The seven second kernel compile

Anton Blanchard

IBM OzLabs Linux Technology Center, <anton@au.ibm.com>

Kernel hackers like to optimise for the common case, and what could be more common to them than compiling kernels.

Timing a kernel compile is often a useful (if somewhat unscientific) metric when making Linux kernel changes. It has been used more recently to identify scalability problems - thus began the kernel compile benchmark wars.

This paper looks at the work required to achieve a seven second kernel compile using the 2.5 Linux kernel on a 32 way PowerPC64 machine as well as the ongoing scalability work to go below seven seconds on this important benchmark (this is a preliminary version, the final version for the Congress may still suffer some changes. The Editors).

Table of Contents

1. Introduction

The kernel compile benchmark is a benchmark often used by Linux kernel developers to assess the performance of changes they make. It has the the advantages of being both easy to run as well as having reasonably repeatable results.

Although heavily CPU bound, it stresses a number of areas in the kernel, like the process, filesystem and virtual memory subsystems.

2. Let the wars begin

Martin Bligh kicked the latest round of kernel compile benchmarks off with the following email:

```
From: Martin J. Bligh <fletch@aracnet.com>
To: lse-tech@lists.sourceforge.net
Cc: linux-kernel@vger.kernel.org
Subject: [Lse-tech] 23 second kernel compile (aka which patches
         help scalibility on NUMA)
Date: Fri, 08 Mar 2002 21:47:04 -0800



"time make -j32 bzImage" is now down to 23 seconds.
(16 way NUMA-Q, 700MHz P3's, 4Gb RAM).
...
```

Martin managed to reduce the compile time from 47 seconds on the standard 2.4.18 kernel down to 23 seconds with the help of a number of patches - an impressive achievement. His email served to highlight the current problems with the 2.4 kernel on large NUMA machines and the gains that could be made by adding NUMA memory allocation and scheduling policies into the kernel.

3. Benchmark Hardware

The PowerPC benchmarking was undertaken on an IBM pSeries p690 server. These are POWER4 based servers which implement the PowerPC AS architecture. From a userspace point of view they are binary compatible with the 32 bit PowerPC architecture and existing PowerPC Linux distributions can run on these machines with only a change of kernel.

3.1. POWER4 architecture

A POWER4 chip contains two processors which share a level 2 cache, running at frequencies of up to 1.3GHz. Four of these chips can be combined into a multi chip module (MCM) forming an eight way SMP. There is an each way interconnect between these four chips and the level 3 cache; memory and IO sits behind the MCM.

A p690 combines up to 4 MCMs to form a 32 way SMP and a bus connects these MCMs together. [1]

3.2. Logical Partitioning

Logical partitioning allows a single machine to run a number of operating systems. For example a p690 can run up to 16 different operating system instances, either AIX or Linux. [2]

Logical partitioning introduces a layer above the operating system called a hypervisor. This provides a virtualisation layer between the hardware and operating system, allowing it to share resources. It also provides protection between operating systems such that one operating system cannot harm any of the others.

The p690 can run with logical partitioning on or off and all but the final run was done with it enabled.

4. The PowerPC64 empire strikes back

An initial run on a 24 way p690 resulted in the following post:

```
From: Anton Blanchard <anton@samba.org>
To: lse-tech@lists.sourceforge.net
Cc: linux-kernel@vger.kernel.org
Subject: [Lse-tech] 10.31 second kernel compile
Date: Wed, 13 Mar 2002 19:52:17 +1100



Let the kernel compile benchmarks continue!



hardware: 24 way logical partition, 1.1GHz POWER4, 60G RAM
kernel: 2.5.6 + ppc64 pagetable rework
kernel compiled: 2.4.18 x86 with Martin’s config
compiler: gcc 2.95.3 x86 cross compiler



# MAKE="make -j14" /usr/bin/time make -j14 bzImage
...
130.63user 71.31system 0:10.31elapsed 1957%CPU
        (0avgtext+0avgdata 0maxresident)k
```

Thus the yardstick was 10.3 seconds. A number of interesting results could be distilled from the above information:

- Averaged over the entire run, less than 20 CPUs (19.57) were used

- The benchmark was significantly system bound (130.64 user vs 71.31 system)

While most of the benchmark could be split across all CPUs, the final link stage was single threaded. This explained why we saw only 20 CPUs utilised. To answer the second question however, a closer look at the kernel was required.

4.1. A look inside the kernel

A simple but often very effective way of obtaining a breakdown of system time on a Linux system is to use the builtin profiler. To enable it a command line option is passed to the kernel: `profile=2` where 2 is the shift count applied to the address of each sample. As all instructions in the PowerPC64 architecture are 4 bytes, a shift count of two will result in a profile with single instruction resolution.

A userspace utility: `readprofile` is used to read the profiling information. The results showed a severe problem in `__hash_page` and to a lesser degree `local_flush_tlb_range` and `local_flush_tlb_page`:

```
 201150 total
 129051 idled
  43586 __hash_page
   6714 local_flush_tlb_range
   2773 local_flush_tlb_page
   2203 do_anonymous_page
   2059 lru_cache_add
   1379 __copy_tofrom_user
   1220 hpte_create_valid_pSeriesLP
   1039 save_remaining_regs
    871 do_page_fault
    575 plpar_hcall
```

In the above readprofile output, column one shows the number of timer hits and column two shows the function that was being executed when the timer tick fired. Each CPU has its own local timer, so these results give a summary of where all CPUs spent their time.

