

COMPILING THE
LINUX KERNEL 2.6
USING
INTEL® C/C++ COMPILER FOR LINUX 8.0

A PRELIMINARY GUIDE

Dipl.- Inform. Ingo A. Kubbilun
www.pyrillion.org
- Germany -

Version 0.2, 06/18/2004

(created with OpenOffice 1.1.1)

0. Table of contents

0. TABLE OF CONTENTS.....	2
0.1 REVISION HISTORY.....	3
0.2 DISCLAIMER.....	3
1. PREFACE.....	4
1.1 SCOPE OF THIS DOCUMENT.....	4
1.2 PREREQUISITES.....	4
1.3 HISTORY OF THE KERNEL PATCH TOOLS.....	4
2. THE INTEL® COMPILER AND THE LINUX KERNEL.....	5
2.1 INTRODUCTION.....	5
2.2 OPTIMIZER UNITS AND THE KERNEL.....	5
2.3 PATCHING THE KERNEL.....	6
2.3.1 Installation of the patch archive.....	6
2.3.2 The compiler wrapper kicc.....	6
2.3.3 Inter Procedural Optimization (IPO).....	7
2.3.4 Minimal kernel patches.....	8
2.3.5 Best kernel patch.....	9
2.3.6 Experimental kernel patches.....	10
3. KERNEL PGO.....	11
3.1 THE KERNEL MODULE INTLPGO.....	11
3.1.1 Preparation of the kernel module.....	11
3.2 HOW DOES KERNEL PGO WORK?.....	12
3.3 BENEFITS OF PGO.....	12
3.4 MOVING THE PGO INSTRUMENTATION TO OTHER KERNEL PARTS.....	12
3.5 EXAMPLE OF A PGO RUN.....	13
3.5.1 Situation.....	13
3.5.2 Example modifications of the kernel Makefiles.....	13
3.5.3 Generation of the dynamic profile data.....	13
3.5.4 Feedback compilation.....	13
3.5.5 Sample output of a feedback compilation.....	14
4. APPENDICES.....	15
4.1 KERNEL PATCH FOR 2.4.XX KERNELS.....	15
4.2 PGO INSTRUMENTATION OF KERNEL MODULES.....	15

0.1 Revision history

Version	Date	Author(s)	Remarks
0.1	04/20/04	Ingo A. Kubbilun	Initial version
0.2	06/18/04	Ingo A. Kubbilun	lots of updates

0.2 Disclaimer

THIS DOCUMENT IS PROVIDED BY INGO A. KUBBILUN "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION).

Copyright ©2004 Ingo A. Kubbilun. All rights reserved.

All logos and trademarks in this document are property of their respective owner.

1. Preface

1.1 Scope of this document

This document presents kernel patches and tools covering the compilation and performance optimization of the Linux kernel **2.6.5/2.6.6** using Intel® C/C++ Compiler for Linux **8.0.055**.

Please read “*Aufgedreht: Kernel- Tuning*” in the German Linux Magazine, issue 07/2004 for a more comprehensive discussion of the kernel patch. There will be a translation of the article in the English Linux Magazine (issue 08/2004).

1.2 Prerequisites

You need the latest GNU binutils¹, the kernel 2.6.5/2.6.6 source distribution², and the module init tools³ if you did not use a 2.6 kernel before.

Furthermore, if you do not own a licensed version of the Intel® C/C++ Compiler for Linux (version 8.0.055 or later), you can download⁴ the “*free non- commercial unsupported version*” for private use only.

You should be familiar with C and some basic knowledge of the Linux kernel (at least how to unpack, build, and install it).

Please read the comprehensive documentation that comes with the Intel® C/C++ Compiler for Linux (man pages and PDF documents).

The kernel patch archive is available at:

<http://www.pyrillion.org/linuxkernelpatch.html>

1.3 History of the kernel patch tools

The first published version of the kernel patch for the 2.6 kernel was 0.8, all prior versions handled the Linux kernel 2.4.x, which is deprecated from now on.

All kernel patches need intermediate helper tools because icc up to 8.0.055 does not support certain features, which need to be emulated, e.g. common symbols (switch: “*- fno- common*”).

