/statm
1024 3803 3803 3186 249 33 3521 617
1025 $ killall Xvfb kdeinit kmail

```

Listing 13: Statistics for prelinked kmail

Statistics for kmail on completely prelinked box with C++ conflict fixup optimizations turned off:

```

1027 $ LD_DEBUG=statistics /usr/bin/kmail 2>&1 | sed '2,8!d;s/^ *///'
1028 20645:      total startup time in dynamic loader: 9704168 clock cycles
1029 20645:      time needed for relocation: 4734715 clock cycles (48.7%)
1030 20645:      number of relocations: 0
1031 20645:      number of relocations from cache: 59871
1032 20645:      number of relative relocations: 0
1033 20645:      time needed to load objects: 4487971 clock cycles (46.2%)
1034 $ ls -l /usr/bin/kmail
1035 -rwxr-xr-x  1 root    root      2877360 Oct  2 12:05 /usr/bin/kmail
1036 $ ( Xvfb :3 & ) >/dev/null 2>&1 </dev/null; sleep 20
1037 $ ( DISPLAY=:3 kmail & ) >/dev/null 2>&1 </dev/null; sleep 10; killall kmail
1038 $ ( DISPLAY=:3 kmail & ) >/dev/null 2>&1 </dev/null; sleep 10
1039 $ cat /proc//sbin/pidof kmail'/statm
1040 3957 3957 3329 398 33 3526 628
1041 $ killall Xvfb kdeinit kmail

```

Listing 14: Statistics for prelinked kmail without conflict fixup reduction

On this application, C++ conflict fixup optimizations saved 50910 unneeded conflict fixups, speeded up startup by 13.3% and decreased number of dirty pages by 11, which means the application needs 44KB less memory per-process.

10 Thread Local Storage support

Thread Local Storage ([12], [13], [14]) support has been recently added to GCC, GNU binutils and GNU C Library. TLS support is a set of new relocations which together with dynamic linker and POSIX thread library additions provide faster and easier to use alternative to traditional POSIX thread local data API (`pthread_getspecific`, `pthread_setspecific`, `pthread_key_*`).

TLS necessitated several changes to `prelink`. Thread Local symbols (with type `STT_TLS`) must not be relocated, as they are relative to the start of `PT_TLS` segment and thus not virtual addresses. The dynamic linker had to be enhanced so that it tells `prelink` at `LD_TRACE_PRELINKING` time what TLS module IDs have been assigned and what addresses relative to start of TLS block have been given to `PT_TLS` segment of each library or executable. There are 3 classes of new TLS dynamic relocations `prelink` is interested in (with different names on different architectures).

In first class are module ID relocations, which are used for TLS Global Dynamic and Local Dynamic models (for Global Dynamic model they are supposed to resolve to module ID of the executable or shared library of particular `STT_TLS` symbol, for Local Dynamic model this resolves to module ID of the containing shared library). These relocations are hard to `prelink` in any useful way without moving TLS module ID assignment from the dynamic linker to `prelink`. Although `prelink` can find out what shared library will contain particular `STT_TLS` symbol unless there will be conflicts for that symbol, it doesn't know how many shared libraries with `PT_TLS` segment will precede it or whether executable will or will not have `PT_TLS` segment. Until TLS is widely deployed by many libraries, `prelink` could guess that only `libc.so` will have `PT_TLS` and store 1 (first module ID the dynamic linker assigns), but given that `libc.so` uses just one such relocation it is not probably worth doing this when soon other shared libraries besides `libc.so` and `libGL.so` start using it heavily. Because of this `prelink` doesn't do anything special when prelinking shared libraries with these relocations and for each relocations in this class creates one conflict fixup.

In second class are relocations which resolve to `st_value` of some `STT_TLS` symbol. These relocations are used in Global Dynamic TLS model (in Local Dynamic they are resolved at link time already) and from `prelink` point of view they are much more similar to normal relocations than the other two classes. When the `STT_TLS` symbol is looked up successfully in shared library's natural search scope, `prelink` just stores its `st_value` into the relocation. The chances there will be a conflict are even smaller than with normal symbol lookups, since overloading TLS symbols means wasted memory in each single thread and thus library writers will try to avoid it if possible.

The third class includes relocations which resolve to offsets within program's initial TLS block¹⁵ Relocation in this class are used in Initial Exec TLS model (or in Local Exec model if this model is supported in shared libraries). These offsets are even harder to predict than module IDs and unlike module IDs it wouldn't be very helpful if they were assigned by `prelink` instead of dynamic linker (which would just read them from some dynamic tag). That's because TLS block needs to be packed tightly and any assignments in `prelink` couldn't take into account other shared libraries linked into the same executable and the executable itself. Similarly to module ID relocations, `prelink` doesn't do anything about them when prelinking shared libraries and for each such relocation creates a conflict fixup.

11 Prelinking of executables and shared libraries

Rewriting of executables is harder than for shared libraries, both because there are more changes necessary and because shared libraries are relocatable and thus have dynamic relocations for all absolute addresses.

After collecting all information from the dynamic linker and assigning virtual address space slots to all shared libraries, prelinking of shared libraries involves following steps:

- Relocation of the shared library to the assigned base address.
- REL to RELA conversion if needed (the only step which changes sizes of allocated sections in the middle).
- On architectures which have `SHT_NOBITS` .plt sections, before relocations are applied the section needs to be converted to `SHT_PROGBITS`. As the section needs to be at the end (or after it) of file backed part of some `PT_LOAD` segment, this just means that the file backed up part needs to be enlarged, the file filled with zeros and all following section file offsets or program header entry file offsets adjusted. All `SHT_NOBITS` sections in

¹⁵Negative on architectures which have TLS block immediately below thread pointer (e.g. IA-32, AMD64, SPARC, S/390) and positive on architectures which have TLS block at thread pointer or a few bytes above it (e.g. PowerPC, Alpha, IA-64, SuperH).

the same PT_LOAD segment with virtual addresses lower than the .plt start address need to be converted from SHT_NOBITS to SHT_PROGBITS too. Without making the section SHT_PROGBITS, prelink cannot apply relocations against it as such sections contain only zeros. Architectures with SHT_NOBITS .plt section supported by prelink are PowerPC and PowerPC64.

- Applying relocations. For each dynamic relocation in the shared library, address of relocation's symbol looked up in natural symbol lookup search scope of the shared library (or 0 if the symbol is not found in that search scope) is stored in an architecture and relocation type dependent way to memory pointed by r_offset field of the relocation. This step uses symbol lookup information provided by dynamic linker.
- Addition or modification of DT_CHECKSUM and DT_GNU_PRELINKED dynamic tags.¹⁶ The former is set to checksum of allocated sections in the shared library, the latter to time of prelinking.
- On architectures which don't use writable .plt, but instead use .got.plt (this section is merged during linking into .got) section, prelink typically stores address into the first PLT slot in .plt section to the reserved second word of .got section. On these architectures, the dynamic linker has to initialize .plt section if lazy binding. On non-prelinked executables or shared libraries this typically means adding load offset to the values in .got.plt section, for prelinked shared libraries or executables if prelinking information cannot be used it needs to compute the right values in .got.plt section without looking at this section's content (since it contains prelinking information). The second word in .got section is used for this computation.
- Addition of .gnu_prelink_undo unallocated section if not present yet. This section is used by prelink internally during undo operation.
- Addition of .gnu_liblist and .gnu_libstr unallocated sections or, if they are already present, their update including possible growing or shrinking. These sections are used only by prelink to compare the dependent libraries (and their order) at the time when the shared library was prelinked against current dependencies. If a shared library has no dependencies (e.g. dynamic linker), these sections are not present.

Adding or resizing unallocated section needs just file offsets of following unallocated sections recomputed (ensuring proper alignment), growing section header table and .shstrtab and adding new section names to that section.

Prelinking of executables involves following steps:

- REL to RELA conversion if needed.
- SHT_NOBITS to SHT_PROGBITS conversion of .plt section if needed.
- Applying relocations.
- Addition or resizing of allocated .gnu_conflict section containing list of conflict fixups.
- Addition or resizing of allocated .gnu_liblist section which is used by the dynamic linker at runtime to see if none of the dependencies changed or were reordered. If they were, it continues normal relocation processing, otherwise they can be skipped and only conflict fixups applied.
- Growing of allocated .dynstr section, where strings referenced from .gnu_liblist section need to be added.
- If there are any COPY relocations (which prelink wants to handle rather than deferring them as conflict fixups to runtime), they need to be applied.
- Modifying second word in .got section for .got.plt using architectures.
- Addition or adjusting of dynamic tags which allow the dynamic linker to find the .gnu_liblist and .gnu_conflict sections and their sizes. DT_GNU_CONFLICT and DT_GNU_CONFLICTSZ should be present if there are any conflict fixups. It should contain the virtual address of the .gnu_conflict section start resp. its size in bytes. DT_GNU_LIBLIST and DT_GNU_LIBLISTSZ need to be present in all prelinked executables and must be equal the to virtual address of the .gnu_liblist section and its size in bytes.
- Addition of .gnu_prelink_undo unallocated section if not present.

¹⁶Prelink is not able to grow .dynamic section, so it needs some spare dynamic tags (DT_NULL) at the end of .dynamic section. GNU linker versions released after August 2001 leave space by default.

Executables can have absolute relocations already applied (and without a dynamic relocation) to virtually any allocated SHT_PROGBITS section ¹⁷, against almost all allocated SHT_PROGBITS and SHT_NOBITS sections. This means that when growing, adding or shrinking allocated sections in executables, all SHT_PROGBITS and SHT_NOBITS section must keep their original virtual addresses and sizes ¹⁸. Prelink tries various places where to put allocated sections which were added or grew:

- In the unlikely case if there is already some gap between sections in read-only PT_LOAD segment where the section fits.
- If the SHT_NOBITS sections are small enough to fit into a page together with the preceding SHT_PROGBITS section and there is still some space in the page after the SHT_NOBITS sections. In this case, prelink converts the SHT_NOBITS sections into SHT_PROGBITS sections, fills them with zeros and adds the new section after it. This doesn't increase number of PT_LOAD segments, but unfortunately those added sections are writable. This doesn't matter much for e.g. .gnu.conflict section which is only used before control is transferred to the program, but could matter for .dynstr which is used even during dlopen.
- On IA-32, executables have for historical reasons base address 0x8048000. The reason for this was that when stack was put immediately below executables, stack and the executable could coexist in the same second level page table. Linux puts the stack typically at the end of virtual address space and so keeping this exact base address is not really necessary. Prelink can decrease the base address and thus increase size of read-only PT_LOAD segment while SHT_PROGBITS and SHT_NOBITS section can stay at their previous addresses. Just their file offsets need to be increased. All these segment header adjustments need to be done in multiplies of ELF page sizes, so even if prelink chose to do similar things on architectures other than IA-32 which typically start executables on some address which is a power of 2, it would be only reasonable if ELF page size on that architecture (which can be much bigger than page size used by the operating system) is very small.
- Last possibility is to create a new PT_LOAD segment. ¹⁹ Section immediately above program header table (typically .interp) has to be moved somewhere else, but if possible close to the beginning of the executable. The new PT_LOAD segment is then added after the last PT_LOAD segment. The segment has to be writable even when all the sections in it are read-only, unless it ends exactly on a page boundary, because brk area starts immediately after the end of last PT_LOAD segment and the executable expects it to be writable.

So that verification works properly, if there is .gnu.prelink_undo section in the executable, prelink first reshuffles the sections and segments for the purpose of finding places for the sections to the original sequence as recorded in the .gnu.prelink_undo section. Examples of the above mentioned cases:

```
$ SEDCMD='s/^.* \.plt.*$/...;/\.[.*\text/,/\.[.*\got/d'
$ SEDCMD2='/Section to Segment/, $d;/^Key to/,/^Program/d;/^[A-Z]/d;/^ *$/d'
$ cat > test1.c <<EOF
int main (void) { return 0; }
EOF
$ gcc -Wl,--verbose 2>&1 \
  | sed '/^===/,/^===/!d;/^===/d;s/\rel\dyn/. += 512; &/' > test1.lds
$ gcc -s -O2 -o test1 test1.c -Wl,-T,test1.lds
$ readelf -Sl ./test1 | sed -e "$SEDCMD" -e "$SEDCMD2"
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048114	000114	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048128	000128	000020	00	A	0	0	4
[3]	.hash	HASH	08048148	000148	000024	04	A	4	0	4
[4]	.dynsym	DYNSYM	0804816c	00016c	000040	10	A	5	1	4
[5]	.dynstr	STRTAB	080481ac	0001ac	000045	00	A	0	0	1
[6]	.gnu.version	VERSYM	080481f2	0001f2	000008	02	A	4	0	2
[7]	.gnu.version_r	VERNEED	080481fc	0001fc	000020	00	A	5	1	4
[8]	.rel.dyn	REL	0804841c	00041c	000008	08	A	4	0	4

¹⁷One exception is .interp special section. It shouldn't have relocations applied to it, nor any other section should reference it.

