Improving Application Performance Through Space Optimizations

Michael Carpenter
Doumit Ishak

Project Sponsor:
Will Cohen, Red Hat Inc.

Table of Contents

Project Description

Commonly used benchmarks such as SPEC2000 are often dominated by small loops which can fit entirely into the processors cache. Typical applications, such as Mozilla, are very large and have very different characteristics. As the gap between memory and processor speeds continues to widen, the cost of cache misses on application performance becomes greater. Optimizing an application for space can help to reduce the number of cache misses as well as the amount of paging to disc.

The purpose of this project is to quantify the effects of space optimizations on the Mozilla web browser in Fedora Core Linux. The study will seek to measure the performance changes by looking at such characteristics as resulting code size, cache and memory performance, and application startup time. The specific goals for this project are as follows:

- Develop test procedures for gauging performance
- Provide concrete evidence on whether performance under optimizations is better or worse
- Show how to incorporate space optimizations into RPM
- Determine root causes of performance change
- Produce guidelines on when space optimizations should be used in packages
- Incorporate the results of this project into making Linux applications faster

The Linux distribution chosen to serve as the test environment for this project is Fedora Core 3 test 1. This will allow for a consistent testing environment which could easily be replicated for further study. In addition, this distribution contains all of the tools necessary to measure performance. The Fedora Project is a Red-Hat-sponsored and community-supported open source project. More information on the Fedora Project can be obtained from the project website (http://fedora.redhat.com).

Performance Metrics

In order to quantify the effects of space optimization on application performance a set of metrics is needed. The metrics used in this study can be broken down into two categories: static and dynamic. Static metrics are those which measure characteristics of the space optimized code which do not change during the execution of the code, such as code size. In contrast, dynamic metrics are those which may vary during execution of the code or across multiple executions of the code. These include such characteristics as time of execution and resident memory size.

The static metrics used are RPM size, executable size, and function size. All of these static metrics aim at measuring how effective the compiler is at reducing code size. The obvious benefit of code size reduction is that the code is more likely to fit within the processors cache and on the same page in memory, thereby reducing the number of cache misses and page faults. Another added benefit of smaller code size is that the package becomes easier to distribute. In a large distribution such as Fedora Core, having smaller RPMs will allow users to download the distribution in a smaller amount of time and could possibly reduce the number of CDs needed to install the operating system.

Measurement of the executable size and RPM size will be done with the Linux 'ls' command. The function sizes will be measured using the Linux 'nm' command. Included with this report is a script used to automate the process of gathering function size information. Additional information on using this script can be found in "Appendix A: Performance Measurement Tools."

In order to verify that the smaller code size will have the predicted effect on the runtime performance, a number of dynamic metrics will be studied including instruction cache misses, instruction translational look-aside buffer (ITLB) misses, page faults, resident memory size, and application startup time. In order to verify that the change in performance is applicable to both a trace cache and a traditional instruction cache the measurements will be taken on an Intel Pentium III and an Intel Pentium 4 processor. The specs for the two systems are summarized in Table 3.

Profiling the caches and the ITLB will be accomplished with the use of OProfile. Due to limitations of the profiler to accurately measure the instruction misses in the trace cache of the Pentium 4, the L2 unified cache references are used instead. The assumption being made is that the data references for the space optimized version and the non-space optimized version will remain consistent across both runs. Therefore, the difference in L2 cache references will be due to misses in the trace cache. Also the event "`TC_DELIVER_MODE`" is used to measure how much time the trace cache spends in deliver mode. These two events will provide a reasonable estimation as to the performance of the trace cache. The Pentium III, L1 instruction cache misses will be measured with the "`IFU_IFETCH_MISS`" event. The events used for profiling the ITLB and time based measurements for both processor types are summarized in Table 1.

	Pentium III			
	Event	**Count**	**Unit-Mask**	**Mask Description**
Time-Based	`CPU_CLK_UNHALTED`	200000		
ITLB Misses	`ITLB_MISS`	1000		
L1 Instruction Cache Misses	`IFU_IFETCH_MISS`	10000		
	Pentium 4			
	Event	**Count**	**Unit-Mask**	**Mask Description**
Time-Based	`GLOBAL_POWER_EVENTS`	200000	0x1	Mandatory
ITLB Misses	`ITLB_REFERENCE`	3000	0x2	Misses to the ITLB
L2 Cache	`BSQ_CACHE_REFERENCE`	10000	0x107	All reads to second level

References				cache
Trace Cache Deliver Mode	TC_DELIVER_MODE	10000	0x4	Processor is in deliver mode

Table 1: OProfile events used to measure dynamic performance metrics.

Measurement of application startup time is done with the use of a script provided with this report. This script uses a timeline enabled version of Mozilla (also included in the accompanying software) to compute an average startup time for the browser. For more information on the usage of the script refer to “Appendix A: Performance Measurement Tools.”

The benchmark used in performing the tests is described below in Table 2. This benchmark is a subset of the Mozilla Quality Assurance SmokeTest (http://www.mozilla.org/quality/smoketests/) and tests a majority of the functionality of the browser including features such as page loading/rendering, ssl security, JavaScript handling, and file system interaction.

Step	**Description**	**Action**
1	Launch the browser using the default profile	Either click on the icon on the menu bar or launch from the terminal window
2	Load a local file	File->Open File… Select a local file to load (e.g. /usr/share/doc/HTML/index.html)
3	Load mozilla.org via the “throbber” icon	Click on the “throbber” icon located in the upper right hand corner of the browser window
4	Load a remote site from the file menu	File-> Open Location… Enter a web address (e.g. http://www.yahoo.com)
5	Load a remote site via a hyperlink	Click on a link in the page opened in the previous step
6	Load a remote https site via the address bar	Enter https://sourceforge.net into the address bar and press enter
7	Generate a domain name mismatch warning	Enter https://sf.net into the address bar and press enter. Hit enter when the domain name mismatch dialog box comes up.
8	Open a page with embedded JavaScript	Enter the address of a page which uses JavaScript (e.g. http://java.sun.com) into the address bar and press enter
9	Test one of the menu dialogs	Edit -> Preferences -> History -> Clear History -> Ok
10	Exit	File -> Exit

Table 2: Benchmark used for performance testing on Mozilla.

Experiment Procedures

Testing of the dynamic metrics was first done on the non-space optimized packages. The tests were run three times with a reboot in between each run. The results shown for the instruction cache misses, instruction TLB misses, page faults, and L2 cache reads are an average of the three runs.

The packages were then optimized for space and re-installed on the machines. Once the optimized packages were installed a pre-linking was done so as to ensure that the tests were run under the same conditions as the non-optimized tests were. The tests were then run again using the procedure mentioned above. More information on the testing procedures can be found in Appendix A: Performance Measurement Tools.

Testing Environment

This study was conducted under Fedora Core 3 test1. The following RPMs were rebuilt for space optimizations:

- freetype-2.1.7-5.i386.rpm
- glib2-2.4.2-1.i386.rpm
- glibc-2.3.3-36.i386.rpm
- gtk2-2.4.1-3.i386.rpm
- mozilla-1.7-0.3.2.i386.rpm
- mozilla-nspr-1.7-0.3.2.i386.rpm

In addition, gcc-3.4.1-2 was used to rebuild the RPMs for space optimization. As mentioned earlier, the tests were conducted on both the Pentium III and Pentium 4 architectures. The system specs for both of these machines are described in Table 3.

Hostname	`perftest1`	`perftest2`
CPU Type	`Intel Pentium 4`	`Intel Pentium III (Katmai)`
CPU Speed	`1700MHz`	`500 Mhz`
Total Memory	`256 MB`	`256 MB`
L1 Instruction Cache/Trace Cache Configuration	`12K uOps, 4-way associative`	`16 KB, 4-way associative, 32 byte line size`
L2 Unified Cache Configuration	`256 KB, Sectored, 8-way associative, 64 byte line size`	`512 KB, 4-way associative, 32 byte line size`
Instruction TLB Configuration	`4KB, 2MB or 4MB pages, fully associative, 64 entries`	`4KB pages, 4-way assocative, 32 entries; 4MB pages, fully associative, 2 entries`

Table 3: Specs for the two testing machines.

