Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Using Hugetlbfs for Mapping Application Text Regions

H.J. Lu
Intel Corporation
hongjiu.lu@intel.com

Kshitij Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Rohit Seth*
Google Inc.
rohitseth@google.com

Jantz Tran
Intel Corporation
jantz.c.tran@intel.com

Abstract

Many enterprise applications such as Database and File- and Application-Servers have large text and data footprints. For efficient execution, these applications need the processor to efficiently cache address translations for many text and data pages. Translation Lookaside Buffers (TLBs) are a very critical resource on any processor and all effort should be made to use them as optimally as possible. Linux kernel uses huge TLBs (x86, IA-64, etc.) for mapping its own text and data. HUGETLBFS support in Linux allows the use of huge TLBs (for example 2M/4M on x86, 256MB on IA-64) for mapping an application's dynamic data. In this paper we will describe an approach that leverages HUGETLBFS support in kernel for mapping a program's text region. We will detail the modifications applied to different components (Linux kernel, glibc and binutils) for this solution, and discuss the performance improvements it delivers on an industry standard transaction processing workload.

*Work was done while working at Intel.

1 Introduction

Data footprints for many enterprise workloads range from several tens of megabytes to a few terabytes. For supporting these workloads efficiently, it is critical to deploy large amounts of primary memory to effectively cache their data working sets. Accesses distributed over wide ranges of primary memory need to be translated efficiently as well, and to do so with the limited translation resources on a processor, many systems use large granular page mappings so that a single translation resource can map a wide range of contiguous data. Significantly, over successive releases many enterprise applications have grown steadily in code size and thus the use of large grained instruction address translation for their efficient execution has also become attractive to explore. This paper describes how to extend the use of HUGETLBFS mappings in Linux, so that in addition to mapping large ranges of data, it is possible also to cover large spans of program text with few address translation resources.

The paper is organized as follows. As background, Section 2 briefly goes over the rationale for employing HUGETLBFS to map data, and offers reasons for using it to map text as

well. Section 3 describes the changes to program linking and loading mechanisms in order to accomplish code placement in large pages. Section 4 illustrates the use of the mechanisms with an example. Section 5 discusses the performance impact, using an industry standard workload for measurement and analysis. Sections 6 relates current status and planned work, and Section 7 concludes the paper.

2 Use of Large Grained Translations

Enterprise software systems manage vast amounts of data, maintained usually in complex and highly interconnected information sets. They also implement many layers of sophisticated and concurrent processing of the information they manage, and are designed for large scale and mission critical use. Such systems, which include for example, database management systems, groupware backends, web servers, and, supply chain and workflow systems, commonly need to touch data spread across large amounts of secondary or tertiary storage. Generally these systems are configured with large amounts of physical memory, in order to achieve efficient buffering of I/O. Accesses to the primary memory, for fetching either instructions or data, are themselves accelerated by high speed caches that retain recently used information close to the processors. It is common for present day machines used for data warehousing to be configured with several hundred gigabytes of physical memory.

Resource and time efficient addressing of these large physical memories is also critical to achieiving good performance. Modern processors implement small, high speed translation caches, also called Translation Lookaside Buffers (TLBs), to reduce the time it takes to translate the virtual page addresses for data and instructions to corresponding physical page addresses. Programs with good locality of access benefit from TLB use considerably, while programs with sparse memory reference characteristics suffer high TLB miss rates and run less efficiently as a result. Increasing the number of the TLBs in order to improve their hit rates is not a satisfactory solution, as it drives up their complexity and the number of clocks it takes to produce translations, and also contributes to power consumption [7].

The number of TLBs available on most processors is generally much smaller than the number of normal sized pages needed to cover the large data working sets of most enterprise applications [2], [3], [7]. To remedy this situation, superpages or huge-pages—which map physical memory in much larger grained units than ordinary pages—are now supported by most processors. Beginning with the 2.6 kernel, the Linux operating system introduced `HUGETLBFS`, a pseudo-file system through which appropriately privileged entities can map their data in hugepages [3], [4].

In addition to having large data footprints, these software systems also have large text working sets, characteristic of their inherent complexity. It is common, for example, for a database management system to have a text footprint of a few hundred megabytes, and a text working set ranging from several hundred kilobytes to a few megabytes. In covering these code ranges with ordinary pages, such software stresses the translation caches of a modern microprocessor. In out-of-order processor pipelines, the resulting stalls pose a serial bottleneck due to the in-order instruction issuing front ends [1]. On simultaneously multi-threaded engines such as Intel's hyperthreaded processors [5], the division of TLBs among the logical processors sharing a core further reduces the number of TLBs available to each logical processor.