¹⁸With a notable exception of splitting one section into two covering the same virtual address range.

¹⁹Linux kernels before 2.4.10 loaded executables which had middle PT_LOAD segment with p_memsz bigger than p_filesz incorrectly, so prelink should be only used on systems with 2.4.10 or later kernels.

```

1179 [ 9] .rel.plt      REL      08048424 000424 000008 08  A  4  b  4
1180 [10] .init         PROGBITS 0804842c 00042c 000017 00  AX  0  0  4
1181 ...
1182 [22] .bss          NOBITS   080496f8 0006f8 000004 00  WA  0  0  4
1183 [23] .comment       PROGBITS 00000000 0006f8 000132 00      0  0  1
1184 [24] .shstrtab     STRTAB   00000000 00082a 0000be 00      0  0  1
1185 Type            Offset    VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
1186 PHDR             0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
1187 INTERP           0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
1188 [Requesting program interpreter: /lib/ld-linux.so.2]
1189 LOAD              0x000000 0x08048000 0x08048000 0x005fc 0x005fc R E 0x1000
1190 LOAD              0x0005fc 0x080495fc 0x080495fc 0x000fc 0x00100 RW  0x1000
1191 DYNAMIC           0x000608 0x08049608 0x08049608 0x000c8 0x000c8 RW   0x4
1192 NOTE              0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
1193 STACK             0x000000 0x00000000 0x00000000 0x00000 0x00000 RW   0x4
1194 $ prelink -N ./test1
1195 $ readelf -Sl ./test1 | sed -e "$SEDCMD" -e "$SEDCMD2"
1196 [Nr] Name          Type            Addr      Off      Size    ES Flg Lk Inf Al
1197 [ 0]                NULL            00000000 000000 000000 00      0  0  0
1198 [ 1] .interp          PROGBITS        08048114 000114 000013 00      A  0  0  1
1199 [ 2] .note.ABI-tag    NOTE            08048128 000128 000020 00      A  0  0  4
1200 [ 3] .hash            HASH            08048148 000148 000024 04      A  4  0  4
1201 [ 4] .dynsym          DYNSYM          0804816c 00016c 000040 10      A  8  1  4
1202 [ 5] .gnu.liblist     GNU_LIBLIST     080481ac 0001ac 000028 14      A  8  0  4
1203 [ 6] .gnu.version     VERSYM          080481f2 0001f2 000008 02      A  4  0  2
1204 [ 7] .gnu.version_r   VERNEED         080481fc 0001fc 000020 00      A  8  1  4
1205 [ 8] .dynstr          STRTAB          0804821c 00021c 000058 00      A  0  0  1
1206 [ 9] .gnu.conflict    RELA            08048274 000274 0000c0 0c      A  4  0  4
1207 [10] .rel.dyn         REL             0804841c 00041c 000008 08      A  4  0  4
1208 [11] .rel.plt         REL             08048424 000424 000008 08      A  4  d  4
1209 [12] .init           PROGBITS        0804842c 00042c 000017 00     AX  0  0  4
1210 ...
1211 [24] .bss          NOBITS   080496f8 0006f8 000004 00  WA  0  0  4
1212 [25] .comment       PROGBITS 00000000 0006f8 000132 00      0  0  1
1213 [26] .gnu.prelink_undo PROGBITS 00000000 00082c 0004d4 01      0  0  4
1214 [27] .shstrtab     STRTAB   00000000 000d00 0000eb 00      0  0  1
1215 Type            Offset    VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
1216 PHDR             0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
1217 INTERP           0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
1218 [Requesting program interpreter: /lib/ld-linux.so.2]
1219 LOAD              0x000000 0x08048000 0x08048000 0x005fc 0x005fc R E 0x1000
1220 LOAD              0x0005fc 0x080495fc 0x080495fc 0x000fc 0x00100 RW  0x1000
1221 DYNAMIC           0x000608 0x08049608 0x08049608 0x000c8 0x000c8 RW   0x4
1222 NOTE              0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
1223 STACK             0x000000 0x00000000 0x00000000 0x00000 0x00000 RW   0x4

```

Listing 15: Reshuffling of an executable with a gap between sections

1224 In the above sample, there was enough space between sections (particularly between the end of the .gnu.version_r
1225 section and the start of .rel.dyn) that the new sections could be added there.

```

1226 $ SEDCMD='s/^.* \.plt.*$/.*/;/\[.*\text/,/\[.*\got/d'
1227 $ SEDCMD2='/Section to Segment/, $d; /Key to/, ^Program/d; /\[A-Z]/d; /\^ *$/'
1228 $ cat > test2.c <<EOF
1229 int main (void) { return 0; }
1230 EOF
1231 $ gcc -s -O2 -o test2 test2.c
1232 $ readelf -Sl ./test2 | sed -e "$SEDCMD" -e "$SEDCMD2"
1233 [Nr] Name          Type            Addr      Off      Size    ES Flg Lk Inf Al
1234 [ 0]                NULL            00000000 000000 000000 00      0  0  0
1235 [ 1] .interp          PROGBITS        08048114 000114 000013 00      A  0  0  1

```

```

1236 [ 2] .note.ABI-tag      NOTE           08048128 000128 000020 00  A 0  0  4
1237 [ 3] .hash             HASH           08048148 000148 000024 04  A 4  0  4
1238 [ 4] .dynsym           DYNSYM          0804816c 00016c 000040 10  A 5  1  4
1239 [ 5] .dynstr           STRTAB          080481ac 0001ac 000045 00  A 0  0  1
1240 [ 6] .gnu.version       VERSYM          080481f2 0001f2 000008 02  A 4  0  2
1241 [ 7] .gnu.version_r     VERNEED          080481fc 0001fc 000020 00  A 5  1  4
1242 [ 8] .rel.dyn           REL            0804821c 00021c 000008 08  A 4  0  4
1243 [ 9] .rel.plt           REL            08048224 000224 000008 08  A 4  b  4
1244 [10] .init              PROGBITS        0804822c 00022c 000017 00  AX 0  0  4
1245 ...
1246 [22] .bss              NOBITS          080494f8 0004f8 000004 00  WA 0  0  4
1247 [23] .comment          PROGBITS        00000000 0004f8 000132 00  0  0  1
1248 [24] .shstrtab          STRTAB          00000000 00062a 0000be 00  0  0  1
1249 Type                Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
1250 PHDR                  0x000034   0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
1251 INTERP                0x000114   0x08048114 0x08048114 0x00013 0x00013 R  0x1
1252 [Requesting program interpreter: /lib/ld-linux.so.2]
1253 LOAD                  0x000000   0x08048000 0x08048000 0x003fc 0x003fc R E 0x1000
1254 LOAD                  0x0003fc   0x080493fc 0x080493fc 0x000fc 0x00100 RW  0x1000
1255 DYNAMIC               0x000408   0x08049408 0x08049408 0x000c8 0x000c8 RW  0x4
1256 NOTE                  0x000128   0x08048128 0x08048128 0x00020 0x00020 R  0x4
1257 STACK                 0x000000   0x00000000 0x00000000 0x00000 0x00000 RW  0x4
1258 $ prelink -N ./test2
1259 $ readelf -Sl ./test2 | sed -e "$SEDCMD" -e "$SEDCMD2"
1260 [Nr] Name              Type                Addr      Off      Size    ES Flg Lk Inf Al
1261 [ 0]                      NULL               00000000 000000 000000 00  0  0  0
1262 [ 1] .interp              PROGBITS           08048114 000114 000013 00  A 0  0  1
1263 [ 2] .note.ABI-tag        NOTE               08048128 000128 000020 00  A 0  0  4
1264 [ 3] .hash                HASH               08048148 000148 000024 04  A 4  0  4
1265 [ 4] .dynsym              DYNSYM             0804816c 00016c 000040 10  A 23  1  4
1266 [ 5] .gnu.liblist          GNU_LIBLIST        080481ac 0001ac 000028 14  A 23  0  4
1267 [ 6] .gnu.version          VERSYM             080481f2 0001f2 000008 02  A 4  0  2
1268 [ 7] .gnu.version_r        VERNEED            080481fc 0001fc 000020 00  A 23  1  4
1269 [ 8] .rel.dyn              REL                0804821c 00021c 000008 08  A 4  0  4
1270 [ 9] .rel.plt              REL                08048224 000224 000008 08  A 4  b  4
1271 [10] .init                PROGBITS           0804822c 00022c 000017 00  AX 0  0  4
1272 ...
1273 [22] .bss                PROGBITS           080494f8 0004f8 000004 00  WA 0  0  4
1274 [23] .dynstr              STRTAB             080494fc 0004fc 000058 00  A 0  0  1
1275 [24] .gnu.conflict         RELA               08049554 000554 0000c0 0c  A 4  0  4
1276 [25] .comment              PROGBITS           00000000 000614 000132 00  0  0  1
1277 [26] .gnu.prelink_undo     PROGBITS           00000000 000748 0004d4 01  0  0  4
1278 [27] .shstrtab             STRTAB             00000000 000c1c 0000eb 00  0  0  1
1279 Type                Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
1280 PHDR                  0x000034   0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
1281 INTERP                0x000114   0x08048114 0x08048114 0x00013 0x00013 R  0x1
1282 [Requesting program interpreter: /lib/ld-linux.so.2]
1283 LOAD                  0x000000   0x08048000 0x08048000 0x003fc 0x003fc R E 0x1000
1284 LOAD                  0x0003fc   0x080493fc 0x080493fc 0x00218 0x00218 RW  0x1000
1285 DYNAMIC               0x000408   0x08049408 0x08049408 0x000c8 0x000c8 RW  0x4
1286 NOTE                  0x000128   0x08048128 0x08048128 0x00020 0x00020 R  0x4
1287 STACK                 0x000000   0x00000000 0x00000000 0x00000 0x00000 RW  0x4

```

Listing 16: Reshuffling of an executable with small .bss

1288 In this case .bss section was small enough that prelink converted it to SHT_PROGBITS.

```

1289 $ SEDCMD='s/^.* \.plt.*$/.*/;/\[.*\text/,/\[.*\got/d'
1290 $ SEDCMD2='/Section to Segment/, $d; /Key to/, /Program/d; /^[A-Z]/d; /^ *$/d'
1291 $ cat > test3.c <<EOF
1292 int foo [4096];

