Reducing Startup Latency in Web and Desktop Applications

Dennis Lee, Jean-Loup Baer, Brian Bershad, and Tom Anderson
Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350
{dlee,baer,bershad,tom}@cs.washington.edu

Abstract

Application startup latency has become a performance problem for both desktop applications and web applications. In this paper, we show that much of the latency experienced during application startup can be avoided by more efficiently packing application code pages. To take advantage of more efficient packing, we describe the implementation of demand paging for web applications. Finally, we show that combining demand paging with code reordering can improve application startup latency by more than 58%.

1 Introduction

Program startup latency is an important performance problem for both desktop and web applications. Desktop applications are becoming larger and improvements in disk and network speeds have not kept up with the improvements in CPU speed. For example, Microsoft Word 2.0 on an Intel 66 Mhz 486 takes about 14 seconds to start, while Microsoft Word 7.0 on a 200 Mhz Pentium Pro takes almost 17 seconds to start.

The situation is worse for web applications on the Internet. For many users, web applications take a long time (a few minutes) to start because they connect to the Internet through high latency, low bandwidth links (i.e., modems). For example, Vivo, a popular video display and control application, takes over 8 minutes to download over a 56 Kbps modem link and over 2 minutes on a 256 Kbps DSL link. For applications on corporate Intranets, frequent updates of internal corporate applications require web professionals to frequently download the applications through slow modem or wireless links.

A large portion of startup latency involves transferring a large number of code bytes through slow links. For web applications, the dominant cost is downloading the entire application over slow network links. For desktop applications, program startup involves loading a large number of code pages from the relatively slow disk. This paper proposes a technique that attempts to transfer the minimum amount of code (and consequently bytes) through expensive disk or network links.

Researchers have recently proposed several ways for improving startup latency including compression [Enst et al. 97, Franz & Kistler 97], non-strict execution [Krintz et al. 98], just-in-time code layout [Chen & Leupen 97], and optimizing disk layout [Melanson 98] . Our approach is orthogonal and uses *code reordering* [Pettis & Hansen 90] and *demand paging* [Levy & Redell 82] to improve the startup latency of web and desktop applications, and reduce the load on web servers and the network.

We show that code reordering can significantly improve upon the performance of pure demand paging systems. To allow execution without requiring the entire binary to be present, we implement an extension of demand paging for web applications. The combination of these two techniques can improve the startup latency of web applications by more than 58% and that of desktop applications by more than 53%.

The rest of this paper

Section 2 presents the motivation and architecture for improving application startup latency. Section 3 describes our experimental methodology and setup. In Section 4, we present the results of our measurements. Finally, Section 5 concludes.

2 Approach

Program startup typically involves waiting for the program binary to be loaded into memory through relatively expensive links (i.e., the network or disk). For web applications, this involves downloading the whole application through the Internet. For desktop applications, this involves paging in all the code and data needed to initialize the application.

In this section, we describe our approach to improving startup latency which attempts to reduce the number of code bytes that passes through expensive links. Our approach places procedures that will probably be used by the application into a single contiguous block in the binary. This improves the effectiveness of demand paging systems and consequently reduces startup latency.

As motivation, we first present the results of profiling several web and desktop applications. These profiles show that existing applications can be better laid out to optimize startup latency. We then present our architecture for code reordering and our architecture for allowing partial downloads of web applications using demand paging.

2.1 Motivation

Web Applications

Application	Description	Size (MB)
envoy	Document viewing control	1.09
scout	VRML parser and renderer	0.98
vivo	Applet for watching movies	0.44
whip	AutoCAD drawing display control	0.49

Desktop Applications [1]

Application	Description	Size (MB)
acrobat	Adobe Acrobat Reader 3.0: Reader for portable document format (PDF) files.	2.26
netscape	Netscape Navigator 3.1 web browser.	3.17
photoshop	Adobe Photoshop 4.0 image editing package.	3.65
powerpoint	Microsoft PowerPoint 7.0b presentation package.	4.36
word	Microsoft Word 7.0 word processor.	3.78

Table 1: *Web and Desktop Applications. The table describes the applications used in this paper. Our web applications consists of four ActiveX [Chappell 96] controls which display various document types. The size column gives the size of the main application binary. For web applications, the given size is the uncompressed size of the main application binary and does not include dynamically loaded libraries (DLLs) used by the applications.*

Application	Binary Size (KB)	Breakdown		
		Data (KB)	Used (KB)	Unused (KB)
envoy	1,089	248 (23%)	164 (15%)	685 (62%)
scout	979	285 (29%)	262 (27%)	440 (45%)
vivo	444	274 (62%)	102 (23%)	69 (16%)
whip	487	197 (40%)	71 (14%)	224 (46%)

