Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

"Turning the Page" on Hugetlb Interfaces

Adam G. Litke
IBM
agl@us.ibm.com

Abstract

HugeTLBFS was devised to meet the needs of large database users. Applications use filesystem calls to explicitly request superpages. This interface has been extended over time to meet the needs of new users, leading to increased complexity and misunderstood semantics. For these reasons, hugeTLBFS is unsuitable for potential users like HPC, embedded, and the desktop. This paper will introduce a new interface abstraction for superpages, enabling multiple interfaces to coexist, each with separate semantics.

To begin, a basic introduction to virtual memory and how page size can affect performance is described. Next, the differing semantic properties of common memory object types will be discussed with a focus on how those differences relate to superpages. A brief history of hugeTLBFS and an overview of its design is presented, followed by an explanation of some of its problems. Next, a new abstraction layer that enables alternative superpage interfaces to be developed is proposed. We describe some extended superpage semantics and a character device that could be used to implement them. The paper concludes with some goals and future work items.

1 Introduction

Address translation is a fundamental operation in virtual memory systems. Virtual addresses must be converted to physical addresses using the system page tables. The *Translation Lookaside Buffer (TLB)* is a hardware cache that stores these translations for quick retrieval. While system memory sizes increase exponentially, the TLB has remained small. *TLB coverage*, the percent of total virtual memory that can be translated into physical addresses directly through the cache, has decreased by a factor of 100 in the last ten years [4]. This has resulted in a greater number of *TLB misses* and reduced system performance.

This paper will discuss extensions provided by hardware which can serve to alleviate TLB coverage issues. Properly leveraging these extensions in the operating system is challenging due to the persistent trade-offs between code complexity, performance benefits, and the need to maintain system stability. We propose an extensible mechanism that enables TLB coverage issues to be solved without adverse effects.

2 Hardware Management

Virtual memory is a technique employed by most modern computer hardware and operating systems. The concept can be implemented in many ways but this paper focuses on the *Linux virtual memory manager (VM)*. The physical memory present in the system is divided into equally-sized units called *page frames*. Memory within these page frames can be referred to by a hardware-assigned location called a *physical address*. Only the operating system kernel has direct access to page frames via the physical address. Programs running in user mode must access memory through a *virtual address*. This address is software-assigned and arbitrary. The VM is responsible for establishing the mapping from virtual addresses to physical addresses so that the actual data inside of the page frame can be accessed. In addition to address translation, the VM is responsible for knowing how each page frame is being used. Quite a few data structures exist for tracking this information. A *page table entry (PTE)*, besides storing a physical address, lists the access permissions for a page frame. The assigned permissions are enforced at the hardware level. A *struct page* is also maintained for each page frame in the system [2]. This structure stores status information such as the number of current users and whether the page frame is undergoing disk I/O.

When a program accesses a block of memory for the first time, a translation will not be available to the CPU so control is passed to the VM. A page frame is allocated

for the process and a PTE is inserted into the *page tables*. The TLB is filled with the new virtual to physical translation and execution continues. Subsequent references to this page will trigger a TLB look-up in hardware. A *TLB hit* occurs if the translation is found in the cache and execution can continue immediately. Failure to find the translation is called a *TLB miss*. When a TLB miss occurs, a significant amount of overhead is incurred. The page tables must be traversed to find the page frame which results in additional memory references.

The TLB is conceptually a small array, each slot containing translation information for one page of memory. Over time it has remained small, generally containing 128 or fewer entries [6]. On a system where the page size is 4 KiB, a TLB with 128 slots could cache translations for 512 KiB of memory. This calculation provides a measure of *TLB reach*. Maximizing TLB reach is a worthy endeavor because it will reduce TLB misses.

The increasing complexity of today's applications require large working sets. A *working set* is the smallest collection of information that must be present in main memory to ensure efficient operation of the program [1]. When TLB reach is smaller than the working set, a problem may arise for programs that do not exhibit *locality of reference*. The principle states that data stored in the same place is likely to be used at the same time. If a large working set is accessed sparsely, the limited number of TLB entries will suffer a cycle of eviction and refilling called *TLB thrashing*.

