SLINKY: Static Linking Reloaded

Christian Collberg, John H. Hartman, Sridivya Babu, Sharath K. Udupa
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{collberg,jhh,divya,sku}@cs.arizona.edu

Abstract

Static linking has many advantages over dynamic linking. It is simple to understand, implement, and use. It ensures that an executable is self-contained and does not depend on a particular set of libraries during execution. As a consequence, the user executes exactly the same executable image as was tested by the developer, diminishing the risk that the user's environment will affect correct behavior.

The major disadvantages of static linking are increases in the memory required to run an executable, network bandwidth to transfer it, and disk space to store it.

In this paper we describe the SLINKY system that uses digest-based sharing to combine the simplicity of static linking with the space savings of dynamic linking: although SLINKY executables are completely self-contained, minimal performance and disk-space penalties are incurred if two executables use the same library. We have developed a SLINKY prototype that consists of tools for adding digests to executables, a slight modification of the Linux kernel to use those digests to share code pages, and tools for transferring files between machines based on digests of their contents. Results show that our prototype has no measurable performance decrease relative to dynamic linking, a comparable memory footprint, a 20% storage space increase, and a 34% increase in the network bandwidth required to transfer the packages. We believe that SLINKY obviates many of the justifications for dynamic linking, making static linking a superior technology for software organization and distribution.

1 Introduction

Most naïve users' frustrations with computers can be summarized by the following two statements: "I installed this new program and it just didn't work!" or "I downloaded a cool new game and suddenly this other program I've been using for months stopped working!" In many cases these problems can be traced back to missing, out-of-date, or incompatible dynamic libraries on the user's computer. While this problem may occur in any system that supports dynamically linked libraries, in the Windows community it is affectionately known as *DLL Hell* [12].

In this paper we will argue — against conventional wisdom — that in most cases dynamic linking should be abandoned in favor of static linking. Since static linking ensures that programs are self-contained, users and developers can be assured that a program that was compiled, linked, and tested on the developer's machine will run unadulterated on the user's machine. From a quality assurance and release management point of view this has tremendous advantages: since a single, self-contained, binary image is being shipped, little attention must be made to potential interference with existing software on the user's machine. From a user's point of view there is no chance of having to download additional libraries to make the new program work.

A major argument against static linking is the size of the resulting executables. We believe that this will likely be offset by ever-increasing improvements in disk-space, CPU speed, and main memory size. Nonetheless, this paper will show that the cost of static linking, in terms of file-system storage, network bandwidth, and run-time memory usage, can be largely eliminated through minor modifications to the operating system kernel and some system software.

Our basic technique is *digest-based sharing*, in which *message digests* identify identical chunks of data that can be shared between executables. Digests ensure that a particular chunk of data is only stored once in memory or on disk, and only transported once over the network. This sharing happens without intervention by the software developer or system administrator; SLINKY automatically finds and shares identical chunks of different executables.

1.1 Slinky

Slinky is a relatively simple system that provides the space-saving benefits of dynamic linking without the complexities. There are four components to Slinky:

1. The `slink` program creates a statically-linked executable from one that is dynamically linked. It does so by linking in the dynamic libraries on which the program depends, resolving references, and performing necessary page alignment.

2. The `digest` program adds SHA-1 digests of the code pages to the statically-linked executable produced by `slink`.

3. A modified Linux kernel shares code pages between processes based on the digests added by `digest`. During a page fault the digest of the faulting page is used to share an in-memory copy of the page, if one exists.

4. The `ckget` program downloads software packages based on the digests of variable-sized chunks. Chunk boundaries are computed using Rabin fingerprints. Each unique chunk is only downloaded once, greatly reducing the network bandwidth required to transfer statically-linked executables.

What is particularly attractive about Slinky is its simplicity. The `slink` program consists of 1000 lines of code, `digest` 200 lines, the kernel modifications 100 lines, and `ckget` 100 lines.

1.2 Background

"Linking" refers to combining a program and its libraries into a single executable and resolving the symbolic names of variables and functions into their resulting addresses. Generally speaking, *static linking* refers to doing this process at link-time during program development, and incorporating into each executable the libraries it needs. *Dynamic linking* refers to deferring linking until the executable is actually run. An executable simply contains references to the libraries it uses, rather than copies of the libraries, dramatically reducing its size. As a result, only a single copy of each library must be stored on disk. Dynamic linking also makes it possible for changes to a library to propagate automatically to the executables that use it, since they will be linked against the new version of the library the next time they are run.

Dynamic linking is typically used in conjunction with *shared libraries*, which are libraries whose in-memory images can be shared between multiple processes. The combination of dynamic linking and shared libraries ensures that there is only one copy of a shared library on disk and in memory, regardless of how many executables make use of it.

Dynamic linking is currently the dominant practice, but this was not always the case. Early operating systems used static linking. Systems such as Multics [3] soon introduced dynamic linking as a way of saving storage and memory space, and increasing flexibility. The resulting complexity was problematic, and subsequent systems such as Unix [16] went back to static linking because of its simplicity. The pendulum has since swung the other way and dynamic linking has become the standard practice in Unix and Linux.

1.3 DLL Hell

Although dynamic linking is the standard practice, it introduces a host of complexities that static linking does not have. In short, running a dynamically linked executable depends on having the proper versions of the proper libraries installed in the proper locations on the computer. Tools have been developed to handle this complexity, but users are all too familiar with the "DLL Hell" [12] that can occur in ensuring that all library dependencies are met, especially when different executables depend on different versions of the same library.

DLL Hell commonly occurred in early versions of the Windows operating when program installation caused an older version of a library to replace an already installed newer one. Programs that relied on the newer version then stopped working — often without apparent reason. Unix and newer versions of Windows fix this problem using complex versioning schemes in which the library file name contains the version number. The numbering scheme typically allows for distinction between compatible and incompatible library versions. However, DLL Hell can still occur if, for example, the user makes a minor change to a library search `PATH` variable, or compatible library versions turn out not to be. Sometimes programs depend on undocumented features (or even bugs) in libraries, making it difficult to determine when two versions are compatible. No matter what the cause, DLL Hell can cause a previously working program to fail.

