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`

Technical Report TR04-02

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 require a particular set of libraries during execution. As a consequence, the executable image that was tested by the developer is exactly the same as gets executed by the user, 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: Slinky executables are completely self-contained and 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, and a slight modification of the Linux kernel to use those digests to share code pages. Results show that our unoptimized prototype has a performance decrease of at most 4% and a space increase of 40% relative to dynamic linking.

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

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 In many cases these problems can be traced back to missing, out-of-date, or incompatible dynamic libraries on the user's computer. In the Windows community this problem is affectionately known as *DLL Hell* [11].

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.

In this paper we will also show that the cost of static linking, in terms of file-system storage, network bandwidth, and run-time memory usage, can be largely eliminated using 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. Digests ensure that a particular chunk of data is only stored once in memory or on disk, and only transported once over the network.

1.1 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 linking a program at load-time or run-time, and sharing libraries between executables both on disk and in memory. 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 so the follow-on to Multics, Unix [15], originally 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.

Dynamic linking has several perceived benefits that make it so popular. One is that with proper care libraries can be updated without relinking the executables that use it. This makes it easier to update libraries. Dynamic linking also makes it possible to share a single copy of a library among multiple executables, both in memory and on disk. While it is possible to statically link a shared library, the complexity and inflexibility of doing so make it much less prevalent than dynamic linking.

In dynamic linking, symbol references are not resolved until the executable runs. At link-time the linker simply records in the executable the names of the libraries it needs. These names are used at runtime to find the libraries and resolve the symbols. Since executables refer to libraries by name, only one copy of a library is needed on disk. In addition, the libraries are compiled in such a way that only a single copy in memory is shared by all processes using that library. The net result is a dramatic reduction in the amount of disk space and memory required as compared to static linking, as well as the amount of network bandwidth required to transfer an executable.

1.2 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 programmers are all too familiar with the "DLL Hell" that can occur in ensuring that all library dependencies are met, especially when different executables depend on different versions of the same library.

In early versions of the Windows operating system a common cause of DLL Hell was the installation of a new program that caused an older version of a library to replace an already installed newer version. Programs that relied on the newer version then stopped working — often without apparent reason. Unix and newer versions of Windows fix this problem by applying version numbers to libraries. However, DLL Hell can still occur if, for example, the user makes a minor change to a library search `PATH` variable. This can cause a different and incompatible version of a library to be loaded, so that a previously working program now fails.

While the "instant update" feature of dynamic linking is 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. Since a dynamically-linked executable can be run with the new library version without having gone through a regression test, there is a risk that the bug fix will have unforeseen consequences. For example, if the library bug was known to the developer he may have devised a "workaround", possibly in a way no longer compatible with the bug fix. Thus, after a library update some programs may work better and some programs may cease to work.

Software removal is also complicated by dynamic linking. 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. Tools are needed to resolve these dependences and install and remove software properly.

1.3 Contributions

For the reasons outlined above we believe that static linking is superior to dynamic linking. The biggest problem with static linking is the size of the resulting executables. We are developing a system called Slinky that supports the efficient execution, transport, and storage of statically-linked executables. 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 [17]. 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 discuss related work. In Section 5 we show that empirically our system makes static linking as efficient as dynamic linking. Section 6, finally, summarizes 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* [7]. 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 a copy of each library to which it is linked. For large, popular libraries, such as the C library, the amount of wasted space 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.

It is also possible for the executable to specify at run-time which libraries to link using a facility such as `dlopen()`. A good example is plugins for a Web browser. SLINKY does not address libraries that are specified at run-time, although there is no reason why the two techniques can not coexist.

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 *indirection 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 a 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 the executable itself, as well as 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 [16] and Debian's dpkg [4] 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 [20] or Debian's apt [1] to fetch the desired packages from on-line repositories and install them.

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 or on disk. Slinky uses SHA-1 [17] 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. This means that for all practical purposes if two chunks of data have the same SHA-1 hash, then they are identical. Although there is a negligible chance that two different chunks will hash to the same value, it is much more likely that a hardware or software failure will corrupt a chunk's content.