The evolution of the kernel patch is the main cause for changing these tools from version to version, which shows the following table:

version	tool	description
0.8	gccicc	gcc to icc delegate, which offered the forwarding of non- compilable C sources from icc back to gcc
0.9[.1]	ccd/lkd	C compiler delegate (ccd) and linker delegate (lkd), definition of the environment variables <i>CCC</i> and <i>CCC_OPTS</i>
>=1.0	kicc	kernel icc, no environment variables any more, less invasive (less Makefile modifications)

¹ version 2.14 as of writing this document (<http://www.gnu.org/directory/GNU/binutils.html>)

² <http://www.kernel.org/pub/linux/kernel/v2.6/>

³ <ftp://ftp.kernel.org/pub/linux/kernel/people/rusty/>

⁴ <http://www.intel.com/software/products/compilers/clin/noncom.htm>

2. The Intel® Compiler and the Linux kernel

2.1 Introduction

The Intel® C/C++ Compiler for Linux is a highly optimizing compiler that supersedes the GNU gcc (even version 3.3.3) in most cases regarding the performance of application code. The compiler is comprised of two versions: a 32bit version (I386 platform) and a 64bit version (Itanium/Itanium 2, i.e. IA64 platform).

Several Intel® employees are collaborating with the Linux kernel community to create kernel patches for the Intel compiler. It is hard to find or access this information especially if you do not have access to the “*Intel Premier Support*”.

This document covers the I386 platform and thus the 32bit version of the compiler only. The first patches were developed for the 2.4 kernels and the Intel® C/C++ Compiler for Linux 7. A lot of patches were needed in order to successfully compile these kernels. The situation became better with the appearance of the Intel® C/C++ Compiler for Linux version 8. Intel did a great job enhancing the compiler in terms of gcc compatibility. Some problems do remain that will be solved here.

Do not expect miracles using the Intel® C/C++ Compiler for Linux to compile the Linux kernel. The kernel is heavily I/O bound: its performance is heavily dependent on the underlying hardware. Furthermore, any optimization, which is applied on the source level of the kernel results in much better kernel performance than using an optimizing compiler. Nevertheless, if you activate both key mechanisms of icc, i.e. IPO (*Inter Procedural Optimization*) and PGO (*Profile Guided Optimization*), then a performance boost up to 40% for certain kernel parts and an overall average performance boost of approx. 8-9% is possible.

2.2 Optimizer units and the kernel

The Intel® C/C++ Compiler for Linux “*icc*” roughly offers the following optimization mechanisms in addition to a standard C compiler optimizer:

1. code emitting for specific CPU architectures: ⁵Pentium Pro/II, Pentium III, Pentium 4, Pentium 4 codename “Prescott”, Pentium M;
2. highly sophisticated “*vectorizer*” (vectorizes loops and creates SIMD⁶ code);
3. IPO (**I**nter **P**rocedural **O**ptimization);
4. PGO (**P**rofile **G**uided **O**ptimization).

The first two items are more or less supported by gcc, too. The vectorizer must be disabled compiling the Linux kernel. The kernel saves the complete FPU/SSE context to support MMX and SSE instrumented applications. The expensive save and restore operations are delayed and executed if and only if the current application makes use of FPU, MMX, or SSE instructions. The current kernel sources do not support SIMD code in the kernel. The workaround used e.g. by the 3DNow! extensions consists of embracing SIMD code by calls to `fpu_kernel_begin` and `fpu_kernel_end`. Because icc creates SIMD code sequences anywhere in the code, this workaround is not applicable. A future release of this kernel patch might support SIMD kernel code by a compiler post processor, which identifies SIMD sequences inserted by icc and pro-

⁵ Pentium and Pentium MMX are of small interest.

⁶ SIMD stands for “**S**ingle **I**nstruction, **M**ultiple **D**ata” and means MMX, SSE, SSE2, and SSE3 CPU extensions

tests them using the above mentioned functions (or substitutes of them). Until now, it has not been checked if the SIMD kernel code would result in better kernel performance. Keep in mind that such SIMD code sections must disable kernel preemption temporarily, which always results in some performance degradation.

The IPO and PGO mechanisms are applicable to the kernel code. PGO is supported by a special kernel module and an user daemon, which are included in the patch archive.