The demand for TLBs is further amplified

by the frequent need to support text working set mappings among several concurrent processes that do not share a common address space even as they share the physical pages that hold the text. For superior performance and resource sharing, applications that share the same text image might ideally be recast as threads; but frequently other considerations—fault isolation, recoverability, and deployment flexibility—make the concurrent process model a preferred approach. In the applications that use multiple process instances that share text, the use of large-grained text mappings also reduces the amount of page table memory consumed—multiple times, once for each process instance. These factors make it very desirable to extend the benefits of large grained translations to text regions.

In addition to kernel static data, the Linux kernel arranges to have its own text also placed into large pages. In order to allocate and use large pages from HUGETLBFS for the purpose of mapping the text of an enterprise application, we need to similarly shape the text layout in the application's address space. Currently the HUGETLBFS support in Linux does not provide a transparent way to let user applications use huge pages for mapping program text. The solution to this problem is described in the next section.

3 Large Text Page Implementation

In current Linux, the kernel lays out program text according to directions encoded into the executable by the link editor. Once the kernel completes the mapping of a program's segments, it passes control to the runtime linker to complete dynamic resolutions and initializations. Because of this split responsibility between the kernel and the dynamic linker, changes are required to each in order to use HUGETLBFS for text mappings in a natural way. We considered the alternative of working with an unmodified kernel and repealing its actions later in order to relocate the desired segments to large text mappings, but refrained from pursuing it as we found it cumbersome, error-prone and maintainence risk.

To create a different layout, our approach is to capture the placement directive at program linkage time. The placement directive indicates whether the application developer prefers that the program text be laid out in large pages. Section 3.1 describes the changes to the application binary interface (ABI) and to the linker, to accomplish this objective. We then act upon this direction at program load time. The kernel is extended very modestly. The kernel change, described in Section 3.2, allows it to defer code placement for the indicated segments.

The bulk of the modifications are limited to the dynamic linker, and are described in Section 3.3. In Section 3.4, we describe compatible execution of binaries compiled for large page placement of code, under the conditions that large pages are either unavailable or the target system does not contain the changes described here.

3.1 ABI and Linker Additions

We added a new segment type, PT_GNU_HUGE_PAGE, to the program header, in order to specify the location of a huge page text segment. An executable cannot have more than one PT_GNU_HUGE_PAGE segment. A PT_GNU_HUGE_PAGE segment, if present, must precede any PT_LOAD segments in the program header. The PT_GNU_HUGE_PAGE segment, which is aligned at the huge page size boundary, has a corresponding PT_LOAD segment, which is aligned normally.

A new linker option, `-z huge`, is added, which will create a `PT_GNU_HUGE_PAGE` segment in executable.

3.2 Kernel

We added three entries to the auxiliary vector: `AT_EXECFILENAME`, `AT_HUGEPAGESZ`, and `AT_HUGEPAGEPHDR`. `AT_EXECFILENAME` specifies the absolute pathname of the program. `AT_HUGEPAGESZ` specifies huge page size and `AT_HUGEPAGEPHDR` provides the address of the `PT_GNU_HUGE_PAGE` segment entry.

When it sees a `PT_GNU_HUGE_PAGE` segment, the kernel does not map in the corresponding `PT_LOAD` segment. Instead, it writes the address of the `PT_GNU_HUGE_PAGE` segment entry into `AT_HUGEPAGEPHDR` and the huge page size into `AT_HUGEPAGESZ`. The kernel also places into `AT_EXECFILENAME` the absolute pathname of the program, before transferring control to user.

3.3 Dynamic Linker

We modified the run-time start up code to recognize the new segment types and take corresponding actions. In the following we describe the sequence of actions from the dynamic linker under these modifications:

- Locate the `PT_GNU_HUGE_PAGE` segment by checking `AT_HUGEPAGEPHDR`. If it is not available, continue with normal processing instead of going through the steps listed below.
- Check the environment variable, `LD_GNU_HUGE_PAGE_FS`, for the mounting point of huge page file system. That directory, if specified, is used instead of the default directory of huge page file system.
- Get the absolute pathname of the executable from `AT_EXECFILENAME`, and open it for processing.
- If the huge page file system is not configured, or cannot furnish pages, then map the segment identified by `PT_GNU_HUGE_PAGE` as a normal segment, and revert to normal processing.
- Lock the original executable exclusively to prevent other processes from mapping its `PT_GNU_HUGE_PAGE` segment.
- If a shadow text file does not exist for the `PT_GNU_HUGE_PAGE` segment, create a shadow text file on the huge page file system with the same permissions as those of the original executable. We use the `<device_id, inode_num>` identity as the part of the pathname for the shadow text file.
- Map and copy the `PT_GNU_HUGE_PAGE` segment to the shadow text file, if either there is not a pre-existing shadow text file, or the original executable has changed. If this map-and-copy attempt fails for any reason, then unlock the executable and map the segment as a normal segment, and revert again to non-special handling instead of continuing as listed below.
- Map the shadow text file in accordance with the flags that are associated with the `PT_GNU_HUGE_PAGE` segment. Again, if there is an error from the mapping attempt, then unlock the executable, and map the `PT_GNU_HUGE_PAGE` segment as a normal segment to continue with normal processing.
- Close the shadow text file, set its time stamps to match those of the executable, and unlock the executable.