Table 2: *Procedure Utilization in Web Applications. The percentages in parenthesis show the fraction of the entire binary covered by the particular component. The data column shows the size of all the non-code sections of the binary including section headers. The unused column shows the potential benefit of not downloading the entire application binary.*

Application	Code Pages		
	Total	Touched	Utilization
acrobat	404	246 (60%)	28%
netscape	388	388 (100%)	26%
photoshop	594	479 (80%)	28%
powerpoint	766	164 (21%)	32%
word	743	300 (40%)	47%

Table 3: *Desktop Application Profile Results. Table gives the number of code pages touched during program startup. Utilization gives the average fraction of used procedures in touched code pages. Page fault rates during startup could be reduced by better packing code pages.*

We profiled four web applications and five desktop applications (c.f., Table 1) to determine if there was an opportunity to improve startup time by improving the layout of procedures in a program binary.

Table 2 shows the statistics derived from profiling four web applications. We ran these applications to completion with a typical workload to determine how many procedures in the application are actually used. The table shows that the applications utilize only between 38% (*envoy*) to 84% (*scout*) of the bytes in their program binaries. Typically, web applications download their entire program binaries before starting. The utilization statistics suggest that startup times could be significantly improved if we download only the procedures actually used by the application.

Table 3 shows statistics for code pages of different desktop applications during startup. The binary for desktop applications is demand paged so we are examine the utilization of the code pages brought in during startup. The table shows that only 26% (*netscape*) to 47% (*word*) of the code pages brought in during program startup are utilized. Interestingly, *netscape* and *photoshop* touch almost every code page in their main binary during startup. For these two applications, demand paging does not reduce the number of bytes accessed on the disk. The low page utilization suggests that like web applications, we could improve startup latency for desktop applications by only loading the actual procedures used by the application.

2.2 Architecture

The previous subsection showed that much of the code transferred over the network or transferred from the disk is not used by the application. Our approach attempts to transfer only the used procedures in the application. We use profile information and an object rewriting system to move the likely-used procedures in the binary together, essentially packing pages better to make the demand paging system more efficient.

Figure 1 shows a diagram of the object rewriting phases of our approach. We use profile information to predict with high accuracy which part of the application would be used. Using an object rewriting engine, we then move the likely-used procedures together at the top of the code section. For our experiments, we simply arrange the code section in first-touch order. Ordering using procedure affinity [Pettis & Hansen 90] might be better for locality but first touch order works well enough for our goal of improving startup latency.

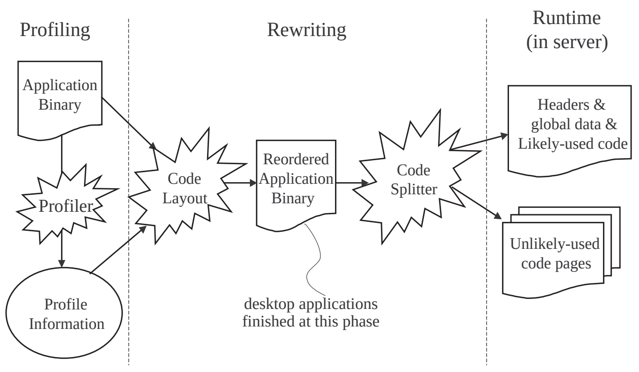


Figure 1: *Object Rewriting Phases*

For desktop applications, the system is done after generating the binary from the code reordering phase. The built-in O/S demand paging system will load in only the fraction of the pages containing the used procedures.

For web applications, the system needs to be able to download only part of the application required for execution. The code-splitting module splits the binary into 1) a large main binary that contains the data portion of the binary and the likely-used procedures, and 2) several page-sized files containing the unlikely-used procedures. At runtime, the server only returns the main binary. When the client needs an unlikely-used procedure, it can easily ask for the page-sized file that contains the procedure.

Web Client Architecture

Our client architecture extends demand paging to web applications to provide a convenient mechanism to detect missing pages and to allow the system to function correctly when control passes to functions that are not present in the initial download.

Figure 2 shows a diagram of what happens on the web client during program runtime. When a web application is accessed by the client browser, only pages containing the data and likely-used portion of the binary are downloaded. The part of the binary that has not been downloaded is marked PAGE_NO_ACCESS by the system.

