Multiple Architecture Characterization of the Linux Build Process with OProfile

W. E. Cohen, *Member, IEEE*

Abstract—**Building software packages is a very common step in software production. The speed and efficiency of producing the software packages can impact the schedule for producing a complete Linux distribution consisting of hundreds of software packages.**

Characterizing the build process is required to determine what performance problems exist and suggest actions to reduce the time required for the package build. There are usually a number of different programs used during the build process and it is often not known precisely what programs are used during the build process. Thus, it is desirable to collect system-wide profile information with a tool that does not require specially compiled executables to obtain an accurate view of where time is spent.

Red Hat Enterprise Linux includes the OProfile system-wide profiler. OProfile collected data on the package build of the Mozilla web browser on the AMD Athlon, Intel Pentium 4, and Intel Itanium 2 processors. The measurements showed that the majority of the time was spent in the C++ to assembly language translator and hot spots were found in the GNU compiler's register allocator and linker for the Itanium 2.

Index Terms—**Linux, OProfile, and performance-characterization**

1. INTRODUCTION

One of the most frequently performed task in software development and production is building the executables for the software packages. In operating systems distributions such as Red Hat Enterprise Linux there are more than a thousand software packages. At times mass rebuilds of all the packages are required. Although some packages can be built in parallel there are dependencies between various packages which order the packages and serialize the build process. Some individual packages, such as the Mozilla browser and Open Office, can take a significant amount of time to build (an hour or more).

The first step in reducing the build times is analyzing the build process to determine where time is spent during the build process and why. Data collection of the build process is complicated by the scripts used to start a number of processes during the build; it is not always clear what programs are executed during the build process. Thus, a system-wide profiler that works on uninstrumented code is required to collect data. OProfile [OPR03] provides system-wide profiling on a variety of Linux platforms and is used to collect this data.

The Mozilla web browser build was selected to demonstrate the OProfile data collection and analysis available on Red Hat Enterprise Linux. Section 2 describes the package build process used to build and to package the executables. Section 3 discusses the OProfile data collection system. Section 4 contains the procedures used for the experiments. The collected data and discussion of the data are in Sections 5, 6, 7, and 8. Future work and conclusions are in Sections 9 and 10, respectively.

2. RPMS AND THE RPM BUILD PROCESS

Red Hat uses Red-Hat Package Manager (RPM) [BAI97]to install software on its Linux distributions. RPM packages allow the software to be installed and removed in a controlled manner. Each RPM package contain a list of dependencies that must be satified for the RPM to be installed, a manifest describing what files the RPM provides, the actual files to be installed, the pre- and post-install scripts, and the uninstall script.

Source RPMs contain the materials required to build a binary RPM. Source RPMs contain a pristine tarball of the source code, a series of patches to apply to the pristine tarball, and a spec file. The spec file describes how to apply the patches, configure the software for the build, and build the binary RPMs. The spec file lists the dependencies for the build and for the resulting binary RPMs. By default the RPM build process occurs in `/usr/src/redhat`, but it is possible to make the RPM build occur in another directory as was done for these experiments.

To save space the executables in the binary RPMs are stripped of debugging information. However, during the spring of 2003 the Red Hat RPM build process was modified to store the debugging information removed from the binaries in a separate RPM package called a debuginfo RPM and a link to the debugging information is placed in the binaries. The debuginfo RPM allows developers to debug the traditional executables in the binary RPMs without changing the executable. Another benefit is the performance monitoring tools can use the debuginfo RPMs to map the profiling data back to the original source code. Thus, developers and system administrators can determine which sections of code consume the most processor resources.

3. OPROFILE

OProfile [OPR03] is a sampling system-wide profiling tool. Support for OProfile has been integrated into the Linux 2.5 kernel for a variety of architectures including Pentium Pro, Pentium II, Pentium III, Pentium 4, Athlon, Duron, Opteron, and PowerPC 64. This support has been backported to the 2.4 Linux kernel for Red Hat Enterprise Linux.