After the PT_GNU_HUGE_PAGE segment has been processed, the dynamic linker closes the executable.

3.4 Compatibility

The above changes are relatively minor, forward compatible, and mostly backward compatible as clarified next. An application for whose text large pages are not desired can be compiled with either the original or the modified link editor in the ordinary way. Such an application is processed uniformly as before, by either the original or the new kernels and dynamic linkers. An application that is compiled with the new link editor and which is linked to request large text pages is handled correctly by an unmodified kernel on the target system with one exception. The exception is in case of Intel® Itanium®: here, the constraints of Intel® Itanium®'s HUGETLBFS implementation compel us to use a modified kernel and a modified linker in order to process a binary that uses the extended ABI.

Another unavoidable departure from compatible execution is the following. An application that has a PT_GNU_HUGE_PAGE segment cannot run correctly if the kernel supports PT_GNU_HUGE_PAGE segment but the dynamic linker doesn't. We consider it a modest requirement that the dynamic linker must be in step with the changes in the kernel.

4 Usage Example

In this section, we show a simple example to illustrate the use of HUGETLBFS code placement and execution on x86-64. We describe this example in two parts. In the first part, we show the construction of the executable using the huge directive. In the second part, we demonstrate the effect of executing the program, first with, and then without, the privilege for allocating HUGETLBFS memory.

The program is shown below as example.c. It is coded merely to list the mapping under /proc for its own text segment, and is intended to illustrate the behavior both when the text segment is mapped as desired (in memory furnished from HUGETLBFS) and when it is mapped in ordinary pages.

```
$ cat example.c
#include <stdio.h>
#include <stdlib.h>

int main () {
  char buf [120];
  printf ("Huge page text segment"
          "map:\n");
  sprintf (buf, "grep 00600000- "
           "/proc/%d/maps | "
           "sed -e \"s/ \\{26\\}//\"",
           getpid ());
  system (buf);
  return 0;
}
```

We next compile the program as shown below. Note the use of huge directive during linking.

```
$ gcc -O   -c -o example.o example.c
$ gcc -Wl,-z,huge,-s -o hugex example.o
```

Next we examine the program headers and the section to segment mapping, using readelf,

```
$ readelf -l --wide hugex
```

For better readability, we select a subset of the information emitted by readelf, below:

```
Program Headers:

Type           VirtAddr MemSiz Flg Align
PHDR           0x400040 0x0230 R E 0x8
INTERP         0x400270 0x001c R   0x1
GNU_HUGE_PAGE  0x600000 0x031c R E 0x200000
LOAD           0x400000 0x0500 R E 0x100000
LOAD           0x600000 0x031c R E 0x100000
LOAD           0x800000 0x0220 RW  0x100000
DYNAMIC        0x800028 0x0190 RW  0x8
NOTE           0x40028c 0x0020 R   0x4
GNU_EH_FRAME   0x600258 0x0024 R   0x4
GNU_STACK      0x000000 0x0000 RW  0x8
```

```
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .text .init .fini .rodata .eh_frame_hdr
   .eh_frame
03 .interp .note.ABI-tag .hash .dynsym .dynstr
   .gnu.version .gnu.version_r .rela.dyn
   .rela.plt .plt
04 .text .init .fini .rodata .eh_frame_hdr
   .eh_frame
05 .ctors .dtors .jcr .dynamic .got .got.plt
   .data .bss
06 .dynamic
07 .note.ABI-tag
08 .eh_frame_hdr
09
```

One can see from the program header and section-to-segment mapping details that segments 2 and 4 map to the same set of sections. The `GNU_HUGE_PAGE` type for segment 2 identifies it as the huge page segment that was requested at link time, while the ordinary LOAD type for segment 4 identifies it as the normal segment that is provided for backward compatibility.

The `HUGETLBFS` file system is mounted at `/mnt/hugepagebydefault`; but if not, in this example we proceed to mount it:

```
# mount none -t hugetlbfs
    /mnt/hugepage
# mount | grep -i hugetlb
none on /mnt/hugepage type hugetlbfs (rw)
```

When the program hugex is run as root (i.e., with the privilege for `HUGETLBFS` use), we see its text map under `/mnt/hugepage`, as expected. The components 5180... and 2ce2110... in the pathname are derived from the device identifier and the inode identifier of the file /tmp/hugex, which is the executable.

```
# ./hugex
Huge page text segment map:
00600000-00800000 r-xp 00000000 00:15
 9166 /mnt/hugepage/5180000000000000/
 2ce2110000000000/text
```