Scientific applications, databases, and various other workloads exhibit this behavior and the resulting marginalization of the TLB. One way to mitigate the problem is to increase TLB reach. As a simple product, it can be increased either by enlarging the TLB or by increasing the page size. Hardware vendors are addressing the issue by devoting more silicon to the TLB and by supporting the use of an increasing number of page sizes. It is up to the operating system to leverage these hardware features to realize the maximum benefits.

2.1 Superpages

Superpages is the term used to refer to any page with a size that is greater than the base page size. Significant research has been done to develop algorithms for, and

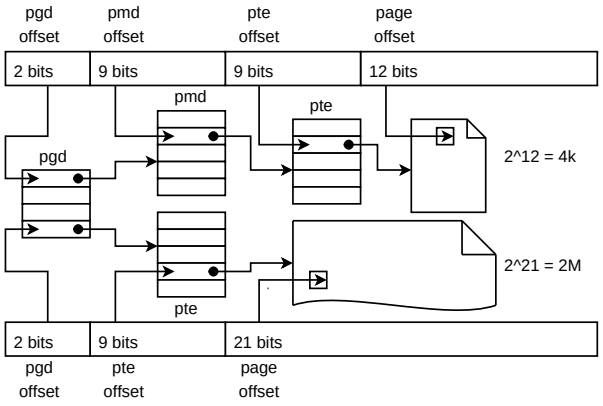

Figure 1: Example page table structure given different page sizes

measure the performance of superpages. Some workloads such as scientific applications and large databases commonly see gains between 10 and 15 percent when using superpages. Gains as high as 30% have been measured in some cases [5]. Superpages are ideal for an application that has a persistent, large, densely accessed working set. The best candidates tend to be computationally intensive.

Virtual memory management with superpages is more complex. Due to the way a virtual address indexes the page table structures and the target page, multiple page sizes necessitate multiple page table formats. Figure 1 contrasts the page table layout for two different page sizes using the x86 architecture as an example. As the figure shows, a virtual address is divided into a series of offsets. The page offset must use enough bits to reference every byte in the page to which the virtual address refers. A 4 KiB page requires 12 bits but a 2 MiB page needs 21. Since, in this example, the virtual address is only 32 bits, supporting a big page offset necessitates changes to the other offsets. In this example, the pmd page table level is simply removed which results in a two level page table structure the 2 MiB pages.

For most workloads, superpages have no effect or may actually hinder performance. Due to *internal fragmentation*, larger page sizes can result in wasted memory and additional, unnecessary work for the operating system. For example, to satisfy security requirements a newly allocated page must be filled with zeroes before it is given to a process. Even if the process only intends to use a small portion of the page, the entire page must be allocated and zeroed. This condition is worst for applications that sparsely access their allocated memory.

Although superpages are complex and improve performance only for programs with a specific set of characteristics, where they do help the impact is great. For this reason, any implementation of superpages should be flexible to maximize performance potential while placing as small of a burden on the system as possible.

3 Address Space Management

By insulating the application from the physical memory layout, the kernel can regulate memory usage to improve security and performance. Programs can be guaranteed private memory allowing sharing only under controlled circumstances. Optimizations, such as automatically sharing common read-only data among processes is made possible. This leads to memory with different semantic characteristics, two of which are particularly relevant when discussing superpages and their implementation in Linux.

3.1 Shared Versus Private

A block of memory can be either shared or private to a process. When memory is being accessed in shared mode, multiple processes can read and, depending on access permissions, write to the same set of pages which are shared among all the attached processes. This means that modifications made to the memory by one process will be seen by every other process using that memory. This mode is clearly useful for things such as interprocess communication.

Memory is most commonly accessed with private semantics. A private page appears exclusive to one address space and therefore only one process may modify its contents. As an optimization, the kernel may share a private page among multiple processes as long as it remains unmodified. When a write is attempted on such a page it must first be unshared. This is done by creating a new copy of the original page and permitting changes only to the new copy. This operation is called *copy on write (COW)* [2].

3.2 File-backed Versus Anonymous

The second semantic characteristic concerns the source of the data that occupies a memory area. Memory can be either file-backed or anonymous. File-backed memory is essentially an in memory cache of file data from a disk or other permanent storage. When pages of this memory type are accessed, the virtual memory manager transparently performs any disk I/O necessary to ensure the in memory copy of the data is in sync with the master version on disk. Maintaining the coherency of many copies of the same data is a significant source of complexity in the Linux memory management code.