Incorporating Space Optimization into RPM

Space optimizations will be accomplished with the use of the GNU Compiler Collection (GCC) '-Os' optimization flag. This flag turns on optimizations in the compiler which do not typically increase code size as well as some which are designed to reduce code size. More information on the '-Os' flag can be obtained from the GCC online documentation (http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html). Table 4 summarizes the optimizations used by GCC for different levels of optimization. Items designated with a '+' indicate that the optimization is turned on, and items designated with a '-' indicate that the optimization is turned off. In order to incorporate this change into the RPMs used in Fedora Core Linux the RPMs must be rebuilt from source. For a detailed guide on building RPMs from source refer to http://www.rpm.org/max-rpm/p5208.html.

Optimization	**-O2**	**-O3**	**-Os**
-fno-defer-pop	-	-	-
-fforce-mem	+	+	+
-fomit-frame-pointer	+	+	+
-foptimize-sibling-calls	+	+	+
-finline-functions	-	+	-
-fmerge-constants	+	+	+
-fstrength-reduce	+	+	+
-fthread-jumps	+	+	+
-fcse-follow-jumps	+	+	+
-fcse-skip-blocks	+	+	+
-frerun-cse-after-loop	+	+	+
-frerun-loop-opt	+	+	+
-fgcse	+	+	+
-floop-optimize	+	+	+
-fcrossjumping	+	+	+
-fif-conversion	+	+	+
-fif-conversion2	+	+	+
-fdelete-null-pointer-checks	+	+	+
-fexpensive-optimizations	+	+	+
-fregmove	+	+	+
-fdelayed-branch	+	+	+
-fschedule-insns	+	+	+
-fschedule-insns2	+	+	+
-fcaller-saves	+	+	+
-fpeephole	+	+	+
-fpeephole2	+	+	+
-fguess-branch-probability	+	+	+
-freorder-blocks	+	+	-
-freorder-functions	+	+	+

-fstrict-aliasing	+	+	+
-falign-functions	+	+	-
-falign-labels	+	+	-
-falign-loops	+	+	-
-falign-jumps	+	+	-
-fweb	-	+	-
-fno-cprop-registers	-	-	-
-fprefetch-loop-arrays	+	+	-

Table 4: Optimizations used by GCC.

Before rebuilding the RPMs it is important to know the shared libraries used by the application. This can easily be done with the use of OProfile’s time based measurements. The commands below were used to generate the time-based profile of Mozilla, shown in Figure 1, for the Pentium III test machine. The output shows the percentage of time spent in the specified libraries by the application.

```
# opcontrol –-init
# opcontrol --reset
# opcontrol --setup –-no-vmlinux --separate=library \
     --event=CPU_CLK_UNHALTED:200000::0:1
# opcontrol –start
# mozilla
     ... Perform application benchmark here
# opcontrol --dump
# opcontrol --stop
# opreport --long-filenames \
     image:/usr/lib/mozilla-1.7/mozilla-bin
```

```
53461 100.000 /usr/lib/mozilla-1.7/mozilla-bin
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
            11396 21.3165 /usr/lib/mozilla-1.7/components/libgklayout.so
             7046 13.1797 /usr/lib/mozilla-1.7/libmozjs.so
             4234  7.9198 /lib/libc-2.3.3.so
             3869  7.2371 /usr/lib/mozilla-1.7/mozilla-bin
             3368  6.2999 /lib/libpthread-0.10.so
             3043  5.6920 /usr/lib/mozilla-1.7/libxpcom.so
             2499  4.6744 /usr/lib/libfreetype.so.6.3.5
             2445  4.5734 /usr/lib/libgdk-x11-2.0.so.0.400.1
             1649  3.0845 /usr/X11R6/lib/libXft.so.2.1.2
             1646  3.0789 /usr/lib/libnspr4.so
             1183  2.2128 /usr/lib/mozilla-1.7/components/libxpconnect.so
             1120  2.0950 /usr/lib/libgobject-2.0.so.0.400.2
             1037  1.9397 /usr/lib/mozilla-1.7/components/libhtmlpars.so
             1016  1.9005 /usr/lib/libfontconfig.so.1.0.4
              915  1.7115 /usr/lib/mozilla-1.7/components/libgfx_gtk.so
              914  1.7097 /lib/ld-2.3.3.so
              864  1.6161 /usr/lib/libglib-2.0.so.0.400.2
              722  1.3505 /usr/lib/mozilla-1.7/components/libimglib2.so
              666  1.2458 /usr/X11R6/lib/libX11.so.6.2
              633  1.1840 /usr/lib/mozilla-1.7/libgkgfx.so
              562  1.0512 /usr/lib/mozilla-1.7/components/libnecko.so
              462  0.8642 /usr/lib/mozilla-1.7/components/librdf.so
              316  0.5911 /usr/lib/libsoftokn3.so
              199  0.3722 /usr/lib/mozilla-1.7/components/libcaps.so
              193  0.3610 /usr/lib/libgtk-x11-2.0.so.0.400.1

<------------------------Remaining Output Omitted------------------------>
```

Figure 1: Output from opreport for time based profiling on Mozilla

The packages which own these libraries can then be determined by querying the RPM database (i.e. "`rpm –qf <library filename>`"). A script to automate the profiling has been provided in the software accompanying this report. For information on using the script refer to "Appendix A: Performance Measurement Tools." The packages which contained libraries used by Mozilla, five percent of the execution time or more were mozilla, mozilla-nspr, glibc, glib2, gtk, and freetype. Once the packages used by the application have been determined they can then be optimized for space.

There are two general techniques for rebuilding the RPMs with space optimizations depending on how the build process is specified in the RPM spec file. Some RPMs, such as Mozilla, will use a build variable such as "OPT_FLAGS" to specify optimization flags to send to the compiler. If this is the case the spec file can be opened in a text editor and the build variables can be set manually. However, many RPMs use the "%build" macro provided by RPM. This macro will automatically export any compiler flags to the appropriate build files. These compiler flags can be set in an ".rpmrc" configuration file. A sample ".rpmrc" configuration file is shown below. The appropriate line for a given architecture must be added to the configuration file in order to export the compiler flags during build time. The ".rpmrc" file should be placed in the home directory of the user building the RPM. Additional information on RPM optflags can be found at http://www.rpm.org/max-rpm/s1-rpm-multi-optflags.html.

Sample .rpmrc file:

```
optflags: i386 –Os -march=i386
optflags: i486 -Os -march=i486
optflags: i586 –Os -march=i586
optflags: i686 –Os -march=i686
optflags: athlon –Os -march=athlon
optflags: ia64 -Os
optflags: x86_64 -Os
optflags: amd64 -Os
optflags: ia32e -Os
```

Results for Static Metrics

The resulting RPM size data for the six optimized packages is summarized in Figure 2 and in Table 5. As expected the RPM sizes do not change dramatically as the RPM is composed of not only code, but also documentation and configuration files. In addition, the effect of the smaller executables is minimized since the RPM is compressed. Nevertheless, the RPM size decreased by an average of four percent for the six packages

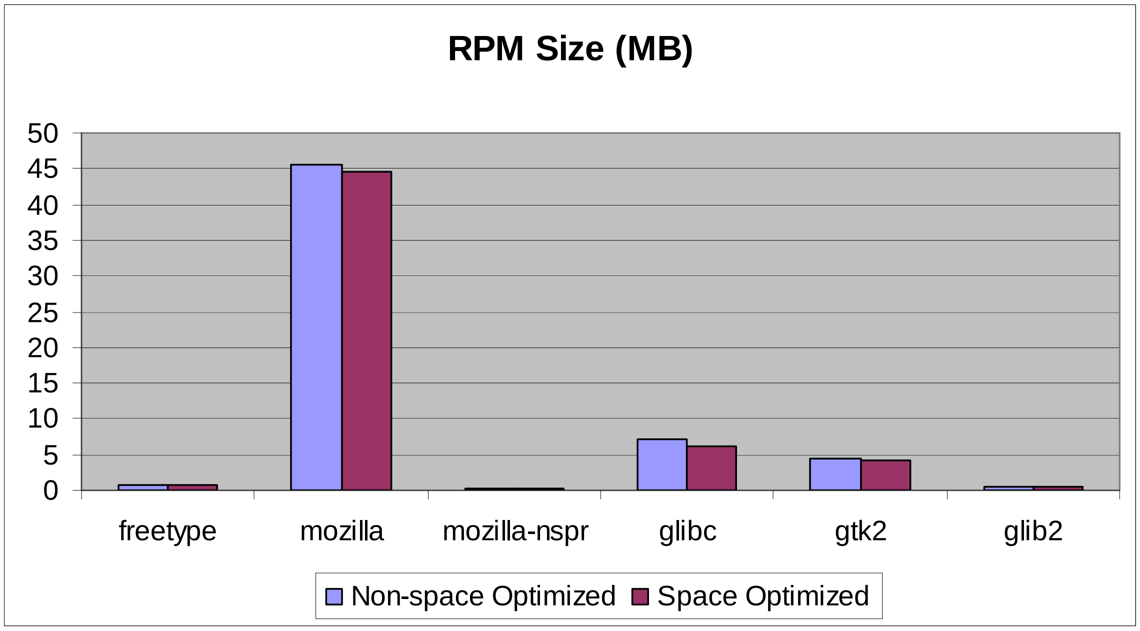

Figure 2: Resulting RPM size decreased by 4% for the six packages.