1.4 Anecdotes

It is easy to argue that the "right solution" to the DLL Hell problem is that *"every package maintainer should just make sure that their package always has the correct set of dependencies!"*, perhaps by using dependency management tools. However, it appears that dependency problems still occur even in the most well-designed package management systems, such as Debian's `apt` [2]. Below we present some anecdotal evidence of this.

Consider, first, the following instructions from the Debian quick reference [1] guide:

```
Package dependency problems may occur
when upgrading in unstable/testing.
Most of the time, this is because a
package that will be upgraded has a
new dependency that isn't met.  These
problems are fixed by using

      # apt-get dist-upgrade

If this does not work, then repeat
one of the following until the problem
resolves itself [sic]:

      # apt-get upgrade -f

...  Some really broken upgrade
scripts may cause persistent trouble.
```

Most Debian users will install packages from the `unstable` distribution, since the `stable` distribution is out-of-date.

Similarly, searching Google with the query `"apt-get dependency problem"`, produces many hits of the following kind:

```
igor> I am trying to upgrade my potato
igor> distro to get the latest KDE. after I
igor> type apt-get update && apt-get
igor> upgrade, here is my error message:
igor> --------------------------------
igor> ...
igor> Sorry, but the following packages
igor> have unmet dependencies:
igor>   debianutils: PreDepends: libc6
igor>   (>= 2.1.97) but 2.1.3-18 is to
igor>   be installed
igor> E: Internal Error, InstallPackages
igor> was called with broken packages!
igor> ---------------------------------
briand> I'm getting everything from the
briand> site you are using and everything
briand> is OK. This is very odd. ...
```

`Ardour` is a 200,000 LOC multi-track recording tool for Linux, which relies on many external libraries for everything from GUI to audio format conversion. Its author, Paul Davis, writes [4]:

```
I spent a year fielding endless
reports from people about crashes,
strange backtraces and the like before
giving up and switching to static
linking.  Dynamic linking just doesn't
work reliably enough.
I will firmly and publically denounce
any packaging of Ardour that use
dynamic linking to any C++ library
except (and possibly including)
libstdc++.
```

1.5 Contributions

In general, package maintenance and installation are difficult problems. A software distribution that appears to work fine on the package maintainer's machine may break for any number of reasons on the user's machine. The inherent complexity of dynamic library versioning coupled with the fact that an installation package has to work under any number of operating system versions and on computers with any number of other packages installed, almost invariably invites a visit from Murphy's famous law.

We have developed a system called SLINKY that combines the advantages of static and dynamic linking without the disadvantages of either. In our system, executables are statically linked (and hence avoid the complexities and frailties of dynamically linked programs) but can be executed, transported, and stored as efficiently as their dynamic counterparts.

The key insight is that dynamic linking saves space by explicitly sharing libraries stored in separate files, and this introduces much of the complexity. SLINKY, instead, relies on implicit sharing of identical chunks of data, which we call *digest-based sharing*. In this scheme chunks of data are identified by a *message digest*, which is a cryptographically secure hash such as SHA-1 [19]. Digest-based sharing allows SLINKY to store a single copy of each data chunk in memory and on disk, regardless of how many executables share that data. The digests are also used to transfer only a single copy of data across the network. This technique allows SLINKY to approach the space savings of dynamic linking without the complexity.

The rest of the paper is organized as follows. In Section 2 we compare static and dynamic linking. In Section 3 we describe the implementation of the SLINKY system. In Section 4 we show that empirically our system makes static linking as efficient as dynamic linking. In Section 5 we discuss related work. In Section 6, finally, we summarize our results.

2 Linking

High-level languages use symbolic names for variables and functions such as `foo` and `read()`. These *symbols* must be converted into the low-level addresses understood by a computer through a process called *linking* or *link editing* [9]. Generally, linking involves assigning data and instructions locations in the executable's address space, determining the resulting addresses for all symbols, and *resolving* each symbol reference by replacing it with the symbol's address. This process is somewhat complicated by *libraries*, which are files containing commonly-used variables and functions. There are many linking variations, but they fall into two major categories, *static linking* and *dynamic linking.*

2.1 Static Linking

Static linking is done at *link-time*, which is during program development. The developer specifies the libraries on which a executable depends, and where to find them. A tool called the *linker* uses this information to find the proper libraries and resolve the symbols. The linker produces a static executable with no unresolved symbols, making it self-contained with respect to libraries.

A statically-linked executable may be self-contained, but it contains portions of each library to which it is linked. For large, popular libraries, such as the C library, the amount of duplicate content can be significant. This means that the executables require more disk storage, memory space, and network bandwidth than if the duplicate content were eliminated.

Statically linking executables also makes it difficult to update libraries. A statically-linked executable can only take advantage of an updated library if it is relinked. That means that the developer must get a new copy of the library, relink the executable and verify that it works correctly with the new library, then redistribute it to the users. These drawbacks with statically-linked executables led to the development of dynamic linking.

2.2 Dynamic Linking

Dynamic linking solves these problems by deferring symbol resolution until the executable is run (*run-time*), and by using a special library format (called a *dynamic library* or *shared library*) that allows processes to share a single copy of a library in memory. At link-time all the linker does is store the names of the necessary libraries in the executable. When the executable runs, a program called a *dynamic linker/loader* reads the library names from the executable and finds the proper files using a list of directories specified by a library search path. Since the libraries aren't linked until run-time, the executable may run with different library versions than were used during development. This is useful for updating a library, because it means the executables that are linked to that library will use the new version the next time they are run.

While the "instant update" feature of dynamic linking appears useful — it allows us to fix a bug in a library, ship that library, and instantly all executables will make use of the new version — it can also have dangerous consequences. There is a high risk of programs failing in unpredictable ways when a new version of a library is installed. Even though two versions of a library may have compatible interfaces and specifications, programs may depend on undocumented side-effects, underspecified portions of the interface, or other features of the library, including bugs. In addition, if a library bug was known to the developer he may have devised a "work-around", possibly in a way no longer compatible with the bug fix. In general, it is impossible for a library developer to determine when two versions are compatible since the dependencies on the library's behavior are not known. We believe that *any* change to a library (particularly one that has security implications) requires complete regression tests to be re-run for the programs that use the library.

