Compiling the Linux kernel with the Intel compiler

Kernel Tuning

The Intel C compiler typically generates much quicker code than the GCC. Unfortunately, previous efforts to run the Linux kernel 2.6 through this tool have been doomed to failure. This exclusive report shows how to do this despite the odds, and also provides evidence of the performance boost.

BY INGO A. KUBBILUN

ICC, the Intel C/C++ Compiler, is finally a competitor to GCC, worthy of mention. It supports modern processors, including the Pentium 4 and AMD CPUs with similar performance characteristics.

It only seems logical to use the Intel compiler to build the Linux kernel. Although the performance boost may be minimal in comparison to the improvements achieved with applications or libraries, the kernel mainly arbitrates between the hardware and the user interface layer and thus depends on I/O to a great extent.

In contrast to the GCC, there is a charge of around 400 US dollars for using the Intel compiler. Linux developers who put the compiler to non-commercial use are provided with a "free non-commercial unsupported version". Experimenting with the kernel 2.6.5 and the ICC 8.0.055 [1] quickly shows that building the kernel is not simply a matter of adding a few patches, despite Intel's claims at [2], at least not with kernel 2.6.

Hard Going

Intel's claim that the ICC 8.0 is source code and binary compatible to GCC 3.2 is true of the Linux kernel in only a very restricted sense. In theory, it is quite simple to change the kernel compiler. You replace *gcc* with a new compiler in the top level makefile. Unfortunately, the Linux source code is hard going for any compiler, and just goes to show that the two are not 100% compatible after all.

The concept of compiler compatibility may need some explanation. As a rule, different compilers behave differently with respect to command line options, the code they create, data emission, integrated functions (so-called intrinsics), assembler sequences, and anything beyond the scope of the ANSI/ISO C standard. It is only natural that replacing the GCC will necessitate various kernel source code modifications.

At this time of writing (using the Intel Compiler 8.0, patch level 055), a few modifications to the sources (of kernel 2.6.5) are insufficient to allow a clean build using ICC. The patch we will be looking at (available from [3]) uses two utility programs to do the job: *ccd* (C Compiler Delegate), and *lkd* (Linker Delegate). Instead of calling the Intel C Compiler or the linker directly, we will be using these tools as intermediate helpers.

Both tools handle some quite sophisticated modifications and thus resolve the compiler incompatibilities. C Compiler Delegate first uses the Intel compiler to translate the source into Assembler code, before going on to do some post-processing, which we will be looking at in detail in the following sections. Following this, *ccd* tells the GNU Assembler to create object code from the modified Assembler code. The Linker Delegate works in a similar way, using the GNU Linker.

Linux kernels built in this way are quite stable on single and multiple CPU machines with Pentium III and Pentium 4 CPUs. This makes all the effort worthwhile, both as an interesting technical exercise, and in order to leverage the performance advantage and thus avoid spending money on new hardware. Caution is still advised: the ICC-built kernel is still officially a developer version, and admins should perform long-term tests before moving their production systems to the new kernel.

Steps to Building the Kernel

There are three steps to migrating your compiler: first configure the kernel to reflect your current hardware, use GCC to build the kernel, and check whether the kernel works. Make sure that any kernel modules with experimental status run well (see Figures 1 and 2).

The second step is to generally modify the source code to allow the Intel compiler to process the source without any errors (see the "Installing ICC" box). Clean is relative with the Intel compiler: if you keep the default warning level, -*Wall*, the compiler is more pedantic than GCC and will print a flood of warnings about missing type conversions, and pointer arithmetic on *void* and function pointers on your console. The kernel patch uses the *-w* flag instead, to output only genuine compiler errors.

Unfortunately, ICC creates faulty object code for some parts of the kernel. You need to find and resolve these errors to complete phase two. The third and last step involves Intel-specific modifications that use two special characteristics of the Intel compiler: IPO (Inter Procedural Optimization) and PGO (Profile Guided Optimization). PGO instrumentalization of the kernel code is technically challenging, but worthwhile.