Package	Non-space Optimized RPM Size (MB)	Space Optimized RPM Size (MB)	Percent Change
freetype	0.715679	0.690044	-3.58188

mozilla	45.66133	44.56362	-2.40403
mozilla-nspr	0.297445	0.270617	-9.01973
glibc	7.05049	6.101048	-13.4663
gtk2	4.423068	4.198192	-5.08417
glib2	0.498142	0.457123	-8.23448

Table 5: Data for the resulting RPM size.

The resulting executable file size data is summarized in Figure 3 and in Table 6. As a result of space optimizations the average size of the executables for these six packages decreased by more than six percent.

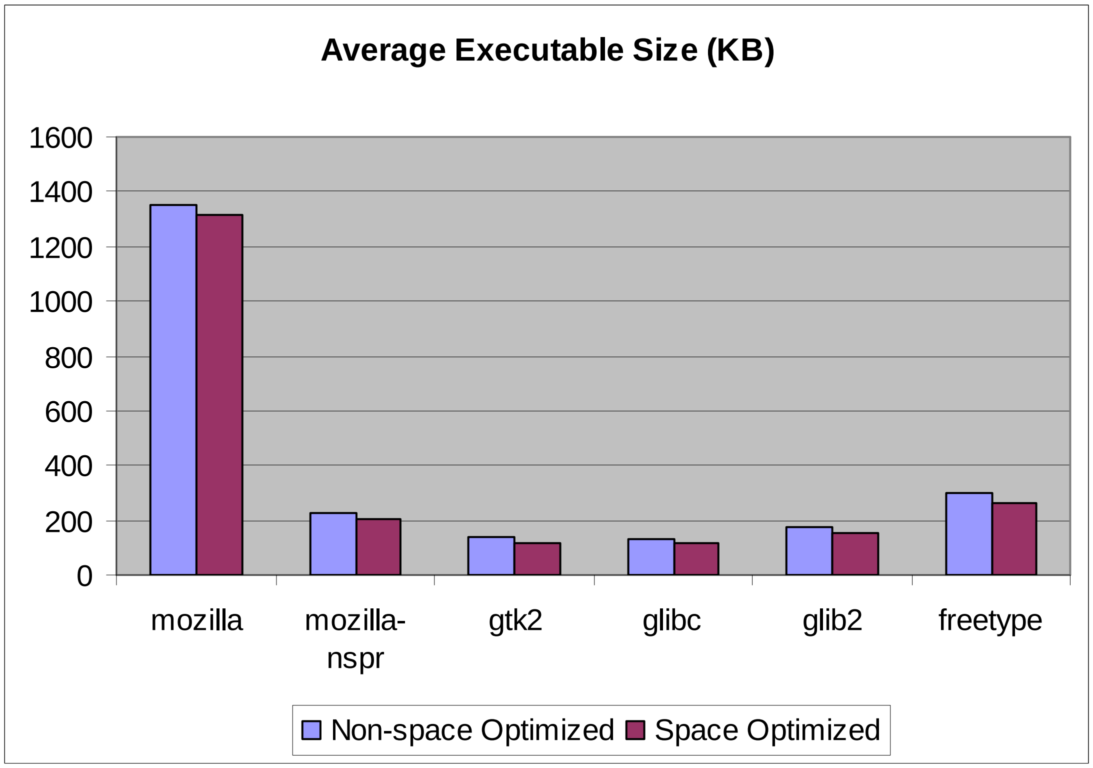

Figure 3: Average executable size decreased by 6.63% for the six packages.

Package	Non-space Optimized Average Executable Size (KB)	Space Optimized Average Executable Size (KB)	Percent Change
mozilla	1348.504	1317.666	-2.28682
mozilla-nspr	225.863	204.7565	-9.3448
gtk2	139.582	114.3771	-18.0574
glibc	132.9298	117.9047	-11.303
glib2	177.3152	151.5402	-14.5363
freetype	301.6177	265.3198	-12.0344

Table 6: Average executable size data.

The resulting function sizes data is summarized by Figure 4 and Table 7. The average function sizes for the six packages decreased by twenty-two percent on average. The reason the executables did not decrease in size proportionately with the function sizes is because the executables contain data and other symbols which can not be optimized for space.

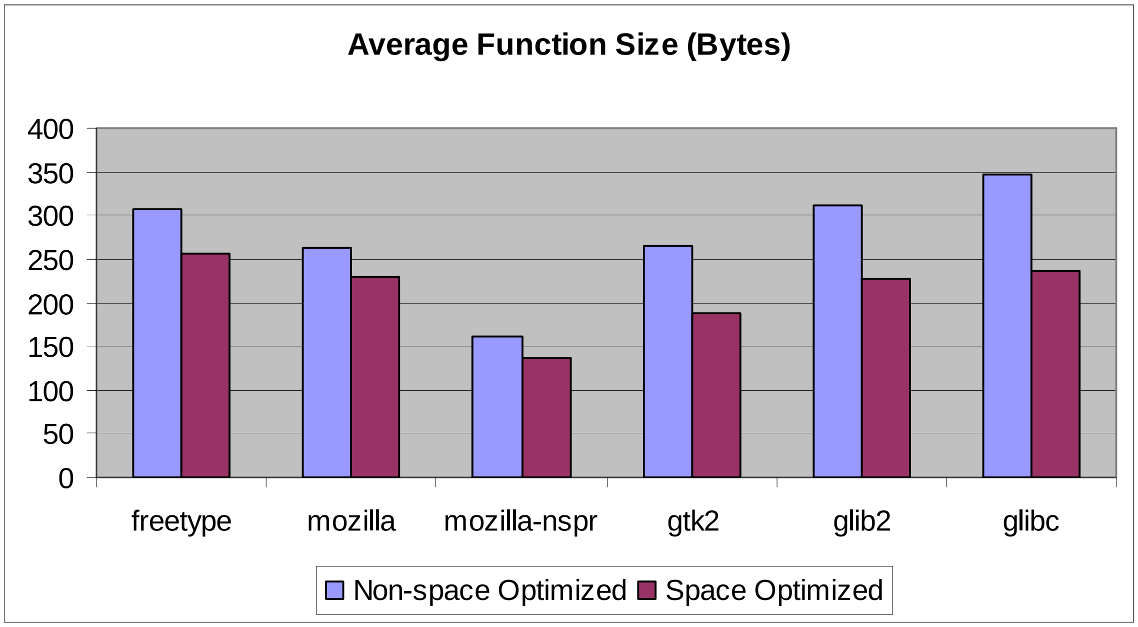

Figure 4: Average function size for the six packages decreased by 22% on average.

Package	**Non-space Optimized Average Function Size (Bytes)**	**Space Optimized Average Function Size (Bytes)**	**Percent Change**
freetype	306.71	256.44	-16.39007532
mozilla	262.57	230.92	-12.05392848
mozilla-nspr	162.32	136.97	-15.61729916
gtk2	265.79	186.85	-29.70013921
glib2	310.88	228.35	-26.54722079
glibc	346.18	235.87	-31.86492576

Table 7: Average function size data.