Dynamic linking also complicates software removal. Care must be taken to ensure that a dynamic library is no longer in use before deleting it, otherwise executables will mysteriously fail. On the other hand, deleting executables can leave unused dynamic libraries scattered around the system. Either way, deleting a dynamically-linked executable isn't as simple as deleting the executable file itself.

2.3 Code-Sharing Techniques

Systems that use dynamic linking typically share a single in-memory copy of a dynamic library among all processes that are linked to it. It may seem trivial to share a single copy, but keep in mind that a library will itself contain symbol references. Since each executable is linked and run independently, a symbol may have a different address in different processes, which means that shared library code cannot contain absolute addresses.

The solution is *position-independent code*, which is code that only contains relative addresses. Absolute addresses are stored in a per-process *indi-*

rection table. Position-independent code expresses all addresses as relative offsets from a register. A dedicated register holds the base address of the indirection table, and the code accesses the symbol addresses stored in the table using a relative offset from the base register. The offsets are the same across all processes, but the registers and indirection tables are not. Since the code does not contain any absolute addresses it can be shared between processes. This is somewhat complex and inefficient, but it allows multiple processes to share a single copy of the library code.

2.4 Package Management

The additional flexibility dynamic linking provides also introduces a tremendous amount of complexity. First, in order to run an executable the libraries on which it depends must be installed in the proper locations in the dynamic linker/loader's library search path. This means that to run an executable a user needs both the executable and the libraries on which it depends, and must ensure that the dynamic linker is configured such that the libraries are on the search path. Additionally, the versions of those libraries must be compatible with the version that was used to develop the executable. If they are not, then the executable will either fail to run or produce erroneous results.

To manage this complexity, package systems such as RedHat's `rpm` [17] and Debian's `dpkg` [5] were developed. A package contains everything necessary to install an executable or library, including a list of the packages on which it depends. For an executable these other packages include the dynamic libraries it needs. A sophisticated versioning scheme allows a library package to be updated with a compatible version. For example, the major number of a library differentiates incompatible versions, while the minor number differentiates compatible versions. In this way a package can express its dependency on a compatible version of another package. The versioning system must also extend to the library names, so that multiple versions of the same library can coexist on the same system.

The basic package mechanism expresses inter-package dependencies, but it does nothing to resolve those dependencies. Suppose a developer sends a user a package that depends on another package that the user does not have. The user is now forced to ask for the additional package, or search the Internet looking for the needed package. More recently, the user could employ sophisticated tools such as RedHat's `up2date` [21] or Debian's `apt` [2] to fetch and install the desired packages from on-line repositories.

2.5 Security Issues

Dynamic linking creates potential security holes because of the dependencies between executables and the libraries they need. First, an exploit in a dynamic library affects every executable that uses that library. This makes dynamic libraries particularly good targets for attack. Second, an exploit in the dynamic linker/loader affects every dynamically-linked executable. Third, maliciously changing the library search path can cause the dynamic linker-loader to load a subverted library. Fourth, when using a package tool such as `up2date` or `apt`, care must be taken to ensure the authenticity and integrity of the downloaded packages. These potential security holes must be weighed against the oft-stated benefit that dynamic linking allows for swift propagation of security fixes.

3 Slinky

Slinky is a system that uses message digests to share data between executables, rather than explicitly sharing libraries. Chunks of data are identified by their digest, and Slinky stores a single copy of each chunk in memory and disk. Slinky uses SHA-1 [19] to compute digests. SHA-1 is a hash algorithm that produces a 160-bit value from a chunk of data. SHA-1 is cryptographically secure, meaning (among other things) that although it is relatively easy to compute the hash of a chunk of data, it is computationally infeasible to compute a chunk of data that has a given hash. There are no known instances of two chunks of data having the same SHA-1 hash, and no known method of creating two chunks that have the same hash. This means that for all practical purposes chunks of data with the same SHA-1 hash are identical. Although there is some controversy surrounding the use of digests to identify data [7], it is generally accepted that a properly-designed digest function such as SHA-1 has a negligible chance of hashing two different chunks to the same value. It is much more likely that a hardware or software failure will corrupt a chunk's content.

Depending on one's level of paranoia with respect to the possibility of hash collisions, different hashing algorithms could be used. The increased costs come in both time and space and depend on the size of the hash (128 bits for MD5; 160 bits for SHA-1; 256, 384, or 512 bits for newer members of the SHA family) and the number of rounds and complexity