As far as I can tell, based on my own experiments, there is one characteristic of modern CPUs that cannot be

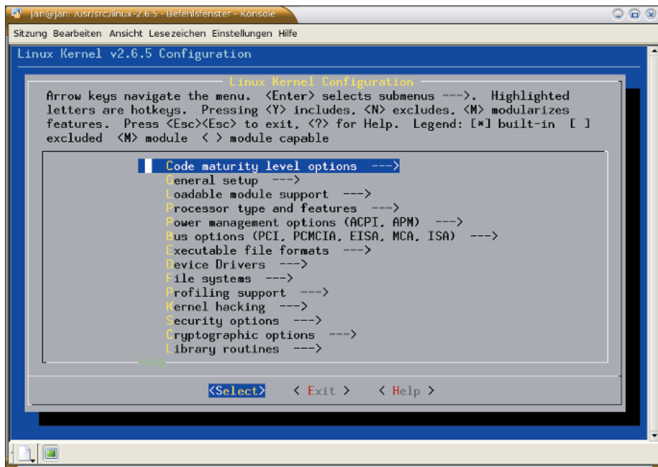

Figure 1: Configure the kernel to reflect your current hardware setup.

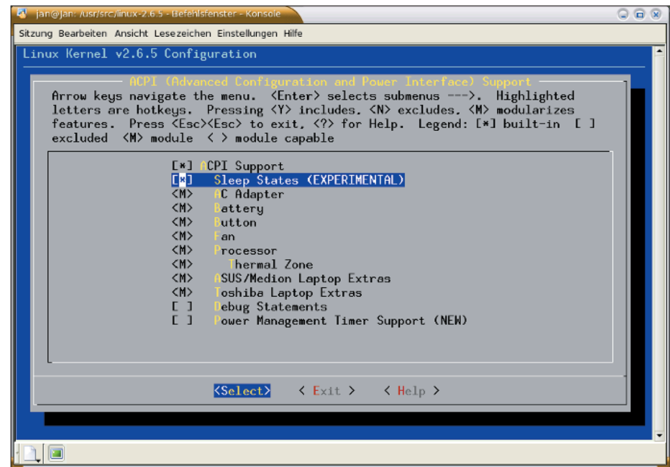

Figure 2: Ensure that experimental kernel modules run with the GCC.

exploited. I was unable to create SIMD (Single Instruction Multiple Data) code for the kernel, using the Intel compiler vectorizer. Refer to the "Why SIMD does not work" box. Even without SIMD, there is enough potential for creating highly optimized kernel code.

Minimally Invasive

The set of patches we will be looking at honors a golden rule of patching: keep the number of modifications to a minimum! This ensures that kernel modifications will migrate to future versions with a minimum of effort. The patch [3] updates a number of kernel makefiles. The *ccd* tool referred to earlier replaces the GNU compiler *gcc* in the top level makefile. The *INSTALL* file from the archive explains how to install the patch.

Some of the Intel compiler's command line options are incompatible to the GCC. This particularly applies to options that control the creation frame pointers, or specify a processor architecture. Again you need to set the right options to disable SIMD instructions for the kernel (in *arch/i386/Makefile*). A few more minor changes will take us to our goal of successfully compiling the kernel.

There is evidence of the Intel compiler's improved compatibility to the GNU compiler. Up to and including version 7, the Intel compiler created empty structures (i.g. for spinlocks), but specified a length of one. The GNU compiler 2.91.x versions had the same error.

There was a workaround that involved pretending to use version 2.91, and adding an element called *gcc_is_buggy* to the purportedly empty structures – in fact, *icc_is_buggy* would have been a more appropriate name. Fortunately, Intel finally got round to removing this bug from version 8 of the compiler, thus removing the need for the workaround.

SIGSEGV

Despite completing all these steps, our kernel is still binary junk in its present state, and it has very restricted functionality. An attempt to load a kernel module with common symbols is doomed to failure. This refers to global variables which are neither static nor initialized. All kernel sources are compiled with the *-fno-common* flag, which the Intel compiler respects. Unfortunately, ICC implements the *-fno-common* option differently. In a post-processing step, the *ccd* tool replaces all common symbols with equivalent constructs and moves the variables to the *.bss* section.

In addition, the Intel compiler has an individual approach to some attributes, such as *__attribute__((used))*. The compiler tags the exported kernel symbol as used, although the kernel code does not reference the symbol. The attribute is a note to the compiler that means "avoid removing variables and functions with this tag during optimizing". The Intel compiler's IPO mechanism cancels the effect of this attribute. The workaround involves introducing dummy access functions, which *ccd* removes prior to assembly.

Loading the *ieee1394.ko* module raises a *SIGSEGV* exception. A quick glance at

Installing ICC

To install Intel's C++ Compiler 8 for Linux, you need at least 100Mbytes of storage space and a glibc version 2.2.5, 2.2.93, or 2.3.2. The first step is to fill out a form on the Intel homepage [1], and agree to the license conditions. The license allows you to compile only private code. Intel will send an email to the address you provide giving you explicit instructions.

