БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ • BULGARIAN ACADEMY OF SCIENCES

ПРОБЛЕМИ НА ТЕХНИЧЕСКАТА КИБЕРНЕТИКА И РОБОТИКАТА, **56**
PROBLEMS OF ENGINEERING CYBERNETICS AND ROBOTICS, **56**

София • 2006 • Sofia

The Impact of Compiling a LINUX Kernel with INTEL C/C^{++} Compiler on Computer Clusters Used by Science

Kalin Lilovski[1], *Nikolay Dokev*[1,2]

[1] *Institute of Information Technologies, 1113 Sofia*
[2] *New Bulgarian University, Sofia*
Emails: kalin@cc.bas.bg n.dokev@nbu.bg

1. Inroduction

Intel C/C^{++} compiler for Linux gives the application developers access to the advanced architecture of Intel Pentium 4 and Intel Xeon processors as well as to Intel Itanium processor family. It is a highly optimizing compiler that generates application code which generally supersedes in performance the one generated by GNU GCC. Intel provides non-commercial license, meaning that anyone can download and use the full compiler for non-profit work. Because of its efficiency and liberal license policy, the Intel C/C^{++} compiler for Linux is often preferred by science for compiling applications that perform heavy computational tasks.

Optimizing the very application is only one of the aspects of achieving high performance. The others refer to the operating system and the hardware. The core of the operating system is its kernel. It is responsible for handling the basic functions of the OS, such as memory management, process and task management, disks and file system management, network communication, etc. Obviously, kernel efficiency is crucial for the overall performance. A suitable hardware solution on the other hand may be a Symmetric Multi-Processing system. Unfortunately such systems are quite expensive and offer no scalability. That is why clusters of inter-connected computers built on commodity hardware are becoming more and more popular.

2. Patching the kernel

The GNU Compiler Collection is a full-featured ANSI C compiler that in addition includes several non-standard features. Intel C/C^{++} compiler supports the ANSI C and C^{++} standards and some but not all of GNU C language extensions [4]. So compiling the kernel with ICC requires a set of patches to be applied.

One of the most popular Linux distributions used by science was RedHat. For example the EGEE (Enabling Grids for E-sciencE) project, in which some institutes from the Bulgarian Academy of Sciences participate, was entirely based on RedHat 7.3. In 2003, Red Hat Inc. announced that Redhat 9 was to be their last release of an operating system that would be freely downloadable in a binary form. Nevertheless, the sources of Redhat Enterprise Linux were available, and it was possible to compile distribution from them. So did Fermilab, CERN, and various other labs and universities around the world who united their efforts in creating Scientific Linux distribution that is binary compatible to RedHat Enterprise with only a few minor additions or changes. Scientific Linux was well accepted by the scientific community and became the new base of the EGEE project. So we decided to concentrate our efforts in patching the RedHat Enterprise Kernel.

RedHat Enterprise Linux uses its own kernel. It is primarily based on a 2.4.21 kernel, but it has a huge number of features backported from the more recent 2.6 kernel. There were some attempts in the past for compiling the 2.4 kernel with icc [1] and patches for some early versions of 2.6 kernel [2, 3]. Unfortunately none of them was completely applicable in the current RedHat Enterprise kernel version that was 2.4.21-27.EL and we had to develop a patch of our own that proved to be not easy but attainable. That process evolves editing of some parts of the kernel that contain GCC extensions and interpreting them with a standard C/C^{++} code. After the patch was successfully done, the next step was to evaluate the performance.

3. Measuring system performance

Tests were performed on:

CPU: Intel(R) Pentium(R) 4
Cpu MHz: 2800
Cache size: 512 kB,
MB: MSI chipset: Intel Corp. 82845 845
MemTotal: 512 MB
HDD: 160 GB ExcelStor Technology J680
NIC 100 Mbit/s Realtek Semiconductor RTL-8139

Software:
OS: Scientific Linux SL Release 3.0.3 (SL)
GCC: gcc version 3.2.3 (Red Hat Linux 3.2.3-42)
ICC: Intel(R) C++ Compiler 8.1 for Linux

Both kernels are compiled with

–march=pentium4

and maximum level of optimization – O2.

System performance is measured using the LMBench utility from BitMover that is a free software program covered by GNU General Public License.