Next we show what happens when the large grained translations are made unavailable. The following invocation of the program is as a normal (unprivileged) user. In this case, large text mapping will not be available, and the normal segment (#4) will be used instead for compatibility.

```
$ ./hugex
Huge page text segment map:
00600000-00601000 r-xp 00100000 08:15 1126082
 /tmp/hugex
```

5 Performance

We measured the impact of changes described in Section 3, on two 64-bit systems. The first was a 4-processor Intel® Itanium® 2, and the second was a 4-processor Intel® Pentium® 4 with Hyperthreading. We employed an industry standard and fully scaled online transaction processing workload and used a workload driver that shared the processors with the database management software, in a single tier configuration for convenience of benchmarking. The buffer pool for the database was placed in `HUGETLBFS`-based shared memory, and was of the same size independent of whether text pages were mapped with normal- or large-grained translations.

Both Intel® Pentium® and Intel® Itanium® systems showed performance gains with the use of large text pages for the database software. Both systems yielded throughput improvements averaging 4.65% as measured by transactions performed per unit of time. The table below captures the percent difference in selected processor event metrics on an Intel® Pentium® machine, between using and not using large pages for mapping text [6]. In this table and in the description that follows, ITLB and DTLB are respectively acronyms for Instruction and Data TLBs.

While the number of ITLB misses reduced by 5%, they produced a much higher drop in the number of page table traversals for servicing

Gain in throughput (transactions per minute)	4.6
Improvement in first level data cache miss ratio	3.0
Improvement in second level data cache miss ratio	5.0
Reduction in ITLB miss handling overhead	50.0
Reduction in number of bus accesses	3.0
Reduction in ITLB misses	5.0
Reduction in second level cache misses	8.0
Reduction in DTLB miss handling overhead	5.0
Reduction in DTLB misses	5.0

Table 1: Percent Improvement from `HUGETLBFS` Mapped Text

the ITLB misses, since the use of large page translation removes the need to perform an additional traversal level and cuts the overhead of handling ITLB misses in half.

The large drop in the second level cache misses comes from a sharp reduction in the number of page table entries occupying the processor's cache. A high value gain reflected in these event metrics is the reduction in bus accesses, due to improved cache hit ratios. The number of hardware threads per die is poised to increase significantly in coming years. Software driven improvements in cache efficiencies in present generation systems can be expected to yield critical reductions in traffic along shared paths between the cores on each die, and other caches or memory modules.

One side benefit of the reduced page table traversals for servicing the ITLB misses is a reduction in the number of DTLB misses arising from the traversals. This yields the 5% reduction in DTLB misses and in the DTLB miss handling overheads.

6 Current Status and Future Work

Our current implementation supports IA-32, x86-64, and Intel® Itanium® processor architectures. The kernel, glibc, and binutils changes described in Section 3 are all available at: `http://www.kernel.org/pub/linux/devel/hugepage`.

We believe that our changes can be easily extended to other architectures. The kernel and glibc changes are architecture independent. Only our linker changes need to be ported.

The current implementation only supports huge page text in executable. We are looking into feasibility of supporting huge page text in shared library. We are also planning a feasibility study for placing writable data sections into huge pages and assessing the resulting performance impact.

7 Conclusion

In summary, capitalizing upon `HUGETLBFS` by mapping code in large pages and thereby improving translation efficiencies of processors in executable regions helps enterprise applications with large text footprints. This capability is achieved with small changes to the linking and loading framework, and removes a significant performance hurdle for such applications. The resulting page table efficiency improves ITLB hit ratios, and produces downstream benefits for first and second level caches. By reducing the stresses on these caches and on other

hardware resources shared on the same chip, the use of large grained text pages facilitates performance scaling with increasing on-chip concurrencies.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood. DBMSs On A *Modern Processor: Where Does Time Go?*, Proc. VLDB, 1999.

[2] Martin J. Bligh and David Hansen. *Linux Memory Management on Larger Machines*, Proc. Linux Symposium, July 2003.

[3] Kenneth Chen, Rohit Seth, Hubert Nueckel, *Improving Enterprise Database Performance on Intel® Itanium® Architecture*, Proc. Linux Symposium, July 2003.

[4] Wim A. Coekarts, *Big Servers—2.6 compared to 2.4*, Proc. Linux Symposium, July 2004.

[5] *Hyper-threading technology*, `http://www.intel.com/technology/hyperthread`

[6] *IA-32 Intel® Architecture Optimization Reference Manaual: Appendix B: Intel Pentium 4 Processor Performance Metrics*, `ftp://download.intel.com/design/Pentium4/manuals/24896612.pdf`

[7] Simon Winwood, Yefim Shuf, Hubertus Franke, *Multiple Page Size Support in the Linux Kernel*, Proc. Linux Symposium, June 2002.