```

```

1293 int main (void) { return 0; }
1294 EOF
1295 $ gcc -s -O2 -o test3 test3.c
1296 $ readelf -Sl ./test3 | sed -e "$SEDCMD" -e "$SEDCMD2"
1297 [Nr] Name Type Addr Off Size ES Flg Lk Inf Al
1298 [ 0] NULL 00000000 000000 000000 00 0 0 0
1299 [ 1] .interp PROGBITS 08048114 000114 000013 00 A 0 0 1
1300 [ 2] .note.ABI-tag NOTE 08048128 000128 000020 00 A 0 0 4
1301 [ 3] .hash HASH 08048148 000148 000024 04 A 4 0 4
1302 [ 4] .dynsym DYNYSYM 0804816c 00016c 000040 10 A 5 1 4
1303 [ 5] .dynstr STRTAB 080481ac 0001ac 000045 00 A 0 0 1
1304 [ 6] .gnu.version VERSYM 080481f2 0001f2 000008 02 A 4 0 2
1305 [ 7] .gnu.version_r VERNEED 080481fc 0001fc 000020 00 A 5 1 4
1306 [ 8] .rel.dyn REL 0804821c 00021c 000008 08 A 4 0 4
1307 [ 9] .rel.plt REL 08048224 000224 000008 08 A 4 b 4
1308 [10] .init PROGBITS 0804822c 00022c 000017 00 AX 0 0 4
1309 ...
1310 [22] .bss NOBITS 08049500 000500 004020 00 WA 0 0 32
1311 [23] .comment PROGBITS 00000000 000500 000132 00 0 0 1
1312 [24] .shstrtab STRTAB 00000000 000632 0000be 00 0 0 1
1313 Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
1314 PHDR 0x000034 0x08048034 0x08048034 0x0000e0 0x0000e0 R E 0x4
1315 INTERP 0x000114 0x08048114 0x08048114 0x00013 0x00013 R 0x1
1316 [Requesting program interpreter: /lib/ld-linux.so.2]
1317 LOAD 0x000000 0x08048000 0x08048000 0x003fc 0x003fc R E 0x1000
1318 LOAD 0x0003fc 0x080493fc 0x080493fc 0x000fc 0x04124 RW 0x1000
1319 DYNAMIC 0x000408 0x08049408 0x08049408 0x000c8 0x000c8 RW 0x4
1320 NOTE 0x000128 0x08048128 0x08048128 0x00020 0x00020 R 0x4
1321 STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
1322 $ prelink -N ./test3
1323 $ readelf -Sl ./test3 | sed -e "$SEDCMD" -e "$SEDCMD2"
1324 [Nr] Name Type Addr Off Size ES Flg Lk Inf Al
1325 [ 0] NULL 00000000 000000 000000 00 0 0 0
1326 [ 1] .interp PROGBITS 08047114 000114 000013 00 A 0 0 1
1327 [ 2] .note.ABI-tag NOTE 08047128 000128 000020 00 A 0 0 4
1328 [ 3] .dynstr STRTAB 08047148 000148 000058 00 A 0 0 1
1329 [ 4] .gnu.liblist GNU_LIBLIST 080471a0 0001a0 000028 14 A 3 0 4
1330 [ 5] .gnu.conflict RELA 080471c8 0001c8 0000c0 0c A 7 0 4
1331 [ 6] .hash HASH 08048148 001148 000024 04 A 7 0 4
1332 [ 7] .dynsym DYNYSYM 0804816c 00116c 000040 10 A 3 1 4
1333 [ 8] .gnu.version VERSYM 080481f2 0011f2 000008 02 A 7 0 2
1334 [ 9] .gnu.version_r VERNEED 080481fc 0011fc 000020 00 A 3 1 4
1335 [10] .rel.dyn REL 0804821c 00121c 000008 08 A 7 0 4
1336 [11] .rel.plt REL 08048224 001224 000008 08 A 7 d 4
1337 [12] .init PROGBITS 0804822c 00122c 000017 00 AX 0 0 4
1338 ...
1339 [24] .bss NOBITS 08049500 0014f8 004020 00 WA 0 0 32
1340 [25] .comment PROGBITS 00000000 0014f8 000132 00 0 0 1
1341 [26] .gnu.prelink_undo PROGBITS 00000000 00162c 0004d4 01 0 0 4
1342 [27] .shstrtab STRTAB 00000000 001b00 0000eb 00 0 0 1
1343 Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
1344 PHDR 0x000034 0x08047034 0x08047034 0x0000e0 0x0000e0 R E 0x4
1345 INTERP 0x000114 0x08047114 0x08047114 0x00013 0x00013 R 0x1
1346 [Requesting program interpreter: /lib/ld-linux.so.2]
1347 LOAD 0x000000 0x08047000 0x08047000 0x013fc 0x013fc R E 0x1000
1348 LOAD 0x0013fc 0x080493fc 0x080493fc 0x000fc 0x04124 RW 0x1000
1349 DYNAMIC 0x001408 0x08049408 0x08049408 0x000c8 0x000c8 RW 0x4
1350 NOTE 0x000128 0x08047128 0x08047128 0x00020 0x00020 R 0x4
1351 STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4

```

Listing 17: Reshuffling of an executable with decreasing of base address

1352 In test3 the base address of the executable was decreased by one page and the new sections added there.

```

1353 $ SEDCMD='s/^.* \.plt.*$/...;/\[*.*\text/,/\[*.*\got/d'
1354 $ SEDCMD2='/Section to Segment/, $d;/^Key to/,/^Program/d;/^[A-Z]/d;/^ *$/d'
1355 $ cat > test4.c <<EOF
1356 int foo [4096];
1357 int main (void) { return 0; }
1358 EOF
1359 $ gcc -Wl,--verbose 2>&1 \
1360 | sed '/^===/,/^===/!d;/^===/d;s/0x08048000/0x08000000/' > test4.lds
1361 $ gcc -s -O2 -o test4 test4.c -Wl,-T,test4.lds
1362 $ readelf -Sl ./test4 | sed -e "$SEDCMD" -e "$SEDCMD2"
1363 [Nr] Name Type Addr Off Size ES Flg Lk Inf Al
1364 [ 0] NULL 00000000 000000 000000 00 0 0 0
1365 [ 1] .interp PROGBITS 08000114 000114 000013 00 A 0 0 1
1366 [ 2] .note.ABI-tag NOTE 08000128 000128 000020 00 A 0 0 4
1367 [ 3] .hash HASH 08000148 000148 000024 04 A 4 0 4
1368 [ 4] .dynsym DYNYSYM 0800016c 00016c 000040 10 A 5 1 4
1369 [ 5] .dynstr STRTAB 080001ac 0001ac 000045 00 A 0 0 1
1370 [ 6] .gnu.version VERSYM 080001f2 0001f2 000008 02 A 4 0 2
1371 [ 7] .gnu.version_r VERNEED 080001fc 0001fc 000020 00 A 5 1 4
1372 [ 8] .rel.dyn REL 0800021c 00021c 000008 08 A 4 0 4
1373 [ 9] .rel.plt REL 08000224 000224 000008 08 A 4 b 4
1374 [10] .init PROGBITS 0800022c 00022c 000017 00 AX 0 0 4
1375 ...
1376 [22] .bss NOBITS 08001500 000500 004020 00 WA 0 0 32
1377 [23] .comment PROGBITS 00000000 000500 000132 00 0 0 1
1378 [24] .shstrtab STRTAB 00000000 000632 0000be 00 0 0 1
1379 Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
1380 PHDR 0x000034 0x08000034 0x08000034 0x000e0 0x000e0 R E 0x4
1381 INTERP 0x000114 0x08000114 0x08000114 0x00013 0x00013 R 0x1
1382 [Requesting program interpreter: /lib/ld-linux.so.2]
1383 LOAD 0x000000 0x08000000 0x08000000 0x003fc 0x003fc R E 0x1000
1384 LOAD 0x0003fc 0x080013fc 0x080013fc 0x000fc 0x04124 RW 0x1000
1385 DYNAMIC 0x000408 0x08001408 0x08001408 0x000c8 0x000c8 RW 0x4
1386 NOTE 0x000128 0x08000128 0x08000128 0x00020 0x00020 R 0x4
1387 STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
1388 $ prelink -N ./test4
1389 $ readelf -Sl ./test4 | sed -e "$SEDCMD" -e "$SEDCMD2"
1390 [Nr] Name Type Addr Off Size ES Flg Lk Inf Al
1391 [ 0] NULL 00000000 000000 000000 00 0 0 0
1392 [ 1] .interp PROGBITS 08000134 000134 000013 00 A 0 0 1
1393 [ 2] .note.ABI-tag NOTE 08000148 000148 000020 00 A 0 0 4
1394 [ 3] .hash HASH 08000168 000168 000024 04 A 4 0 4
1395 [ 4] .dynsym DYNYSYM 0800018c 00018c 000040 10 A 22 1 4
1396 [ 5] .gnu.version VERSYM 080001f2 0001f2 000008 02 A 4 0 2
1397 [ 6] .gnu.version_r VERNEED 080001fc 0001fc 000020 00 A 22 1 4
1398 [ 7] .rel.dyn REL 0800021c 00021c 000008 08 A 4 0 4
1399 [ 8] .rel.plt REL 08000224 000224 000008 08 A 4 a 4
1400 [ 9] .init PROGBITS 0800022c 00022c 000017 00 AX 0 0 4
1401 ...
1402 [21] .bss NOBITS 08001500 0004f8 004020 00 WA 0 0 32
1403 [22] .dynstr STRTAB 080064f8 0004f8 000058 00 A 0 0 1
1404 [23] .gnu.liblist GNU_LIBLIST 08006550 000550 000028 14 A 22 0 4
1405 [24] .gnu.conflict RELA 08006578 000578 0000c0 0c A 4 0 4
1406 [25] .comment PROGBITS 00000000 000638 000132 00 0 0 1
1407 [26] .gnu.prelink_undo PROGBITS 00000000 00076c 0004d4 01 0 0 4
1408 [27] .shstrtab STRTAB 00000000 000c40 0000eb 00 0 0 1
1409 Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
1410 PHDR 0x000034 0x08000034 0x08000034 0x000e0 0x000e0 R E 0x4
1411 INTERP 0x000134 0x08000134 0x08000134 0x00013 0x00013 R 0x1
1412 [Requesting program interpreter: /lib/ld-linux.so.2]
1413 LOAD 0x000000 0x08000000 0x08000000 0x003fc 0x003fc R E 0x1000
1414 LOAD 0x0003fc 0x080013fc 0x080013fc 0x000fc 0x04124 RW 0x1000
1415 LOAD 0x0004f8 0x080064f8 0x080064f8 0x00140 0x00140 RW 0x1000
1416 DYNAMIC 0x000408 0x08001408 0x08001408 0x000c8 0x000c8 RW 0x4
1417 NOTE 0x000148 0x08000148 0x08000148 0x00020 0x00020 R 0x4

```

Listing 18: Reshuffling of an executable with addition of a new segment

1419 In the last example, base address was not decreased but instead a new PT_LOAD segment has been added.

1420 R_<arch>_COPY relocations are typically against first part of the SHT_NOBITS .bss section. So that prelink can
 1421 apply them, it needs to first change their section to SHT_PROGBITS, but as .bss section typically occupies much larger
 1422 part of memory, it is not desirable to convert .bss section into SHT_PROGBITS as whole. A section cannot be partly
 1423 SHT_PROGBITS and partly SHT_NOBITS, so prelink first splits the section into two parts, first .dynbss which covers
 1424 area from the start of .bss section up to highest byte to which some COPY relocation is applied and then the old .bss.
 1425 The first is converted to SHT_PROGBITS and its size is decreased, the latter stays SHT_NOBITS and its start address and
 1426 file offset are adjusted as well as its size decreased. The dynamic linker handles relocations in the executable last, so
 1427 prelink cannot just copy memory from the shared library where the symbol of the COPY relocation has been looked
 1428 up in. There might be relocations applied by the dynamic linker in normal relocation processing to the objects, so
 1429 prelink has to first process the relocations against that memory area. Relocations which don't need conflict fixups
 1430 are already applied, so prelink just needs to apply conflict fixups against the memory area, then copy it to the newly
 1431 created .dynbss section.

1432 Here is an example which shows various things which COPY relocation handling in prelink needs to deal with:

```

1433 $ cat > test1.c <<EOF
1434 struct A { char a; struct A *b; int *c; int *d; };
1435 int bar, baz;
1436 struct A foo = { 1, &foo, &bar, &baz };
1437 int *addr (void) { return &baz; }
1438 EOF
1439 $ cat > test.c <<EOF
1440 #include <stdio.h>
1441 struct A { char a; struct A *b; int *c; int *d; };
1442 int bar, *addr (void), big[8192];
1443 extern struct A foo;
1444 int main (void)
1445 {
1446     printf ("%p: %d %p %p %p %p\n", &foo, foo.a, foo.b, foo.c, foo.d,
1447         &bar, addr ());
1448 }
1449 EOF
1450 $ gcc -nostdlib -shared -fpic -s -o test1.so test1.c
1451 $ gcc -s -o test test.c ./test1.so
1452 $ ./test
1453 0x80496c0: 1 0x80496c0 0x80516e0 0x4833a4 0x80516e0 0x4833a4
1454 $ readelf -r test | sed '/\..rel\..dyn/, /\..rel\..plt/!d;/^0/!d'
1455 080496ac 00000c06 R_386_GLOB_DAT 00000000 __gmon_start__
1456 080496c0 00000605 R_386_COPY 080496c0 foo
1457 $ readelf -S test | grep bss
1458 [22] .bss NOBITS 080496c0 0006c0 008024 00 WA 0 0 32
1459 $ prelink -N ./test ./test1.so
1460 $ readelf -s test | grep foo
1461 6: 080496c0 16 OBJECT GLOBAL DEFAULT 25 foo
1462 $ readelf -s test1.so | grep foo
1463 15: 004a9314 16 OBJECT GLOBAL DEFAULT 6 foo
1464 $ readelf -r test | sed '/.gnu.conflict/, /\..rel\..dyn/!d;/^0/!d'
1465 004a9318 00000001 R_386_32 080496c0
1466 004a931c 00000001 R_386_32 080516e0
1467 005f9874 00000001 R_386_32 ffffffff0
1468 005f9878 00000001 R_386_32 00000001
1469 005f98bc 00000001 R_386_32 ffffffff4
1470 005f9900 00000001 R_386_32 ffffffffec
1471 005f9948 00000001 R_386_32 ffffffffdc