If the application transfers control to a page that is marked PAGE_NO_ACCESS, the page-fault handler is invoked. We modified the handler to contact the web server, download the file containing the page, place the page in the appropriate location in application memory. [2]

[2] As an alternative to downloading individual files each containing a single code page, the client could use the range option in `HTTP 1.1` [Nielsen et al. 97]. This would avoid splitting the application binary into multiple files.

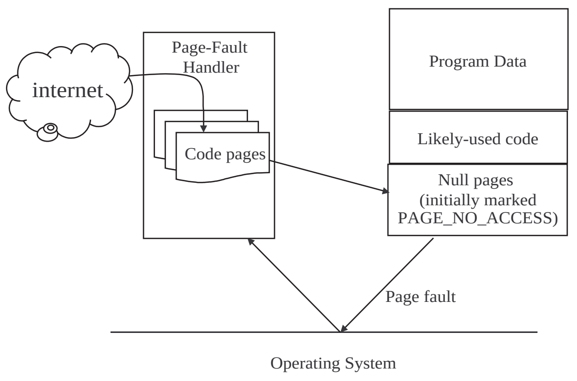


Figure 2: *Web Client Architecture*

3 Experimental Methodology

3.1 Startup Latency

To determine the startup latency, our timing system 1) invokes the application, and 2) simulates a user initiated event by sending a message to an application window. We define startup latency as the time from the invocation of the application to the time the application replies to the message sent by the timing system.

Our timing method works because of the Windows NT event queue model and the way most Windows applications are written. Under Windows NT, windows are assigned to threads, and messages are sent to thread-local event queues. Messages are delivered to this queue and do not interrupt the execution of the owning thread. For most applications, threads process their message queue only when they have finished initializing and are ready to respond to user input.

Occasionally, a thread responsible for the main application window will respond to a user message even when other threads are still drawing other windows (e.g., tool bar windows). Since users are unlikely to interact with the application until after all the initial windows have been drawn, the timing system sends a message not to the main application window but rather to the window last drawn during application startup and waits for the reply.

3.2 Environment

Our client system was a Pentium Pro 200 system running Windows NT 4.0 Service Pack 3, with 128 MB of memory, and a Seagate ST34371W disk. We used a slightly modified version of Internet Explorer 4.0 as our browser. Our measurements were taken using the processor cycle counter and the performance counters built into Windows NT. All our network measurements were taken on isolated networks with no external traffic.

For our web server, we use Apache 1.3b5 running on FreeBSD 2.2.6. To control the bandwidth and latency between the web server and the client, we installed the dummynet [Rizzo 99] patch to the BSD kernel. Our Internet application experiments looked at a range of bandwidths (from 56 Kbits/second to 3 Mbits/second) and latencies (from 10 ms. to 200 ms.) which cover the range of network conditions on the Internet.

The application server used in our experiments with application startup latency was a Pentium Pro 200 system running Windows NT 4.0 service pack 3 with 64 MB of memory. Unfortunately, we could not control the bandwidth or latency to the application server on NT so our measurements for desktop applications only involve communication on a single shared 10 Mbit Ethernet link.

We used Etch [Romer et al. 97, Lee et al. 98], a binary instrumentation and rewriting engine, to profile and rewrite the applications used in this study. For all our experiments, we profile and reorder only the main application binary. For our prototype implementation, we simulate having an augmented page fault handler using the Windows NT debugger API [Microsoft 98]. We run the web browser in the context of a custom debugger.

4 Results

In this section, we present the results of our experiments optimizing the startup latency. Section 4.1 describes the results for web applications, and Section 4.2 describes the results for desktop applications.

4.1 Web Applications

In this subsection, we show the performance of our optimization on web applications. We first present a performance model describing the startup latency of web applications and show how our optimization improves startup latency. We then compare four different schemes for starting web applications.

4.1.1 Performance Model

A simple model for predicting the startup latency of web applications:

$$Startup = \frac{Bytes}{Bandwidth} + Requests \times Latency + Overhead \tag{1}$$

where:

- *Bandwidth* and *Latency* are the observed network bandwidth and latency between the client and server,
- *Bytes* is the number of bytes transfered,
- *Requests* is the number of requests for files to the web server, and
- *Overhead* is the fixed overhead of executing instructions to start the application.

Equation 1 suggests that we can improve startup time by reducing the number of bytes transfered and transfer all the needed bytes in a single requests. Unfortunately, we cannot predict with perfect accuracy the actual bytes that the application would use. Download schemes thus have to strike a balance between including as little as possible into the initially downloaded package and paying the cost of extra requests.