The OProfile support of the kernel consists of a module to collect samples and a system call to map numbers stored in the

OProfile sample buffer back to file names. The OProfile driver provides access to the performance monitoring hardware on processors such at the Intel Pentium 4, the AMD Athlon, and the Intel Itanium 2 processors through a mountable file system, usually mounted as directory `/dev/oprofile`. The directory contains a subdirectory for each performance counter available in the processor. Controlling software sets the machine specific events to monitor, whether to count in user and/or kernel space, and the interval between samples. The kernel reads this information from the file system. When the performance counter counts the interval specified, an interrupt is generated. The program counter value and the performance monitoring counter triggering the sample are recorded in a buffer. The kernel also records context switches and changes between user and kernel space in the buffer when needed. The context change includes a dcookie number that can be mapped back to a file name via the `lookup_dcookie` system call.

The daemon is responsible for extracting the information from the kernel buffer and storing the sample data in user accessible files. The daemon reads the kernel buffer when the buffer is full. If the buffer does not fill after 10 minutes, the daemon reads the buffer anyway. The daemon can also be forced to read the buffer. The daemon processes the sequence of samples. For context changes the daemon uses the dcookie number to find the associated file name. For each executable with samples a matching file is created in `/var/lib/oprofile/sample`. The sample files use hashing and only require space proportional to the number of unique program counter locations in the code with samples. These sample files are readable by all; the performance analysis tools read these files. The sample files can also be saved as a session for later analysis.

The OProfile analysis tools have changed between versions 0.5.4 and 0.6. Red Hat Enterprise Linux uses OProfile 0.5.4. Thus, this discussion will be restricted to the OProfile 0.5.4 tools. The `op_time` program lists the samples for all the executable ordering the number of samples either from fewest samples to most samples or most samples to fewest samples. The `oprofpp` program gives a more detailed analysis of an executable and can either provide sample counts on a function-by-function level or by individual instruction addresses. There is a library API available for accessing the sample files and this has been used to write other tools, e.g. a plugin for the Eclipse Integrated Development Environment (IDE) [ECL03]. OProfile analysis tools make use of binary to source code mapping information from the debuginfo RPMs. This allows the OProfile analysis tools to provide analysis data on the installed binaries distributed as RPMs. Note that the debuginfo RPMs can be installed after the data is collected; they are only needed for the data analysis, not for the data collection.

4. DATA COLLECTION AND ANALYSIS

Three hardware platforms (AMD Athlon [AMD02], Intel Pentium 4 [INT03a], and Intel Itanium 2 [INT03b]) were set up to build the Mozilla web browser package. The hardware characteristics are listed in Table 1.

Table 1. Hardware characteristics.

Processor	***Athlon***	***Pentium 4***	***Itanium 2***
#processors	1	1	4
CPU frequency	1.7 GHz	2.4 GHz	900 MHz
L1 Data Cache	64 KB	8 KB	16 KB
L1 Data Cache line size	64 Bytes	64 Bytes	64 Bytes
L2 Cache	256 KB	512 KB	256KB
L2 line size	64 Bytes	128 Bytes	128 Bytes
L3 Cache	---	---	3 MB
L3 line size	---	----	128 bytes
RAM	512 MB	512 MB	1024 MB
Disk drive brand	MAXTOR	IBM-ESXS	MAXTOR
Disk drive model	6L080J4	MAN3184MP	ATLASU320_18_SCA
HD capacity	80 GB	18 GB	18 GB

The base software installed on the hardware was Red Hat Enterprise Linux Beta 1. The kernel was locally rebuilt on each machine to work around a transient problem with the generation of debugging information in the kernel. A more recent version of the OProfile package was also installed on the machines to allow the OProfile to be controlled from a script. Table 2 lists the specific version of the software installed on the machines. For the binary RPM packages installed the associated debuginfo RPMs were also installed to allow the data analysis tools to map the samples back to the source code.

Table 2. Software.

RHEL Beta 1 Components	***Version***
OProfile	oprofile-0.5.4-6
Kernel	kernel-source-2.4.21-1.1931.2.349.2.2.en
Compiler	gcc-3.2.3-12 gcc-c++-3.2.3-12 binutils-2.14.90.0.4-22
RPM	rpm-4.2.1-0.29 redhat-rpm-config-8.0.24-3.1 rpm-build-4.2.1-0.29
Mozilla	mozilla-1.4-3.0.11

The .rpmmacros file was set in the user's home directory to allow the Mozilla web browser package to be built as a normal user rather than as root in `/usr/src/redhat`. The Mozilla source RPM (mozilla-1.4-3.0.11.src.rpm) was installed in the user's rpmbuild directory. The user could start the build of the package in the spec directory with:

```
rpmbuild -ba mozilla.spec
```

On each machine the rpmbuild of Mozilla was run under the `time` command. Table 3 lists the real, user, and system time for each. The real time are within 10% of each other. However, the build on the Itanium 2 makes use of the multiple processors; the make in the build process runs with four threads. The result is the user time on the Itanium is 87m26s, significantly more time than the wall real time, 54m36s.