```

```

1472 005f995c 00000001 R_386_32 ffffffffe0
1473 005f9980 00000001 R_386_32 ffffffff8
1474 005f9988 00000001 R_386_32 fffffffe4
1475 005f99a4 00000001 R_386_32 fffffffd8
1476 005f99c4 00000001 R_386_32 fffffffe8
1477 005f99d8 00000001 R_386_32 08048584
1478 004c2510 00000007 R_386_JUMP_SLOT 00534460
1479 004c2514 00000007 R_386_JUMP_SLOT 00534080
1480 004c2518 00000007 R_386_JUMP_SLOT 00534750
1481 004c251c 00000007 R_386_JUMP_SLOT 005342c0
1482 004c2520 00000007 R_386_JUMP_SLOT 00534200
1483 $ objdump -s -j .dynbss test
1484
1485 test: file format elf32-i386
1486
1487 Contents of section .dynbss:
1488 80496c0 01000000 c0960408 e0160508 a4934a00 .....J.
1489 $ objdump -s -j .data test1.so
1490
1491 test1.so: file format elf32-i386
1492
1493 Contents of section .data:
1494 4a9314 01000000 14934a00 a8934a00 a4934a00 .....J...J...J.
1495 $ readelf -S test | grep bss
1496 [24] .dynbss PROGBITS 080496c0 0016c0 000010 00 WA 0 0 32
1497 [25] .bss NOBITS 080496d0 0016d0 008014 00 WA 0 0 32
1498 $ sed 's/8192/1/' test.c > test2.c
1499 $ gcc -s -o test2 test2.c ./test1.so
1500 $ readelf -S test2 | grep bss
1501 [22] .bss NOBITS 080496b0 0006b0 00001c 00 WA 0 0 8
1502 $ prelink -N ./test2 ./test1.so
1503 $ readelf -S test2 | grep bss
1504 [22] .dynbss PROGBITS 080496b0 0006b0 000010 00 WA 0 0 8
1505 [23] .bss PROGBITS 080496c0 0006c0 00000c 00 WA 0 0 8

```

Listing 19: Relocation handling of .dynbss objects

Because `test.c` executable is not compiled as position independent code and takes address of `foo` variable, a COPY relocation is needed to avoid dynamic relocation against executable's read-only PT_LOAD segment. The `foo` object in `test1.so` has one field with no relocations applied at all, one relocation against the variable itself, one relocation which needs a conflict fixup (as it is overridden by the variable in the executable) and one with relocation which doesn't need any fixups. The first and last field contain already the right values in prelinked `test1.so`, while second and third one need to be changed for symbol addresses in the executable (as shown in the `objdump` output). The conflict fixups against `foo` in `test1.so` need to stay (unless it is a C++ virtual table or *RTTI* data, i.e. not in this testcase). In `test`, `prelink` changed `.dynbss` to `SHT_PROGBITS` and kept `SHT_NOBITS .bss`, while in slightly modified testcase (`test2`) the size of `.bss` was small enough that `prelink` chose to make it `SHT_PROGBITS` too and grow the read-write PT_LOAD segment and put `.dynstr` and `.gnu.conflict` sections after it.

12 Prelink undo operation

Prelinking of shared libraries and executables is designed to be reversible, so that `prelink` operation followed by `undo` operation generates bitwise identical file to the original before prelinking. For this operation `prelink` stores the original ELF header, all the program and all section headers into a `.gnu.prelink_undo` section before it starts prelinking an unprelinked executable or shared library. When undoing the modifications, `prelink` has to convert `RELA` back to `REL` first if `REL` to `RELA` conversion was done during prelinking and all allocated sections above it relocated down to adjust for the section shrink. Relocation types which were changed when trying to avoid `REL` to `RELA` conversion need to be changed back (e.g. on IA-32, it is assumed `R_386_GLOB_DAT` relocations should be only those against `.got` section and `R_386_32` relocations in the remaining places). On `RELA` architectures, the memory pointed by `r_offset` field of the relocations needs to be reinitialized to the values stored there by the linker originally. For `prelink` it doesn't matter much what this value is (e.g. always 0, copy of `r_addend`, etc.), as long as it is computable from the

information `prelink` has during undo operation ²⁰. The GNU linker had to be changed on several architectures, so that it stores there such a value, as in several places the value e.g. depended on original addend before final link (which is not available anywhere after final link time, since `r_addend` field could be adjusted during the final link). If second word of `.got` section has been modified, it needs to be reverted back to the original value (on most architectures zero). In executables, sections which were moved during prelinking need to be put back and segments added while prelinking must be removed.

There are 3 different ways how an undo operation can be performed:

- Undoing individual executables or shared libraries specified on the command line in place (i.e. when the undo operation is successful, the prelinked executable or library is atomically replaced with the undone object).
- With `-o` option, only a single executable or shared library given on the command line is undone and stored to the file specified as `-o` option's argument.
- With `-ua` options, `prelink` builds a list of executables in paths written in its config file (plus directories and executables or libraries from command line) and all shared libraries these executables depend on. All executables and libraries in the list are then unprelinked. This option is used to unprelink the whole system. It is not perfect and needs to be worked on, since e.g. if some executable uses some shared library which no other executable links against, this executable (and shared library) is prelinked, then the executable is removed (e.g. uninstalled) but the shared library is kept, then the shared library is not unprelinked unless specifically mentioned on the command line.

13 Verification of prelinked files

As `prelink` needs to modify executables and shared libraries installed on a system, it complicates system integrity verification (e.g. `rpm -V`, TripWire). These systems store checksums of installed files into some database and during verification compute them again and compare to the values stored in the database. On a prelinked system most of the executables and shared libraries would be reported as modified. `Prelink` offers a special mode for these systems, in which it verifies that unprelinking the executable or shared library followed by immediate prelinking (with the same base address) creates bitwise identical output with the executable or shared library that's being verified. Furthermore, depending on other `prelink` options, it either writes the unprelinked image to its standard output or computes MD5 or SHA1 digest from this unprelinked image. Mere undo operation to a file and checksumming it is not good enough, since an intruder could have modified e.g. conflict fixups or memory which relocations point at, changing a behavior of the program while file after unprelinking would be unmodified.

During verification, both `prelink` executable and the dynamic linker are used, so a proper system integrity verification first checks whether `prelink` executable (which is statically linked for this reason) hasn't been modified, then uses `prelink --verify` to verify the dynamic linker (when verifying `ld` so the dynamic linker is not executed) followed by verification of other executables and libraries.

Verification requires all dependencies of checked object to be unmodified since last prelinking. If some dependency has been changed or is missing, `prelink` will report it and return with non-zero exit status. This is because prelinking depends on their content and so if they are modified, the executable or shared library might be different to one after unprelinking followed by prelinking again. In the future, perhaps it would be possible to even verify executables or shared libraries without unmodified dependencies, under the assumption that in such case the `prelink` information will not be used. It would just need to verify that nothing else but the information only used when dependencies are up to date has changed between the executable or library on the filesystem and file after unprelink followed by prelink cycle. The `prelink` operation would need to be modified in this case, so that no information is collected from the dynamic linker, the list of dependencies is assumed to be the one stored in the executable and expect it to have identical number of conflict fixups.

14 Measurements

There are two areas where `prelink` can speed things up noticeably. The primary is certainly startup time of big GUI applications where the dynamic linker spends from 100ms up to a few seconds before giving control to the application.

²⁰Such as relocation type, `r_addend` value, type, binding, flags or other attributes of relocation's symbol, what section the relocation points into or the offset within section it points to.

1570 Another area is when lots of small programs are started up, but their execution time is rather short, so the startup time
 1571 which prelink optimizes is a noticeable fraction of the total time. This is typical for shell scripting.

1572 First numbers are from lmbench benchmark, version 3.0-a3. Most of the benchmarks in lmbench suite measure kernel
 1573 speed, so it doesn't matter much whether prelink is used or not. Only in lat_proc benchmark prelink shows up
 1574 visibly. This benchmark measures 3 different things:

- 1575 • *fork proc*, which is `fork()` followed by immediate `exit(1)` in the child and `wait(0)` in the parent. The results
 1576 are (as expected) about the same between unprelinked and prelinked systems.
- 1577 • *exec proc*, i.e. `fork()` followed by immediate `close(1)` and `execve()` of a simple hello world program
 1578 (this program is compiled and linked during the benchmark into a temporary directory and is never prelinked).
 1579 The numbers are $160\mu\text{s}$ to $200\mu\text{s}$ better on prelinked systems, because there is no relocation processing needed
 1580 initially in the dynamic linker and because all relative relocations in `libc.so.6` can be skipped.
- 1581 • *sh proc*, i.e. `fork()` followed by immediate `close(1)` and `execlp("/bin/sh", "sh", "-c", "/tmp/hello",
 1582 0)`. Although the hello world program is not prelinked in this case either, the shell is, so out of the $900\mu\text{s}$ to
 1583 $1000\mu\text{s}$ speedup less than $200\mu\text{s}$ can be accounted on the speed up of the hello world program as in *exec proc*
 1584 benchmark and the rest to the speedup of shell startup.

1585 First 4 rows are from running the benchmark on a fully unprelinked system, the last 4 rows on the same system, but
 1586 fully prelinked.

```

1587           L M B E N C H 3 . 0   S U M M A R Y
1588           -----
1589 (Alpha software, do not distribute)
1590
1591 Processor, Processes - times in microseconds - smaller is better
1592 -----
1593 Host          OS  Mhz null null      open slct sig  sig  fork exec sh
1594                  call I/O stat clos TCP  inst hndl proc proc proc
1595 -----
1596 pork Linux 2.4.22 651 0.53 0.97 6.20 8.10 41.2 1.44 4.30 276. 1497 5403
1597 pork Linux 2.4.22 651 0.53 0.95 6.14 7.91 37.8 1.43 4.34 274. 1486 5391
1598 pork Linux 2.4.22 651 0.56 0.94 6.18 8.09 43.4 1.41 4.30 251. 1507 5423
1599 pork Linux 2.4.22 651 0.53 0.94 6.12 8.09 41.0 1.43 4.40 256. 1497 5385
1600 pork Linux 2.4.22 651 0.56 0.94 5.79 7.58 39.1 1.41 4.30 271. 1319 4460
1601 pork Linux 2.4.22 651 0.56 0.92 5.76 7.40 38.9 1.41 4.30 253. 1304 4417
1602 pork Linux 2.4.22 651 0.56 0.95 6.20 7.83 37.7 1.41 4.37 248. 1323 4481
1603 pork Linux 2.4.22 651 0.56 1.01 6.04 7.77 37.9 1.43 4.32 256. 1324 4457

```

Listing 20: lmbench results without and with prelinking

1604 Below is a sample timing of a 239K long configure shell script from GCC on both unprelinked and prelinked system.
 1605 Preparation step was following:

```

1606 cd; cvs -d :pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc login
1607 # Empty password
1608 cvs -d :pserver:anoncvs@subversions.gnu.org:/cvsroot/gcc -z3 co -D20031103 gcc
1609 mkdir ~/gcc/obj
1610 cd ~/gcc/obj; ../configure i386-redhat-linux; make configure-gcc

```

Listing 21: Preparation script for shell script tests

1611 On an unprelinked system, the results were:

```

1612 cd ~/gcc/obj/gcc
1613 for i in 1 2; do ./config.status --recheck > /dev/null 2>&1; done
1614 for i in 1 2 3 4; do time ./config.status --recheck > /dev/null 2>&1; done
1615
1616 real    0m4.436s
1617 user    0m1.730s
1618 sys     0m1.260s
1619
1620 real    0m4.409s
1621 user    0m1.660s
1622 sys     0m1.340s
1623
1624 real    0m4.431s
1625 user    0m1.810s
1626 sys     0m1.300s
1627
1628 real    0m4.432s
1629 user    0m1.670s
1630 sys     0m1.210s

```

Listing 22: Shell script test results on unprelinked system

1631 and on a fully prelinked system:

```

1632 cd ~/gcc/obj/gcc
1633 for i in 1 2; do ./config.status --recheck > /dev/null 2>&1; done
1634 for i in 1 2 3 4; do time ./config.status --recheck > /dev/null 2>&1; done
1635
1636 real    0m4.126s
1637 user    0m1.590s
1638 sys     0m1.240s
1639
1640 real    0m4.151s
1641 user    0m1.620s
1642 sys     0m1.230s
1643
1644 real    0m4.161s
1645 user    0m1.600s
1646 sys     0m1.190s
1647
1648 real    0m4.122s
1649 user    0m1.570s
1650 sys     0m1.230s