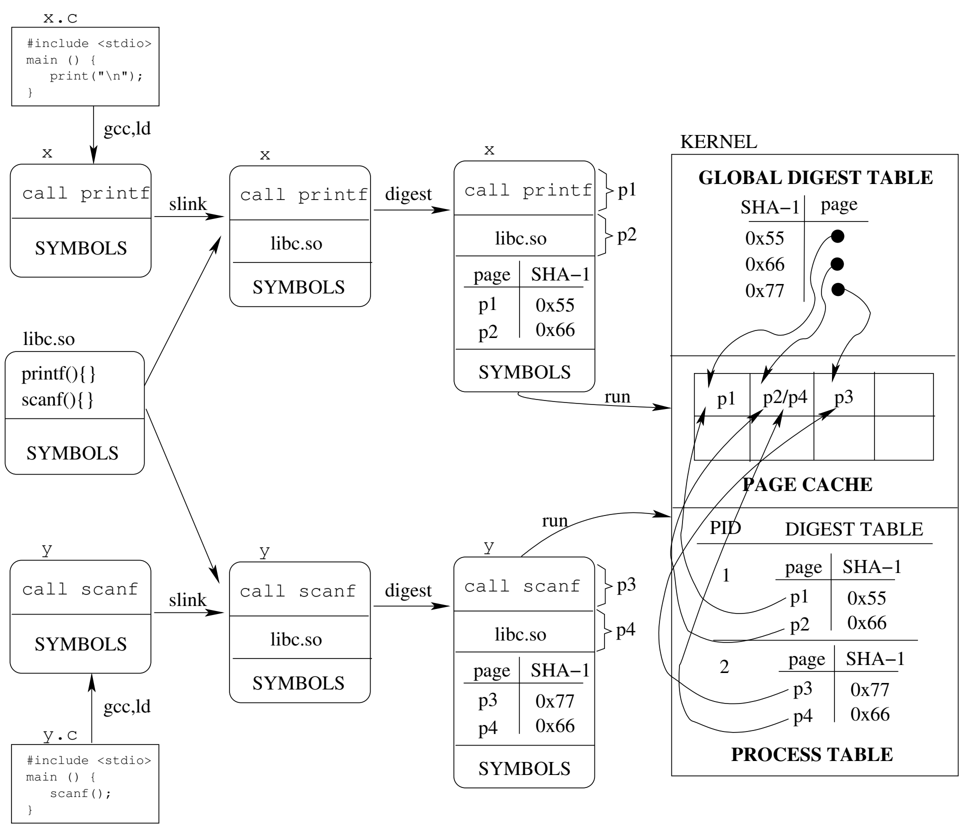

Figure 1: Sharing memory pages based on digest. Only one copy of each page exists in memory (page cache). The per-process digest table contains the page digests of the process, and is initialized from a table in the executable. The global digest table contains the digest of every page in memory.

of computation. Schneier [18, p. 456] shows, for example, that MD5 is roughly 2.3 times faster than SHA-1, while SHA-1 is less prone to collisions due to its larger hash-size.

Digests allow Slinky to store a single copy of each chunk in memory and on disk. In addition, when transferring an executable over the network only those chunks that do not already exist on the receiving end need be sent. The use of digests allows Slinky to share data between executables efficiently, without introducing the complexities of dynamic linking. In addition, Slinky avoids the security holes endemic to dynamic linking.

The following sections describe how Slinky uses digests to share data in memory and on disk, as well as to reduce the amount of data required to transfer an executable over the network. We developed a Slinky prototype that uses digests to share memory pages and eliminate redundant network transfers. Sharing disk space based on digest is currently work-in-progress; we describe how Slinky will provide that functionality, although it has not yet been implemented.

3.1 Sharing Memory Pages

Slinky shares pages between processes by computing the digest of each code page, and sharing pages that have the same digest. If a process modifies a page, then the page's digest changes, and the page can no longer be shared with other processes using the old version of the page. One way to support this is to share the pages copy-on-write. When a process modifies a page it gets its own copy. Slinky employs the simpler approach of only sharing read-only code pages. Slinky assumes that data pages are likely to be written, and therefore unlikely to be shared anyway.

The current Slinky prototype is Linux-based, and consists of three components that allow processes to share pages based on digests. The first is a linker called `slink` that converts a dynamically-linked executable into a statically-linked executable. The second is a tool called `digest` that computes the digest of each code page in an executable. The digests are stored in special sections in the executable. The third component is a set of Linux kernel modifications that allow processes to share pages with identical digests.

Figure 1 illustrates how SLINKY functions. A source file `x.c` that makes use of the C library is compiled into a dynamically-linked executable file `x`. Our program `slink` links `x` with the dynamic library `libc.so`, copying its contents into `x`. The `digest` program adds a new ELF section that maps pages to their digests. The process is repeated for a second example program `y.c`. When `x` is run its pages will be loaded, along with their digests. When `y` is run, page `p4` will not be loaded since a page with the same digest already exists (page `p2` from `x`'s copy of `libc.so`).

3.1.1 Slink

Shared libraries require position-independent code to allow sharing of code pages between processes. SLINKY also requires position-independent code because static linking may cause the same library to appear at different addresses in different executables. Therefore, SLINKY must statically link executables with shared libraries, since those libraries contain position-independent code and traditional static libraries do not. `Slink` is a program that does just that — it converts a dynamically-linked executable into a statically-linked executable by linking in the shared libraries on which the dynamically-linked executable depends. The resulting executable is statically linked, but contains position-independent code from the shared libraries. `Slink` consists of about 830 lines of C and 160 lines of shell script.

The input to `slink` is a dynamically-linked executable in ELF format [10]. `Slink` uses the `prelink` [13] tool to find the libraries on which the executable depends, organize them in the process's address space, and resolve symbols. `Slink` then combines the prelinked executable and its libraries into a statically-linked ELF file, aligning addresses properly, performing some relocations that `prelink` cannot do, and removing data structures related to dynamic linking that are no longer needed.

Although position-independent code is necessary for sharing library pages between executables, it is not sufficient. If the same library is linked into different executables at different page alignments, the page contents will differ and therefore cannot be shared. Fortunately, the page alignment is specified in the library ELF file, ensuring that all copies of the library are linked with the same alignment.

3.1.2 Digest

`Digest` is a tool that takes the output from `slink` and inserts the digests for each code page. For every executable read-only ELF segment, `digest` computes the SHA-1 hash of each page in that segment and stores them in a new ELF section. This section is indexed by page offset within the associated segment, and is used by the kernel to share pages between processes. A Linux page is 4KB, and the digest is 20 bytes, so the digests introduce an overhead of 20/4096 or less than 0.5% per code page. `Digest` consists of about 200 lines of C code.

3.1.3 Kernel Modifications

SLINKY requires kernel modifications so that the loader and page fault handler make use of the digests inserted by `digest` to share pages between processes. These modifications consist of about 100 lines of C. When a program is loaded, the loader reads the digests from the file and inserts them in a per-process digest table (PDT) that maps page number to digest. This table is used during a page fault to determine the digest of the faulting page.

We also modified the Linux 2.4.21 kernel to maintain a global digest table (GDT) that contains the digest of every code page currently in memory. The GDT is used during a page fault to determine if there is already a copy of the faulting page in memory. If not, the page is read into memory from disk and an entry added to the GDT. Otherwise the reference count for the page is simply incremented. The page table for the process is then updated to refer to the page, and the process resumes execution. When a process exits the reference count for each of its in-memory pages is decremented; a page is removed from the GDT when its reference count drops to zero.