OProfile's measurements are only for the CPU. If the build is I/O bound, then the OProfile analysis and optimization of the CPU portion will have little impact on the overall performance. The upper bound on the time spent for I/O can be estimated on the uniprocessor Athlon and Pentium 4 by comparing the sum of the user and sys time to the real time. For the Athlon the difference is 4m46s and for the Pentium 4 it is 3m28s. In both cases less I/O is less than 10% of the build time.

Table 3. Time spent for mozilla package build.

time	Athlon	Pentium 4	Itanium 2
real	55m0.251s	51m52.128s	54m36.099s
user	46m27.920s	43m26.790s	87m26.290s
sys	3m46.970s	4m57.380s	2m59.737s

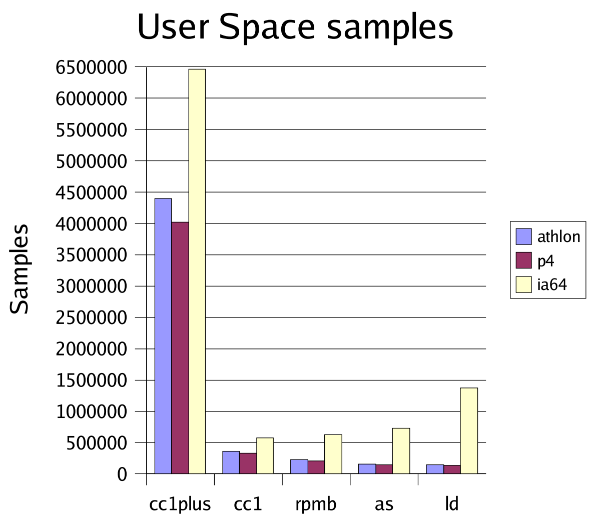
A script was written to install the OProfile driver, read the performance monitoring setups from a file, make a separate data collection run for the setup, and save the data from the data collection run as a session. For the time-based measurements the sampling intervals were set to be 500 microseconds on all processors to allow the samples to be directly compared. The Athlon and Pentium 4 do not count the time that the processor was halted (idle); the Itanium 2 counts all cycles resulting in many samples in the kernel idle loop. Table 4 lists the events for time-based measurements. OProfile was also configured to map samples back to the currently running kernel and to associate the library samples with the executable using the shared library.

Table 4. OProfile settings for time-based data collection.

Processor	***events***
Athlon	-ctr0-event=CPU_CLK_UNHALTED --ctr0-count=843500
Pentium 4	--ctr0-event=GLOBAL_POWER_EVENTS --ctr0-unit-mask=1 --ctr0-count=1193000
Itanium 2	--ctr0-event=CPU_CYCLES --ctr0-count=448500

Figure 1 shows the number of samples of the top five user-space programs run during the build of Mozilla. The programs are cc1plus (C++ to assembler translator), cc1 (C to assembler translator), rpmb (rpmbuild program), as (the assembler), and ld (the loader). Mozilla is a C++ application. Thus, most of the samples are related to the conversion of C++ source code into executable and most of the samples are for cc1plus, the core of the G++ compiler. The as program converts the assembly language in object code and ld links the object code into executable binaries. The 1.7GHz Athlon and 2.4GHz Pentium 4 have similar number of samples and spend comparable amounts of time in the various user-space programs with the Pentium 4 having 91% of the samples of the Athlon. The Itanium 2 has 85% more samples than the Athlon, indicating that 85% more cpu time is used by these five executables on Itanium 2 than the Athlon. Sections 5, 6, 7, and 8 discuss the individual programs in more detail.

Figure 1. Samples for top user space executables.



5. G++ AND GCC ANALYSIS

The bulk of the Mozilla build time is spent in the programs that convert source code into assembly language, cc1plus and cc1. More detailed analysis is required to see which functions consume processor time converting the source code to assembly language. For each architecture the functions with the ten highest sample counts were selected. The top ten functions varied between architectures. Figure 2 plots the counts for the functions for cc1plus and Figure 3 plots the counts for the functions for cc1.