In the next subsection, we will consider the case of "improved" schemes that reduces the number of bytes downloaded at the cost of more requests. An improved scheme would be faster than the original scheme, if and only if:

$$Startup_{Original} > Startup_{Improved} \tag{2}$$

$$\frac{Bytes_{Original} - Bytes_{Improved}}{Requests_{Improved} - Requests_{Original}} > Bandwidth \times Latency \tag{3}$$

Equation 3 implies that the performance of one scheme relative to another is closely related to the bandwidth-latency product. An improved scheme may be faster when the bandwidth-latency product is small but the same scheme might actually be slower when the bandwidth-latency product is large.

4.1.2 Different Approaches

We compare the performance of four approaches to starting web applications. These approaches examine the performance of demand paging and reordering, and probe the space of trade-offs between 1) eager approaches which download more bytes in a few requests, and 2) lazy approaches which download less bytes in many requests.

- *Original* downloads the entire binary at once. This approach assures that there will only be a single request.
- *Paged* downloads a code page of the binary only when it is needed. All of the program data is still downloaded initially. This approach reduces the number of bytes transfered (i.e., unused pages are not transfered) but may have to pay a high cost because of request latency.
- *Reordered-Paged* is like *paged* in that it downloads a code page only when needed. But this approach first reorders the procedures in the application to more densely pack likely-used procedures into fewer code pages. Compared to *paged*, *reordered-paged* minimizes the number of pages needed by the application, effectively reducing the number bytes transfered and the number of requests to the server that have to be made.
- *Reordered* initially downloads the likely-used portion of the code section with all the data in the binary. It still has to pay the cost of a request when control transfers to a page that is not in the initially downloaded portion of the code but this would be much rarer. *Reordered* may transfer more bytes than the *reordered-paged* approach since some of the pages in the likely-used portion of the binary may not be used.

We implemented a prototype of each of these approaches. Each prototype reorders, pages, and downloads only the main application binary and not the libraries that the binaries depend upon.

Figure 3 shows the results of our experiment starting applications using the different approaches and network conditions. [3] The figure show the improvement in startup latency for a workload that is different from the profiled workload used to reorder the binary (c.f., Section 2.2).

We highlight a few trends from the figure:

- *Reordered* almost always does better than *original.*
- For low bandwidth (e.g., 60+Kbps) and low latency (e.g., 10 ms) connections, *paging* does better than *original.* However, as the bandwidth-latency product increases (graphs towards the bottom right), *paged* makes too many requests from the server and doesn't reduce the number of bytes sufficiently to compensate.
- Comparing *paged* with *reordered-paged* shows that demand paging still leaves much room for improvement. In all cases, *reordered-paged* does better than *paged*, especially in the cases of *envoy* and *scout.*
- *Reordered* attempts to do better than *reordered-paged* by reducing the number of server requests. The results show that *reordered* is better than *reordered-paged* for high latency-bandwidth product networks. However, for low-bandwidth networks, the reduced number of bytes transferred by *reordered-paged* make that option better for startup.

The figure also shows a case where our methods are not very effective. For most cases with *vivo*, *reordered* and *reordered-paged* do not do significantly better than *original.* Since *vivo* is already fairly well compacted (84% of the binary is used, c.f., Table 2), we are hard pressed to make the binary more efficient.

[3] Figure 6 in the Appendix shows the raw startup latency numbers for all network conditions.