3.1.4 Security

SLINKY shares pages based on the digests stored in executables, so the system correctness and security depend on those digests being correct. A malicious user could modify a digest or a page so that the digest no longer corresponds to the page's contents, potentially causing executables to use the wrong pages. SLINKY avoids this problem by verifying the digest of each text page when it is brought into memory and before it is added to the GDT.

This slows down the the handling of page faults that result in a page being read from disk, but we show in Section 4.1.2 that this overhead is negligible when compared to the cost of the disk access. Once a page is added to the GDT, SLINKY relies on the memory protection hardware to ensure that processes cannot modify the read-only text page.

Figure 2 shows pseudo-code for the modified page fault handler that is responsible for searching the GDT for desired pages, and adding missing pages to the GDT when they are read from disk.

3.2 Sharing Disk Space

Digests can also reduce the disk space required to store statically-linked executables. One option is to store data based on the per-page digests stored in the executable. Although this will reduce the space required, it is possible to do better. This is because the digests only refer to the executable's code pages, and the virtual memory system requires those pages be fixed-size and aligned within the file. Additional sharing is possible if arbitrary-size chunks of unaligned data are considered.

SLINKY shares disk space between executables by breaking them into variable-size chunks using Rabin's fingerprinting algorithm [15], and then computing the digests of the chunks. Only one copy of each unique chunk is stored on disk. The technique is based on that used in the Low-Bandwidth File system (LBFS) [11]. A small window is moved across the data and the Rabin fingerprint of the window is computed. If the low-order N bits of the fingerprint match a pre-defined value, a chunk boundary is declared. The sliding window technique ensures that the effects of insertions or deletions are localized. If, for example, one file differs from another only by missing some bytes at the beginning, the sliding window approach will synchronize the chunks for the identical parts of the file. Alternatively, if fixed-size blocks were used, the first block of each file would not have the same hash due to the missing bytes, and the mismatch would propagate through the entire file.

SLINKY uses the same 48-byte window as LBFS as this was found to produce good results. SLINKY also uses the same 13 low-order bits of the fingerprint to determine block boundaries, which results in an 8KB average block size. The minimum block size is 2KB and the maximum is 64KB. Using the same parameters as LBFS allows us to build on LBFS's results.

The SLINKY prototype contains tools for breaking files into chunks, computing chunk digests, and comparing digests between files. It does not yet use the digests to share disk space between executables. We intend to extend SLINKY so that executables share chunks based on digest, but that work is in progress. The current tools allow SLINKY's space requirements to be compared to dynamic libraries, as described in Section 4.

3.3 Reducing Network Bandwidth

The final piece of the puzzle is to reduce the amount of network bandwidth required to transport statically-linked executables. Digests can also be used for this purpose. The general idea is to only transfer those chunks that the destination does not already have. This is the basic idea behind LBFS, and SLINKY uses a similar mechanism. Suppose we want to transfer an executable from X to Y over the network. First, X breaks the executable into chunks and computes the digests of the chunks. X then sends the list of digests to Y. Y compares the provided list with the digests of chunks that it already has, and responds with a list of missing digests. X then sends the missing chunks to Y.

We developed a tool called `ckget` that transfers files across the network based on chunks. Continuing the above example, each file on X has an associated *chunk file* containing the chunk digests for the file. X and Y also maintain individual *chunk databases* that indicate the location of each chunk within their file systems. To transfer a file, Y runs `ckget URL`, which uses the URL to contact X. X responds with the corresponding chunk file. `ckget` cross-references the chunks listed in the file with the contents of its chunk database to determine which chunks it lacks. It then contacts X to get the missing chunks, and reconstitutes the file from its chunks. `ckget` then adds the new chunks to Y's chunk database.

Figure 3 illustrates this process. The client issues the command `ckget x-1.1.deb` to download version 1.1 of the `x` application. `ckget` retrieves the chunk file `x-1.1.ck` from the proper server. The chunk file indicates that two chunks (`A` and `B`) are the same as in version 1.0 and already exist locally in `x-1.0.deb`, but a new chunk `E` needs to be downloaded from the server. `ckget` gets this chunk and then reconstitutes `x-1.1.deb` from `x-1.0.deb` and the downloaded chunk `E`. `ckget` updates the client's `ChunkDB` to indicate that chunk `E` now exists in `x-1.1.deb`. Should the user subsequently download `y-2.3.deb` only chunk `D` will be retrieved.

```
// filemap_nopage is called when the page is not
// recorded in the page table entries of the process
func filemap_nopage (vm, address){
  if (page is in the page cache) return page
  if (page is from a slinky binary) {
    get digest of page from PDT
    use digest to search GDT
      if (matching digest is found) return page
  }
  //page not in memory
  read page from disk
  if (page is from a slinky binary) {
    verify page digest
    add page to GDT
  }
  return page
}
```

Figure 2: Page fault handler. The page's digest is used to search the GDT. If the page is not found it is read from disk, its digest verified, and an entry added to the GDT.