Figures 7-12 provide a different perspective of the decrease in function size. In these graphs the original function size is plotted on the x-axis and the space optimized function size is plotted on the y-axis. All points which lie above the x=y line indicate a function which has gotten larger due to space optimization and all points below the line indicate a function which has gotten smaller due to space optimization. For most of the packages the larger functions tend to get smaller under space optimization, with the exception of glibc. Notice the density of points below the line in Figure 12. This indicates that even the smaller functions are benefiting from space optimizations.

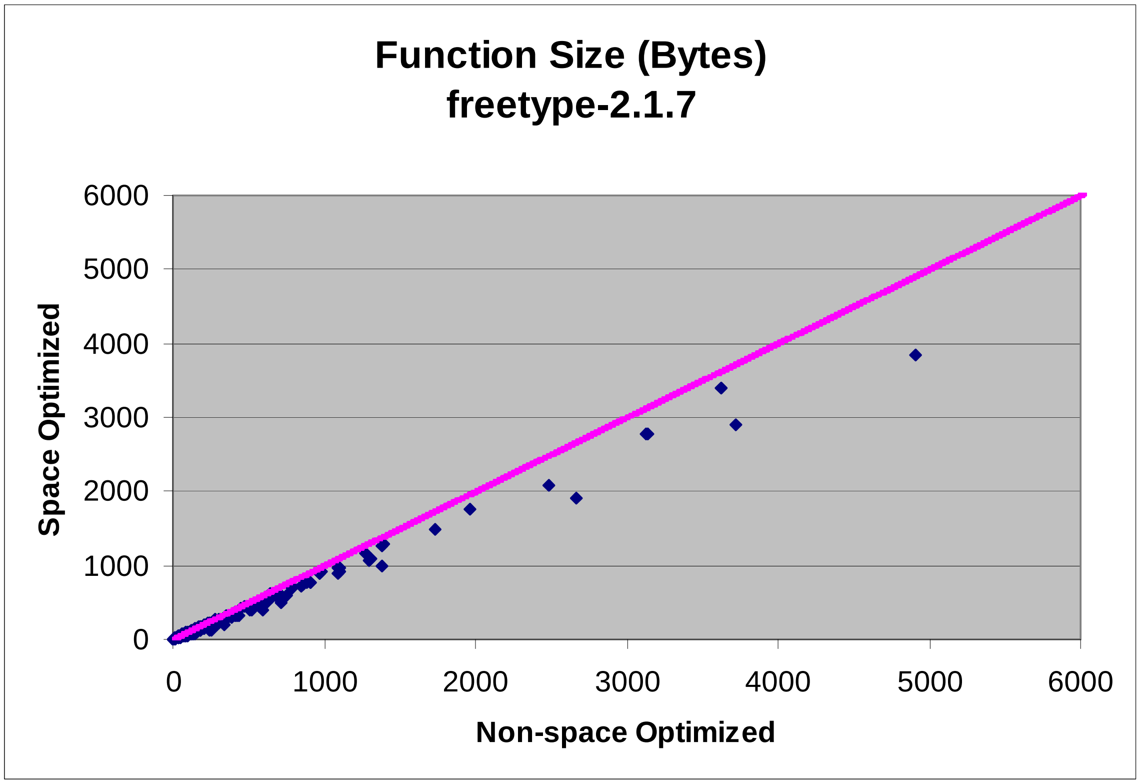

Figure 7: Resulting function sizes for freetype-2.1.7.

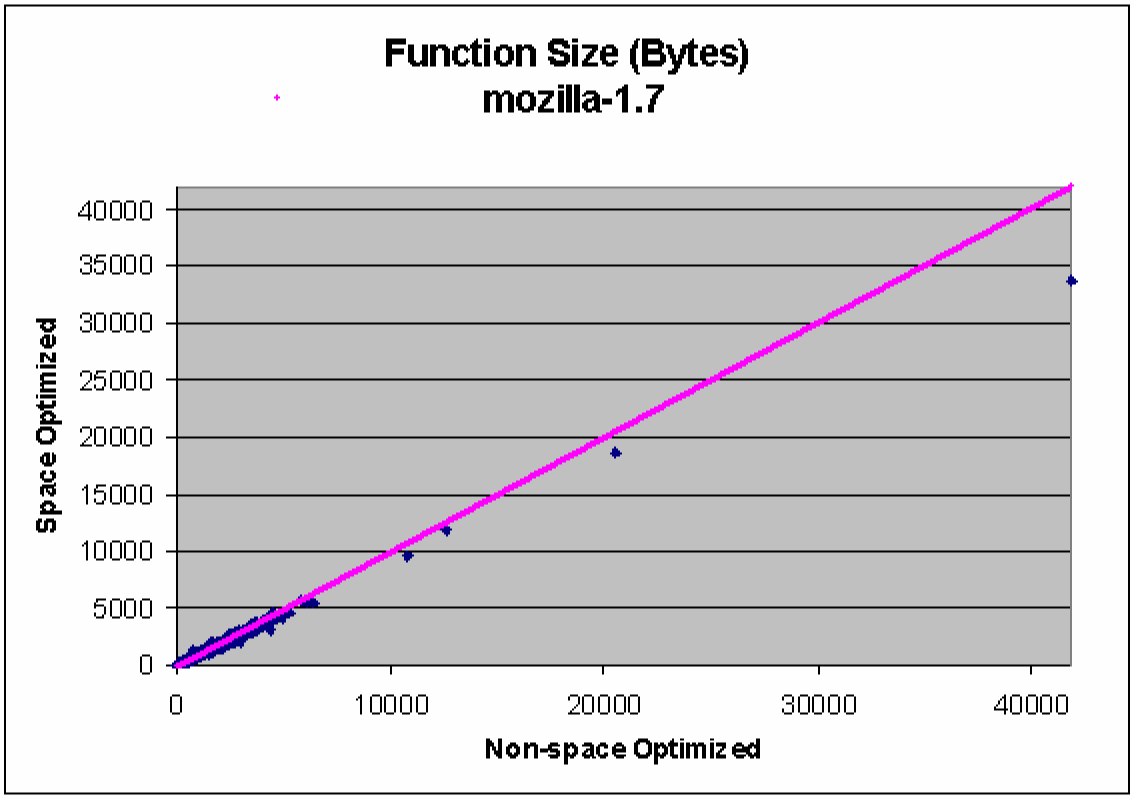

Figure 8: Resulting function sizes for mozilla-1.7.

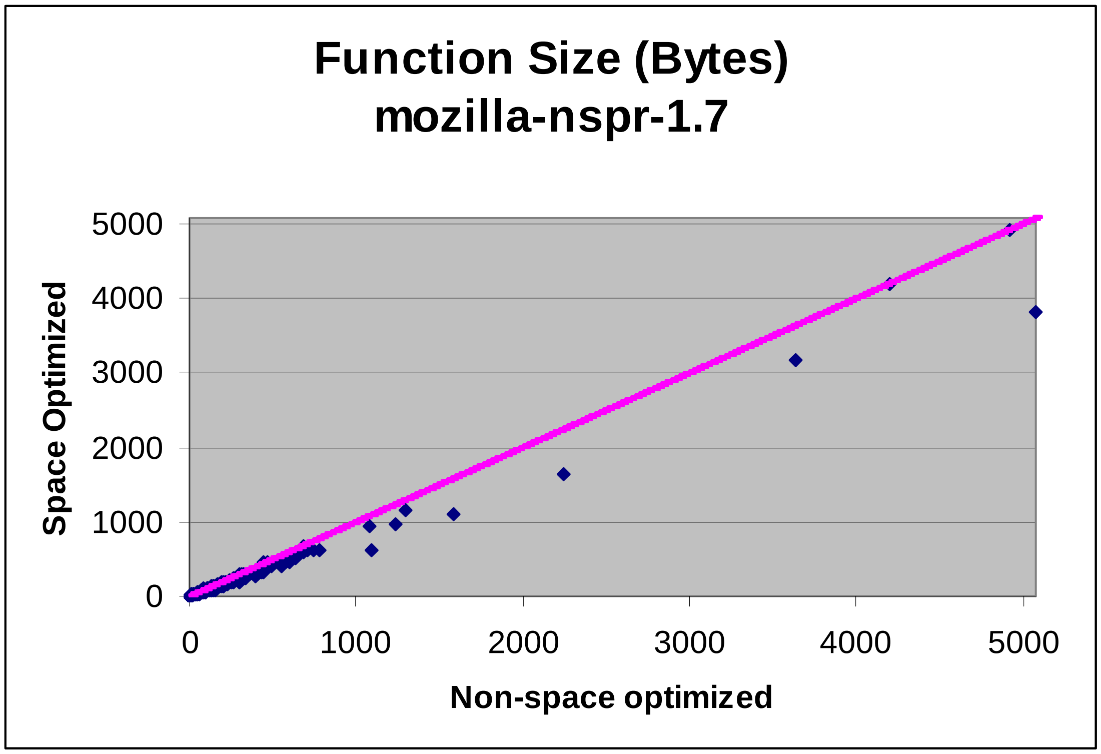

Figure 9: Resulting function sizes for mozilla-nspr-1.7.