2.3 Patching the kernel

As of now, the kernel patch does not simply patch the kernel sources but also need some external tools, e.g. the `icc` wrapper tool. Currently, three different kernel archives are available:

archive	covered kernel(s)	remarks
linux-2.6-icc-0.8	2.6.3, 2.6.4, 2.6.5	limited IPO support, no PGO support, uses the C compiler delegate <code>gccicc</code> , this archive is deprecated
linux-2.6-icc-0.9 [1]	2.6.3, 2.6.4, 2.6.5, 2.6.6	full IPO and PGO support, compiler delegate <code>gccicc</code> replaced by <code>ccd</code> (C compiler delegate) and <code>lkd</code> (linker delegate), PGO kernel module <code>intlpgo</code> and daemon <code>pgod</code> added
linux-2.6-icc-1.0 (latest)	2.6.5, 2.6.6	full (changed) IPO and full (changed) PGO support, patch is less invasive, delegates <code>ccd/lkd</code> replaced by <code>kicc</code> (<i>kernel "icc"</i>), several patch files available (minimal impact, best performance, experimental (test versions))

This document covers the latest kernel patch `linux-2.6-icc-1.0` only.

2.3.1 Installation of the patch archive

Download and unpack the kernel archive. Change the working directory to the top-level patch directory and type (bash):

```
./configure && make && make install
```

This builds and installs the compiler wrapper `kicc` (`/usr/local/bin`), the PGO daemon `pgod` (`/usr/local/sbin`), and the man pages `kicc.1`, `pgod.8`, `intlpgo.9` and `lkpatch.9` (`/usr/local/man`).

The folder `/patches` contains the patch files. The folder `/ExtractPGO` contains another tool to extract the original PGO code needed by the kernel module `intlpgo`. Please refer to chapter 3 (starting on page 11) for details on profile guided optimization in the kernel.

2.3.2 The compiler wrapper `kicc`

The Intel® C/C++ Compiler for Linux is not used by the kernel patch directly. Instead, the compiler wrapper “`kicc`” (*kernel icc*) is executed, which in turn executes `icc` as a child process.

`kicc` is responsible for:

- i. converting `gcc` compiler switches to their `icc` pendants;
- ii. post processing the generated code;
- iii. debugging purposes.

It first compiles a C source to an assembler file (`.S`), applies generic (built-in) patches to the assembler source, and finally assembles it to object code using `gas`, the GNU assembler. Some in-

compatibilities of the Intel® Compiler cannot be fixed by patching the kernel sources or by using other `icc` compiler switches (see below).

`kicc` performs the following steps:

- converts switches:
 - ignores: `-fomit-frame-pointer`, `-Wstrict-prototypes`, `-Wwrite-strings`, `-Winline`, `-Wno-uninitialized`, `-Wno-format`, `-Wno-trigraphs`, `-Wno-unused`, `-fno-inline-functions`, `-finhibit-size-directive`, `-fno-exceptions`, `-fno-inline`, `-ffloat-store`, `-fno-builtin`, `-fexceptions`, `-pipe`, `--param`, `-frename-registers`, `-fno-strict-aliasing`, `-fno-common`, `-ffixed-r13`, `-mb-step`, `-traditional`, `-gstabs`, `-falign-functions=`, `-mpreferred-stack-boundary=`, `-malign-functions=`, `-malign-jumps=`, `-malign-loops=`;
 - adds `-no-gcc` and the manual definition of the compiler version to 3.3.0;
 - switch `-s` yields to a forward of the operation to `gcc`;
 - switch `-v` yields to the print of the Intel compiler version;
 - `-Os` is converted to `-O1`;
 - `-O2` and `-O3` are converted to `-O3 -ip`;
 - `-Wall` is converted to `-w` (only errors);
 - `-fno-omit-frame-pointer` is converted to `-fp`;
 - `-march=xxx` switches are converted to `-tpp5|6|7`;
 - switch `-static` always added;
 - switch `-ansi-alias-` always added;
 - `-idirafter<dir>` always added (Intel® Compiler header files);
- compiles to assembler source using `icc`;
- patches the assembler source:
 - converts common symbols (always emitted by `icc`; the switch `-fno-common` is supported by `icc` but implemented in a different way compared to the `gcc` switch);
 - removes “junk functions” inserted by the kernel patch and the modified utility `modpost`; these functions are needed to retain certain symbols otherwise discarded by `icc` in IPO mode (cause: the attribute “used” is not honored by `icc` in all situations)
- assembles to object code.