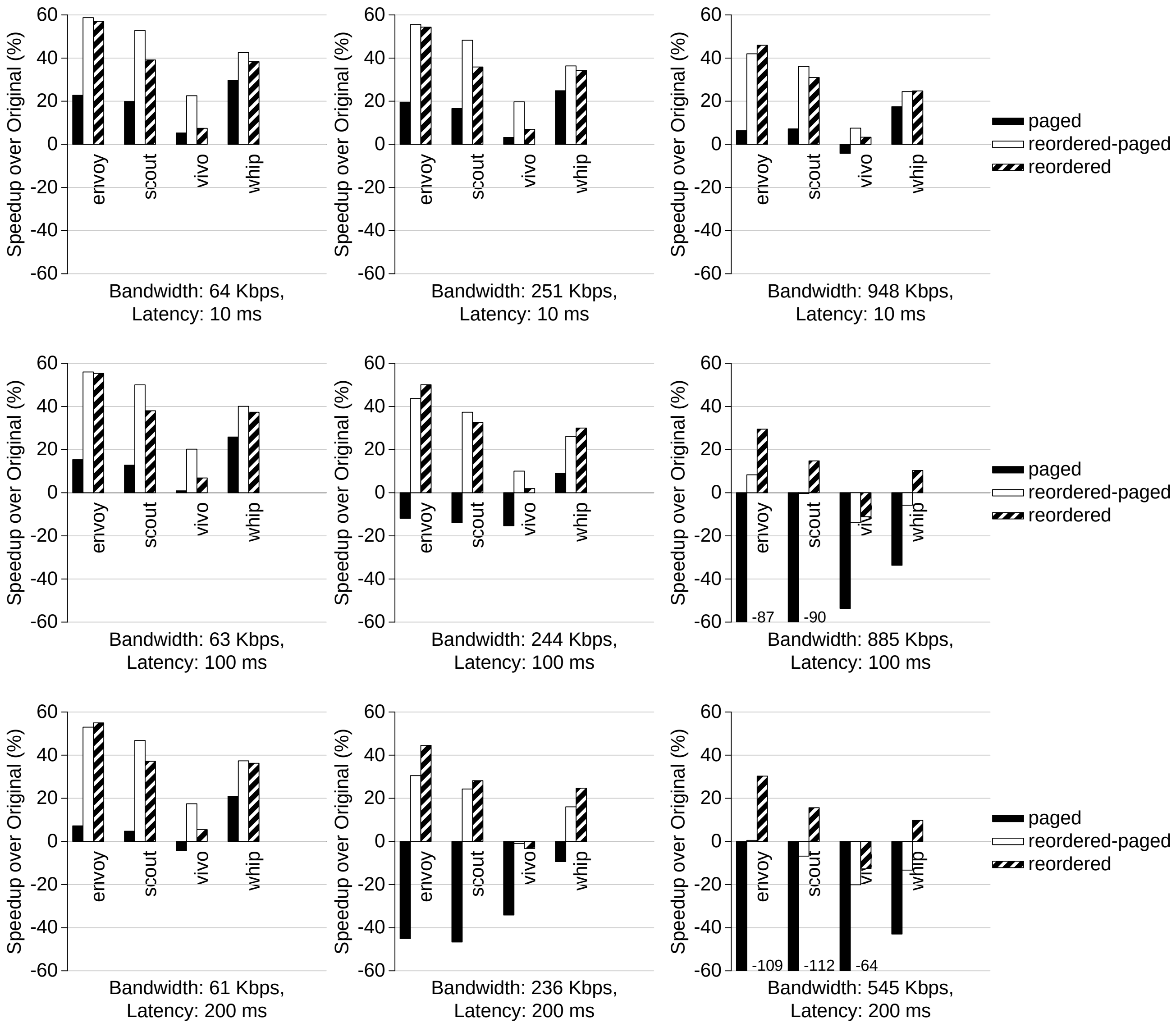


Figure 3: *Web Application Results. Graphs show the improvement in startup latency of web applications under various network conditions. The measured workload was different from the profiled workload used to reorder the web binary. The titles show the measured bandwidths between the client and the server. This is different from the "available" bandwidth (56 Kbps, 256 Kbps, and 1 Mbps) because of the effects of TCP buffering and congestion control [Peterson & Davie 96], and the limitations of dummynet. We attempted to get data points at higher bandwidth-latency products but were limited by the TCP receive buffer size used in Internet Explorer.*

Analysis of Downloaded Bytes

Figure 4 shows the breakdown of the bytes downloaded during the startup of the different applications. *Paged* downloads less bytes than *original* but generates a fair number of server requests. As shown in Figure 3, this does not work well in the presence of high latency. As expected, *reordered-paged* downloads the least number of bytes and has significantly fewer server requests compared to *paged*. This explains why *reordered-paged* is always better than *paged*.

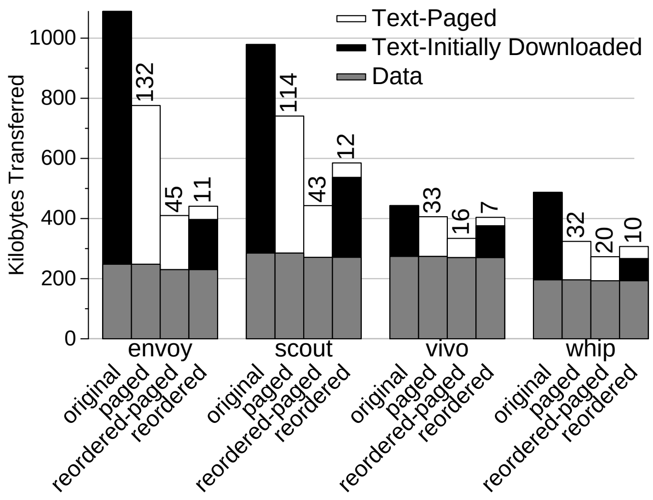


Figure 4: *Download Statistics. Numbers on top of the bars give the number of faults requiring access to the web server. Reordered-paged downloads less bytes than paged and accesses the web server less. Reordered downloads more bytes than reordered-paged but accesses the web server more.*