Anonymous memory areas generally have no associated backing data store. Under memory pressure, the data may be associated with a swap file and written out to disk, but this is not a persistent arrangement as with file-backed areas. When pages are allocated for this type of memory, they are filled with zeroes.

Linux is focused on furnishing superpage memory with anonymous and shared semantics. This means that superpages can be trivially used with only a small subset of the commonly used memory objects.

3.3 Memory Objects

Memory is used by applications for many different purposes. The application places requirements on each area, for example the code must be executable and the data writable. The operating system enforces additional constraints, such as preventing writes to the code or preventing execution of the stack. Together these define the required semantics for each area.

The stack is crucial to the procedural programming model. It is organized into *frames*, where each frame stores the local variables and return address for a function in the active function call chain. Memory in this area is private and anonymous and is typically no larger than a few base pages. The access pattern is localized since executing code tends to only access data in its own frame at the top of the stack. It is unlikely that such a small, densely accessed area would benefit from superpages.

Another class of objects are the memory resident components of an executable program. These can be further divided into the *machine executable code (text)* and the *program variables (data)*. These objects use file-backed, private memory. The access pattern depends heavily on the specific program, but when they are sufficiently large, superpages can provide significant performance gains. The text and data segments of shared libraries are handled slightly different but have similar semantic properties to executable program segments.

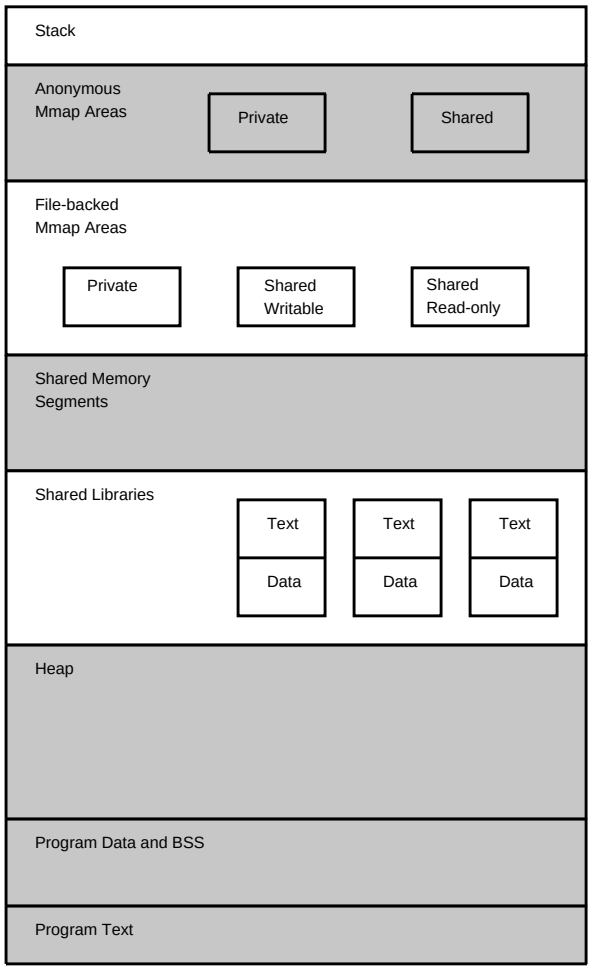

Figure 2: Memory object types

The *heap* is used for dynamic memory allocation such as with malloc. It is private, anonymous memory and can be fairly trivially backed with superpages. Programs that use lots of dynamic memory may see performance gains from superpages.

Shared memory segments are used for interprocess communication. Processes attach them by using a global shared memory identifier. Database management systems use large shared memory segments to cache databases in memory. Backing that data with superpages has demonstrated substantial benefits and inspired the inclusion of superpage support in Linux.

It is already possible to use superpages for some memory types directly. Shared memory segments and special, explicit memory maps are supported. By using libraries, superpages can be extended in a limited way to a wider variety of memory objects such as the heap and program segments.

4 Introduction to HugeTLBFS

In 2002, developers began posting patches to make superpages available to Linux applications. Several approaches were proposed and, as a result of the ensuing discussions, a set of requirements evolved. First and foremost, the existing VM code could not be unduly complicated by superpage support. Second, databases were the application class most likely to benefit from superpage usage at the time, so the interface had to be suitable for them. *HugeTLBFS* was deemed to satisfy these design requirements and was merged at the end of 2002.