PPVA (Post Process Via Assembly) is the core module of `kicc`. If you define the environment variable `PPVA_DUMP_STAT`, then it dumps patch statistics. If you define `PPVA_KEEP_ASM`, then the intermediate assembler files are retained on disk for debugging purposes (naming convention `<source>.c.S`). If you encounter problems using `icc`, then you can inspect the assembler files to e.g. detect wrong code emitted by `icc`.

2.3.3 Inter Procedural Optimization (IPO)

Please refer to the original Intel® Compiler documentation to learn more about IPO in general.

The Intel Compiler offers two different IPO modes: IPO within files (single file IPO) and IPO across files (multiple files IPO). This version (1.0) of the kernel patch uses the less powerful mode “IPO for single- file compilations”, which does not result in real performance degradations (compared to multi- file IPO).

Beside several problems with IPO discovered in subsequent sections of this document, the multi- file IPO is useless if the kernel option `CONFIG_MODVERSIONS` is set. Each C source is first compiled to a temporary object code, which results in the “real” object code by performing a second link step to add the version symbols. Due to this compile and link scheme, the IL files created by the Intel® Compiler in multi- file IPO mode are not referenced any more. The biggest performance gains are realized by the kernel PGO instrumentation (refer to chapter 3 starting on page 11).

2.3.4 Minimal kernel patches

Use the patch file `patches/patch-2.6.5-ICC-MINIMAL` to apply minimal patches required by the Intel® Compiler.

This patch does the following (**mandatory patches needed by icc are bold**):

1. patches the top- level Makefile: `EXTRAVERSION` is set to `ICC`, `gcc` is replaced by `kicc`, `ld` is replaced by `xild`⁷, additional (`icc`) dummy symbols are deleted in `System.map`, the clean files are modified to include `.il`⁸ and `.c.S` files;
2. patches `arch/i386/Kconfig` and `arch/i386/Makefile` to include the `intlpgo` kernel module (PGO support);
3. adds the `arch/i386/intlpgo` directory and files (PGO support);
4. **patches two `asm` constraints in `arch/i386/kernel/efi.c` (fixes an `icc` incompatibility);**
5. **rewrites the function “`atomic_dec_and_lock`” in `arch/i386/lib/dec_and_lock.c` completely (cause: `icc` emits wrong code, see remarks below);**
6. **adds the function “`memcpy`” to `arch/i386/lib/memcpy.c` (cause: otherwise linker error, symbol not found);**
7. **rewrites the C macros in `include/asm-i386/byteorder.h` (cause: `icc` throws internal compiler errors `0_(4900 + 5)` if the inline assembly is not replaced by the compiler intrinsic `_bswap`);**
8. **patches the C macro `_IOC_TYPECHECK` in `include/asm-i386/ioctl.h` (cause: `icc` does not compile);**
9. **converts and patches the inline function “`prefetch`” in `include/asm-i386/processor.h` (cause: `icc` has problems with inline functions declared as “extern inline” with attached function body; otherwise wrong relocations emitted, which cause the occurrence of `SIGSEGV`’s loading certain kernel modules, e.g. `ieee1394.ko`);**
10. patches `include/linux/module.h` and `scripts/modpost.c`; dummy access functions are inserted not to discard certain kernel symbols during IPO (these functions are automatically removed by `kicc` in the post processing phase);

⁷ only needed if you plan to use multi- file IPO

⁸ only needed if you plan to use multi- file IPO

11. patches `kernel/module.c` (PGO support) to allow PGO instrumented kernel modules to be unloaded.
12. patches `include/linux/reiserfs_fs.h` to enable the compilation of the Reiser file system (the function attribute “`__attribute__((noreturn))`” raises a SIGSEGV in `icc`)

This minimal patch contains everything that is needed to successfully compile the kernel using `icc`. **Please keep in mind that the majority of the modifications affects the PGO instrumentation of the kernel only!**

The biggest problems of the Intel® C/C++ Compiler for Linux 8.0.055 seem to be the inline assembly and the expansion of inline functions. Intel® recommends to avoid inline assembly and to use intrinsic functions instead. That is a nice hint, which e.g. fixed the internal compiler error problem mentioned above⁹ (enumeration item 7).