3.1. Processes

We measured the processes performance when making program calls and handling signals as well as the time that it takes to create a basic thread of control.

3.1.1. Program calls

The null call is the most basic call a program can make. This benchmark measures how long it takes for the getppid () function to return the process ID of the parent of the current process. Since the null call is very basic, it is an important indicator of kernel performance.

The null I/O benchmark measures the average of times for a one-byte read from /dev/zero and a one-byte write to /dev/null.

The Stat call (“stat()”) is a call that programs usually make whenever a file’s metadata is accessed. Stat returns information about the file including the access permissions for the file, the date the file was created, last modified, and last accessed. Stat speed depends on the speed of the CPU and the kernel’s efficiency, as well as on the speed of the hard drive as well.

Open/close call measures how long it takes to open () and then close() a file.

3.1.2. Signal handling cost

It measures signal handling by installing a signal handler and then repeatedly sending itself the signal. Note that there are no context switches in this benchmark; the signal goes to the same process that generated the signal:

sig inst measures the time to catch signals;

sig hndl measures the time to handle signals.

3.1.3. Process creation

Process creation test creates processes in three different forms, each one more expensive than the last. The purpose is to measure the time that it takes to create a basic thread of control.

The fork proc benchmark measures the time that it takes to split a process into two identical copies and have one exit.

The exec proc benchmark measures the time that it takes to create a new process and have that new process run a new program which forms the basis of every UNIX command line interface or shell. In this case – a tiny program that prints “hello world” and exits.

The sh proc benchmark measures the time that it takes to create a new process and have that new process run a new program by asking the system shell to find that program and run it. In other words, the shell uses the user’s $PATH variable as a list of places to find the application. It is the most general and the most expensive.

Table 1. Processes (times in µs) the smaller is better

Kernel	Program calls				Signal handling		Process creation		
OS	null call	null I/O	stat	open clos	sig inst	sig hndl.	fork proc	exec proc	sh proc
GCC	0.39	0.43	1.56	2.11	0.67	2.34	110.	434.	1923
ICC	0.39	0.43	1.36	1.89	0.67	2.16	108.	426.	1932

3.2. Basic Integer/ Float/Double operations

Measures the latency of basic CPU operations. Results are reported as the average operation latency divided by the minimum average latency across all levels of parallelism. This benchmark showed no difference in performance.

Table 2. Basic integer, float and double operations (times in ns)

Variable	Bit	Add	Mul	Div	Mod
Integer	0.1800	0.1800	5.0400	20.7	23.1
Float		1.7900	2.5000	15.5	
Double		1.7900	2.5000	15.5	

3.3. Local communications

Local communication benchmark includes inter-process communication latencies, inter-process communication bandwidth and memory performance

3.3.1. Inter-Process Communication Latency

Passing a small message (a byte or a word) back and forth between two processes. No other work is done in the processes. This sort of benchmark is frequently referred to as a “hot potato” benchmark.. The time reported is one round trip.

Pipe benchmark passes a token back and forth between the two processes.

AF UNIX measures inter-process connection latency via UNIX sockets.

TCP measures inter-process communication latency via TCP/IP.

TCP conn benchmark times the creation of an AF_INET (aka TCP/IP) socket to a remote server.

Table 3. Local Communication latencies (in μs) the smaller is better

OS	Pipe	AF Unix	TCP	TCP conn
GCC	4.459	7.29	11.5	34.
ICC	4.289	6.82	11.0	32.4

3.3.2. Inter-Process Communication Bandwidth

Pipe creates an UNIX pipe between two processes and moves 50MB through the pipe in 64KB chunks.

AF UNIX creates a pipe and forks a child process which keeps writing data to the pipe as fast as it can. The benchmark measures how fast the parent process can read the data from the pipe. Nothing is done with the data in either the parent (reader) or the child (writer) processes.

TCP time data movement through TCP/IP sockets. It is a client/server program that moves data over a TCP/IP socket. Nothing is done with the data on either side.

Table 4. Local Communication bandwidths in MBps – the bigger is better

OS	Pipe	AF Unix	TCP
GCC	1498	2703	639
ICC	1480	2568	600

3.3.3. Cached file read

File Reread measures the time of reading and summing of a file. It times the reading of the specified file in 64KB blocks. Each block is summed up as series of 4 byte integers in an unrolled loop. Results are reported in megabytes read per second. The benchmark is intended to be used on a file that is in memory, i.e., the benchmark is a reread benchmark.