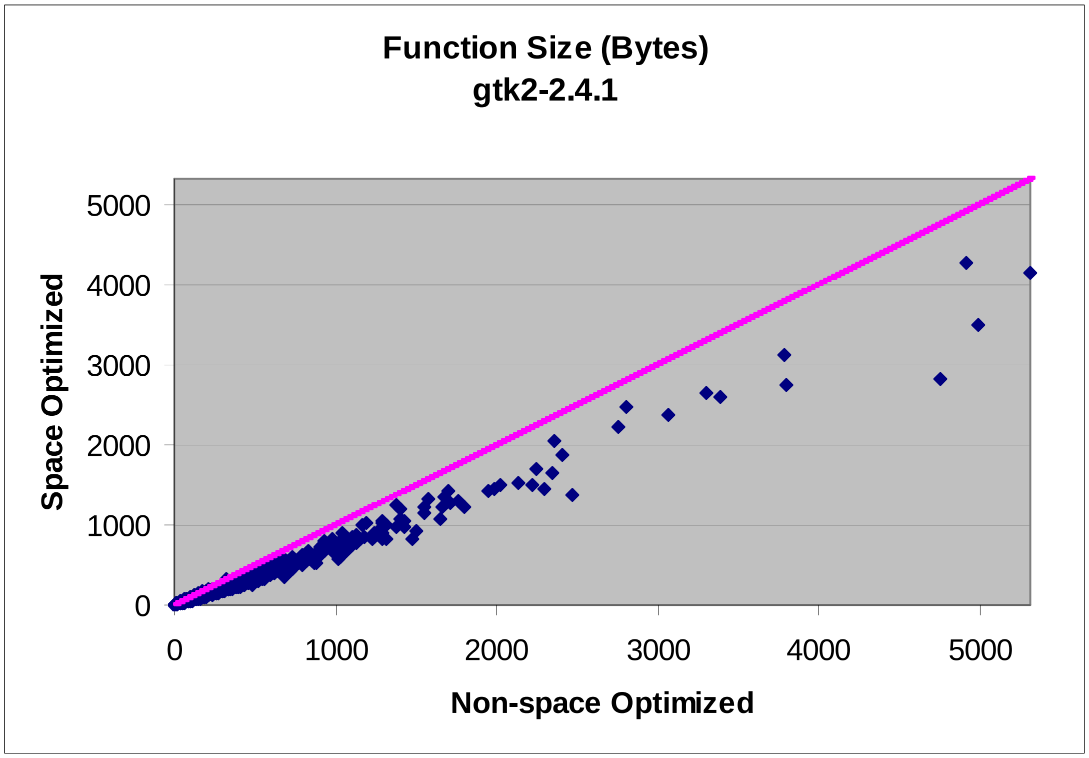

Figure 10: Resulting function sizes for gtk2-2.4.1.

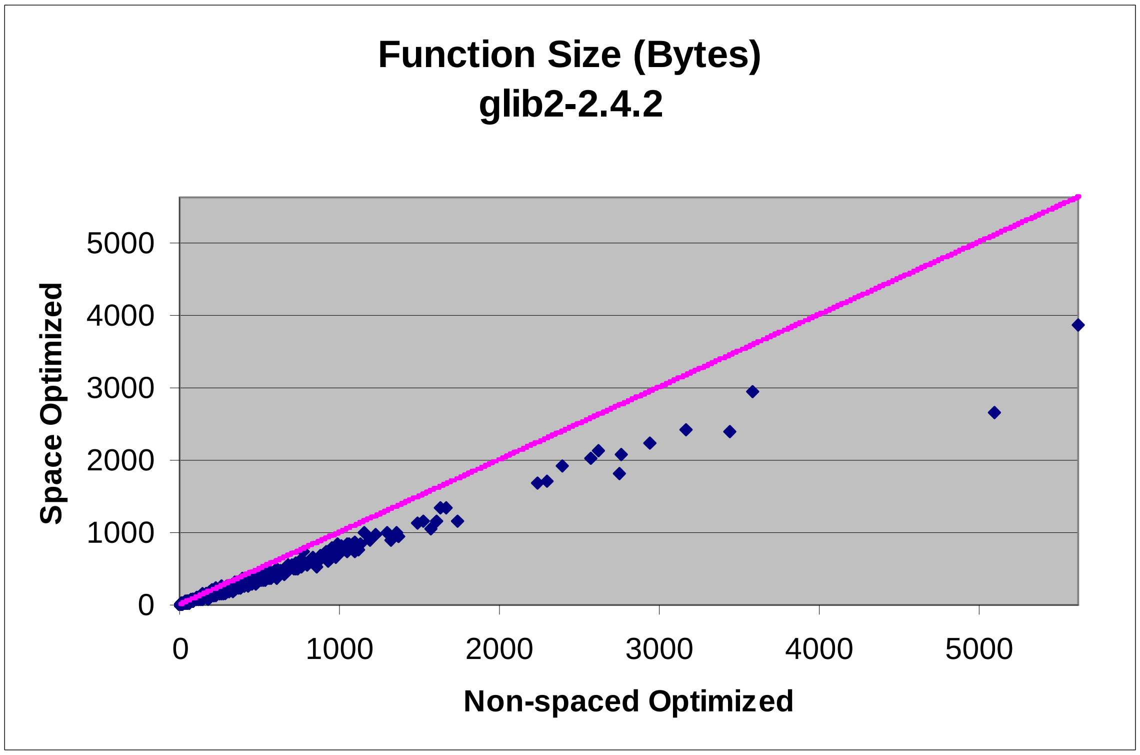

Figure 11: Resulting function sizes for glib2-2.4.2.