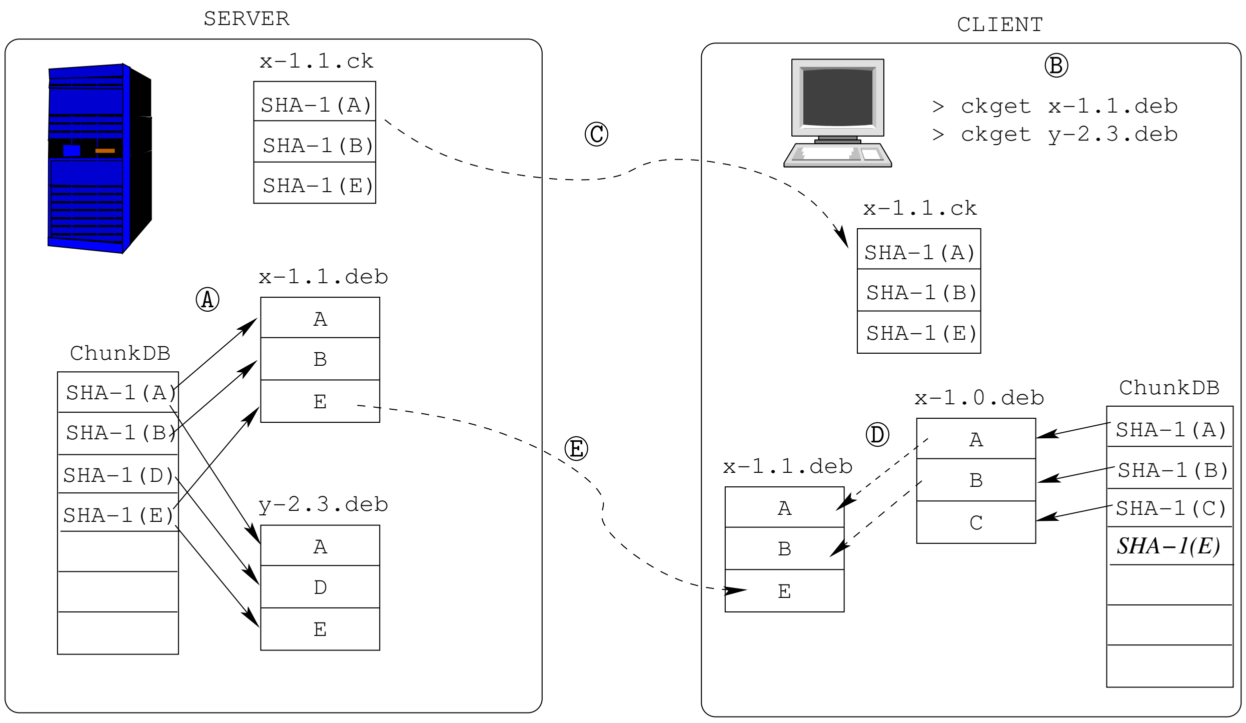

Figure 3: Overview of how SLINKY uses `ckget` to download a software package. Point Ⓐ shows the server with two packages `x-1.1.deb` and `y-2.3.deb`. The chunk database maps SHA-1 digests of chunks in these packages to the locations of the chunks. At point Ⓑ the client has download of `x-1.1` by issuing the command `ckget x-1.1.deb`. At Ⓒ x's chunk-file is downloaded. At Ⓓ chunks `A` and `B` are retrieved from a previous version of `x`. At Ⓔ chunk `E` has to be fetched from the server and its SHA-1 digest added to the client's ChunkDB. If the client subsequently downloads package `y-2.3.deb`, only chunk `D` will be transfered.

4 Evaluation

We performed several experiments on the Slinky prototype to evaluate the performance of statically-linked executables vs. dynamically-linked executables, as well as the space required to store them.

4.1 Performance

We performed several experiments to compare the performance of Slinky with that of a standard dynamically-linked Linux system. First, we measured the elapsed time to build the Linux kernel using `make`. This is representative of Slinky's performance on real-world workloads. Second, we timed page fault handling in both an unmodified Linux kernel and a kernel modified to support Slinky executables. Slinky's CPU overhead is confined to page fault handling, so these timings allow extrapolation of the worst-case overhead for Slinky when running executables whose performance is limited by page fault performance. Third, we measured the number of page faults when running a variety of executables in both systems. This not only provides information on Slinky's effect on the page fault rate, but also information on the memory footprint of Slinky executables.

All experiments were performed on a system with a 2.4 GHz Intel Pentium 4 CPU, 1GB of memory, and a Western Digital WD800BB-53CAA1 80GB disk drive. The kernel was Linux 2.4.21, and the Linux distribution was the "unstable" Debian distribution. The machine was a desktop workstation used for operating system development, so it had a representative set of software development and GUI applications installed. For the Slinky results all executables used in the tests were created using the `slink` and `digest` tools.

4.1.1 Elapsed Time

Our macro-benchmark consisted of measuring the elapsed time to build the Linux 2.4.19 kernel and several standard drivers. The benchmark was run four times on both the Slinky and the standard system. There is no statistically-significant difference in performance. The average elapsed time for the standard system was 139.32 ± 0.12 seconds vs. 139.00 ± 0.35 seconds for Slinky.

4.1.2 Page Fault Handling

We also measured the time required by Slinky to handle page faults. When a page is not found in the page cache Slinky looks for it in the GDT, and when a page is read in from disk Slinky verifies its digest and adds it to the GDT. Both of these cases increase the page fault overhead. In practice, however, the overhead is minimal. For page faults that require a disk read, Slinky adds 44.58 microseconds for a total of 3259.55 microseconds per fault, or 1.3%. Page faults that hit in the GDT required 0.95 microseconds. For comparison, page faults that hit in the page cache required 0.82 microseconds.

4.1.3 Memory Footprint

Table 1 shows the results of running several executables on a Slinky system and a standard system and measuring the number of page faults. For Slinky page faults are classified into those for which the page was found in the page cache or read from disk (*Other*), vs. those for which the page was found in the GDT (*GDT*). For the standard system page faults are classified into those for shared libraries (*Library*) and those for the executable itself (*Other*). Slinky adds overhead to both kinds of faults, as described in the previous section. Table 1(a) shows the results of running `make` with an empty GDT. The Slinky version of `make` generated 95 page faults, compared with 173 faults for the dynamic version. Table 1(b) shows running several commands in sequence starting with an empty GDT. Three conclusions can be drawn from these results. First, for our workloads Slinky executables suffer about 60-80 fewer page faults than their dynamically-linked counterparts. These additional page faults are caused by the dynamic linker, which is not run for the Slinky executables.

Second, `make` suffers 95 non-GDT page faults when it is run with an empty GDT, but only 36 non-GDT faults after several other executables have added pages to the GDT. The remaining 59 faults were handled by the GDT. This shows that pages can effectively be shared by digest, without resorting to shared libraries.