Mmap Reread measures **the** time of reading and summing of a file. bw_mmap_rd creates a memory mapping to the file and then reads the mapping in an unrolled loop. The benchmark is intended to be used on a file that is in memory, i.e., the benchmark is a reread benchmark.

3.3.4. Memory Bandwidths

Allocates twice the specified amount of memory, zeroes it, and then times the copying of the first half to the second half. Results are reported in megabytes moved per second.

Memory copy

Measures how fast the system can bcopy data. Bcopy copies *n* bytes from a string source to string destination.

An 8 MB to 8 MB copy does not fit in the cache Kernel bcopy and C library bcopy.

Memory read/write

Read: Measures the time to read data into the processor. An unrolled loop that sums up a series of integers.

Write: Measures the time to write data to memory. An unrolled loop that stores a value into an integer.

Table 5. Local Memory Communication bandwidths in MBps – the bigger is better

Kernel	File Reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
GCC	1462.6	1861.6	450.0	471.9	1865	668.3
ICC	1448.3	1861.4	454.9	479.3	1864	659.2

3.4. Context switching

The processes are connected in a ring of UNIX pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. Context-switch time doesn't include the overhead of doing the work.

Table 6. Context switching (times in μs) the smaller is better

Procs	2	4	8	16	24
GCC 0k	0.94	1.38	1.35	1.46	1.49
ICC 0k	0.98	1.33	1.40	1.34	1.48
GCC 4k	1.17	1.77	1.78	1.85	2.38
ICC 4k	1.19	1.79	1.83	2.03	2.63
GCC 8k	1.17	1.66	1.69	2.23	4.26
ICC 8k	1.26	1.74	2.19	2.70	3.71

Processes may vary in number. Smaller numbers of processes result in faster context switches. Processes may vary in size. A size of zero is the baseline process that does nothing except pass the token on to the next process. A process size greater than zero means that the process is doing some work before passing on the token. The work is simulated as the summing up of an array of a specified size. The summing is an unrolled loop of about 2.7 thousand instructions.

3.5. File system

File Create Delete creates a number of small files in the current working directory and then removes the files. Both the creation and removal of files is timed.

Mmap latency benchmark maps in and unmaps the first size bytes of the file repeatedly and reports the average time for one mapping/unmapping.

Prot Fault measures the time to catch a protection fault.

Page fault measures the cost of page-faulting pages from a file. The output is the average cost of page – faulting a page.

100 fd selct measures the time to do a selection on *n* file descriptors. In the summary, the result of 100 file descriptors is shown.

Table 7. File & VM system latencies in µs – the smaller is better

OS	0K File		10K File		Mmap Latency	Prot Fault	Page Fault	100fd selct
	Create	Delete	Create	Delete				
GCC	7.4734	3.5432	39.2	189.7	2582.0	0.684	1.69660	3.068
ICC	4.7612	3.2211	34.6	162.4	2582.0	0.770	1.68510	2.475

3.6. Memory latencies

Measures memory read latency for varying memory sizes and strides.

The entire memory hierarchy is measured, including onboard cache latency and size, external cache latency and size, main memory latency, and TLB miss latency.

Only data accesses are measured; the instruction cache is not measured.

The size of the array varies from 512 bytes to (typically) eight megabytes. For the small sizes, the cache will have an effect, and the loads will be much faster. This becomes much more apparent when the data is plotted.

Table 8. Memory latencies in ns – the smaller is better

OS	L1 $	L2 $	Main mem	Rand mem
GCC	0.7150	6.5490	94.0	152.7
ICC	0.7150	6.5480	93.8	164.1

4. Measuring network performance

Network performance test was done between two directly connected nodes. As MPICH is the standard communication library used by clusters, we measured throughput and latency of each package using MPPTEST tool included in MPICH distribution. MPPTEST performs point to point communications that is basically the classic ping-pong test of messages with different size, repeated several times. Network latency was evaluated by repeating 4 times a sequence of round trip messages from 0 up to 64 bytes with increment of 4 bytes (mpirun –np 2 mpptest –reps 4 –size 0 64 4) and throughput by messages from 0 up to 16000 bytes with increment of 4 bytes (mpirun –np 2 mpptest –reps 4 –size 0 16 000 1000).