After completing the download, you should have a 64MByte tar.gz archive on your disk. The unpacked files include the required RPM files, and the *install.sh* script, which you can call by entering *source*. The installation script uses the *rpm* command, and thus needs root privileges. If you do not have root privileges, or do not use an RPM-based distribution, follow the instructions in *C++ReleaseNotes.htm* on using *rpm2cpio* to dissect the RPM files. Before you launch into the install, make sure you install the license file from the email into the directory that the *INTEL_LICENSE_FILE* variable points to. You can edit the variable, although keeping the default, */opt/intel/licenses*, is recommended.

Configuration

You need to change a few settings after installing *icc*. If the Intel compiler resides in */opt/intel/*, the default settings will be located in */opt /intel/bin/icc.cfg*. You can modify this file to suit your requirements.

Intel provides scripts that automatically set the *PATH* and *LD_LIBRARY_PATH* environment variables. Load the script for your shell before compiling. The following example is for bash:

```
./opt/intel/bin/iccvars.sh
```

Admins of developer machines will want to add these files to the start files for their own shell, for example */etc/bashrc*.

the ELF binary file for the module tells us that the Intel compiler generates faulty relocations for alternative instructions. If you compile a module for a specific processor architecture, and transfer the binary to a similar platform, the Linux kernel will be able to replace existing CPU instructions with equivalent, compatible instructions. Modules of this kind are identified by the ELF sections *.altinstr_replacement* and *.altinstructions*.

Checking *include/asm-i386/processor.h* reveals the cause of the exception. The *prefetch* is declared *extern inline*, but has a function stub. Changing this to *static inline* provides the required relocations. This does not explain why the compiler does not perform at this point.

Mixed Bunch

So far, so good, but we are not finished yet. If you use a kernel built in this way, the next problem is just around the corner. The Ext-2 subsystem fails when attempting to unmount a partition. The reason is the Intel compiler has difficulty processing mixed C/Assembler blocks, and generates faulty object code reproducibly in this case.

This behavior is not restricted to the Linux kernel, as is evidenced by compiling the OpenSSL library 0.9.7d. My own experiments after compiling with the ICC again led to errors. A detailed search led me to a code segment that implemented the RC5 symmetric cipher. It uses inline Assembler to perform bitwise rotation, and this is where things start to go wrong.

It looks like Intel does not support Assembler in C code, instead expecting developers to use the compiler's intrinsic functions to avoid mixed code. In the case of the OpenSSL RC5 algorithm, switching to the intrinsic functions, *_lrotl* and *_lrotr*, did the trick.

Tricks 'n' Traps

Although this workaround is fine for the OpenSSL library, it does not help the Linux kernel. The kernel is so low level that a mixture of C and Assembler is unavoidable at times. The Intel compiler does not provide intrinsic functions that could be used instead. After a search for the error in the Ext-2 subsystem that involved a number of kernel components, I discovered the cause of the issue in the *atomic_dec_and_lock()* function.

Listing 1 shows the section from the C file *arch/i386/lib/dec_and_lock.c*; the faulty Assembler code courtesy of the Intel compiler is shown in Listing 2. The Intel compiler analyzes the inline Assembler parameters incorrectly and applies the *cmpxchgl* command to a local copy of *atomic->counter*, rather than to the counter itself. This is a critical error in the stricter sense of the word, as it means that synchronous access to counters on SMP computers cannot be guaranteed. Resolving this problem means replacing the critical section in *atomic_dec _and_lock* with new Assembler code as soon as the symbol *__INTEL _COMPILER* is defined.

Temporarily fixed: The ICC-built kernel runs without any glitches on the author's three computers:

- a Pentium 4 laptop running at 2GHz with 512MBytes RAM,
- a dual Pentium III Server running at 2x500MHz with 512MBytes RAM and
- a Pentium 4 desktop running at 2.26GHz with 1024MBytes RAM.

Interprocedural Aspects

The kernel in its current state is not noticeably quicker than a GCC-built kernel. The reason is that the Intel compiler's mechanisms, IPO (Interprocedural Optimization) and PGO (Profile Guided Optimization), have not been applied so far. In the course of interprocedural optimization, the Intel compiler decides which functions to expand inline, and if storing temporary results in the CPU registers it will speed up execution flow to an extent that makes the effort worthwhile. Decisions are reached by reference to pre-defined heuristics for the individual processor architectures.