```

Listing 23: Shell script test results on prelinked system

1651 Now timing of a few big GUI programs. All timings were done without X server running and with DISPLAY environ-
1652 ment variable not set (so that when control is transfered to the application, it very soon finds out there is no X server
1653 it can talk to and bail out). The measurements are done by the dynamic linker in ticks on a 651MHz dual Pentium III
1654 machine, i.e. ticks have to be divided by 651000000 to get times in seconds. Each application has been run 4 times and
1655 the results with smallest total time spent in the dynamic linker was chosen. Epiphany WWW browser and Evolution
1656 mail client were chosen as examples of Gtk+ applications (typically they use really many shared libraries, but many
1657 of them are quite small, there aren't really many relocations nor conflict fixups and most of the libraries are written
1658 in C) and Konqueror WWW browser and KWord word processor were chosen as examples of KDE applications (typ-
1659 ically they use slightly fewer shared libraries, though still a lot, most of the shared libraries are written in C++, have
1660 many relocations and cause many conflict fixups, especially without C++ conflict fixup optimizations in prelink).
1661 On non-prelinked system, the timings are done with lazy binding, i.e. without LD_BIND_NOW=1 set in the environment.
1662 This is because that's how people generally run programs, on the other side it is not exact apples to apples comparison,

```

1663 since on prelinked system there is no lazy binding with the exception of shared libraries loaded through dlopen. So
1664 when control is passed to the application, prelinked programs should be slightly faster for a while since non-prelinked
1665 programs will have to do symbol lookups and processing relocations (and on various architectures flushing instruction
1666 caches) whenever they call some function they haven't called before in particular shared library or in the executable.

1667 $ ldd 'which epiphany-bin' | wc -l
1668      64
1669 $ # Unprelinked system
1670 $ LD_DEBUG=statistics epiphany-bin 2>&1 | sed 's/^ *///'
1671 18960:
1672 18960: runtime linker statistics:
1673 18960: total startup time in dynamic loader: 67336593 clock cycles
1674 18960: time needed for relocation: 58119983 clock cycles (86.3%)
1675 18960: number of relocations: 6999
1676 18960: number of relocations from cache: 4770
1677 18960: number of relative relocations: 31494
1678 18960: time needed to load objects: 8696104 clock cycles (12.9%)
1679
1680 (epiphany-bin:18960): Gtk-WARNING **: cannot open display:
1681 18960:
1682 18960: runtime linker statistics:
1683 18960: final number of relocations: 7692
1684 18960: final number of relocations from cache: 4770
1685 $ # Prelinked system
1686 $ LD_DEBUG=statistics epiphany-bin 2>&1 | sed 's/^ *///'
1687 25697:
1688 25697: runtime linker statistics:
1689 25697: total startup time in dynamic loader: 7313721 clock cycles
1690 25697: time needed for relocation: 565680 clock cycles (7.7%)
1691 25697: number of relocations: 0
1692 25697: number of relocations from cache: 1205
1693 25697: number of relative relocations: 0
1694 25697: time needed to load objects: 6179467 clock cycles (84.4%)
1695
1696 (epiphany-bin:25697): Gtk-WARNING **: cannot open display:
1697 25697:
1698 25697: runtime linker statistics:
1699 25697: final number of relocations: 31
1700 25697: final number of relocations from cache: 1205
1701
1702 $ ldd 'which evolution' | wc -l
1703      68
1704 $ # Unprelinked system
1705 $ LD_DEBUG=statistics evolution 2>&1 | sed 's/^ *///'
1706 19042:
1707 19042: runtime linker statistics:
1708 19042: total startup time in dynamic loader: 54382122 clock cycles
1709 19042: time needed for relocation: 43403190 clock cycles (79.8%)
1710 19042: number of relocations: 3452
1711 19042: number of relocations from cache: 2885
1712 19042: number of relative relocations: 34957
1713 19042: time needed to load objects: 10450142 clock cycles (19.2%)
1714
1715 (evolution:19042): Gtk-WARNING **: cannot open display:
1716 19042:
1717 19042: runtime linker statistics:
1718 19042: final number of relocations: 4075
1719 19042: final number of relocations from cache: 2885
1720 $ # Prelinked system
1721 $ LD_DEBUG=statistics evolution 2>&1 | sed 's/^ *///'
1722 25723:
1723 25723: runtime linker statistics:
1724 25723: total startup time in dynamic loader: 9176140 clock cycles

```

```

1725 25723:          time needed for relocation: 203783 clock cycles (2.2%)
1726 25723:          number of relocations: 0
1727 25723:          number of relocations from cache: 525
1728 25723:          number of relative relocations: 0
1729 25723:          time needed to load objects: 8405157 clock cycles (91.5%)
1730
1731 (evolution:25723): Gtk-WARNING **: cannot open display:
1732 25723:
1733 25723: runtime linker statistics:
1734 25723:          final number of relocations: 31
1735 25723: final number of relocations from cache: 525
1736
1737 $ ldd 'which konqueror' | wc -l
1738      37
1739 $ # Unprelinked system
1740 $ LD_DEBUG=statistics konqueror 2>&1 | sed 's/^ *///'
1741 18979:
1742 18979: runtime linker statistics:
1743 18979:   total startup time in dynamic loader: 131985703 clock cycles
1744 18979:   time needed for relocation: 127341077 clock cycles (96.4%)
1745 18979:   number of relocations: 25473
1746 18979:   number of relocations from cache: 53594
1747 18979:   number of relative relocations: 31171
1748 18979:   time needed to load objects: 4318803 clock cycles (3.2%)
1749 konqueror: cannot connect to X server
1750 18979:
1751 18979: runtime linker statistics:
1752 18979:   final number of relocations: 25759
1753 18979: final number of relocations from cache: 53594
1754 $ # Prelinked system
1755 $ LD_DEBUG=statistics konqueror 2>&1 | sed 's/^ *///'
1756 25733:
1757 25733: runtime linker statistics:
1758 25733:   total startup time in dynamic loader: 5533696 clock cycles
1759 25733:   time needed for relocation: 1941489 clock cycles (35.0%)
1760 25733:   number of relocations: 0
1761 25733:   number of relocations from cache: 2066
1762 25733:   number of relative relocations: 0
1763 25733:   time needed to load objects: 3217736 clock cycles (58.1%)
1764 konqueror: cannot connect to X server
1765 25733:
1766 25733: runtime linker statistics:
1767 25733:   final number of relocations: 0
1768 25733: final number of relocations from cache: 2066
1769
1770 $ ldd 'which kword' | wc -l
1771      40
1772 $ # Unprelinked system
1773 $ LD_DEBUG=statistics kword 2>&1 | sed 's/^ *///'
1774 19065:
1775 19065: runtime linker statistics:
1776 19065:   total startup time in dynamic loader: 153684591 clock cycles
1777 19065:   time needed for relocation: 148255294 clock cycles (96.4%)
1778 19065:   number of relocations: 26231
1779 19065:   number of relocations from cache: 55833
1780 19065:   number of relative relocations: 30660
1781 19065:   time needed to load objects: 5068746 clock cycles (3.2%)
1782 kword: cannot connect to X server
1783 19065:
1784 19065: runtime linker statistics:
1785 19065:   final number of relocations: 26528
1786 19065: final number of relocations from cache: 55833
1787 $ # Prelinked system
1788 $ LD_DEBUG=statistics kword 2>&1 | sed 's/^ *///'
1789 25749:

```

```

1790 25749: runtime linker statistics:
1791 25749:     total startup time in dynamic loader: 6516635 clock cycles
1792 25749:         time needed for relocation: 2106856 clock cycles (32.3%)
1793 25749:             number of relocations: 0
1794 25749:         number of relocations from cache: 2130
1795 25749:             number of relative relocations: 0
1796 25749:         time needed to load objects: 4008585 clock cycles (61.5%)
1797 kword: cannot connect to X server
1798 25749:
1799 25749: runtime linker statistics:
1800 25749:         final number of relocations: 0
1801 25749:     final number of relocations from cache: 2130

```

Listing 24: Dynamic linker statistics for unprelinked and prelinked GUI programs

1802 In the case of above mentioned Gtk+ applications, the original startup time spent in the dynamic linker decreased into
1803 11% to 17% of the original times, with KDE applications it decreased even into around 4.2% of original times.

1804 The startup time reported by the dynamic linker is only part of the total startup time of a GUI program. Unfortunately it
1805 cannot be measured very accurately without patching each application separately, so that it would print current process
1806 CPU time at the point when all windows are painted and the process starts waiting for user input. The following table
1807 contains values reported by `time(1)` command on each of the 4 GUI programs running under X, both on unprelinked
1808 and fully prelinked system. As soon as each program painted its windows, it was killed by application's quit hot key
1809 ²¹. Especially the real time values depend also on the speed of human reactions, so each measurement was repeated
1810 10 times. All timings were done with hot caches, after running the applications two times before measurement.

Type	Values (in seconds)										Average	Std.Dev.
	unprelinked epiphany											
real	3.053	2.84	2.996	2.901	3.019	2.929	2.883	2.975	2.922	3.026	2.954	0.0698
user	2.33	2.31	2.28	2.32	2.44	2.37	2.29	2.35	2.34	2.41	2.344	0.0508
sys	0.2	0.23	0.23	0.19	0.19	0.12	0.25	0.16	0.14	0.14	0.185	0.0440
	prelinked epiphany											
real	2.773	2.743	2.833	2.753	2.753	2.644	2.717	2.897	2.68	2.761	2.755	0.0716
user	2.18	2.17	2.17	2.12	2.23	2.26	2.13	2.17	2.15	2.15	2.173	0.0430
sys	0.13	0.15	0.18	0.15	0.11	0.04	0.18	0.14	0.1	0.15	0.133	0.0416
	unprelinked evolution											
real	2.106	1.886	1.828	2.12	1.867	1.871	2.242	1.871	1.862	2.241	1.989	0.1679
user	1.12	1.09	1.15	1.19	1.17	1.23	1.15	1.11	1.17	1.14	1.152	0.0408
sys	0.1	0.11	0.13	0.07	0.1	0.05	0.11	0.11	0.09	0.08	0.095	0.0232
	prelinked evolution											
real	1.684	1.621	1.686	1.72	1.694	1.691	1.631	1.697	1.668	1.535	1.663	0.0541
user	0.92	0.87	0.92	0.95	0.79	0.86	0.94	0.87	0.89	0.86	0.887	0.0476
sys	0.06	0.1	0.06	0.05	0.11	0.08	0.07	0.1	0.12	0.07	0.082	0.0239
	unprelinked kword											
real	2.111	1.414	1.36	1.356	1.259	1.383	1.28	1.321	1.252	1.407	1.414	0.2517
user	1.04	0.9	0.93	0.88	0.89	0.89	0.87	0.89	0.9	0.8	0.899	0.0597
sys	0.07	0.04	0.06	0.05	0.06	0.1	0.09	0.08	0.08	0.12	0.075	0.0242
	prelinked kword											
real	1.59	1.052	0.972	1.064	1.106	1.087	1.066	1.087	1.065	1.005	1.109	0.1735
user	0.61	0.53	0.58	0.6	0.6	0.58	0.59	0.61	0.57	0.6	0.587	0.0241
sys	0.08	0.08	0.06	0.06	0.03	0.07	0.06	0.03	0.06	0.04	0.057	0.0183
	unprelinked konqueror											
real	1.306	1.386	1.27	1.243	1.227	1.286	1.262	1.322	1.345	1.332	1.298	0.0495
user	0.88	0.86	0.88	0.9	0.87	0.83	0.83	0.86	0.86	0.89	0.866	0.0232
sys	0.07	0.11	0.12	0.1	0.12	0.08	0.13	0.12	0.09	0.08	0.102	0.0210
	prelinked konqueror											
real	1.056	0.962	0.961	0.906	0.927	0.923	0.933	0.958	0.955	1.142	0.972	0.0722
user	0.56	0.6	0.56	0.52	0.57	0.58	0.5	0.57	0.61	0.55	0.562	0.0334

²¹Ctrl+W for Epiphany, Ctrl+Q for Evolution and Konqueror and Enter in Kword's document type choice dialog.

Type	Values (in seconds)										Average	Std.Dev.
sys	0.1	0.13	0.08	0.15	0.07	0.09	0.09	0.09	0.1	0.08	0.098	0.0244

Table 1: GUI program start up times without and with prelinking

1811

1812 OpenOffice.org is probably the largest program these days in Linux, mostly written in C++. In OpenOffice.org
1813 1.1, the main executable, soffice.bin, links directly against 34 shared libraries, but typically during startup it loads
1814 using dlopen many others. As has been mentioned earlier, prelink cannot speed up loading shared libraries using
1815 dlopen, since it cannot predict in which order and what shared libraries will be loaded (and thus cannot compute
1816 conflict fixups). The soffice.bin is typically started through a wrapper script and depending on what arguments
1817 are passed to it, different OpenOffice.org application is started. With no options, it starts just empty window with
1818 menu from which the applications can be started, with say private:factory/swriter argument it starts a word
1819 processor, with private:factory/scalc it starts a spreadsheet etc. When soffice.bin is already running, if you
1820 start another copy of it, it just instructs the already running copy to pop up a new window and exits.