Table 9. MPI Network latencies (time in μs)

Bytes	GCC	ICC
0	36.390	36.210
4	36.830	36.610
8	37.140	36.910
12	37.550	37.360
16	37.900	37.740
20	38.260	38.050
24	38.570	38.400
28	38.990	38.790
32	39.360	39.210
36	39.710	39.550
40	40.050	39.840
44	40.480	40.350
48	40.970	40.830
52	41.420	41.210
56	41.700	41.500
60	42.190	42.020
64	42.630	42.500

Table 10. MPI Network Bandwidth in μs (Mbps)

Bytes	GCC	ICC
0	0.000	0.000
1000	7.862	7.872
2000	9.128	9.128
3000	9.982	9.986
4000	10.447	10.447
5000	10.512	10.515
6000	10.900	10.900
7000	10.943	10.943
8000	10.919	10.921
9000	11.161	11.165
10000	11.152	11.152
11000	11.113	11.115
12000	11.271	11.274
13000	11.181	11.186
14000	11.232	11.236
15000	11.351	11.351
16000	11.315	11.315

5. Conclusion

Compiling the 2.4.21-27.EL kernel with Intel C/C^{++} Compiler provides improvement in some, but not all system characteristics. Basic numerical operations are not effected. In context switching benchmark the kernel compiled with GCC supersedes the one compiled with ICC in many tests with less than 24 processes. Program calls, signal handling and process creation (except sh) are unaffected or improved. There is slight improvement in local communication latencies and slight deterioration in local communication bandwidths. The file system shows the most noticeable performance boost. The network performance improvement in network communication trough MPI is also sustainable and shows improvement in both latency and bandwidth. We mustkeep in mind that benchmark programs are giving only a general picture and for some specific applications there still might be some noticeable changes in performance.

References

1. Ookubo, Katuhiko. The Linux Kernel Build and Performance Evaluation with the Intel C^{++} Compiler. – In: 49th Linux Seminar (2003/08/01), Linux Network Japan.
http://www.linet.gr.jp/lswg/contents/index.html
http://www.suri.co.jp/~ohkubo-k/linux/icclinux.pdf
2. White Paper: Kubbilun Ingo A. Compiling the LINUX Kernel 2.6 Using INTEL® C/C^{++} COMPILER for LINUX 8.0. Version 0.2, 06/18/2004.
http://www.pyrillion.org/downloads/icckernpatch.pdf
3. Kubbilun, Ingo, A. Compiling the LINUX Kernel with the Intel Compiler. – LINUX Magazine, Issue **45**, August 2004.
http://www.linux-magazine.com/issue/45/Intel_C_Compiler.pdf
4. White Paper. John O'Neill, Software Products Division Intel Corporation "Intel® Compilers for Linux. Compatibility with GNU Compilers".
5. Didem, Unat. Performance Analysis of Supercomputers in NCSA with LMbench.
http://netfiles.uiuc.edu/dunat2/www

Влияние компиляции ядра LINUX при помощи INTEL C/C^{++} компилятора на компьютерные кластеры, применяемые в науке

Калин Лиловски [1], *Николай Докев*[1,2]

[1] *Институт информационных технологий, 1113 София*
[2] *Нов Болгарский университет, София*
E-mails: kalin@cc.bas.bg , n.dokev@nbu.bg

(Резюме)

Компилятор Intel C/C^{++} для LINUX (ICC) производит высокоэффективный код, который оптимизиран для процессорной фамилии Intel. Во многих случаях улучшение поведения применений, компилированных при помощи ICC, значительно. Возможно модифицировать LINUX ядро так, что компилировать его при помощи ICC. Если такое ядро может заменить построенное при помощи GNU Compiler Collection (GCC), это будет важно для компьютерных систем, которые испитывают значительную вычислительную и сетевую нагрузку при использовании компьютерных кластеров, выполняющих научные вычисления. Работа сравнивает поведение RedHat Enterprise LINUX ядра, используемого в Scientific LINUX распределении, компилированного при помощи GCC и ICC тоже, учитывая поведение системы и сети в кластерной среде.