For the Athlon and P4 the samples counts for cc1plus and cc1 are similar. The same executable is being used for both Athlon and P4. The ia64 architecture is very different from the i386 architecture and one would expect the sample counts to be a bit different because the compiler is generating native code for the host processor. The most notable differences are the bundling of explicit parallelism, the large register set for the ia64, and the lack of complex addressing modes. The sample counts for the ia64 cc1plus and cc1 are very different from the i386 processors. The compilers spend a large amount of time in functions related to register allocation: `ia64_secondary_reload_class`, `regclass`, and `true_regnum`. cc1plus spends 17% of the its runtime in those three functions. cc1 spends 9.4% of its runtime in `ia64_secondary_reload_class`, `regclass`, and `record_reg_classes`.

Examination of the `regclass` function shows that a triple-nested loop is used for the ia64 due to the post increment and decrement addressing modes available on the ia64; this triple-nested loop is not used on the i386. One of the loop iterates over the set of physical registers in the processor. The GNU Compiler ia64 backend lists 335 physical registers, while the i386 backend used for P4 and Athlon only has 54 physical registers. Another of the loops in the triple-nested loops iterates over the register classes (groups of registers that can be used in the same manner). The register classes on the i386 are complicated because of the nouniform use of the i386 register, yielding 25 register classes on the i386. The ia64 is simpler in this respect and has only 11 register classes. These loop bounds for the ia64 makes the loops iterate a great number of times in `regclass`. Additionally, the ia64 compiler needs to allocate registers for address computations because the ia64 only support simple indirect addressing.

Hashing on the compiler uses the modulo and division operations. The i386 processor provide hardware instructions to perform modulo and division operations. The ia64 calls functions to implement modulo and division operations. On the ia64 over 4% of the samples for cc1plus are due to functions implementing unsigned 64-bit division (__udivdi3) and modulo (__umoddi3).

The `finish_struct` function produce 5.6% of the cc1 samples on Pentium 4 and 5.8% of the cc1 samples on the Athon. Another study of GCC [HOA03] studying the time required to build the Linux kernel with various versions of GCC found this to be a significant consumer of cycles on i386 machines for GCC 3.2. In the past there has been problems with `finish_structs` using inefficient algorithms [SHE02].

Figure 2. Samples for cc1plus.