The problem is that the Linux kernel needs a lot of inline assembly constructs for which the Intel Compiler does not offer any intrinsic. This affects e.g. the atomic operations needed by the SMP implementation. The fix of the problem stated in subsection 2.3.4 is the result of a one-month-debugging-session. The Intel® Compiler performs the inline assembly on local stack copies of the variables stated by the `asm` constraints, which yields to wrong code and is not SMP-safe anymore.

Another problem is the implementation of the C attributes. The attribute “`used`” causes trouble (see above) and the attribute “`always_inline`” is not honored by `icc`. The bad message is: there is no way to force `icc` to always expand an inline function (please refer to subsection 2.3.5 for more information). During the development of the patch, I tried everything including the undocumented keyword `__forceinline`, which is supported by `icc`¹⁰. Furthermore, any inline function comprised of inline assembly and containing jump labels is never expanded inline.

The Intel® Compiler performs the inline expansion on behalf of the IPO mechanism. This minimal patch results in a bunch of (originally inline) functions, which are not expanded inline. A pure C example is the inline function “`get_current_thread_info`”, which is called frequently and results in a real performance degradation. Whereas the (atomic) bit manipulation functions represent the examples in the assembler domain. These inline functions consist of one assembler statement only but `icc` emits a full function with prologue, epilogue, call instruction and stack cleanup. Was this intended by the compiler engineers?

2.3.5 Best kernel patch

Use the patch file `patches/patch-2.6.5-ICC-BEST` to apply the above mentioned minimal patches plus certain performance patches, which yield to a faster kernel.

The performance patches convert certain inline functions to C macros to bypass the inline expansion decision of `icc`¹¹. Please refer to the patch file for details. The main modifications are applied to `include/asm-i386/bitops.h`. Do not modify the `asm` constraints used in the assembler statements of the C macros. They were chosen carefully not to yield to wrong code

⁹ btw this error is not limited to the kernel sources. If you try to compile e.g. OpenSSL 0.9.7d, then the symmetric cipher RC5 fails performing the “`make test`”, cause: the RC5 source uses inline assembly to rotate 32bit registers. If you substitute the according constructs by the intrinsics `_lrotl` and `_lrotr` in `rc5_locl.h`, then you’re all set. This example emphasizes dramatically the “*inline assembly problem*” of `icc`. It does not come up with an internal error but emits wrong code instead...

¹⁰ this keyword is MS Windows specific and used by Win32 C sources to force the inline expansion regardless of whether optimization is enabled or not

¹¹ Furthermore, some of the string memory operations (e.g. `movs`, `cmps`, `lods`, `stos`) are replaced by non-string operations, which are faster. Other parts of the kernel (e.g. page copy/fill), which use string memory operations are not modified because the string operations are faster if the referenced data is cached, which seems to occur quite often.

emitted by `icc`. The constraints are a little bit more restrictive than the original ones regarding the usage of CPU registers. You can use the original constraints and define the environment variable `PPVA_KEEP_ASM` to see what happens, e.g. in `mm/rmap.c[.S]` or `mm/vmscan.c[.S]`. Your kernel build ends up on the scrapheap!

2.3.6 Experimental kernel patches

The folder `patches` contains some more patch files, which apply more patches to the kernel source tree (more inline functions converted to C macros) but did not prove to perform better than the “*best kernel patch*” (subsection 2.3.5). They are included for completeness (only for kernel version 2.6.5).

3. Kernel PGO

[This is a topic for **advanced** users only.]

PGO (**Profile Guided Optimization**) is **the** highlight of the Intel® C/C++ Compiler for Linux. Normally, optimization decisions are made by evaluating a static profile of the code, which is created by a heuristic algorithm of the Intel® Compiler.

You can help the Intel® Compiler to optimize your code in a more sophisticated way by supplying *dynamic* profile information. The dynamic information is created in phase ii of the following three-phases-compilation-scheme:

- i. compile the source with PGO instrumentation enabled (switches: `-prof_gen[x]`, `-prof_dir`);
- ii. run the resulting code with different input parameters; the PGO instrumented code dumps dynamic profile data to disk in the background;
- iii. perform a feedback compilation using the dynamic profile data.

The third step emits optimized code, which reflects the real execution behavior of the application and obviously yields to better (faster) code. Unfortunately, the PGO instrumentation of kernel code is not natively supported by the Intel® C/C++ Compiler for Linux.