Why SIMD Does Not Work

Both compilers, GCC and ICC, can handle SIMD instructions (Single Instruction Multiple Data) for Intel processors. These commands were introduced step by step for Pentium MMX, Pentium II, and 4 CPUs. Intel distinguishes between MMX (Multimedia Extensions), SSE (Streaming SIMD Extensions), SSE2, and SSE3. The principle is as follows. A SIMD command tells the CPU to process multiple data snippets of the same type at the same time (see Figure 3). These extensions can mean major performance gains for normal applications. The extensions mainly affect floating point arithmetic, but almost any program will contain loops which the compiler can vectorize. The compiler can also analyze the datastream to collate multiple memory access to achieve 128 bit data transfer, as supported by SSE. Reports from users and my own observations indicate that the ICC vectorizer is superior to the GNU counterpart.

These instruction sets should mean performance gains for the Intel compiler in the case of the Linux kernel – but they don't. The background is as follows. The CPU registers used by the new instruction units are the same as the MMX floating point registers. Whenever the Linux kernel changes context, it stores both the general registers and the MMX and SSE floating point registers to allow every process to leverage the extensions.

When changing from the userspace to kernel mode, the entry points store only the general registers. Whenever the kernel program code uses the processor extensions, without saving the appropriate registers, the kernel code overwrites application values without so much as a by your leave. That is fatal for SIMD code in the kernel space.

The *kernel_fpu_begin* and *kernel_fpu_end* functions that protect a SIMD area against preemption, and store the complete FPU/SSE context, are no help here. The Intel compiler generates SIMD code sequences at arbitrary points in the kernel, and without this protection.

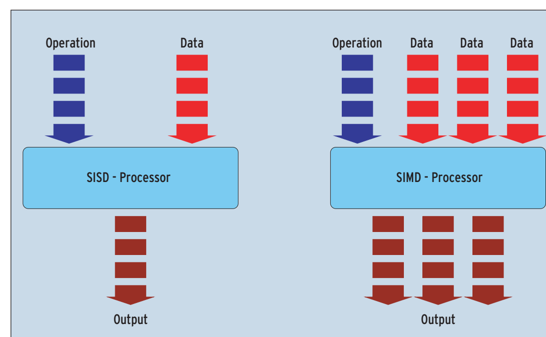

Figure 3: An SISD processor (left) processes a single data field per instruction whereas an SIMD processor (right) processes multiple data at each step.

A combination of options, *-ipo* and *-ipo_obj*, enables the IPO mechanism across all source code. This reveals another program with mixed C and Assembler. The ICC quits with an internal compiler error *(0_(4900+5))* in *include/asm-i386/byteorder.h*. Replace the inline Assembler instructions with intrinsic functions to solve the problem.

IPO Pitfalls

The GNU compiler uses the function attribute *always_inline* for the *inline* keyword, and this is also implemented by the Intel compiler. The kernel sources use this attribute for all inline functions. Every compiler should expand functions with this tag, although this may appear to stand in the way of optimization features. The Intel compiler regards the tag as a friendly note, and relies on IPO decisions instead. The large number of bit manipulation operations in the kernel (defined in *include/asm-i386/bitops.h*) indicates that this was not one of the Intel developers' best design decisions.

Inspecting *System.map* shows that the Intel compiler generates many instances of the function as separate (static) functions, but not as inline functions. The Intel developers surely can not have intended the compiler to create a separate function with prolog and epilog, call and stack cleanup, for an inline function with a single line.

The Intel compiler fights the obvious solution, that is forcing the compiler to expand inline functions by rewriting the inline functions as C macros, tooth and nail. Only a few inline functions can be modified in this way without the compiler generating faulty object code.

Converting every single bit manipulation function into a macro would leave a trail of code destruction. The *ccd* introduced previously helps resolve the problem. The *—assem* option tells *ccd* to generate intermediary Assembler code and store it for inspection in *.c.S* files.

Dreamteam: IPO and PGO

A combination of IPO with profile driven ICC optimization provides the performance boost, when compared with gcc, including version 3.3.3. ICC uses a three-phases compiler model that lets it access information on the execution state, and thus reorder and optimize commands.