An extension to readprofile by Andrew Tridgell takes the per instruction profiling information and places it on top of a disassembly of the function. In this way timer ticks can be attributed to particular instruction sequences and then back to the source code in question.

As a result of this analysis, the problem with `__hash_page` was clear. It was caused by contention on the `hash_table_lock` spinlock. In order to understand what the `hash_table_lock` protects, a quick background in Linux memory management is required.

5. 2.5 Development

5.1. Linux memory management

The 2.5 Linux can run on 13 architectures and so must have a clean abstraction that every memory management unit can be hidden behind. Linux does this by always using a tree representation for pagetables even when the hardware does not. On some architectures like x86, the tree representation is shared by the hardware, whereas on other architectures it is used by Linux only.

PowerPC64 has a hashtable based MMU which is not compatible with the Linux pagetable approach. As a result the PowerPC64 hashtable is treated as an extended TLB with Linux software pagetables backing it. Whenever a Linux pagetable entry is modified, architecture specific code is called to keep the PowerPC64 hashtable in sync. The architecture specific code must provide its own locking since the generic code does not.

5.2. PowerPC64 memory management rework

In the 2.4 kernel, all operations on the PowerPC64 hashtable are serialised with a global lock. This lock is the `hash_table_lock` and as seen above is a significant bottleneck on large SMP.

While the global spinlock approach is easy to verify it was obvious that a different approach was required to improve SMP scalability in 2.5. The approach taken was to use a spare bit in each hashtable entry as a per PTE lock. Each operation on a hashtable entry first atomically sets this bit and if it succeeds it can continue to modify the entry. If it fails to atomically set the bit it must back off because another CPU is currently modifying the entry.

5.3. TLB invalidation batching

The PowerPC64 architecture requires a sequence of instructions to invalidate a TLB entry:

```
ptesync
tlbie
eieio
tlbsync
ptesync
```

The ptesync, eieio, tlbsync and ptesync are all synchronising instructions. The actual TLB invalidation is done with the tlbie instruction. This five instruction sequence can take a significant amount of time and it is possible to batch invalidations up:

```
ptesync
tlbie 1
tlbie 2
...
eieio
```

```
        tlbsync
        ptesync
```

In doing so we incur the overhead of the synchronising instructions once. Support in the generic Linux kernel is required to batch TLB invalidations and the 2.5 kernel has something called a “TLB gather” which does just this.

The PowerPC64 architecture also requires there to be only one outstanding TLB invalidate on the system at any time. As such all invalidations have to be serialised by a global spinlock.

There are two potential problems with such a global spinlock. Firstly if many CPUs are trying to get the spinlock, they will waste CPU cycles waiting for the lock to become free. Clearly nothing can be done about this problem since it is required by the architecture.

The second problem with a global spinlock is the potential for it to be bounced from CPU to CPU. It can take a large amount of time for a lock to move from one CPU to another and so a lock with an otherwise low amount of contention can become a bottleneck.

The solution for the second problem is to reduce the number of spinlock acquisitions. The TLB batching interface above was modified so the spinlock was taken once at the start of the invalidations and dropped at the end. This addresses the problem with `local_flush_tlb_range` and `local_flush_tlb_page`.

This set of changes plus the addition of eight more CPUs gave us a respectable 7.52s compile:

```
From: Anton Blanchard <anton@samba.org>
To: lse-tech@lists.sourceforge.net
Cc: linux-kernel@vger.kernel.org
Subject: [Lse-tech] 7.52 second kernel compile
Date: Sat, 16 Mar 2002 17:15:35 +1100



> Let the kernel compile benchmarks continue!



I think Im addicted. I need help!



In this update we added 8 CPUs and rewrote the ppc64 pagetable management
code to do lockless inserts and removals (there is still locking at
the pte level to avoid races).
```

```
hardware: 32 way logical partition, 1.1GHz POWER4, 60G RAM
kernel: 2.5.7-pre1 + ppc64 pagetable rework
kernel compiled: 2.4.18 x86 with Martin’s config
compiler: gcc 2.95.3 x86 cross compiler
```

```
make[1]: Leaving directory ‘/home/anton/intel_kernel/linux/arch/i386/boot’
128.89user 40.23system 0:07.52elapsed 2246%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (437084major+572835minor)pagefaults 0swaps
```

```
...
```

6. Current Record

The latest results were done on a 32way pSeries p690 with 1.3GHz CPUs and logical partitioning turned off (only one operating system was running):

```
105.02user 14.50system 0:04.83elapsed 2474%CPU
       (0avgtext+0avgdata 0maxresident)k0inputs+0outputs
       (394245major+570713minor)pagefaults 0swaps
```

4.8 seconds should keep even the most impatient of kernel hackers happy.

7. Conclusions

The kernel compile benchmark has been a useful benchmark for isolating PowerPC64 Linux scalability issues. With the latest results the challenge is now open for other architectures to better it.

And finally, its cool to have the worlds fastest Linux kernel compile machine :)

8. References

1. Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le and Balaram Sinharoy. POWER4 System Microarchitecture. IBM Server Group, October 2001 (http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html)
2. IBM Corporation. Partitioning for the IBM eserver pSeries 690 System, 2001 (http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/lpar.html).