1821 In an experiment, soffice.bin has been invoked 7 times against running X server, with no arguments, private:factory/swriter,
1822 private:factory/scalc, private:factory/sdraw, private:factory/simpress, private:factory/smath
1823 arguments (in all these cases nothing was pressed at all) and last with the private:factory/swriter argument
1824 where the menu item New Presentation was selected and the word processor window closed. In all these cases,
1825 /proc/'pidof soffice.bin'/maps file was captured and the application then killed. This file contains among
1826 other things list of all shared libraries mmaped by the process at the point where it started waiting for user input
1827 after loading up. These lists were then summarized, to get number of the runs in which particular shared library was
1828 loaded up out of the total 7 runs. There were 38 shared libraries shipped as part of OpenOffice.org package which
1829 have been loaded in all 7 times, another 3 shared libraries included in OpenOffice.org (and also one shared library
1830 shipped in another package, libdb.cxx-4.1.so) which were loaded 6 times.²² There was one shared library loaded
1831 in 5 runs, but was locale specific and thus not worth considering. Inspecting OpenOffice.org source, these shared
1832 libraries are never unloaded with dlclose, so soffice.bin can be made much more prelink friendly and thus save
1833 substantial amount of startup time by linking against all those 76 shared libraries instead of just 34 shared libraries it is
1834 linked against. In the timings below, soffice1.bin is the original soffice.bin as created by the OpenOffice.org
1835 makefiles and soffice3.bin is the same executable linked dynamically against additional 42 shared libraries. The
1836 ordering of those 42 shared libraries matters for the number of conflict fixups, unfortunately with large C++ shared
1837 libraries there is no obvious rule for ordering them as sometimes it is more useful when a shared library precedes its
1838 dependency and sometimes vice versa, so a few different orderings were tried in several steps and always the one with
1839 smallest number of conflict fixups was chosen. Still, the number of conflict fixups is quite high and big part of the
1840 fixups are storing addresses of PLT slots in the executable into various places in shared libraries²³ soffice2.bin is
1841 another experiment, where the executable itself is empty source file, all objects which were originally in soffice.bin
1842 executable with the exception of start files were recompiled as position independent code and linked into a new shared
1843 library. This reduced number of conflicts a lot and speeded up start up times against soffice3.bin when caches are
1844 hot. It is a little bit slower than soffice3.bin when running with cold caches (e.g. for the first time after bootup), as
1845 there is one more shared library to load etc.

1846 In the timings below, numbers for soffice1.bin and soffice2.bin resp. soffice3.bin cannot be easily com-
1847 pared, as soffice1.bin loads less than half of the needed shared libraries which the remaining two executables load
1848 and the time to load those shared libraries doesn't show up there. Still, when it is prelinked it takes just slightly more
1849 than two times longer to load soffice2.bin than soffice1.bin and the times are still less than 7% of how long it
1850 takes to load just the initial 34 shared libraries when not prelinking.

```
1851 $ S='s/^ *///'
1852 $ ldd /usr/lib/openoffice/program/soffice1.bin | wc -l
1853 34
```

²²In all runs but when ran without arguments. But when the application is started without any arguments, it cannot do any useful work, so one loads one of the applications afterward anyway.

²³This might get better when the linker is modified to handle calls without ever taking address of the function in executables specially, but only testing it will actually show it up.

```

1854 $ # Unprelinked system
1855 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice1.bin 2>&1 | sed "$S"
1856 19095:
1857 19095: runtime linker statistics:
1858 19095:     total startup time in dynamic loader: 159833582 clock cycles
1859 19095:         time needed for relocation: 155464174 clock cycles (97.2%)
1860 19095:             number of relocations: 31136
1861 19095:         number of relocations from cache: 31702
1862 19095:             number of relative relocations: 18284
1863 19095:         time needed to load objects: 3919645 clock cycles (2.4%)
1864 /usr/lib/openoffice/program/soffice1.bin X11 error: Can't open display:
1865 Set DISPLAY environment variable, use -display option
1866 or check permissions of your X-Server
1867 (See "man X" resp. "man xhost" for details)
1868 19095:
1869 19095: runtime linker statistics:
1870 19095:         final number of relocations: 31715
1871 19095:     final number of relocations from cache: 31702
1872 $ # Prelinked system
1873 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice1.bin 2>&1 | sed "$S"
1874 25759:
1875 25759: runtime linker statistics:
1876 25759:     total startup time in dynamic loader: 4252397 clock cycles
1877 25759:         time needed for relocation: 1189840 clock cycles (27.9%)
1878 25759:             number of relocations: 0
1879 25759:         number of relocations from cache: 2142
1880 25759:             number of relative relocations: 0
1881 25759:         time needed to load objects: 2604486 clock cycles (61.2%)
1882 /usr/lib/openoffice/program/soffice1.bin X11 error: Can't open display:
1883 Set DISPLAY environment variable, use -display option
1884 or check permissions of your X-Server
1885 (See "man X" resp. "man xhost" for details)
1886 25759:
1887 25759: runtime linker statistics:
1888 25759:         final number of relocations: 24
1889 25759:     final number of relocations from cache: 2142
1890 $ ldd /usr/lib/openoffice/program/soffice2.bin | wc -l
1891 77
1892 $ # Unprelinked system
1893 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice2.bin 2>&1 | sed "$S"
1894 19115:
1895 19115: runtime linker statistics:
1896 19115:     total startup time in dynamic loader: 947793670 clock cycles
1897 19115:         time needed for relocation: 936895741 clock cycles (98.8%)
1898 19115:             number of relocations: 69164
1899 19115:         number of relocations from cache: 94502
1900 19115:             number of relative relocations: 59374
1901 19115:         time needed to load objects: 10046486 clock cycles (1.0%)
1902 /usr/lib/openoffice/program/soffice2.bin X11 error: Can't open display:
1903 Set DISPLAY environment variable, use -display option
1904 or check permissions of your X-Server
1905 (See "man X" resp. "man xhost" for details)
1906 19115:
1907 19115: runtime linker statistics:
1908 19115:         final number of relocations: 69966
1909 19115:     final number of relocations from cache: 94502
1910 $ # Prelinked system
1911 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice2.bin 2>&1 | sed "$S"
1912 25777:
1913 25777: runtime linker statistics:
1914 25777:     total startup time in dynamic loader: 10952099 clock cycles
1915 25777:         time needed for relocation: 3254518 clock cycles (29.7%)
1916 25777:             number of relocations: 0
1917 25777:         number of relocations from cache: 5309
1918 25777:             number of relative relocations: 0

```

```

1919 25777:                time needed to load objects: 6805013 clock cycles (62.1%)
1920 /usr/lib/openoffice/program/soffice2.bin X11 error: Can't open display:
1921 Set DISPLAY environment variable, use -display option
1922 or check permissions of your X-Server
1923 (See "man X" resp. "man xhost" for details)
1924 25777:
1925 25777: runtime linker statistics:
1926 25777:                final number of relocations: 24
1927 25777: final number of relocations from cache: 5309
1928 $ ldd /usr/lib/openoffice/program/soffice3.bin | wc -l
1929      76
1930 $ # Unprelinked system
1931 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice3.bin 2>&1 | sed "$S"
1932 19131:
1933 19131: runtime linker statistics:
1934 19131: total startup time in dynamic loader: 852275754 clock cycles
1935 19131: time needed for relocation: 840996859 clock cycles (98.6%)
1936 19131: number of relocations: 68362
1937 19131: number of relocations from cache: 89213
1938 19131: number of relative relocations: 55831
1939 19131: time needed to load objects: 10170207 clock cycles (1.1%)
1940 /usr/lib/openoffice/program/soffice3.bin X11 error: Can't open display:
1941 Set DISPLAY environment variable, use -display option
1942 or check permissions of your X-Server
1943 (See "man X" resp. "man xhost" for details)
1944 19131:
1945 19131: runtime linker statistics:
1946 19131: final number of relocations: 69177
1947 19131: final number of relocations from cache: 89213
1948 $ # Prelinked system
1949 $ LD_DEBUG=statistics /usr/lib/openoffice/program/soffice3.bin 2>&1 | sed "$S"
1950 25847:
1951 25847: runtime linker statistics:
1952 25847: total startup time in dynamic loader: 12277407 clock cycles
1953 25847: time needed for relocation: 4232915 clock cycles (34.4%)
1954 25847: number of relocations: 0
1955 25847: number of relocations from cache: 8961
1956 25847: number of relative relocations: 0
1957 25847: time needed to load objects: 6925023 clock cycles (56.4%)
1958 /usr/lib/openoffice/program/soffice3.bin X11 error: Can't open display:
1959 Set DISPLAY environment variable, use -display option
1960 or check permissions of your X-Server
1961 (See "man X" resp. "man xhost" for details)
1962 25847:
1963 25847: runtime linker statistics:
1964 25847: final number of relocations: 24
1965 25847: final number of relocations from cache: 8961

```

Listing 25: Dynamic linker statistics for unprelinked and prelinked OpenOffice.org

Below are measurement using time(1) for each of the soffice.bin variants, prelinked and unprelinked. OpenOffice.org was killed immediately after painting Writer's window using Ctrl+Q.

Type	Values (in seconds)										Average	Std.Dev.
	unprelinked soffice1.bin private:factory/swriter											
real	5.569	5.149	5.547	5.559	5.549	5.139	5.55	5.559	5.598	5.559	5.478	0.1765
user	4.65	4.57	4.62	4.64	4.57	4.55	4.65	4.49	4.52	4.46	4.572	0.0680
sys	0.29	0.24	0.19	0.21	0.21	0.21	0.25	0.25	0.27	0.26	0.238	0.0319
	prelinked soffice1.bin private:factory/swriter											
real	4.946	4.899	5.291	4.879	4.879	4.898	5.299	4.901	4.887	4.901	4.978	0.1681
user	4.23	4.27	4.18	4.24	4.17	4.22	4.15	4.25	4.26	4.31	4.228	0.0494

Type	Values (in seconds)										Average	Std.Dev.
sys	0.22	0.22	0.24	0.26	0.3	0.26	0.29	0.17	0.21	0.23	0.24	0.0389
	unprelinked soffice2.bin private:factory/swriter											
real	5.575	5.166	5.592	5.149	5.571	5.559	5.159	5.157	5.569	5.149	5.365	0.2201
user	4.59	4.5	4.57	4.37	4.47	4.57	4.56	4.41	4.63	4.5	4.517	0.0826
sys	0.24	0.24	0.21	0.34	0.27	0.19	0.19	0.27	0.19	0.29	0.243	0.0501
	prelinked soffice2.bin private:factory/swriter											
real	3.69	3.66	3.658	3.661	3.639	3.638	3.649	3.659	3.65	3.659	3.656	0.0146
user	2.93	2.88	2.88	2.9	2.84	2.63	2.89	2.85	2.77	2.83	2.84	0.0860
sys	0.22	0.18	0.23	0.2	0.18	0.29	0.22	0.23	0.24	0.22	0.221	0.0318
	unprelinked soffice3.bin private:factory/swriter											
real	5.031	5.02	5.009	5.028	5.019	5.019	5.019	5.052	5.426	5.029	5.065	0.1273
user	4.31	4.35	4.34	4.3	4.38	4.29	4.45	4.37	4.38	4.44	4.361	0.0547
sys	0.27	0.25	0.26	0.27	0.27	0.31	0.18	0.17	0.16	0.15	0.229	0.0576
	prelinked soffice3.bin private:factory/swriter											
real	3.705	3.669	3.659	3.669	3.66	3.659	3.659	3.661	3.668	3.649	3.666	0.0151
user	2.86	2.88	2.85	2.84	2.83	2.86	2.84	2.91	2.86	2.8	2.853	0.0295
sys	0.26	0.19	0.27	0.25	0.24	0.23	0.28	0.21	0.21	0.27	0.241	0.0303

Table 2: OpenOffice.org start up times without and with prelinking

15 Similar tools on other ELF using Operating Systems

Something similar to `prelink` is available on other ELF platforms. On Irix there is `QUICKSTART` and on Solaris `crle`.

SGI `QUICKSTART` is much closer to `prelink` from these two. The `rqs` program relocates libraries to (if possible) unique virtual address space slot. The base address is either specified on the command line with the `-l` option, or `rqs` uses a `so_locations` registry with `-c` or `-u` options and finds a not yet occupied slot. This is similar to how `prelink` lays out libraries without the `-m` option.

`QUICKSTART` uses the same data structure for library lists (`ElfNN.Lib`) as `prelink`, but uses more fields in it (`prelink` doesn't use `l.version` and `l.flags` fields at the moment) and uses different dynamic tags and section type for it. Another difference is that `QUICKSTART` makes all `liblist` section `SHF_ALLOC`, whether in shared libraries or executables. `prelink` only needs `liblist` section in the executable be allocated, `liblist` sections in shared libraries are not allocated and used at `prelink` time only.