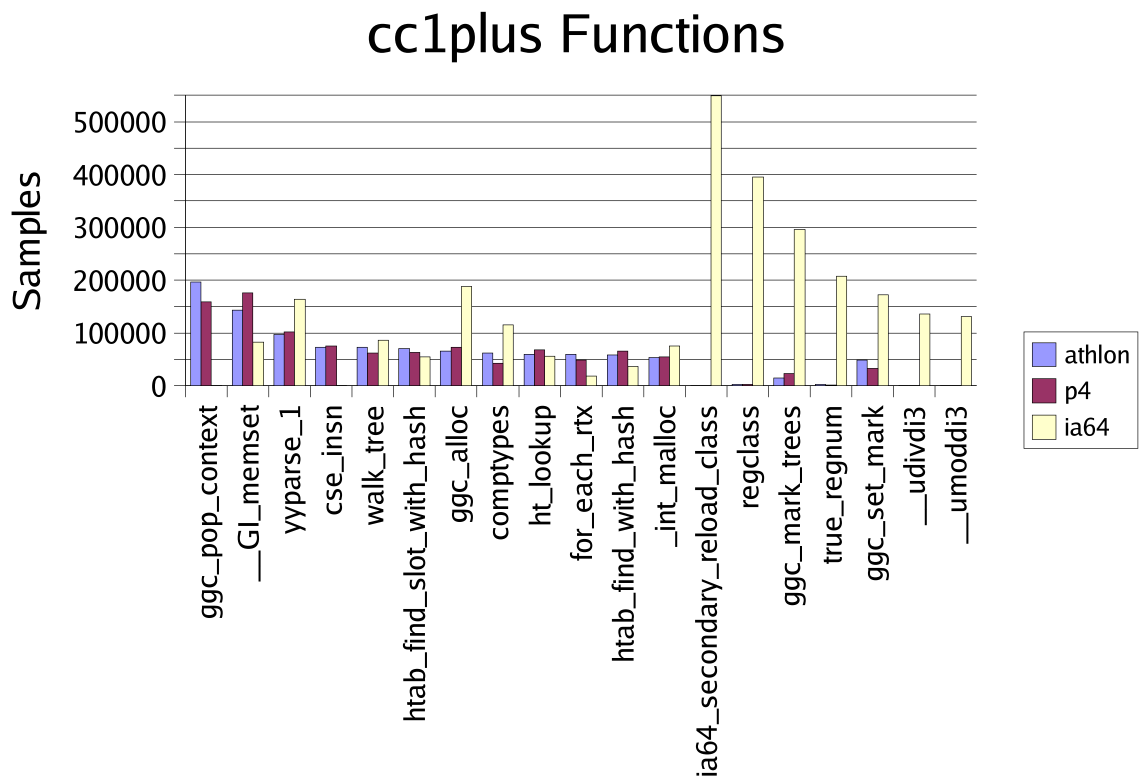
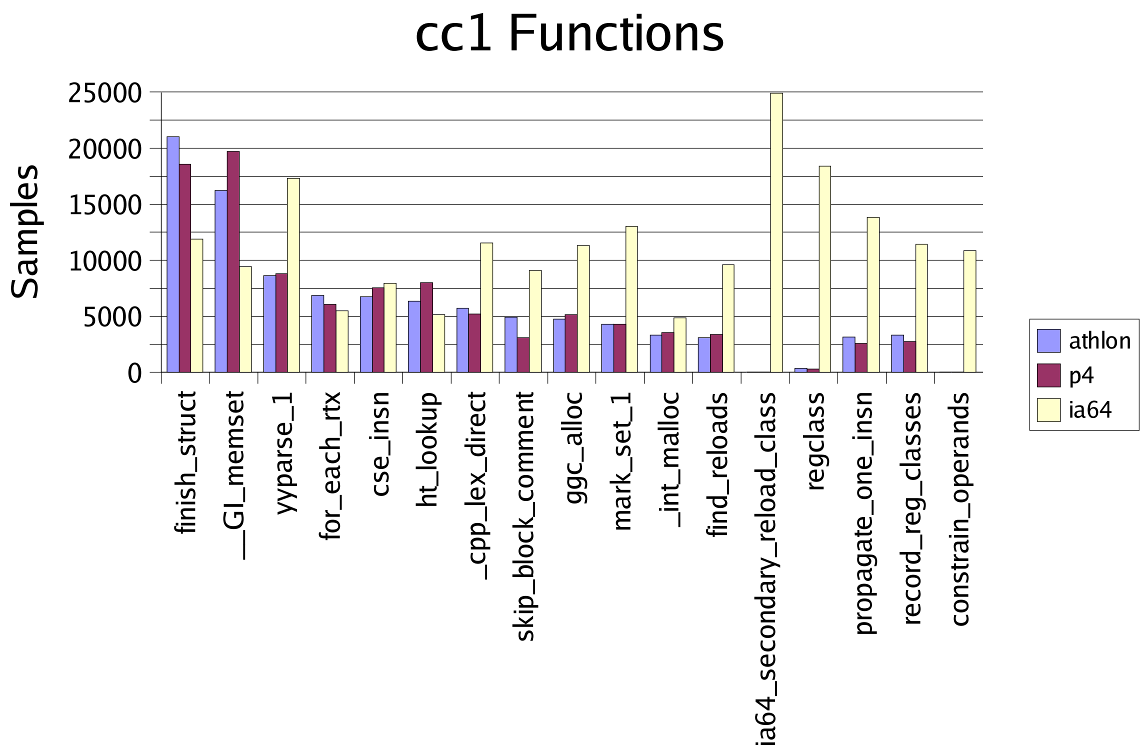