To optimize, you need to enable PGO instrumentalization and leave it running for a while. This tells the Intel compiler to generate so-called PGO segment packets and PGO code with a profile of the current program, and storing time controlled PGO segments in files. In the third phase, feedback compilation, you need to feed these files to the compiler to perform optimized compilation.

The *intlpgo* kernel module in the patch discussed in this article makes the Linux kernel PGO-aware. It interacts with the new *pgod* daemon to store profile data in files. The manpages and the source code for both tools provide more information on adding PGO capabilities to the kernel and building both tools [3]. Profile-driven optimization allows the following:

- Analysis of a kernel's behavior during execution,
- specific optimization and
- creation of specialized kernels.

Listing 1: Excerpt from *dec_and_lock.c*

```
01 int
   atomic_dec_and_lock(atomic_t
   *atomic, spinlock_t *lock)
02 {
03   int counter;
04   int newcount;
05
06 repeat:
07   counter =
   atomic_read(atomic);
08   newcount = counter-1;
09
10   if (!newcount)
11     goto slow_path;
12
13   asm volatile(„lock; cmpxchgl
   %1,%2“
14     :“=a“ (newcount)
15     :“r“ (newcount), „m“
   (atomic->counter), „0“
   (counter));
16
17   /* If the above failed,
   „eax“ will have changed */
18   if (newcount != counter)
19     goto repeat;
20   return 0;
21
22 slow_path:
23   spin_lock(lock);
24   if
   (atomic_dec_and_test(atomic))
25     return 1;
26   spin_unlock(lock);
27   return 0;
28 }
```

Listing 2: Assembler *atomic_dec_and_lock*

```
01    .globl atomic_dec_and_lock
02 atomic_dec_and_lock:
03 # parameter 1: 28 + %esp
   (atomic)
04 # parameter 2: 32 + %esp
   (lock)
05 ..B1.1:
06    pushl  %esi
07    pushl  %ebx
08    subl   $16, %esp
09    movl   28(%esp), %ebx   # %ebx
   = atomic
10 ..B1.2:
11    movl   (%ebx), %esi   # %esi =
   atomic->counter
12    movl   %esi, (%esp)   # (%esp)
   = counter
13    addl   $-1, %esi   # %esi =
   newcount
14    je  ..B1.5
15 ..B1.3:
16    movl   (%ebx), %ecx   # %ecx =
   atomic->counter
17    movl   (%esp), %edx   # %edx =
   counter
18    movl   %ecx, 12(%esp)   #
   12(%esp) = DUPLICATE
19       # atomic->counter
20    movl   %edx, %eax   # %eax =
   counter
21       # inline param: „0“
   (counter)
22 # Begin ASM
23    lock; cmpxchgl %esi,12(%esp)
   # %1 = %esi = newcount
24       # %2 = 12(%esp) =
   DUPLICATE
25       # atomic->counter
26 # End ASM
27    cmpl   %edx, %eax
28    jne   ..B1.2
29 ..B1.4:
30    xorl   %eax, %eax
31    addl   $16, %esp
32    popl   %ebx
33    popl   %esi
34    ret
35 ..B1.5:
```

Figure 4 shows an example of the last point. The Intel *codecov* tool generates HTML pages for all source files from the profile data. The files provide data on the executed code blocks, and execution frequency. Restricting the application of the current kernel to specific scenarios, for example desktops, Web or database servers, allows the PGO mechanism to build special-purpose, optimized kernels, which also reflect the execution frequency of device drivers for the current hardware.

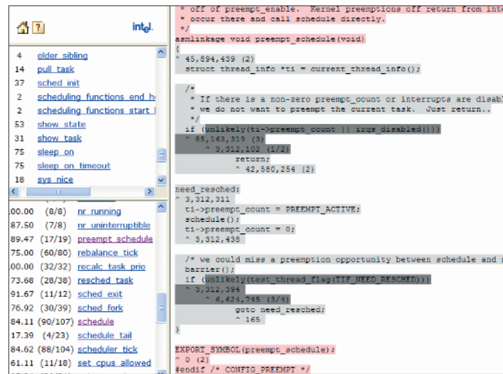

Figure 4: *preempt_schedule* from *sched.c* is one of the most frequently used of all Linux kernel functions. The frequency is shown by colored blocks that indicate which blocks were covered, partially covered, and not executed.

How Big is the Performance Boost?