Reordered reduces the number of server requests further but transfers more bytes. The figure shows that *reordered* still experiences some faults and downloads more bytes than *reordered-paged*. The reason is three-fold. First, the measured workload is different from the profiled workload. We expect that some of the code used in one run would not be used in the other. Second, we profiled the *entire* program run, rather than just startup, to avoid faults even in the middle of the program. Hence, we initially download code that may not be used immediately. This is the case for *reordered* in *scout*. Finally, our profile does not identify the use of data embedded in the code section. This may cause faults on access to code pages containing data.

Reducing the number of bytes transfered has benefits other than reducing the startup latency of web applications. [Banga & Druschel 99, Bradford & Crovella 98] showed that having a large number of open connections on a web server can cause serious performance degradation. By reducing the amount of work that the server has to perform, our techniques can reduce the duration of each connection hence reducing the load on servers serving up these Web applications.

4.2 Desktop Applications

In this subsection, we examine the effect of reordering on desktop application startup latency. We consider the effects of reordering on four different states of the O/S file cache:

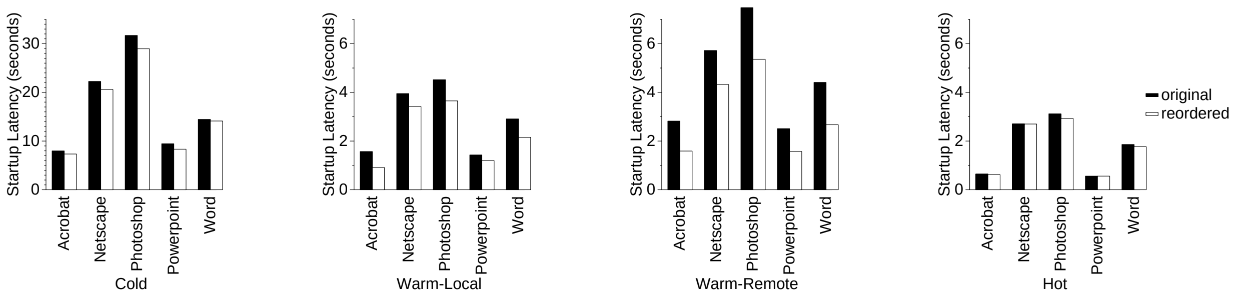
- *Cold* - the file cache is empty. All pages for all files used by the application (including shared libraries) have to be read in from disk.

- *Warm-local* - the file cache contains the pages from shared libraries but not those of the main application binary. This corresponds to the case when a user starts an application either for the first time or after it has been purged from the disk cache. We expect that the shared libraries would be in the file cache due to other applications loading them.

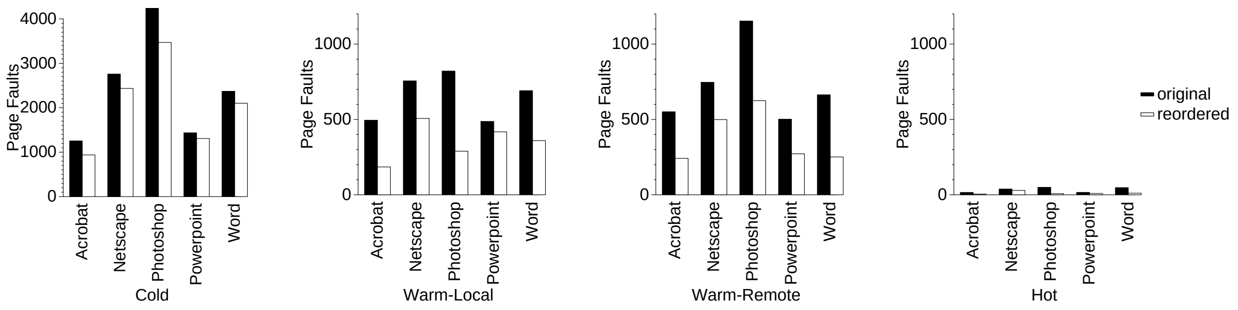
- *Warm-remote* - like *Warm-local* except that the main application binary lives on a remote server. We assume that the application binary is in the file cache of the remote server. This corresponds to the case when a user starts up an application on a shared file server. Since Windows NT purges its local copy of the file when an application exits, [4] we expect this case to happen often in environments with shared application servers.

- *Hot* - the file cache contains all the application and library pages.