Third, the memory footprints for the Slinky executables are comparable to those for shared libraries. The dynamically-linked version of `make` caused 31 faults to pages not in shared libraries. In comparison, the Slinky version caused 36 faults when run after several other commands. Therefore, the Slinky version required 5 more non-shared pages, or 16% more. As more pages are added to the GDT we expect the number for Slinky to drop to 31. This effect can be seen for `gcc`, which caused 20 page faults in both the Slinky and dynamic versions, so that both versions had exactly the same memory footprint.

Execution	SLINKY			Dynamic		
Workload	GDT	Other	Sum	Library	Other	Sum
`1> make`	0	95	95	142	31	173

(a) First workload

Execution	SLINKY			Dynamic		
Workload	GDT	Other	Sum	Library	Other	Sum
`1> ls`	0	79	79	151	14	165
`2> cat`	29	3	31	89	3	92
`3> gcc`	32	20	52	93	20	113
`4> make`	59	36	95	142	31	173

(b) Second workload

Table 1: The number of page faults generated for two experimental workloads, running SLINKY vs. dynamically-linked executables. The first workload consists of running `make`; the second of running a sequence of commands culminating with `make`. For SLINKY the *GDT* column is faults for which the page was found in the GDT, *Other* is faults that either hit in the page cache or required a disk access, and *Sum* is the total faults. For Dynamic, the *Library* column is faults for pages in shared libraries, *Other* is faults to the executable itself, and *Sum* is the total.

4.2 Storage Space

Table 2(a) shows the space required to store dynamically-linked executables vs. statically-linked. These numbers were collected on the "unstable" Debian distribution. The *Dynamic* column shows the space required to store the dynamically-linked ELF executables in the given directories, plus the dynamic libraries on which they depend. Each dynamic library is only counted once. The *All* row is the union of the directories in the other rows, hence its value is not the sum of the other rows (since libraries are shared between rows). The SLINKY column shows the space required to store the statically-linked executables, and *Ratio* shows the ratio of the static and dynamic sizes. The static executables are much larger than their dynamic counterparts because they include images of all the libraries they use.

Table 2(b) shows the amount of space required to store the dynamic and static executables if they are broken into variable-size chunks and only one copy of each unique chunk is stored. The dynamic executables show a modest improvement over the numbers in the previous table due to commonality in the files. The static executables, however, show a tremendous reduction in the amount of space. Most of the extra space in Table 2(a) was due to duplicate libraries; the chunk-and-digest technique is able to share these chunks between executables. The SLINKY space requirements are reasonable – across all directories SLINKY consumes 20% more space than dynamic linking. SLINKY requires 306MB to store the executables instead of 252MB, or 54MB more. This is a very small fraction of a modern disk drive. Nonetheless, we believe that further reductions are possible. The current chunking algorithm does not take into account the internal structure of an ELF file. We believe that this structure can be exploited to improve chunk sharing between files, but we have not yet experimented with this.

4.3 Network Bandwidth

The third component of SLINKY is reducing the amount of network bandwidth required to transfer static executables. SLINKY accomplishes this by breaking the files into chunks, digesting the chunks, and transferring each unique chunk only once. Table 3 shows the results of our experiments with downloading Debian packages containing static vs. dynamic executables. We performed these experiments by writing a tool that takes a standard Debian package containing a dynamic executable (e.g. emacs), unpacks it, statically-links the executable using the `slink` tool, then repacks the package. We then downloaded the packages using `ckget` (Section 3.3). The *Dynamic* numbers in the table show the size of the dynamic packages (*Size*) and the number of bytes transferred when they are downloaded (*Xfer*). The packages were from the Debian "unstable" distribution.

Directory	Slinky	Dynamic	Ratio
`/bin`	90.0	7.1	12.7
`/sbin`	123.1	5.5	22.2
`/usr/bin`	2945.8	219.6	13.4
`/usr/sbin`	244.5	25.1	9.7
`/usr/X11R6/bin`	396.4	37.3	10.6
All	3799.9	264.3	14.4

(a) Storage space required for statically- and dynamically-linked executables.

Directory	Slinky	Dynamic	Ratio
`/bin`	7.0	7.0	1.0
`/sbin`	6.0	5.2	1.2
`/usr/bin`	251.6	208.9	1.2
`/usr/sbin`	26.3	25.0	1.0
`/usr/X11R6/bin`	43.0	36.2	1.2
All	305.9	251.9	1.2

(b) Storage space required for chunked executables.

Table 2: Storage space evaluation. All sizes are in MB.

The Slinky numbers show that although the statically-linked packages are quite large, only a fraction of them need to be transferred when the packages are downloaded. The chunk database is initially empty on the receiving machine, representing a worst-case scenario for Slinky as the dynamic packages already have their dependent libraries installed. The first package downloaded (`emacs`) transfers 11.9MB for the static package vs. 8.6MB for the dynamic, an increase of 39%. This is because the static package includes all of its dependent libraries. Note that the transfer size of 11.9MB for `emacs` is less than the 19.3MB size of the package; this is due to redundant chunks within the package. Subsequent static packages download fewer bytes than the package size because many chunks have already been downloaded. For `xutils` only 1.0MB of the 13.6MB package must be downloaded. Overall, the static packages required 34% more bytes to download than the comparable dynamic packages. This represents the worst-case situation for Slinky as the chunk database was initially empty whereas the shared libraries on which the dynamic packages depend were already installed.

Installation Workload	Slinky		Dynamic	
	Size	Xfer	Size	Xfer
`1> ckget emacs`	19.3	11.9	8.6	8.6
`2> ckget vim`	5.4	4.2	3.6	3.6
`3> ckget xmms`	5.7	2.8	1.9	1.9
`4> ckget xterm`	2.4	0.9	0.5	0.5
`5> ckget xutils`	13.6	1.0	0.67	0.67
`6> ckget xchat`	0.5	0.5	0.5	0.5
Total	46.9	21.2	15.8	15.8

Table 3: Network bandwidth required to download Slinky and dynamic Debian packages. The dynamic packages are the standard Debian packages; the Slinky packages are chunked before compression. The Slinky numbers include the packages' chunk files. All sizes are in MB.