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 reduce the amount of data required to transfer an executable over the network. We developed a Slinky prototype that implements sharing of memory pages based on digest. Sharing of disk space and reduction of network traffic is currently work-in-progress; we describe how Slinky will provide that functionality, but it has not yet been implemented. We also describe the limitations of the Slinky approach.

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 a simpler approach that avoids the complexity of copy-on-write by 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 modifications to the Linux kernel to use the digests in the executables to share pages between processes.

Figure 1 illustrates how Slinky functions. A source file `x.c` that makes use of the C library is compiled into an object file `x.o`. Our program `slink` links `x.o` with the dynamic library `libc.so`, copying its pages. The `digest` program adds a new ELF sec-

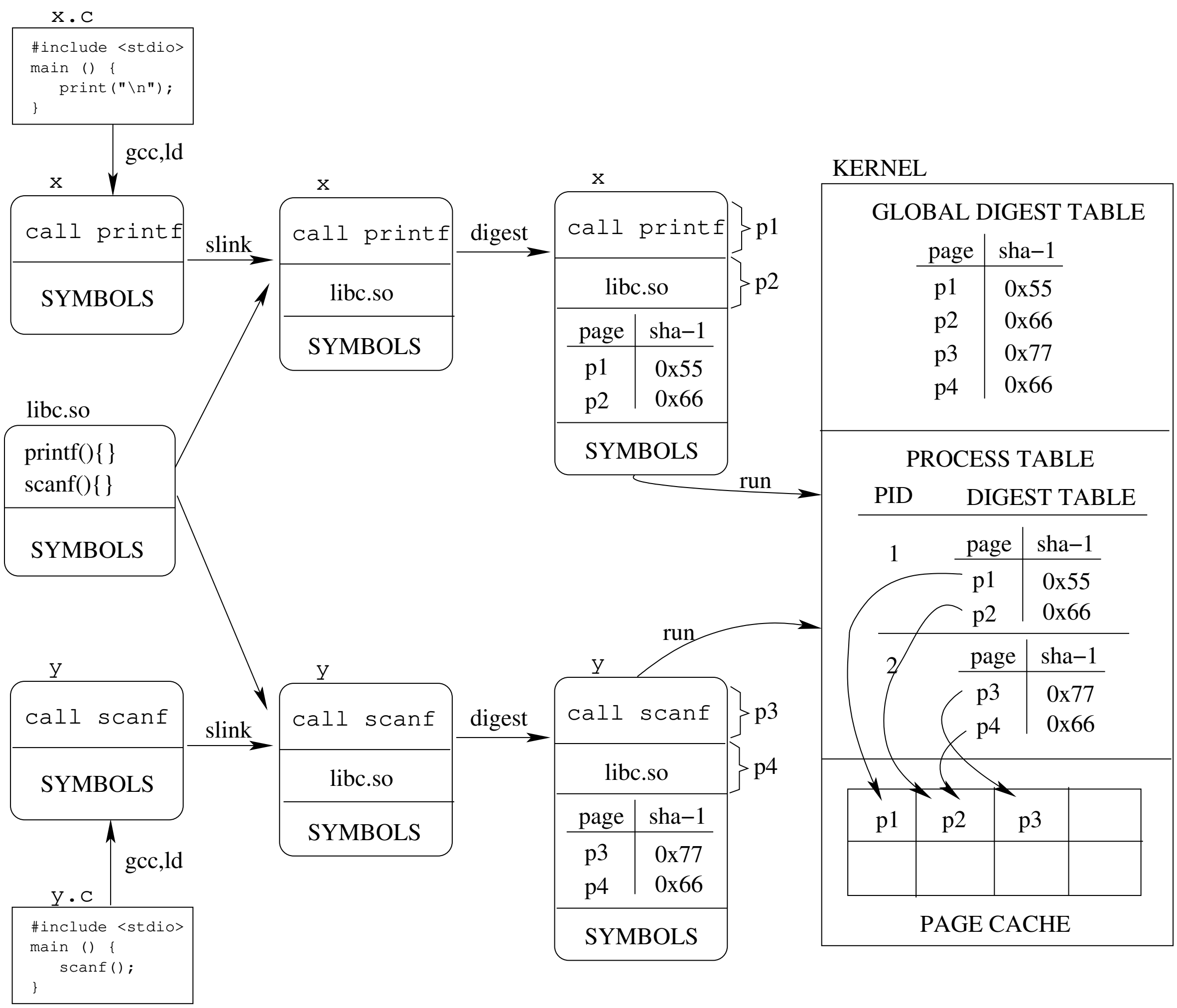

Figure 1: An overview of SLINKY functionality.

tion 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 `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 1400 lines of C and 200 lines of shell script.