We ran the original and reordered application under these four scenarios. To filter out spurious results, we performed our measurements at least ten times for each application and scenario. We dropped the runs with the highest and lowest times, and report the average of the other runs. Figure 5 shows the results of these experiments.



(a) Startup Latencies



(b) Page Faults

Figure 5: *Effect of Reordering. Page fault data was obtained using the performance counters built into Windows NT. Reordering improves application startup latency and reduces the number of page faults experienced by the program.*

[4]Actually, Windows NT purges the local copy of a file when there is no longer an open handle to the file in the local machine [Leach & Naik 97].

For all applications and configurations, reordering improves application startup latency. The two *warm* scenarios show that reordering the binary can significantly reduce startup time. It is especially effective for *powerpoint* and *word* as they improve their application startup time by 53% and 39% respectively. Page fault rates decrease a corresponding amount for the cases where we improve the application startup latency. This verifies that the dominant cost during startup are page-faults.

Since we only reordered the main application binary, we expect to mildly improve the *cold* case. This is shown by Figure 5 as we see from 2% (*word*) to 11% (*powerpoint*) improvement. We also included the *hot* case to see if reordering procedures in first-touch order would slow down this case. The figure shows that for the *hot* case, reordering does not slow down the applications and even slightly improves *photoshop* and *word*.

5 Conclusion

We have implemented a system that uses code reordering and demand paging to improve the startup latency of web and desktop applications. Our measurements on a prototype implementation show that the combination of these optimizations can improve program startup latency of web applications by as much as 58% and desktop applications by as much as 53%.

For web applications, the approaches to improving startup time have to carefully balance the competing requirements of downloading as few bytes as possible and accessing the server the least number of times. This balance is especially important for networks with a high latency-bandwidth product.

For desktop applications, code reordering improves application startup latency for all states of the file cache. It improves startup latency moderately (11%) for *cold* caches, slightly (6%) for *hot* caches, and significantly (53%) for *warm* caches.

References

[Banga & Druschel 99] Banga, G. and Druschel, P. Measuring the Capacity of a Web Server Under Realistic Loads. *World Wide Web Journal*, May 1999. to appear.

[Bradford & Crovella 98] Bradford, P. and Crovella, M. Generating Representative Web Workloads for Network and Sever Performance Evaluation. In *Proceedings of the 1998 ACM SIGMETRICS Internation Conference on Measurement and Modeling of Computer Systems*, pages 151–160, July 1998.

[Chappell 96] Chappell, D. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[Chen & Leupen 97] Chen, J. and Leupen, B. Improving Instruction Locality with Just-in-Time Code Layout. In *Proc. of the USENIX Windows NT Workshop*, pages 25–32, 1997.

[Enst et al. 97] Enst, J., Evans, W., Fraser, C., Lucco, S., and T.Proebsting. Code Compression. In *Proc. ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 358–365, 1997.

[Franz & Kistler 97] Franz, M. and Kistler, T. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Krintz et al. 98] Krintz, C., Calder, B., Lee, H., and Zorn, B. Overlapping Execution with Transfer Using Non-strict Execution for Mobile Programs. In *Proc. 8th Int. Conf on Architectural Support for Programming Languages and Operating Systems*, pages 159–169, 1998.

[Leach & Naik 97] Leach, P. and Naik, D. A Common Internet File System (CIFS/1.0) Protocol, December 1997. Internet Engineering Task Force (IETF) draft document, available from ftp://ietf.org/-internet-drafts/draft-leach-cifs-v1-spec-01.txt.

[Lee et al. 98] Lee, D., Crowley, P., Baer, J.-L., Anderson, T., and Bershad, B. Execution Characteristics of Desktop Applications on Windows NT. In *Proc. 25th Annual International Symposium on Computer Architecture*, pages 27–38, June 1998.

[Levy & Redell 82] Levy, H. M. and Redell, D. D. Virtual Memory Management in VAX/VMS. *Computer*, 15(3):35–41, March 1982.

[Melanson 98] Melanson, E. Tuning up. *PC Magazine*, August 1998.

[Microsoft 98] Microsoft. Microsoft Developer Network Library, April 1998. on CD-ROM.

[Nielsen et al. 97] Nielsen, H., Gettys, J., Baird-Smith, A., Prudhommeau, E., Lie, H., and Lilley, C. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 155–166, September 1997.

[Peterson & Davie 96] Peterson, L. L. and Davie, B. S. *Computer Networks, A Systems Approach*, chapter 6. Morgan Kaufmann Publishers, Inc., 1996.