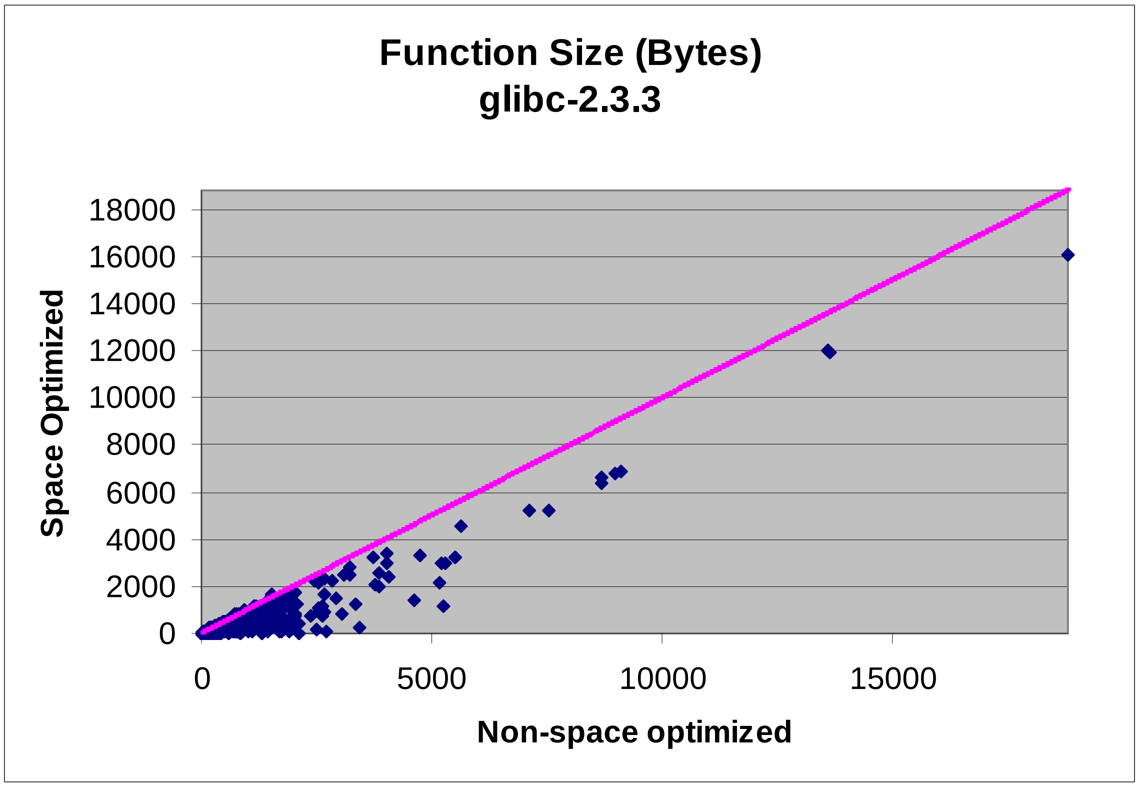

Figure 12: Resulting function sizes for glibc-2.3.3

It is worth noting the results seen in Figure 12. Unlike the other function size plots which tend to follow the x=y line, this plot contains a number of points which move away from this trend. This would indicate that space optimizations were particularly effective, even on small functions. This trend is due to the large number of functions in glibc which have been compiled with the '-O3' option and therefore have functions inlined. Once the function is recompiled with '-Os' the inline function is then converted to a call to the function resulting in a reduction in code size.

The following example illustrates this point. The function shown below is contained within glibc. The first version has been compiled using '-O3' which by default has inline function optimizations turned on. The second was compiled using '-Os' and includes a call to the function 'svcerr_auth'. As a result the space optimized function is considerably smaller.

'svcerr_weakauth' compiled with –O3:

```
000e351c <svcerr_weakauth>:
   e351c:   55                      push   %ebp
   e351d:   89 e5                   mov    %esp,%ebp
   e351f:   83 ec 30                sub    $0x30,%esp
   e3522:   8b 4d 08                mov    0x8(%ebp),%ecx
   e3525:   c7 45 d8 01 00 00 00    movl   $0x1,0xffffffd8(%ebp)
   e352c:   c7 45 dc 01 00 00 00    movl   $0x1,0xffffffdc(%ebp)
   e3533:   c7 45 e0 05 00 00 00    movl   $0x5,0xffffffe0(%ebp)
   e353a:   8d 45 d0                lea    0xffffffd0(%ebp),%eax
   e353d:   8b 51 08                mov    0x8(%ecx),%edx
   e3540:   50                      push   %eax
   e3541:   c7 45 d4 01 00 00 00    movl   $0x1,0xffffffd4(%ebp)
   e3548:   51                      push   %ecx
   e3549:   ff 52 0c                call   *0xc(%edx)
   e354c:   58                      pop    %eax
   e354d:   5a                      pop    %edx
   e354e:   c9                      leave
   e354f:   c3                      ret
```

'svcerr_weakauth' compiled with –Os

```
000ad86d <svcerr_weakauth>:
   ad86d:   55                      push   %ebp
   ad86e:   89 e5                   mov    %esp,%ebp
   ad870:   6a 05                   push   $0x5
   ad872:   ff 75 08                pushl  0x8(%ebp)
   ad875:   e8 c2 ff ff ff          call   ad83c <svcerr_auth>
   ad87a:   c9                      leave
   ad87b:   c3                      ret
```

Results for Dynamic Metrics

Figure 13 summarizes the L1 instruction cache performance for the Pentium III machine. The number of instruction cache misses was reduced by 4.75 percent.

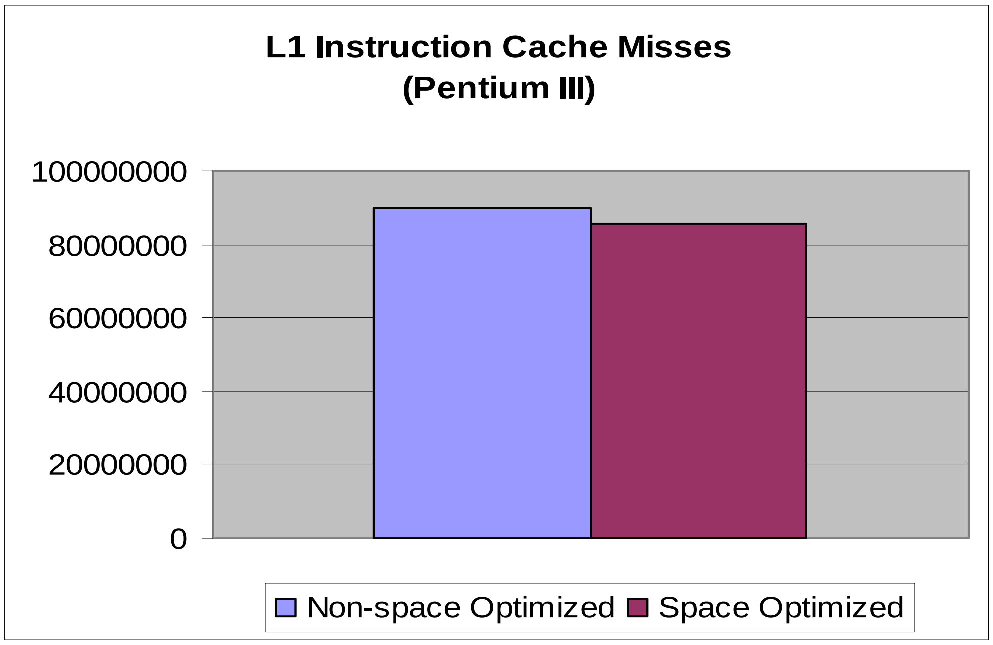

Figure 13: Level 1 instruction cache misses decreased by 4.75% on the Pentium III.

Additionally, the number of ITLB misses on the same machine was decreased by 7.64 percent due to space optimizations. This data is depicted in Figure 14.

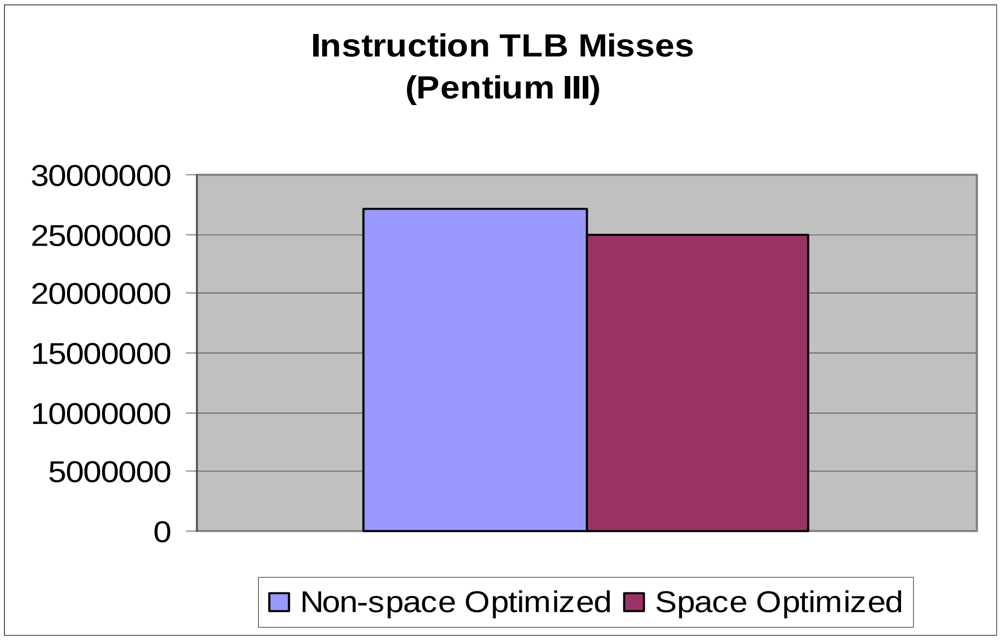

Figure 14: ITLB misses decreased by 7.64% due to space optimizations.