The input to slink is a dynamically-linked executable in ELF format [8]. Slink uses a slightly-modified version of the prelink [12] 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 zeroing out any data structures related to dynamic linking that are no longer needed. Removing these data structures is complicated because doing so changes the addresses of the sections that follow them, requiring additional relocation. For this reason, slink simply fills them with zeros. This allows the data structures to share pages of zeros, reducing the space they consume. A future version of slink will remove them altogether.

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 8KB, and the digest is 20 bytes, so the digests introduce an overhead of 20/8196 or less than 0.3% 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 the executable file 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, and a page is removed from the GDT when its reference count drops to zero.

Slinky uses the digests stored in the executables to share pages, so the system correctness and security depends on the digest correctness. A malicious user could modify a digest or a page so that the digest no longer corresponds to the page's contents. Modifying a digest will cause the page to have the wrong contents when the executable is run, but it is no worse than modifying the executable's page directly. Modifying a page but not its digest is a more serious concern, since if the executable is run and it is the first to use the page, the modified page will incorrectly be entered into the GDT under the wrong digest. This can be used to introduce an exploit into other processes that share the page. One solution is for the kernel to verify the digests of pages that are added to the GDT. This requires the kernel to compute digests, which increases the complexity of the kernel and slows down the handling of page faults that require the page to be added to the GDT. The Slinky prototype adopts a simpler solution in which it only shares pages between executables that are owned by root. This is simple to implement and reasonably secure, as a malicious user who gains root access could modify the digests, but he could also simply modify the pages of the executables he wishes to subvert.

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 [14], then computing the digests of the chunks. The technique is based on that used in the Low-Bandwidth File system (LBFS) [9]. A small window is moved across the data and the Rabin fingerprint of the window 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 5.

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. Suppose we want to transfer an executable from A to B over the network. First, A breaks the executable into chunks and computes the digests of the chunks. A then sends the list of digests to B. B compares the provided list with the digests of chunks that it already has, and responds with a list of missing digests. A then sends the missing chunks to B. This is the basic idea behind LBFS, and Slinky will use a similar mechanism. Slinky will integrate network transport with file storage, allowing both to use the same chunks and digests for a file. This will avoid having to re-chunk and re-hash data in the network transport system, as is done in LBFS. LBFS also compresses chunks before transferring them, something that will likely be useful in Slinky as well. The current Slinky prototype does not use digests to reduce network bandwidth, but we have developed tools to determine how much chunk sharing there is between executables, as described in Section 5.

3.4 Limitations

Since Slinky statically links executables, it does not provide some of the benefits of dynamic linking. In particular, updating a library has no effect on statically-linked executables. This can be a good thing, as it means incompatible library upgrades or inadvertent library deletions do not affect executables, but it also means that library updates have no effect until the executables are relinked. It is questionable how much benefit this flexibility really provides, and is a subject of future work in the Slinky project. Slinky can ameliorate the problem by including library version information in the executables to help in determining which executables use which library versions. It is likely that Slinky will retain dynamic linking of libraries determined at run-time, such as plugins, since that functionality isn't easily provided by static linking.

4 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. Hashing is not collision-free, so pages must be compared when two pages 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 [9]. 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 [19] 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 [13] 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 [5] 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 [16] and dpkg [4] were developed in part to deal with the dependencies between programs and libraries. Tools such as apt [1], up2date [20], 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 [11]. 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.