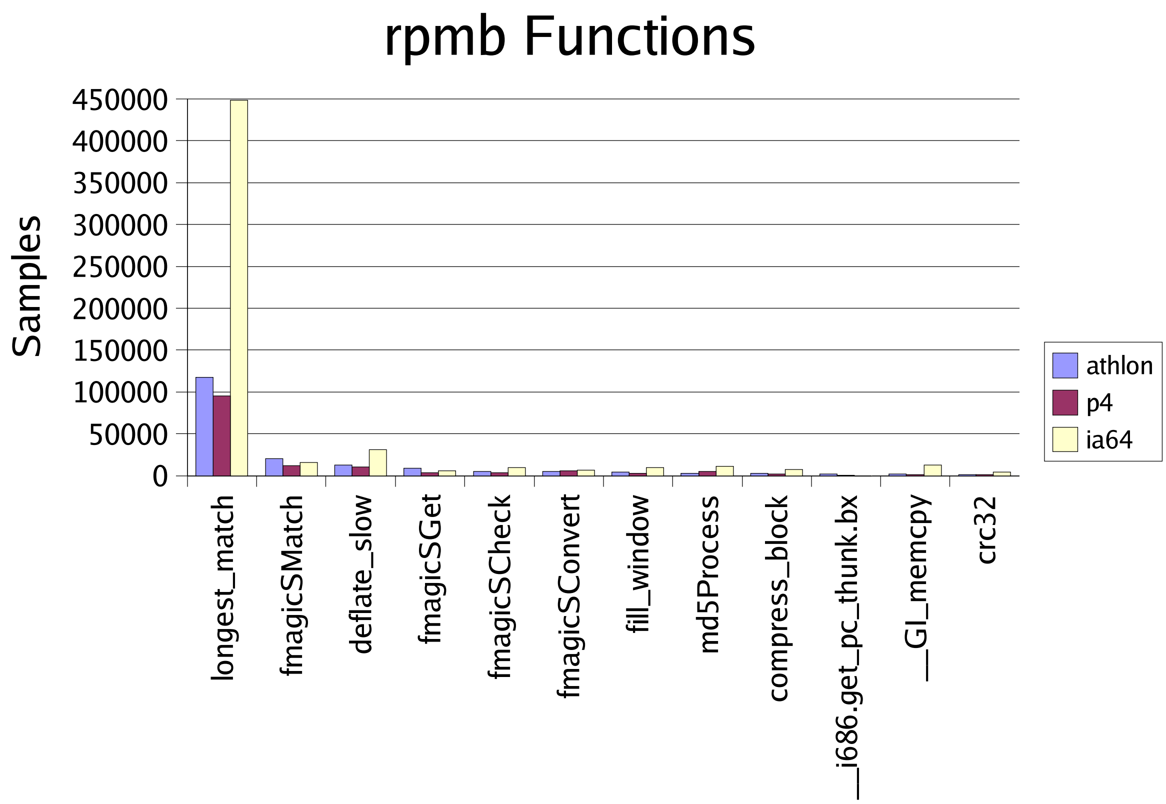
Figure 3. Samples for cc1.



6. RPMB ANALYSIS

The rpmb program implements the rpmbuild command; it coordinates the build process, running the pre- and post- build scripts, compressing the files for the package, and performs the actual packaging. The rpmb program is not a large consumer of time for a compile intensive build. However, there is a hotspot in rpmb, `longest_match`. On the i386 over 60% of the time is spent in longest_match; the ia64 spent 77% of its time in that function. The `longest_match` function is used in the compression algorithm. Given the percentage of runtime rpmb uses, analyzing the other functions is not worthwhile at this time.