HugeTLBFS is a RAM-based filesystem. The data it contains only exists within superpages in memory. There is no backing store such as a hard disk. HugeTLBFS supports one page size called a *huge page*. The size of a huge page varies depending on the architecture and other factors. To access huge page backed memory, an application may use one of two methods. A file can be created and `mmap()`ed on a mounted hugeTLBFS filesystem, or a specially created shared memory segment can be used. HugeTLBFS continues to serve databases well, but other applications use memory in different ways and find this mechanism unsuitable.

The last few years have seen a dramatic increase in enthusiasm for superpages from the scientific community. Researchers run multithreaded jobs to process massive amounts of data on fast computers equipped with huge amounts of memory. The data sets tend to reside in portions of memory with private semantics such as the *BSS* and heap. Heavy usage of these memory areas is a general characteristic of the *Fortran* programming language. To accommodate these new users, private mapping support was added to hugeTLBFS in early 2006. *LibHugeTLBFS* was written to facilitate the remapping of executable segments and heap memory into huge pages. This improved performance for a whole new class of applications, but for a price.

For shared mappings, the huge pages are reserved at creation time and are guaranteed to be available. Private mappings are subject to non-deterministic COW operations which make it impossible to determine the number of huge pages that will actually be needed. For this reason, successful allocation of huge pages to satisfy a private mapping is not guaranteed. Huge pages are a scarce resource and they frequently run out. If this happens while trying to satisfy a huge page fault, the application

will be killed. This unfortunate consequence makes the use of private huge pages unreliable.

The data mapped into huge pages by applications is accessible in files via globally visible mount points. This makes it easy for any process with sufficient privileges to read, and possibly modify, the sensitive data of another process. The problem can be partially solved through careful use of multiple hugeTLBFS mounts with appropriate permissions. Despite safe file permissions, an unwanted covert communication channel could still exist among processes run by the same user.

HugeTLBFS has become a mature kernel interface with countless users who depend on consistency and stability. Adapting the code for new superpage usage scenarios, or even to fix the problems mentioned above has become difficult. As hardware vendors make changes to solve TLB coverage issues such as adding support for multiple superpage sizes, Linux is left unable to capitalize due to the inflexibility of the hugeTLBFS interface.

For a real example of these problems, one must only look back to the addition of *demand faulting* to hugeTLBFS. Before enablement of this feature, huge pages were always *prefaulted*. This means that when huge pages were requested via an `mmap()` or `shmat()` system call, all of them were allocated and installed into the mapping immediately. If enough pages were not available or some other error occurred, the system call would fail right away. Demand faulting delays the allocation of huge pages until they are actually needed. A side effect of this change is that huge page allocation errors are delayed along with the faults.

A commercial database relied on the strict accounting semantics provided by prefault mode. The program mapped one huge page at a time until it failed. This effectively reserved all available huge pages for use by the database manager. When hugeTLBFS switched to demand faulting, the `mmap()` calls would never fail so the algorithm falsely assumed an inflated number of huge pages were available and reserved. Even with a smarter huge page reservation algorithm, the database program could be killed at a non-deterministic point in the future when the huge page pool is exhausted.

5 Page Table Abstraction

HugeTLBFS is complex and its semantics are rigid. This makes it an unsuitable vehicle for future development. To quantify the potential benefits of expanded superpage usage, a mechanism for testing competing and potentially incompatible semantics and interfaces is needed. Making this possible requires overriding the superpage-related special cases that are currently hardwired to hugeTLBFS.