5 Related Work

Slinky is unique in its use of digests to share data in memory, across the network, and on disk. Other systems have used digests or simple hashing to share data in some, but not all, of these areas. Waldspurger [22] describes a system called ESX Server that uses *content-based page sharing* to reduce the amount of memory required to run virtual machines on a virtual machine monitor. A background process scans the pages of physical memory and computes a simple hash of each page. Pages that have the same hash are compared, and identical pages are shared copy-on-write. This allows the virtual machine monitor to share pages between virtual machines without any modification to the code the virtual machines run, or any understanding by the virtual machine monitor of the virtual machines it is running. Although both ESX Server and Slinky share pages implicitly, the mechanisms for doing so are very different. ESX Server finds identical pages in a lazy fashion, searching the pool of existing pages for identical copies. This allows ESX Server to reduce the memory footprint without requiring digests as does Slinky. However, hashing is not collision-free, so ESX server must compare pages when two hash to the same value. In contrast, Slinky avoids creating duplicate copies of a page in the first place. Digests avoid having to compare pages with the same hash. Slinky also shares only read-only pages, avoiding the need for a copy-on-write mechanism.

Slinky's scheme for breaking a file into variable-sized chunks using Rabin fingerprints is based on that of the Low-Bandwidth Network File System [11]. LBFS uses this scheme to reduce the amount of data required to transfer a file over the network, by sharing chunks of the file with other files already on the recipient (most notably previous versions of the same file). LBFS does not use digests to share pages in memory, nor does it use the chunking scheme to save space on disk. Instead, files are stored in a regular UNIX file system with an additional database that maps SHA-1 values to (file,offset,length) tuples to find the particular chunk.

The `rsync` [20] algorithm updates a file across a network. The recipient has an older version of the file, and computes the digests of fixed-size blocks. These digests are sent to the sender, who computes the digests of all overlapping fixed-size blocks. The sender then sends only those parts of the file that do not correspond to blocks already on the recipient.

Venti [14] uses SHA-1 hashes of fixed size blocks to store data in an archival storage system. Only one copy of each unique block need be stored, greatly reducing the storage requirements. Venti is block-based, and does not provide higher-level abstractions.

SFS-RO [6] is a read-only network file system that uses digests to provide secure file access. Entire files are named by their digest, and directories consist of (name, digest) pairs. File systems are named by the public key that corresponds to the private key used to sign the root digest. In this way files can be accessed securely from untrusted servers. SFS-RO differs from Slinky in that it computes digests for entire files, and does not explicitly use the digests to reduce the amount of space required to store the data.

There are numerous tools to reduce the complexity of dynamic linking and shared libraries. Linux package systems such as `rpm` [17] and `dpkg` [5] were developed in part to deal with the dependencies between programs and libraries. Tools such as `apt` [2], `up2date` [21], and `yum` [23] download and install packages, and handle package dependencies by downloading and installing additional packages as necessary. In the Windows world, `.NET` provides facilities for avoiding DLL Hell [12]. The `.NET` framework provides an *assembly* abstraction that is similar to packages in Linux. Assemblies can either be private or shared, the former being the common case. Private assemblies allow applications to install the assemblies they need, independent of any assemblies already existing on the system. The net effect is for dynamically-linked executables to be shipped with the dynamic libraries they need, and for each executable to have its own copy of its libraries. This obviously negates many of the purported advantages of shared libraries. Sharing and network transport is done at the level of assemblies, without any provisions for sharing content between assemblies.

Software configuration management tools such as `Vesta` [8] automatically discover dependencies between components and ensure consistency between builds. Such tools are indispensable to manage large software projects. However, they do not help with the DLL Hell problem that stems from inconsistencies arising on a user's machine as the result of installing binaries and libraries from a multitude of sources.

6 Conclusion

Static linking is the simplest way of combining separately compiled programs and libraries into an executable. A program and its libraries are simply merged into one file, and dependencies between them resolved. Distributing a statically linked program is also trivial — simply ship it to the user's machine where he can run it, regardless of what other programs and libraries are stored on his machine.

In this paper we have shown that the disadvantages associated with static linking (extra disk and memory space incurred by multiple programs linking against the same library, extra network transfer bandwidth being wasted during transport of the executables) can be largely eliminated. Our Slinky system achieves this efficiency by use of digest-based sharing. Relative to dynamic linking, Slinky has no measurable performance decrease, a comparable memory footprint, a storage space increase of 20%, and a network bandwidth increase of 34%. We are confident that additional tuning will improve these numbers. Slinky thus makes it feasible to replace complicated dynamic linking with simple static linking.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CCR-0073483.

References

[1] O. Aoki and D. Sewell. Debian quick reference. `http://www.debian.org/doc/manuals/en/quick-reference.pdf`.

[2] APT Howto. `http://www.debian.org/doc/manuals/apt-howto/index.en.html`.

[3] F. Corbat and V. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS FJCC*, pages 185–196, 1965.

[4] P. Davis. Ardour. `http://boudicca.tux.org/hypermail/ardour-dev/2002-Jun/0068.html`.

[5] The dpkg package manager. `http://freshmeat.net/projects/dpkg`.

[6] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[7] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, July 2003.

[8] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management, 1999.

[9] J. R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 2000.

[10] H. Lu. ELF: From the programmer's perspective, 1995.

[11] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[12] S. Pratschner. Simplifying deployment and solving DLL Hell with the .NET framework. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp`, Nov. 2001.

[13] prelink. `http://freshmeat.net/projects/prelink`.

[14] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX Conference on File and Storage Technologies (FAST)*, Monterey,CA, 2002.

[15] M. Rabin. Combinatorial algorithms on words. F12 of NATO ASI Series:279–288, 1985.

[16] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[17] The RPM package manager. `www.rpm.org`.

[18] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

[19] RFC 3174 - US secure hash algorithm 1 (SHA-1).

[20] A. Tridgell and P. Macherras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[21] NRH-up2date. `http://www.nrh-up2date.org`.

[22] C. A. Waldspurger. Memory resource management in VMware ESX server. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, 2002.

[23] yum. `http://linux.duke.edu/projects/yum`.