The biggest difference between `QUICKSTART` and `prelink` is in how conflicts are encoded. SGI stores them in a very compact format, as array of `.dynsym` section indexes for symbols which are conflicting. There is no information publicly available what exactly SGI dynamic linker does when it is resolving the conflicts, so this is just a guess. Given that the conflicts can be stored in a shared library or executable different to the shared library with the relocations against the conflicting symbol and different to the shared library which the symbol was originally resolved to, there doesn't seem to be an obvious way how to handle the conflicts very cheaply. The dynamic linker probably collects list of all conflicting symbol names, for each such symbol computes ELF hash and walks hash buckets for this hash of all shared libraries, looking for the symbol. Every time it finds the symbol, all relocations against it need to be redone. Unlike this, `prelink` stores conflicts as an array of `ElfNN.Rela` structures, with one entry for each shared relocation against conflicting symbol in some shared library. This guarantees that there are no symbol lookups during program startup (provided that shared libraries have not been changed after prelinking), while with `QUICKSTART` will do some symbol lookups if there are any conflicts. `QUICKSTART` puts conflict sections into the executable and every shared library where `rqs` determines conflicts while `prelink` stores them in the executable only (but the array is typically much bigger). Disk space requirements for prelinked executables are certainly bigger than for requickstarted executables, but which one has bigger runtime memory requirements is unclear. If prelinking can be used, all `.rela*` and `.rel*` sections in the executable and all shared libraries are skipped, so they will not need to be paged in during whole program's life (with the exception of first and last pages in the relocation sections which can be paged in because of other sections on the same page), but whole `.gnu.conflict` section needs to be paged in (read-only) and processed. With `QUICKSTART`, probably all (much smaller) conflict sections need to be paged in and also likely for each conflict whole relocation sections of each library which needs the conflict to be applied against.

1999 In QUICKSTART documentation, SGI says that conflicts are very costly and that developers should avoid them. Un-
2000 fortunately, this is sometimes quite hard, especially with C++ shared libraries. It is unclear whether rqs does any
2001 optimizations to trim down the number of conflicts.

2002 Sun took completely different approach. The dynamic linker provides a `dldump` (`const char *ipath, const`
2003 `char *opath, int flags`); function. `ipath` is supposed to be a path to an ELF object loaded already in the current
2004 process. This function creates a new ELF object at `opath`, which is like the `ipath` object, but relocated to the base address
2005 which it has actually been mapped at in the current process and with some relocations (specified in `flags` bitmask)
2006 applied as they have been resolved in the current process. Relocations, which have been applied, are overwritten in
2007 the relocation sections with `R_*_NONE` relocations. The `cr1e` executable, in addition to other functions not related to
2008 startup times, with some specific options uses the `dldump` function to dump all shared libraries a particular executable
2009 uses (and the executable itself) into a new directory, with selected relocation classes being already applied. The main
2010 disadvantage of this approach is that such alternate shared libraries are at least for most relocation classes not shareable
2011 across different programs at all (and for those where they could be shareable a little bit there will be many relocations
2012 left for the dynamic linker, so the speed gains will be small). Another disadvantage is that all relocation sections need
2013 to be paged into the memory, just to find out that most of the relocations are `R_*_NONE`.

16 ELF extensions for prelink

2014 Prelink needs a few ELF extensions for its data structures in ELF objects. For list of dependencies at the time of
2015 prelinking, a new section type `SHT_GNU_LIBLIST` is defined:

```
2016 #define SHT_GNU_LIBLIST    0x6ffffff7 /* Prelink library list */
2017
2018 typedef struct
2019 {
2020     Elf32_Word l_name;           /* Name (string table index) */
2021     Elf32_Word l_time_stamp;     /* Timestamp */
2022     Elf32_Word l_checksum;       /* Checksum */
2023     Elf32_Word l_version;        /* Unused, should be zero */
2024     Elf32_Word l_flags;          /* Unused, should be zero */
2025 } Elf32_Lib;
2026
2027 typedef struct
2028 {
2029     Elf64_Word l_name;           /* Name (string table index) */
2030     Elf64_Word l_time_stamp;     /* Timestamp */
2031     Elf64_Word l_checksum;       /* Checksum */
2032     Elf64_Word l_version;        /* Unused, should be zero */
2033     Elf64_Word l_flags;          /* Unused, should be zero */
2034 } Elf64_Lib;
```

Listing 26: New structures and section type constants used by prelink

2035 Introduces a few new special sections:

Name	Type	Attributes
<i>In shared libraries</i>		
<code>.gnu.liblist</code>	<code>SHT_GNU_LIBLIST</code>	0
<code>.gnu.libstr</code>	<code>SHT_STRTAB</code>	0
<code>.gnu.prelink_undo</code>	<code>SHT_PROGBITS</code>	0
<i>In executables</i>		
<code>.gnu.liblist</code>	<code>SHT_GNU_LIBLIST</code>	<code>SHF_ALLOC</code>
<code>.gnu.conflict</code>	<code>SHT_RELA</code>	<code>SHF_ALLOC</code>
<code>.gnu.prelink_undo</code>	<code>SHT_PROGBITS</code>	0

Table 3: Special sections introduced by prelink

2036

2037 **.gnu.liblist** This section contains one `ElfNN_Lib` structure for each shared library which the object has been pre-
 2038 linked against, in the order in which they appear in symbol search scope. Section's `sh_link` value should contain
 2039 section index of `.gnu.libstr` for shared libraries and section index of `.dynsym` for executables. `l_name` field
 2040 contains the dependent library's name as index into the section pointed by `sh_link` field. `l_time_stamp` resp.
 2041 `l_checksum` should contain copies of `DT_GNU_PRELINKED` resp. `DT_CHECKSUM` values of the dependent library.

2042 **.gnu.conflict** This section contains one `ElfNN_Rela` structure for each needed prelink conflict fixup. `r_offset`
 2043 field contains the absolute address at which the fixup needs to be applied, `r_addend` the value that needs to be
 2044 stored at that location. `ELFNN_R_SYM` of `r_info` field should be zero, `ELFNN_R_TYPE` of `r_info` field should be
 2045 architecture specific relocation type which should be handled the same as for `.rela.*` sections on the archi-
 2046 tecture. For EM_ALPHA machine, all types with `R_ALPHA_JMP_SLOT` in lowest 8 bits of `ELF64_R_TYPE` should be
 2047 handled as `R_ALPHA_JMP_SLOT` relocation, the upper 24 bits contains index in original `.rela.plt` section of the
 2048 `R_ALPHA_JMP_SLOT` relocation the fixup was created for.

2049 **.gnu.libstr** This section contains strings for `.gnu.liblist` section in shared libraries where `.gnu.liblist`
 2050 section is not allocated.

2051 **.gnu.prelink_undo** This section contains prelink private data used for prelink `--undo` operation. This data in-
 2052 cludes the original `ElfNN_Ehdr` of the object before prelinking and all its original `ElfNN_Phdr` and `ElfNN_Shdr`
 2053 headers.

2054 Prelink also defines 6 new dynamic tags:

```
2055 #define DT_GNU_PRELINKED    0x6ffffdf5 /* Prelinking timestamp */
2056 #define DT_GNU_CONFLICTSZ  0x6ffffdf6 /* Size of conflict section */
2057 #define DT_GNU_LIBLISTSZ   0x6ffffdf7 /* Size of library list */
2058 #define DT_CHECKSUM        0x6ffffdf8 /* Library checksum */
2059
2060 #define DT_GNU_CONFLICT    0x6ffffef8 /* Start of conflict section */
2061 #define DT_GNU_LIBLIST     0x6ffffef9 /* Library list */
```

Listing 27: Prelink dynamic tags

2062 `DT_GNU_PRELINKED` and `DT_CHECKSUM` dynamic tags must be present in prelinked shared libraries. The corresponding
 2063 `d_un.d_val` fields should contain time when the library has been prelinked (in seconds since January, 1st, 1970, 00:00
 2064 UTC) resp. CRC32 checksum of all sections with one of `SHF_ALLOC`, `SHF_WRITE` or `SHF_EXECINSTR` bit set whose
 2065 type is not `SHT_NOBITS`, in the order they appear in the shared library's section header table, with `DT_GNU_PRELINKED`
 2066 and `DT_CHECKSUM` `d_un.v_val` values set to 0 for the time of checksum computation.

2067 The `DT_GNU_LIBLIST` and `DT_GNU_LIBLISTSZ` dynamic tags must be present in all prelinked executables. The
 2068 `d_un.d_ptr` value of the `DT_GNU_LIBLIST` dynamic tag contains the virtual address of the `.gnu.liblist` section
 2069 in the executable and `d_un.d_val` of `DT_GNU_LIBLISTSZ` tag contains its size in bytes.

2070 `DT_GNU_CONFLICT` and `DT_GNU_CONFLICTSZ` dynamic tags may be present in prelinked executables. `d_un.d_ptr` of
 2071 `DT_GNU_CONFLICT` dynamic tag contains the virtual address of `.gnu.conflict` section in the executable (if present)
 2072 and `d_un.d_val` of `DT_GNU_CONFLICTSZ` tag contains its size in bytes.

A Glossary

2074 **ASCII Shield area** First 16MB of address space on 32-bit architectures. These addresses have zeros in upper 8 bits,
2075 which on little endian architectures are stored as last byte of the address and on big endian architectures as first
2076 byte of the address. A zero byte terminates string, so it is hard to control the exact arguments of a function if
2077 they are placed on the stack above the address. On big endian machines, it is even hard to control the low 24
2078 bits of the address,

2079 **Global Offset Table (GOT)** When position independent code needs to build address which requires dynamic relocation,
2080 instead of building it as constant in registers and applying a dynamic relocation against the read-only segment
2081 (which would mean that any pages of the read-only segment where relocations are applied cannot be shared
2082 between processes anymore), it loads the address from an offset table private to each shared library, which
2083 is created by the linker. The table is in writable segment and relocations are applied against it. Position
2084 independent code uses on most architectures a special PIC register which points to the start of the Global
2085 Offset Table,

2086 **Lazy Binding** A way to postpone symbol lookups for calls until a function is called for the first time in particular
2087 shared library. This decreases number of symbol lookups done during startup and symbols which are never
2088 called don't need to be looked up at all. Calls requiring relocations jump into PLT, which is initially set up
2089 so that a function in the dynamic linker is called to do symbol lookup. The looked up address is then stored
2090 either into the PLT slot directly (if PLT is writable) or into GOT entry corresponding to the PLT slot and any
2091 subsequent calls already go directly to that address. Lazy binding can be turned off by setting LD.BIND.NOW=1
2092 in the environment. Prelinked programs never use lazy binding for the executable or any shared libraries not
2093 loaded using dlopen,

2094 **Page** Memory block of fixed size which virtual memory subsystem deals with as a unit. The size of the page depends
2095 on the addressing hardware of the processor, typically pages are 4K or 8K, in some cases bigger,

2096 **PLT** Process Linkage Table. Stubs in ELF shared libraries and executables which allow lazy relocations of function
2097 calls. They initially point to code which will do the symbol lookup. The result of this symbol lookup is
2098 then stored in the Process Linkage Table and control transferred to the address symbol lookup returned. All
2099 following calls to the PLT slot just branch to the already looked up address directly, no further symbol lookup
2100 is needed,

2101 **Position Independent Executable** A hybrid between classical ELF executables and ELF shared libraries. It has a form
2102 of a ET_DYN object like shared libraries and should contain position independent code, so that the kernel
2103 can load the executable starting at random address to make certain security attacks harder. Unlike shared
2104 libraries it contains DT_DEBUG dynamic tag, must have PT_INTERP segment with dynamic linker's path, must
2105 have meaningful code at its e_entry and can use symbol lookup assumptions normal executables can make,
2106 particularly that no symbol defined in the executable can be overridden by a shared library symbol,

2107 **REL** Type of relocation structure which includes just offset, relocation type and symbol. Addend is taken from
2108 memory location at offset,

2109 **RELA** Type of relocation structure which includes offset, relocation type, symbol against which the relocation is and
2110 an integer addend which is added to the symbol. Memory at offset is not supposed to be used by the relocation.
2111 Some architectures got this implemented incorrectly and memory at offset is for some relocation types used by
2112 the relocation, either in addition to addend or addend is not used at all. RELA relocations are generally better
2113 for prelink, since when prelink stores a pre-computed value into the memory location at offset, the addend
2114 value is not lost,

2115 **relative relocation** Relocation, which doesn't need a symbol lookup, just adds a shared library load offset to certain
2116 memory location (or locations),