Similar results were obtained from the Pentium 4 cache architecture. The number of reads to the Pentium 4's unified L2 cache decreased by 4.3 percent as shown in Figure 15. Additionally the trace cache spent nearly three percent more time in deliver mode as summarized in Figure 16.

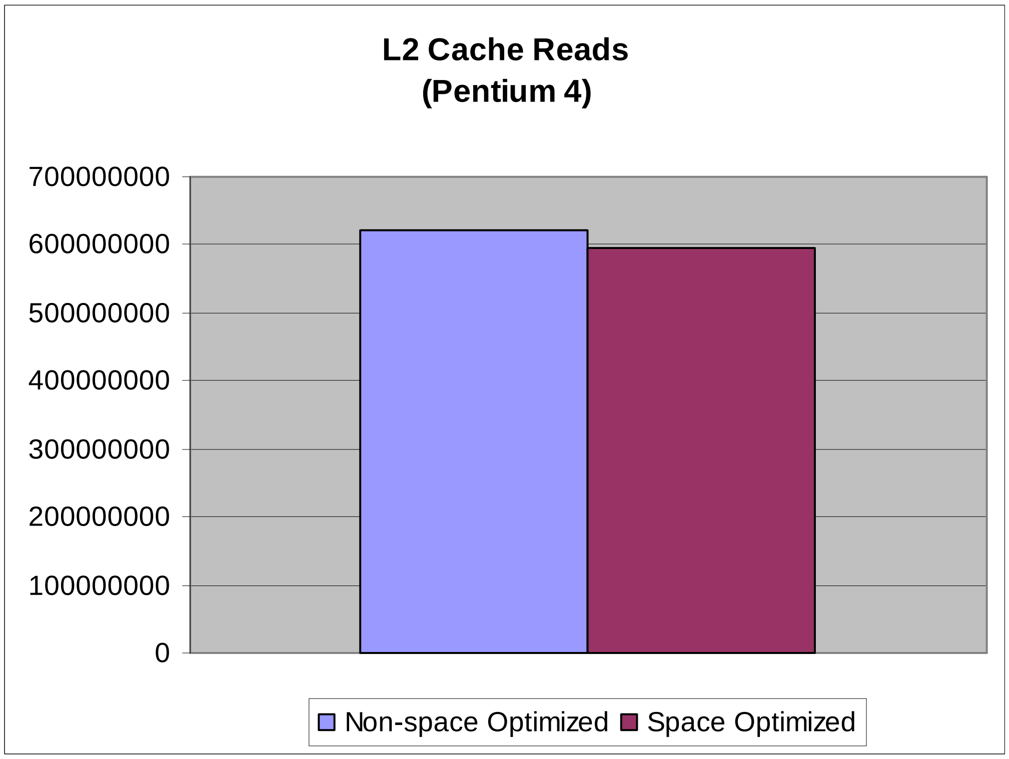

Figure 15: The total number of reads (data + instruction) to the Pentium 4's L2 cache decreased by 4.3%.

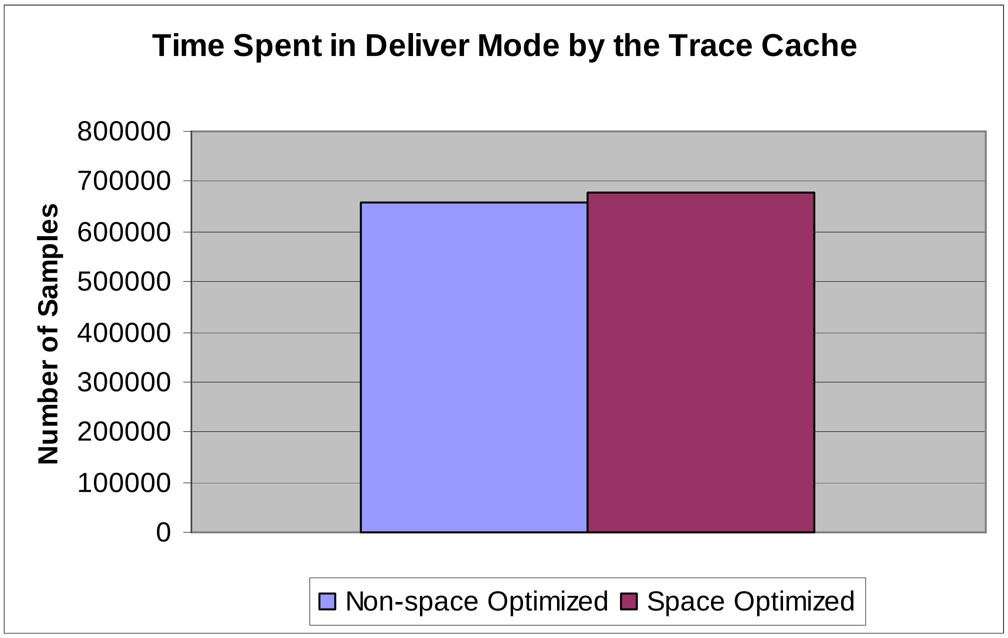

Figure 16: The amount of time the trace cache spent in deliver mode increased by 2.85%.

Figure 17 summarizes the number of page faults for the non-space optimized version of Mozilla and the space optimized version. The number of major page faults decreased by 4.2 percent.

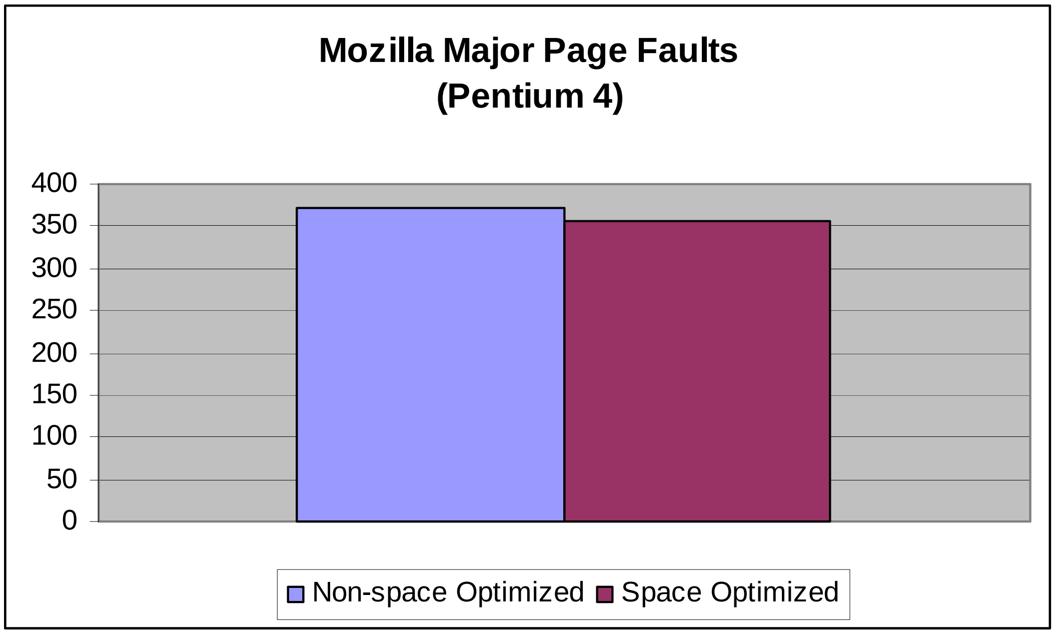

Figure 17: Major page faults were reduced by 4.2% due to space optimization.

Average startup time, as measured using the Mozilla timeline feature, decreased by 2.68 percent as seen in Figure 18. More details on using the Mozilla timeline feature can be found in Appendix A: Performance Measurement Tools.