3.1 The kernel module *intlpgo*

This new kernel module is located in `arch/i386/intlpgo` after the kernel patch has been applied to the kernel source tree. It implements the necessary functions referenced by the PGO instrumented code and registers the new file system “*pgofs*”¹², which is needed by the PGO daemon “*pgod*”. After booting a PGO instrumented kernel, the daemon must be started¹³ to dump the dynamic profile data to disk, which is collected and temporarily stored in memory by the kernel module.

You have to define the kernel option `CONFIG_INTLPGO` to include the PGO kernel module. As of now, the module cannot be loaded/unloaded and must be statically linked to the kernel.

3.1.1 Preparation of the kernel module

The kernel module `intlpgo` included in the patch archive is not functional after unpacking the archive. It needs the original Intel® PGO code to be *injected* in the module before you can use it.

The Intel® copyrighted code is obviously **not** part of the patch archive. You need either the non-commercial unsupported version or a licensed version of the Intel® C/C++ Compiler for Linux to build an “*operational*” version of the kernel module.

Change the working directory to `<top-level patch dir>/ExtractPGO`. Then type:

```
make KRNL_SRCDIR=<top-level kernel source directory>
```

You have to apply one of the patch files presented in subsections 2.3.4, 2.3.5 or 2.3.6 first. The build process extracts and patches the original PGO code and creates the new file `arch/i386/intlpgo/intlPGO.object`, which replaces `arch/i386/intlpgo/intlPGO.S`. The

¹² The file system code is cloned from the great OProfile kernel module (<http://oprofile.sf.net>).

¹³ see man page `pgod(8)` for details

latter one contains the non-functional PGO dummy functions. The Makefile in `arch/i386/intlpgo` is modified, too. The transition `intlPGO.S -> intlPGO.o` is replaced by the transition `intlPGO.object -> intlPGO.o` (just a copy op.).

3.2 How does kernel PGO work?

If you compile a C source with the switch `-prof_gen[x]`, then the Intel® Compiler creates so called “*PGO segment packets*” in the data section and emits PGO code, which fills the packets. The PGO segment packets form a linked list. Upon execution of PGO instrumented code, the function `_PGOPTI_Prof_Begin` is called, which creates a link to the preceding segment packet in the list if the link was not established by a previous call already.

The usage of PGO in the Linux kernel can yield to such big kernels that they cannot be built successfully¹⁴ or crash during the boot sequence. You can limit the PGO instrumentation to certain kernel parts. After collecting the dynamic profile data, the PGO instrumentation can be moved to a different kernel part. Use the Intel® tool `profmerge` to merge the dynamic profiles before entering the feedback compilation phase.

The linking of the PGO segment packets is SMP-safe, whereas the PGO code filling the packets is **not**. You may use the PGO instrumentation in SMP kernels only on uniprocessor machines. If you want to create profiles for multiprocessor machines, then you must disable all but the boot processor. I did not check the PGO instrumentation on my multiprocessor machine with both processors enabled. Do this at your own risk. The core problem of the PGO code is the usage of 64bit counters, which need to be updated by a short sequence of assembler instructions on the 32bit I386 architecture. Intel does not provide spin locks to protect these sequences. They are omitted by the Intel® Compiler. There is no way to change this. In general, it is possible to use the PGO instrumentation even on a multiprocessor system but there is a certain probability that some of the 64bit counters of the PGO mechanism will be corrupted during the update process.

The PGO daemon `pgod` mounts the new file system `pgofs` provided by the kernel module `intlpgo` and requests a dump of the PGO segment packets in configurable intervals. The dynamic profile data is stored in `.dyn` files, which are needed by the Intel® tool `profmerge` to create the file `pgopti.dpi`. This file and the static profile `pgopti.spi` are used by the Intel® Compiler in the feedback compilation stage. The latter one is automatically created during the initial compilation of the kernel with PGO instrumentation enabled. Please refer to section 3.5 for a complete example of the PGO instrumentation and feedback compilation.

3.3 Benefits of PGO

Some PGO instrumented kernels have been benchmarked and compared to `gcc` compiled kernels and `icc` compiled kernels (without PGO). PGO tends to yield to the biggest performance gain. The results are not included here because it is not possible to create “universal” results due to the PGO mechanism. You have to check it on your own. The results heavily depend on your setup and the collected PGO data. Obviously, it is best to run the PGO instrumented kernel under the same conditions as it will be used later in a production environment.