It is quite difficult to achieve reproducible results that allow us to compare runtime behavior of the kernel builds. Linus Torvalds uses the synthetic LMBench [4] benchmark for checking optimization results. Unfortunately, the benchmark does not provide truly granular results that would allow a genuine comparison. Adding the OProfile [5] kernel module changes this. The module uses Pentium 4 registers that allow an extremely granular performance check at CPU processor cycle level.

The following setup was used to verify the efficiency of this approach:

- Pentium 4 machine running at 2.26 GHz with 1024MBytes RAM
- LMBench and OProfile for benchmarking
- Standard kernel 2.6.5 built with GCC 3.3.3
- Standard kernel 2.6.5 built with ICC 8.0.055 using the patches and tools from [3]
- The benchmark was repeated nine times for each kernel and the results computed.

The kernel with profile-driven optimizations was subjected to various, unspecified loads (file system, network traffic, background and foreground activity) for PGO instrumentalization. Table 1 shows the values measured by the OProfile module as a collection of samples, collected by the NMI (Non Maskable Interrupt) while the benchmark was running. This is the number of processor cycles in non-stopped state for a counter overflow of 22600 on the test system, which corresponds to a counter resolution of about 10 microseconds at a clock speed of 2.26GHz.

Table 1: Benchmark Results

LMBench Test	GCC	ICC	Performance boost
Read file	667.353	641.146	+3.93 %
Read file (Memmap)	355.139	332.860	+6.27 %
Data transfer (Pipe)	51.251	46.309	+9.64 %
Data transfer (TCP/IP Socket)	404.764	404.371	+0.10 %
Data transfer (Unix Socket)	61.422	57.660	+6.12 %
IP connection latency (TCP/IP)	8.727	8.719	+0.09 %
Context change	166.269	168.229	-1.18 %
File system (create/delete)	618.093	611.237	+1.11 %
File (Memmap/unmap)	77.508	70.242	+9.37 %
Page errors (File)	1.409	1.351	+4.12 %
IPC latency (Pipe)	31.224	18.243	+41.57 %
Process generation	12.832	12.325	+3.95 %
IPC latency (Sun RPC)	169.344	113.342	+33.07 %
Select (file/TCP connection)	62.676	60.264	+3.85 %
Signals (install/trigger)	53.297	54.053	-1.42 %
Simple system call	30.970	30.466	+1.63 %
IPC latency (TCP/IP)	62.500	60.700	+2.88 %
IPC latency (UDP/IP)	58.845	57.982	+1.47 %
IPC latency (Unix Socket)	54.854	47.464	+13.47 %

On average, the benchmarks revealed a performance advantage of 8.7 percent for the Intel kernel. This may not seem a lot at first, but it means that you could extend the lifetime of a machine that has reached the limit of its processing capacity by 8.7%, before having to replace it with a more powerful box. Thus, the performance boost does have a noticeable economic effect, above all, for servers and clustered machines.

Some tests showed good performance results: *lat_unix* (interprocess communication via Unix sockets) – over 13 percent, *lat_rpc* (communication via Sun RPC) – 33 percent, and *lat_pipe* (communication via pipes) – 41 percent. The Intel kernel lost out with *lat_sig* (install signal handler, trigger signals) – minus 1.4 percent and *lat_ctx* (context change) – minus 1.2 percent.

The former may be due to the fact that signal code is not frequently iterated – this makes the going tough for PGO. The context change results may be due to the base pointer that the kernel requests when compiling *sched.c*. A one percent degree of imprecision, or less, can be assumed for all results.

Better for Dedicated Tasks

There is no doubt about the general applicability of the comparison thanks to the PGO mechanism, and this also emphasizes the power of the Intel compiler. Based on the results, it seems obvious that no GCC-built kernel would be capable of competing with a purpose built, PGO instrumentalized ICC kernel.

Admins should look into their options for creating high-performance Linux environments for dedicated tasks, such as database servers, by compiling any critical components (kernel, server, libraries) with the Intel compiler using the IPO and PGO options, and running the machine for an extended period under real conditions to support PGO instrumentalization. Feedback compilation will then produce the quickest possible combination. ■

INFO

[1] Intel C Compiler for Linux: *http://support.intel.com/support/performancetools/c/linux/*

[2] Intel document on compiler compatibility: *http://developer.intel.com/software/products/compilers/techtopics/LinuxCompilersCompatibility.htm*

[3] Kernel patch and tools: *http://www.pyrillion.org/linuxkernelpatch.html*

[4] LMBench: *http://www.bitmover.com/lmbench*

[5] OProfile: *http://oprofile.sourceforge.net*