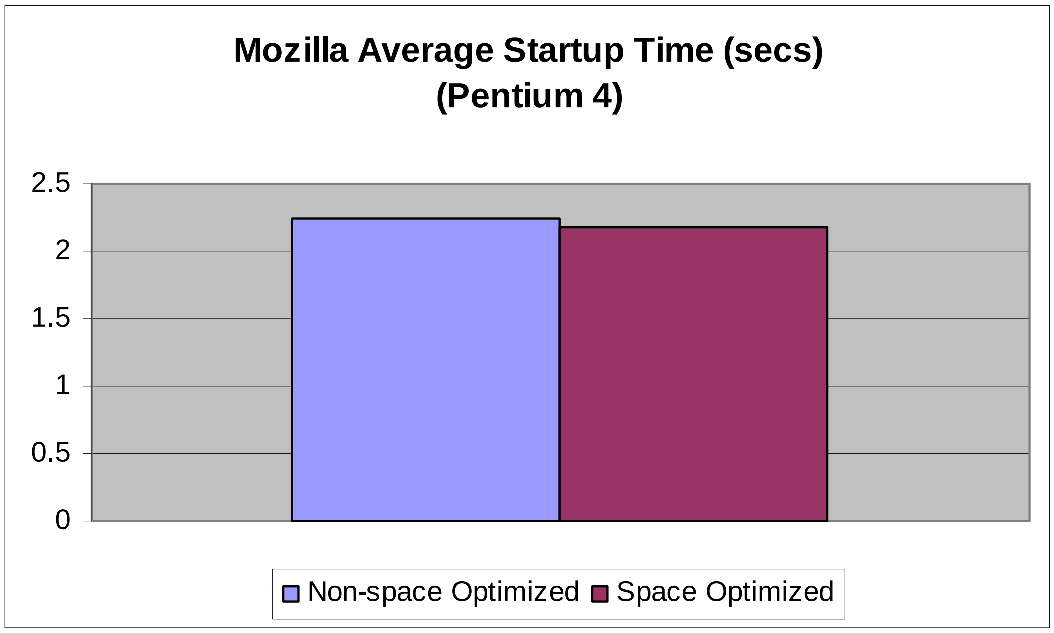

Figure 18: Average startup time for Mozilla decreased by 2.68%.

Conclusions

Overall space optimizations proved to be very effective in improving application performance. The optimizing algorithms were able to reduce code size by significant amounts, thus leading to smaller functions and the predicted improvement in cache and memory performance. These improvements proved to be applicable to both the Pentium III and Pentium 4 architectures. Ultimately these improvements translated into faster startup times for Mozilla. It is our hope that these encouraging results can be applied to making other Linux applications faster.

In addition to the performance gain due to space optimizations, this smaller code can be used to make distributing large Linux distributions, such as Fedora Core, easier. Smaller RPMs require less media and less network bandwidth in getting the product to the customers.

Future Research

This study has shown that gcc’s space optimizing algorithms can be very effective in reducing code size. Likewise, this reduction in code size can lead to increased cache, memory, and TLB performance. However, more testing is needed to determine how these improved dynamic characteristics will translate into execution time improvements. Some areas for future research include:

- Conducting similar studies on non-GUI applications for which automated benchmarks can be established. This will allow for more precise execution time measurements which can’t be taken for GUI driven applications due to variance in user input speeds.
- Establishing upper bounds for the amount of performance improvement that could be expected with space optimizations. This includes quantifying the amount execution time spent handling cache misses, ITLB misses, and page faults.
- Establishing a set of characteristics of source code which lead to effective space optimizations. It would be useful to understand why some of the functions actually got larger due to space optimizations (see Appendix B), while others experienced only moderate reductions in size.
- Using the characteristics mentioned above, generate a set of guidelines on when to use space optimizations. These guidelines could be used by application developers in developing a coding style which could make better use of space optimizations.

Appendix A: Performance Measurement Tools

A number of different tools were used in this study to measure the performance of the space optimized RPMs. This appendix will explain how these tools were used so that future research can be conducted in a similar manner.

The main tool used in this study was OProfile. OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. It makes use of hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics. More information about OProfile as well as source code and tutorials can be found at the project website (http://oprofile.sourceforge.net/news/).

OProfile was used in this study to profile the caches, instruction TLBs, and for time based sampling. The kernel used for this study was 'kernel-smp-2.6.5-1.358.' The drivers for OProfile are enabled only in the SMP kernels; therefore an SMP kernel must be installed in order to utilize oprofile.

OProfile was initially used to gather information about the shared libraries for which Mozilla used, as described in the section "Incorporating Space Optimizations into RPM." Once this was done the cache and ITLB profiling was performed on the non-space optimized packages using the events listed in Table 2. The tests were run three times, with a reboot in between each run, and the numbers shown in the tables are the average of these three runs. The packages were then re-built for space optimization and the tests were performed once again in the same manner. However, before running the tests again on the optimized packages, the system prelinker, which normally runs as a cron job overnight, was run manually. This ensured that the newly installed libraries are prelinked just as the old libraries were. Prelinking should be done before testing anytime new packages have been installed.

The function sizes were obtained using /usr/bin/nm. In order to automate this process, a script called fsizes.pl was written. This script will allow the user to specify a package name and will print out all of the functions and their sizes for the given package. This script has been included with the accompanying software. Usage for the script can be obtained by executing it with no command line arguments.

Executable sizes were also gathered with the use of a perl script. This script, called execsize.pl, accepts the name of a package and then queries the RPM database for a list of all the files associated with the package. It then goes through and sees which of these files are executable and prints the file name as well the file size for those which are executable. Usage for this script can be obtained by executing the script with no command line arguments.

Average resident memory size was obtained using a script called memprof.pl. This script accepts a single command line argument, the command to profile, and will return the maximum resident memory size and the average resident memory size in 4KB pages.

Application startup time was measured with the use of the Mozilla timeline feature; a feature which was re-built into the RPMs. RPMs which have the timeline feature enabled are included with the accompanying software. A timeline enabled RPM has been built as both non-space optimized and space optimized. These RPMs are designated by a '-te' in the filename (i.e. mozilla-1.7-0.3.2-te.i386.rpm). To measure the startup time a script called measure-simple.pl is included. This script will export the necessary environment variables to enable the timeline feature, startup the browser using a file called quit.html, and parse the resulting timeline file for a startup time measurement. A sample call to the script is shown below. It requires two parameters, the first being the path to the Mozilla shell script and the second being the number of times to measure the startup (the first run is omitted and the subsequent runs are averaged together). More information on the timeline feature can be obtained at http://www.mozilla.org/performance/measureStartup.html.

Sample call to startup measurement script:

```
./measure-simple.pl mozilla 5
```

Page fault information was gathered with the use of /usr/bin/time. The page faults shown in the report refer to major page faults specified by the '%F' format string. A major page fault is defined as one in which the page has to be brought in from disk. A sample call to /usr/bin/time is shown below. This command will start Mozilla and print the number of major page faults once the browser window has been closed.

Sample call to /usr/bin/time to measure major page faults:

```
# /usr/bin/time --format=”%F” mozilla
```

Appendix B: Function Size Data

Table 8 summarizes the functions for which space optimization was particularly effective in reducing code size.

Function	Package	Non-space Optimized Size (Bytes)	Space Optimized Size (Bytes)	Percent Change
syslog	glibc	2099	20	- 99.05
rexec	glibc	1307	30	- 97.70
__libc_system	glibc	2728	96	- 96.48
_IO_flush_all	glibc	336	12	- 96.42
__argp_fmtstream_printf	glibc	1892	76	- 95.98
_mcleanup	glibc	1458	60	- 95.88
PL_DHashClearEntryStub	mozilla	48	21	- 56.25
JS_DHashClearEntryStub	mozilla	48	21	- 56.25
JS_Finish	mozilla	408	192	- 52.94
g_static_rec_mutex_init	glib2	194	94	- 51.54
g_value_transforms_init	glib2	5092	2656	- 47.84
g_array_set_size	glib2	173	91	- 47.40

Table 8: A sampling of functions which became significantly smaller due to space optimizations.

Table 9 summarizes the functions for which space optimization was not particularly effective in reducing code size.

Function	Package	Non-space Optimized Size (Bytes)	Space Optimized Size (Bytes)	Percent Change
atof	glibc	20	31	55.00
atoll	glibc	22	33	50.00
atoll	glibc	22	33	50.00
__mpn_construct_float	glibc	48	66	37.50
getenv	glibc	176	220	25.00
CSSParserImpl::ParseProperty	mozilla	815	1286	57.79
nsBoxSize::Add	mozilla	68	102	50.00
nsPseudoFrames::operator	mozilla	107	151	41.12
TT_LookUp_Table	freetype	68	73	7.35
_g_locale_charset_unalias	glib2	141	155	9.93