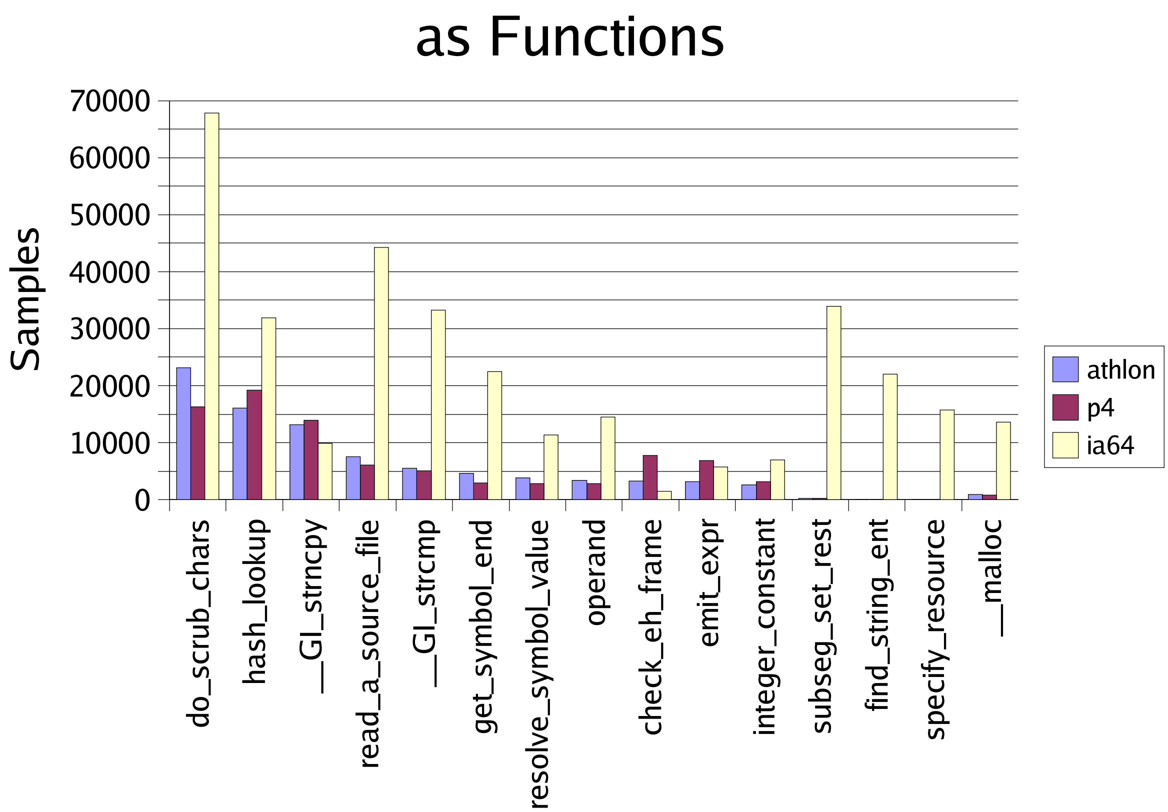
Figure 4. Samples for rpmb.



7. ASSEMBLER ANALYSIS

The GNU compilers do not directly generate object code; they generate assembly language which is translated into object code by the as program. Figure 5 shows that the ia64 as spends much more time in `do_scrub_char` and `read_a_source_file` than the i386 platforms. This may be a result of the compiler generating more assembly language code. The ia64 processor only has simple indirect addressing. Thus, load and store operations such as indexed loads and stores to arrays with scaled offsets which are single instructions on the i386 require sequences of instructions on ia64.

Figure 5. as Function samples.