Task	Dynamic (sec.)	Slinky (sec.)	Slowdown
Untar Linux kernel	148	154	4%
Build Linux kernel	205	206	1%
Format this paper	2.6	2.7	4%

Table 1: Times to perform a variety of tasks.

Dynamically loading code in a secure fashion has received much attention. Type-safe languages such as Java [6] were developed in part to address this concern. Other techniques such as software fault isolation [21] and proof-carrying code [10] try to ensure that loading malicious code does not affect the main program. Systems such as SPIN [2] and Vino [18] focus on loading secure extensions into the operating system.

5 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.

5.1 Performance

We ran several benchmarks to compare the performance of Slinky with that of a standard dynamically-linked Linux system. These 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 as of 12/12/03. The machine was a desktop workstation used for operating system development, so it had a representative set of software development and GUI applications installed. All numbers are the average of three trials.

Table 1 shows the elapsed time of various tasks using dynamically-linked and statically-linked executables. As can be seen, the performance impact is at most 4%. This is not surprising, as the only performance overhead that Slinky introduces is manipulating the PDTs and GDT, both of which are dominated by the cost of handling a page fault. These tables are currently implemented by linked-lists in the Slinky prototype, so it is likely that performance will improve as we tune the system.

5.2 Space Requirements

Table 2 shows the space required to store dynamically-linked executables vs. statically-linked. These numbers were also collected on the "unstable" Debian distribution as of 12/12/03. 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 may be shared between rows). The *Slinky* column shows the space required to store the statically-linked executables, and *Expansion* shows the ratio of space required by the statically-linked executables to the dynamically-linked. Obviously the space overhead is significant unless efforts are made to reduce it, as described next. The current Slinky prototype does not statically link the executables very efficiently (Section 3.1.1); as a result the executables contain a lot of wasted space filled with zeros. The *Waste* column shows this amount, which is significant.

Table 3 shows the amount of space required to store the dynamic and static executables if they are broken into variable-size chunks and the digest of each chunk computed. Only one copy of each chunk is stored. The dynamic executables show a modest improvement over the numbers in the previous table due

Directory	Dynamic (KB)	Slinky (KB)	Expansion	Waste (KB)
/bin	5008	93078	18.6	7633
/sbin	6529	192576	29.5	16050
/usr/bin	218786	3219889	14.7	372176
/usr/sbin	20253	210422	10.4	21611
/usr/X11R6/bin	35613	325936	9.2	31083
All	259628	4041901	15.6	448553

Table 2: Storage space required for statically-linked and dynamically-linked executables.

Directory	Dynamic (KB)	Slinky (KB)	Expansion
/bin	4712	6913	1.5
/sbin	6107	10860	1.8
/usr/bin	210470	295348	1.4
/usr/sbin	20141	24913	1.3
/usr/X11R6/bin	34168	42524	1.3
All	249041	352213	1.4

Table 3: Storage space required for chunks.

to commonality in the files. The static executables, however, show a tremendous reduction in the amount of space. This is because most of the extra space in Table 2 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 requires 40% more space than dynamic linking. SLINKY consumes 350MB to store the executables instead of 250MB, or 100MB more. This is a very small fraction of a modern disk drive, but nonetheless we are confident that tuning the system will significantly reduce the overhead. Part of the problem is the wasted space shown in the previous table, although it is unclear how much of a contribution it makes. Compression can be used to reduce the size of the chunks, as was done in LBFS, and we expect that it would work well for SLINKY too, although we haven't experimented with it. The wasted space will obviously compress very well, though.

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. SLINKY has a performance decrease of at most 4% and a space increase of 40% relative to dynamic linking, and we are confident that these can be improved with system tuning. SLINKY makes it feasible to replace complicated dynamic linking with simple static linking.

References

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

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

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

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

[5] Kevin Fu, M. Frans Kaashoek, and David Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[7] John R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 2000.

[8] Hongjiu Lu. ELF: From the programmer's perspective, 1995.

[9] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[10] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[11] Steven 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`, November 2001.

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

[13] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

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

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

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

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

[18] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Cambridge, MA, 1994.

[19] A. Tridgell and P. Macherras. The rsync algorithm, June 1996.

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

[21] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[22] Carl 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/`.