3.4 Moving the PGO instrumentation to other kernel parts

You can repeat the three-phases-compilation-scheme as often as you want moving the profiling to other kernel parts (and/or modules). The old `.dyn` files dumped by the PGO daemon in a

¹⁴ „your kernel is too big, try using modules“

previous profiling run are not overwritten. You have just to re-run the Intel® tool `profmerge` to update `pgopti.dpi` with the new dynamic profile data. The static profile `pgopti.spi` is automatically updated by the Intel® compiler.

3.5 Example of a PGO run

3.5.1 Situation

You have properly configured and tested an `icc` compiled kernel.

3.5.2 Example modifications of the kernel Makefiles

You want to generate PGO data for the following kernel parts:

- `kernel`
- `arch/i386/kernel`
- `mm`
- `arch/i386/mm`
- `lib`
- `arch/i386/lib`

Enter the directories step by step and add the following line to the Makefiles:

```
EXTRA_CFLAGS += -prof_genx
```

and perform a “`make bzImage && make modules && make modules_install`”. Your kernel is now PGO instrumented. Check the size of the kernel to determine if it is probably too big.

3.5.3 Generation of the dynamic profile data

Install the kernel (as another option in LILO) and reboot. Create a directory for the `.dyn` files, e.g. `/profdata`. After logging in as root, start the PGO daemon:

```
pgod --start --prof-dir=/profdata
```

Now work with the kernel. The PGO daemon collects and dumps the dynamic profile data. After finishing the profiling, stop the PGO daemon:

```
pgod --kill
```

Reboot and select a “stable” Linux kernel (not PGO instrumented).

3.5.4 Feedback compilation

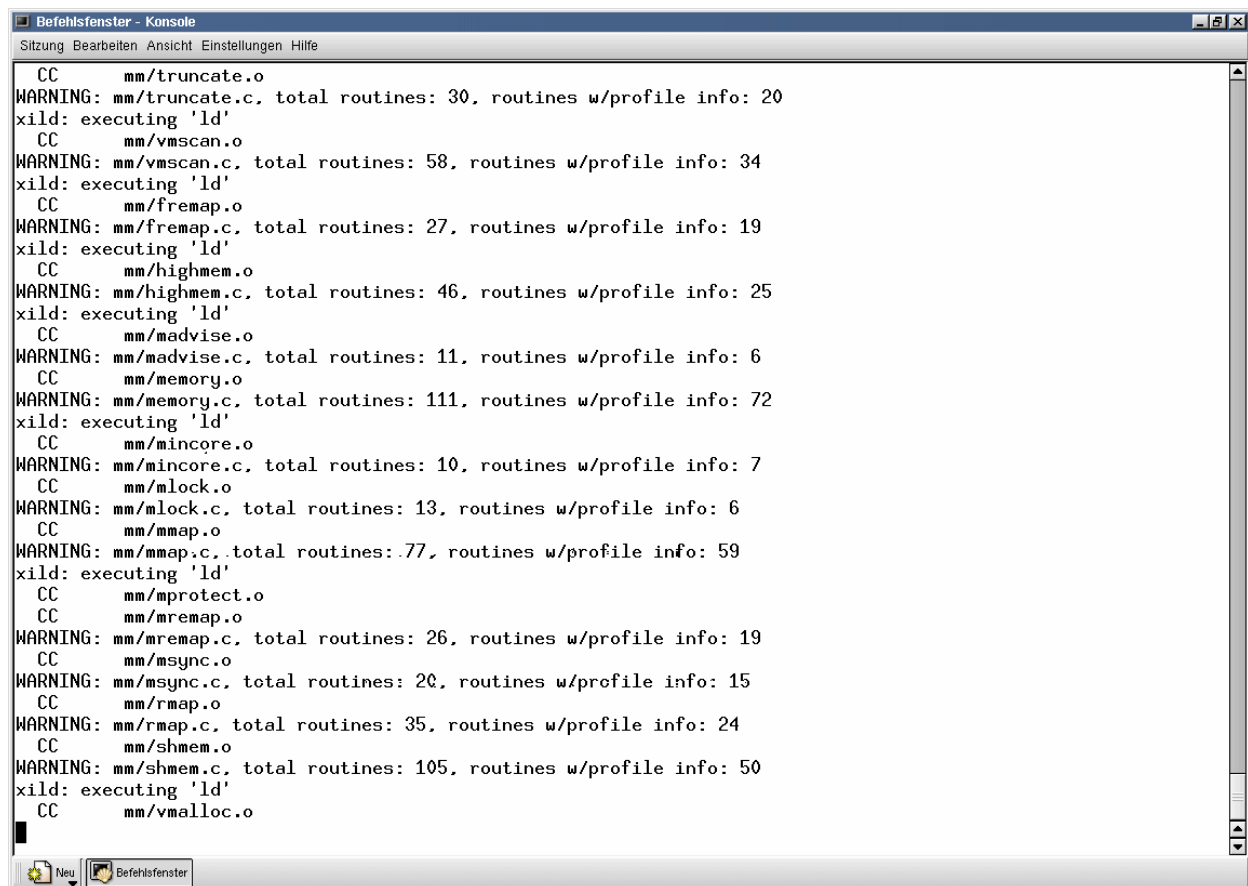
Enter the directory `/profdata` and run Intel®’s `profmerge` utility. It creates the file “`pgopti.dpi`”. Copy this file to the top-level directory of the Linux kernel source tree. You should now have to files “`pgopti.spi`” (created by the Intel® Compiler in the first compilation step) and “`pgopti.dpi`” (created by the `profmerge` utility).