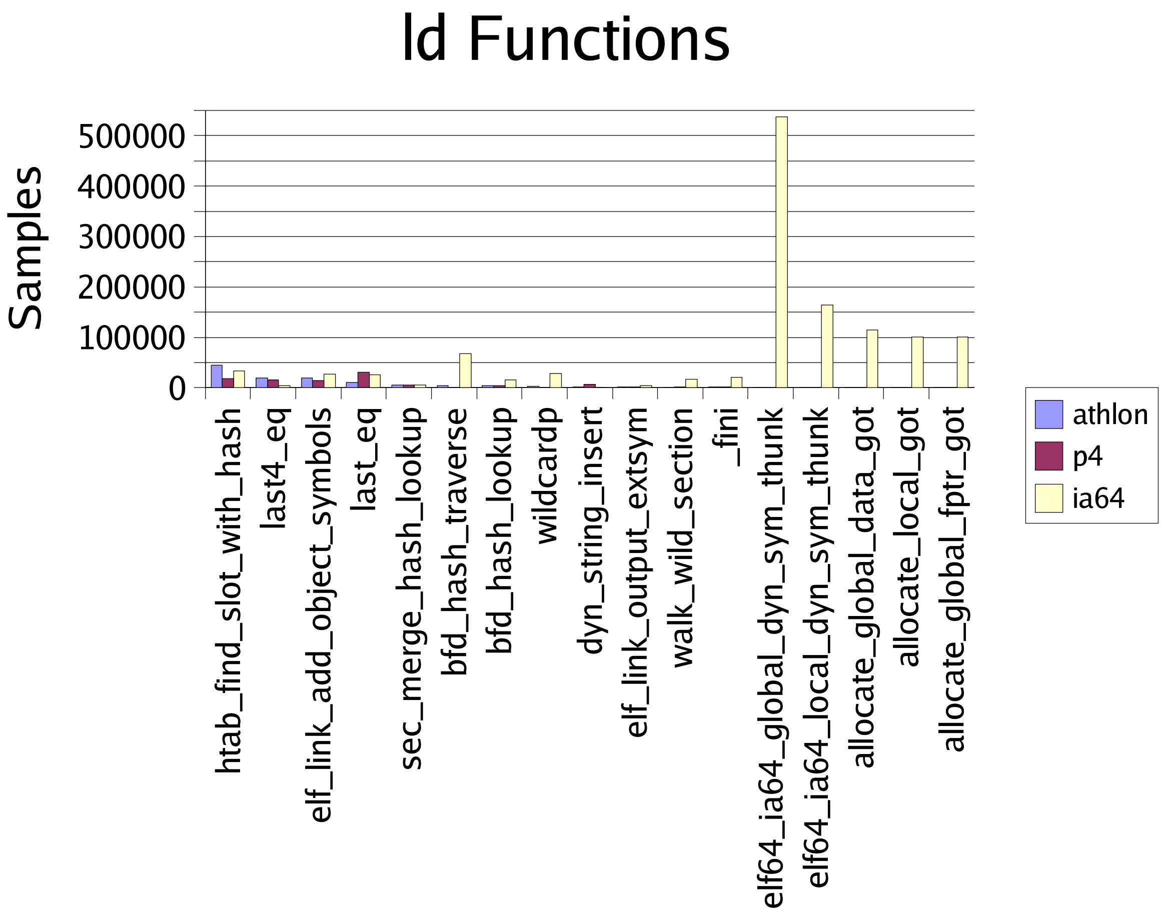


8. LINKER ANALYSIS

The object code files are linked into executables and libraries by the ld program. On the ia64 Mozilla build 5.6% of the build time was consumed by the linker on ia64, on the Pentium 4 and Athlon 2.4%. In terms of absolute time the ia64 ld spends an order of magnitude of time more than the Pentium 4 (1378392 samples vs. 133497 samples).

Figure 6 shows the top ten functions with samples. The ia64 linker uses a number of functions that have no equivalents in the i386 linker, e.g. `elf64_ia64_global_dyn_sym_thunk` and `elf64_ia64_local_dyn_sym_thunk`. These two functions contribute over 50% of the total time for ld on the ia64. The function `elf64_ia64_global_dyn_sym_thunk` does a walk through a linked list calling a function each time to do a test. The function exits with false the first time the test function fails, but if the test is true it must traverse the entire list. The `elf64_ia64_local_dyn_sym_thunk` is coded in the same inefficient manner.

Figure 6. ld Function samples.



9. FUTURE WORK

This work is an early application of OProfile to gain insight into the issues of building executables for production software. More detailed analysis of the data should be performed. The data collected allows the developers to focus on the sections of the build process that consume the most time.

OProfile does not provide call graph information. It would be useful to augment the data collection by OProfile to also provide counts on the number of times that functions are called. This would allow developers to distguish between a too expensive function being called a few times and a relative inexpensive function being called too many times. The optimizations for these cases are different. The developer has to manually determine which category the function falls in, which can be difficult.

Although the current version of the software allows use on a stock distribution of Red Hat Enterprise Linux, there should also be better intergration of the tools with the system and better analysis tools. At times some detailed knowledge of where the mapping information is needed to allow the user to annotate the source code. Finally, the analysis tools for OProfile are relatively simple. Some of analysis for this paper, e.g. comparing runs from different machines, was performed by hand because the tools did not do them. More of the analysis should be automated.

10. CONCLUSIONS

The build process is a common procedure in software development. Analyzing the build process with the OProfile system-wide profiler on multiple platforms provided insight into areas in the software that could be improved to reduce the build time. The data collected for the Mozilla build shows several hotspots in the ia64 compiler.

11. REFERENCES

[AMD02] AMD Athlon Processor X86 Code Optimization Guide, February 2002, AMD Publication 22007.

[BAI97] Bailey, E., Maximum RPM, MacMillan Publishing Company, August 1997.

[ECL03] Eclipse, http://eclipse.org/, August 2003.

[HOA03] Hoare, G., "Optimization case study: GCC", http://people.redhat.com/graydon/gcc-optimizing/, June 2003.

[INT03a] Intel Architecture Optimization Reference Manual, Intel Order Number 245127.

[INT03b] Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, June 2002, Intel Document 251110.

[OPR03] OProfile, http://oprofile.sourceforge.net/, July, 2003.

[SHE02] Shebs, S., "[PATCH] Greatly speed up finish_struct," http://gcc.gnu.org/ml/gcc-patches/2002-05/msg02546.html, May, 2002.