2117 **RTTI** C++ runtime type identification,

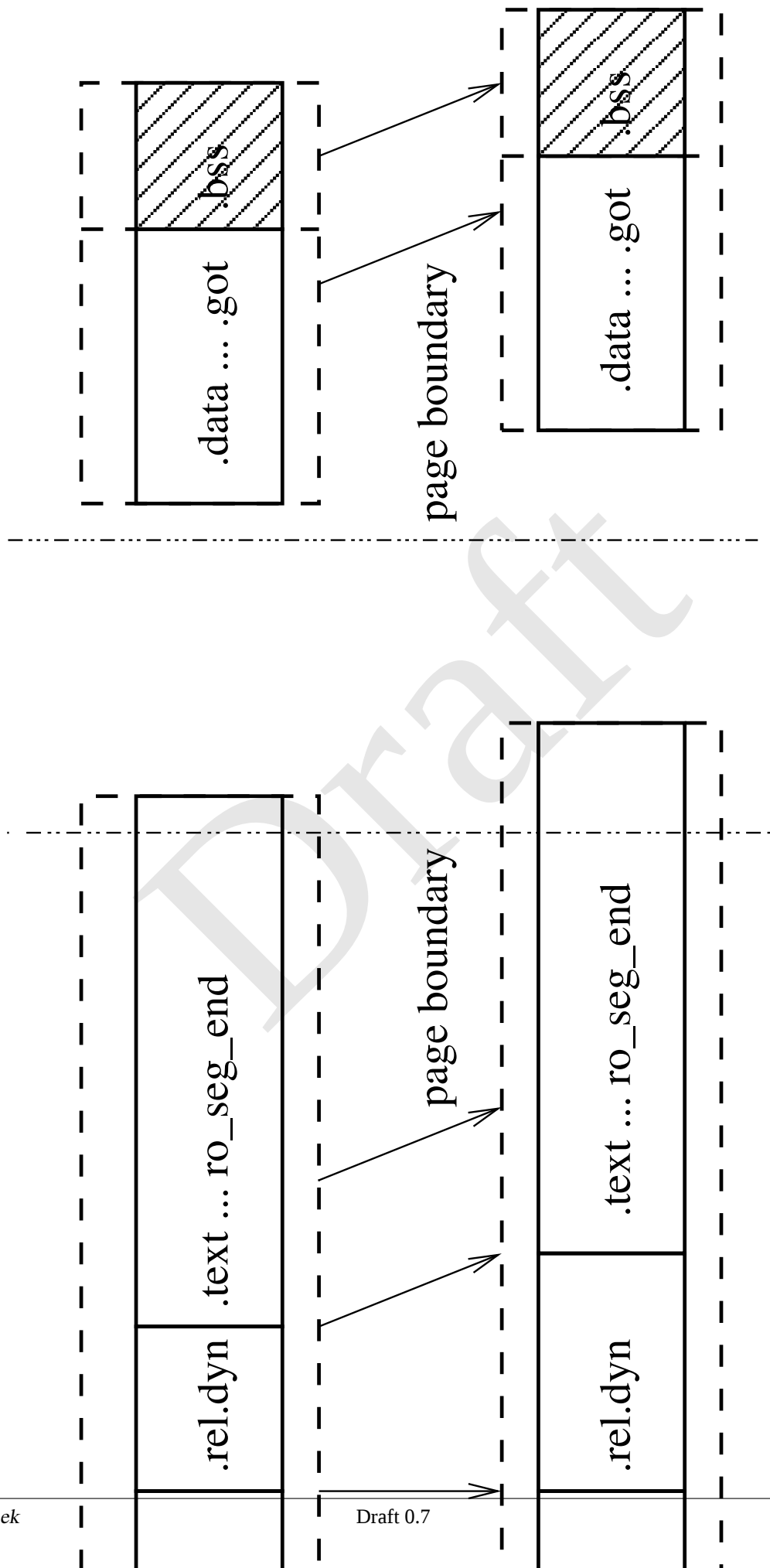
2118 **Symbol Search Scope** The sequence of ELF objects in which a symbol is being looked up. When a symbol definition
2119 is found, the searching stops and the found symbol is returned. Each program has a global search scope,
2120 which starts by the executable, is typically followed by the immediate dependencies of the executable and
2121 then their dependencies in breadth search order (where only first occurrence of each shared library is kept).
2122 If DT_FILTER or DT_AUXILIARY dynamic tags are used the order is slightly different. Each shared library
2123 loaded with dlopen has its own symbol search scope which contains that shared library and its dependencies.
2124 Prelink operates also with natural symbol search scope of each shared library, which is the global symbol
2125 search scope the shared library would have if it were started as the main program,

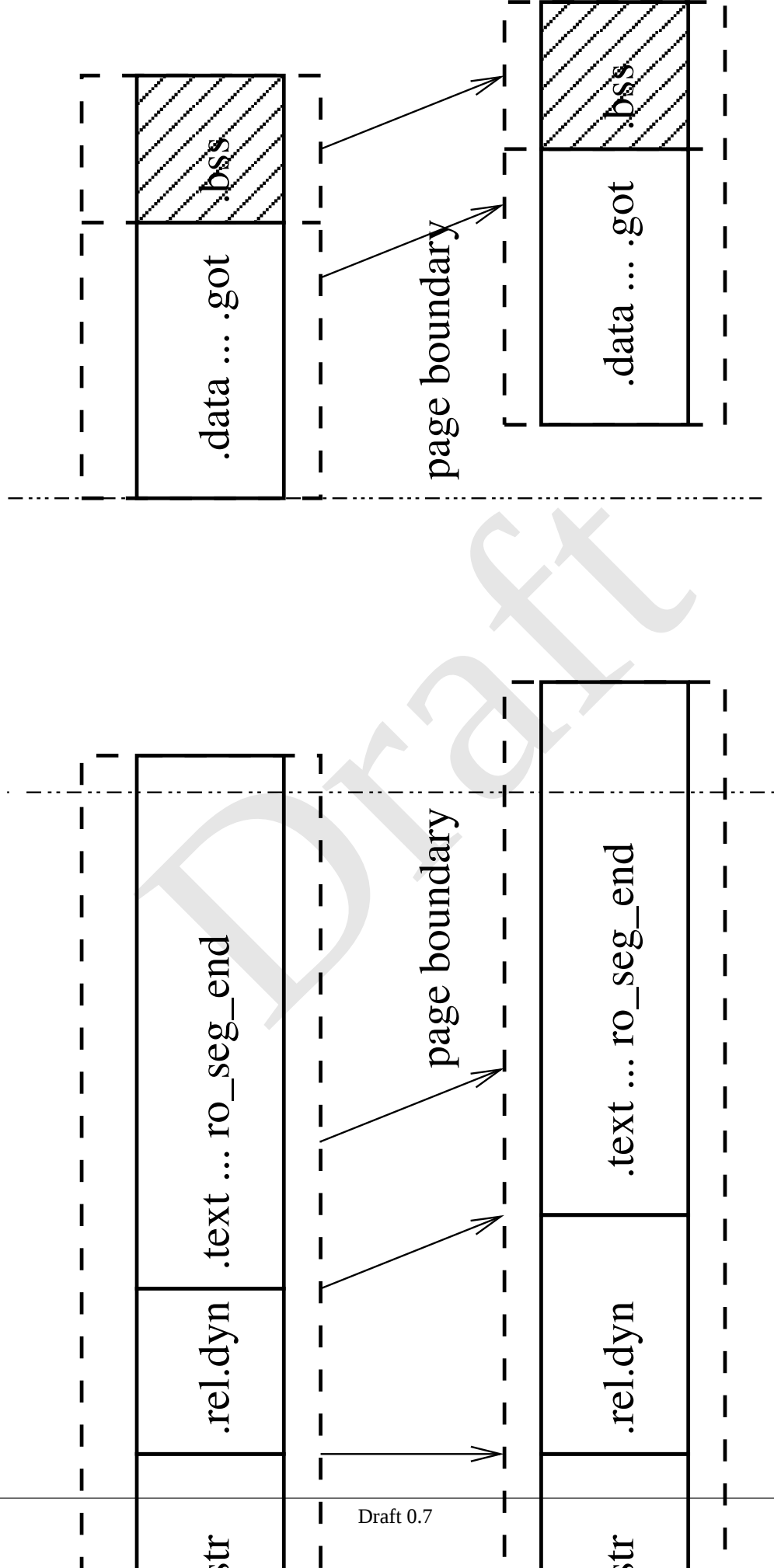
B References

- 2126 [1] *System V Application Binary Interface, Edition 4.1.*
- 2127 [2] *System V Application Binary Interface, Intel 386 Architecture Processor Supplement.*
- 2128 [3] *System V Application Binary Interface, AMD64 Architecture Processor Supplement.*
- 2129 [4] *System V Application Binary Interface, Intel Itanium Architecture Processor Supplement*, Intel Corporation, 2001.
- 2130 [5] Steve Zucker, Kari Karhi, *System V Application Binary Interface, PowerPC Architecture Processor Supplement*,
2131 SunSoft, IBM, 1995.
- 2132 [6] *System V Application Binary Interface, PowerPC64 Architecture Processor Supplement.*
- 2133 [7] *System V Application Binary Interface, ARM Architecture Processor Supplement.*
- 2134 [8] *SPARC Compliance Definition, Version 2.4.1*, SPARC International, Inc., 1999.
- 2135 [9] Ulrich Drepper, *How To Write Shared Libraries*, Red Hat, Inc., 2003.
- 2136 [10] *Linker And Library Guide*, Sun Microsystems, 2002.
- 2137 [11] John R. Levine, *Linkers and Loaders*, 1999.
- 2138 [12] Ulrich Drepper, *ELF Handling For Thread-Local Storage*, Red Hat, Inc., 2003.
- 2139 [13] Alan Modra, *PowerPC Specific Thread Local Storage ABI*, 2003.
- 2140 [14] Alan Modra, *PowerPC64 Specific Thread Local Storage ABI*, 2003.
- 2141 [15] *DWARF Debugging Information Format Version 2.*
- 2142 [16] *DWARF Debugging Information Format Version 3*, Draft, 2001.
- 2143 [17] *The "stabs" debugging information format.*

C Revision History

- 2144 **2003-11-03** First draft.





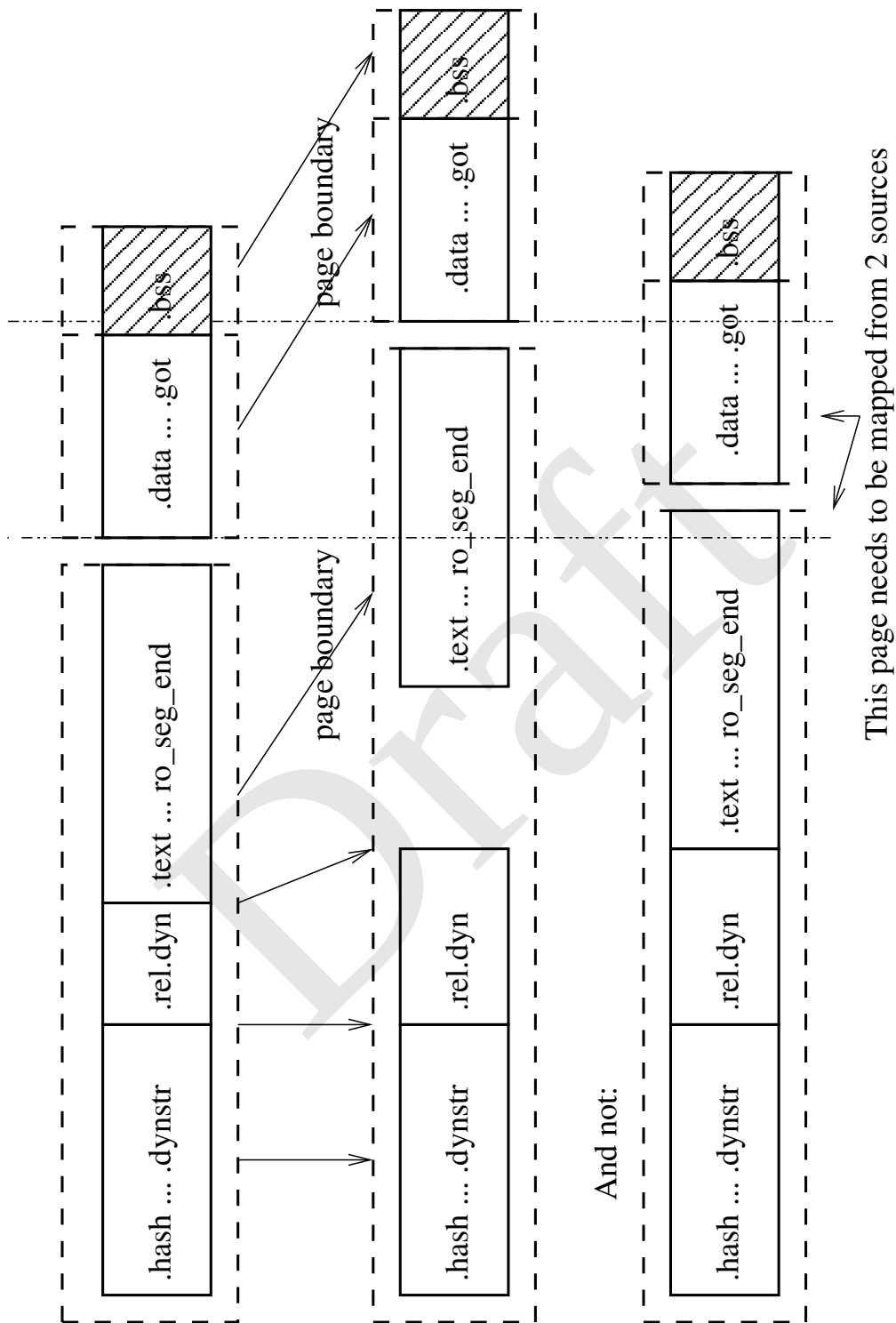


Figure 3: Growing read-only segment if page padding needed