Enter the directories mentioned in subsection 3.5.2 and modify the `EXTRA_CFLAGS` (substitute `-prof_genx` by `-prof_use`):

```
EXTRA_CFLAGS += prof_use
```

then perform a “`make bzImage && make modules && make modules_install`”. This is the feedback compilation step. Finally, install the new kernel. **Do not select the `intlpgo` kernel module to be included in this final kernel build if you want to distribute the kernel. It contains Intel® copyrighted code!**

3.5.5 Sample output of a feedback compilation



```
Befehlsfenster - Konsole
Sitzung Bearbeiten Ansicht Einstellungen Hilfe

CC      mm/truncate.o
WARNING: mm/truncate.c, total routines: 30, routines w/profile info: 20
xild: executing 'ld'
CC      mm/vmscan.o
WARNING: mm/vmscan.c, total routines: 58, routines w/profile info: 34
xild: executing 'ld'
CC      mm/fremap.o
WARNING: mm/fremap.c, total routines: 27, routines w/profile info: 19
xild: executing 'ld'
CC      mm/highmem.o
WARNING: mm/highmem.c, total routines: 46, routines w/profile info: 25
xild: executing 'ld'
CC      mm/madvise.o
WARNING: mm/madvise.c, total routines: 11, routines w/profile info: 6
CC      mm/memory.o
WARNING: mm/memory.c, total routines: 111, routines w/profile info: 72
xild: executing 'ld'
CC      mm/mincore.o
WARNING: mm/mincore.c, total routines: 10, routines w/profile info: 7
CC      mm/mlock.o
WARNING: mm/mlock.c, total routines: 13, routines w/profile info: 6
CC      mm/mmap.o
WARNING: mm/mmap.c, total routines: 77, routines w/profile info: 59
xild: executing 'ld'
CC      mm/mprotect.o
CC      mm/mremap.o
WARNING: mm/mremap.c, total routines: 26, routines w/profile info: 19
CC      mm/msync.o
WARNING: mm/msync.c, total routines: 20, routines w/profile info: 15
CC      mm/rmap.o
WARNING: mm/rmap.c, total routines: 35, routines w/profile info: 24
CC      mm/shmem.o
WARNING: mm/shmem.c, total routines: 105, routines w/profile info: 50
xild: executing 'ld'
CC      mm/vmalloc.o
```

4. Appendices

4.1 Kernel patch for 2.4.xx kernels

This section has been removed (2.4 patch is deprecated).

4.2 PGO instrumentation of kernel modules

The kernel module `intlpgo` interacts with the kernel module loader to ensure that unloaded PGO instrumented kernel modules do not corrupt the linked list of PGO segment packets. If a kernel module is about to be unloaded, the `intlpgo` kernel module backups the current linked list of PGO segment packets in an internal ring buffer. It then destroys the linked list. After the kernel module loader has discarded the module, the linked list is automatically rebuilt by the PGO code.

The PGO daemon `pgod` dumps all entries of the ring buffer during the next dump operation.