[Pettis & Hansen 90] Pettis, K. and Hansen, R. Profile Guided Code Positioning. In *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–26, 1990.

[Rizzo 99] Rizzo, L. Dummynet: A Simple Approach to the Evaluation of Network Protocols, February 1999. available from http://www.iet.unipi.it/ luigi/ip_dummynet.

[Romer et al. 97] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., and Bershad, B. Instrumentation and Optimization of Win32/Intel Executables using Etch. In *Proc. of the USENIX Windows NT Workshop*, pages 1–7, 1997.

A Web Application Download Times

In this appendix, we show the startup latency from all our experiments with web applications.

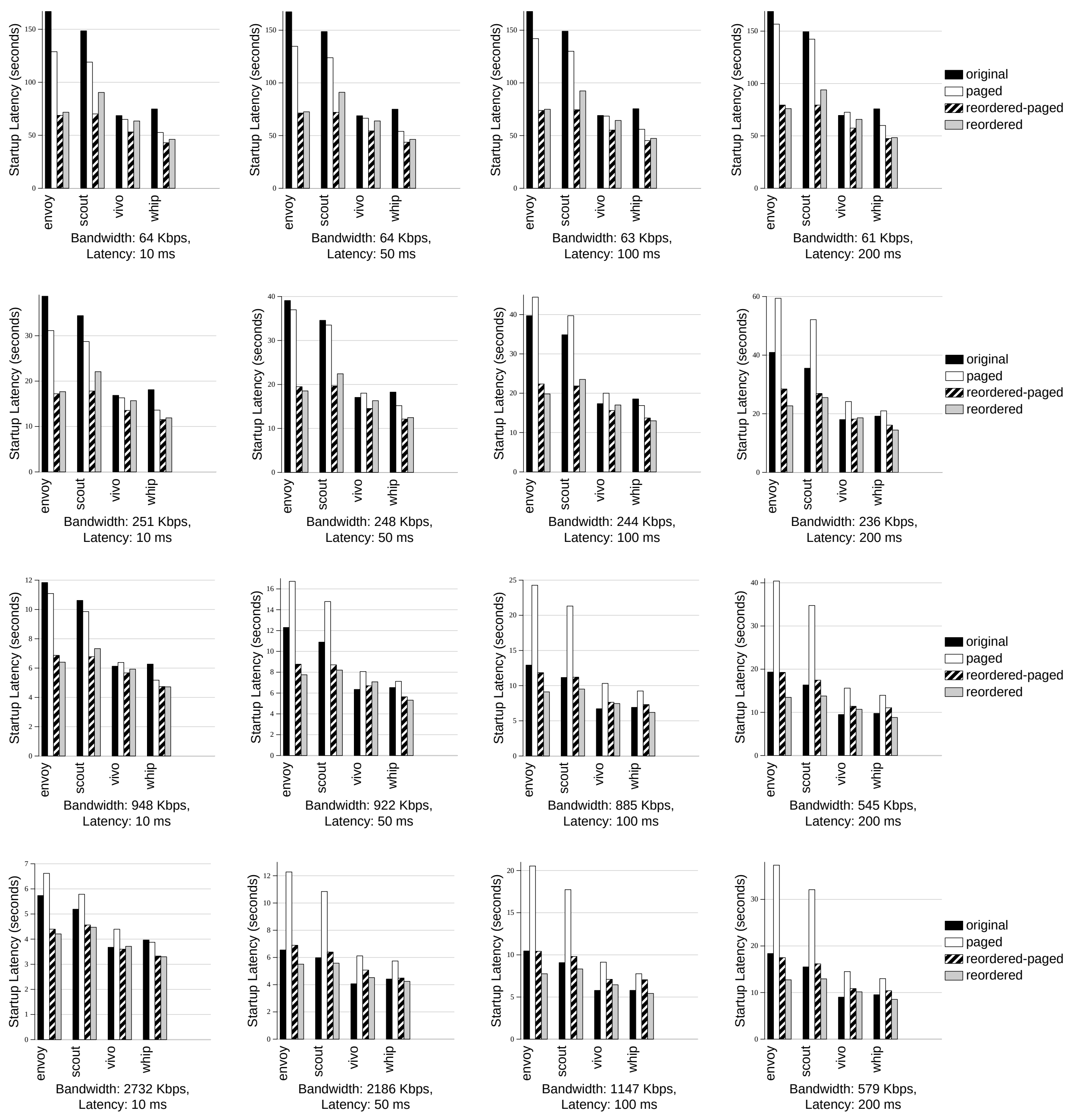


Figure 6: *Web Application Results. Graph shows the startup latency of web applications under various network conditions.*