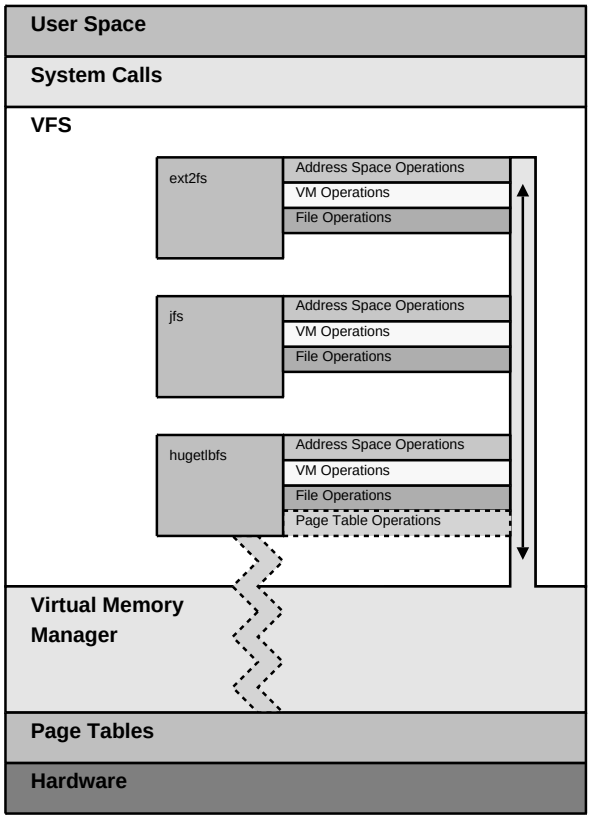

Figure 3: How hugeTLBFS interacts with the rest of the kernel

In finding a solution to this problem, it is important to remember the conditions that grounded the development of hugeTLBFS. Additional complexity cannot be added to the VM. Transparent, fully-integrated superpage support is thus an unrealistic pursuit. As with any kernel change, care must be taken to not disturb existing users. The behavior of hugeTLBFS must not change and performance regressions of any kind must be avoided. Further, a complete solution must fully abstract existing special cases but remain extensible.

Like any other filesystem, hugeTLBFS makes use of an abstraction called the *virtual filesystem (VFS)* layer. This object-oriented interface is what makes it possible for the VM to consistently interact with a diverse array of filesystems. Many of the special cases introduced by supporting superpages are hidden from the VM through

hugeTLBFS' effective use of the VFS API. This interface was not designed to solve page size issues so some parts of the kernel remain hugeTLBFS-aware. Most of these cases are due to the need for an alternate page table format for superpages. Figure 3 shows how hugeTLBFS fits into the VFS layer as compared to other filesystems.

Taking the VFS layer as inspiration, we propose an abstraction for page table operations that is capable of unwiring the remaining hugeTLBFS-specific hooks. It is simply a structure of six function pointers that fully represent the set of special page table operations needed.

`fault()` is called when a page fault occurs somewhere inside a VMA. This function is responsible for finding the correct page and instantiating it into the page tables.

`copy_vma()` is used during fork to copy page table entries from a VMA in the parent process to a VMA in the child process.

`change_protection()` is called to modify the protection bits of PTEs for a range of memory.

`pin_pages()` is used to instantiate the pages in a range of userspace memory. The kernel is not generally permitted to take page faults. When accessing userspace, the kernel must first ensure that all pages in the range to be accessed are present.

`unmap_page_range()` is needed when unmapping a VMA. The page tables are traversed and all instantiated pages for a given memory range are unmapped. The PTEs for the covered range are cleared and any pages which are no longer in use are freed.

`free_pgtable_range()` is called after all of the PTEs in a mapping are cleared to release the pages that were used to store the actual page tables.

5.1 Evaluating the Solution

The page table operations produce a complete and extensible solution. All the infrastructure is in place to enable hugeTLBFS to be built as a module, further separating it from the core VM. Other independent superpage interface modules can be added to the system without causing interference.

The implementation is simple. The existing, hugeTLBFS page table manipulation functions are collected into an operations structure without modifying them in any way. Instead of calling hard-wired functions, the function to call is looked up in the page table operations. Changing only these two elements ensures that hugeTLBFS behavior is unchanged.

Superpages exist solely for performance reasons so an implementation that is inefficient serves only to undermine its own existence. Two types of overhead are considered with respect to the proposed changes. One pointer will be added to the VMA. Already, this structure does not fit into a cache line on some CPUs, so care must be taken to place the new field at a sensible offset within the structure definition. To this end, the `pagetable_ops` pointer has been placed near the `vm_ops` and `vm_flags` fields, which are also used frequently when manipulating page tables.

The second performance consideration is related to the pointer indirection added when a structure of function pointers is used. Instead of simply jumping to an address that can be determined at link time, a small amount of additional work is required. The address of the assigned operations is looked up in the VMA. This address, plus an offset, is dereferenced to yield the address of the function to be called. VMAs that do not implement the page table operations do not incur any overhead. Instead of checking `vm_flags` for `VM_HUGETLB`, `pagetable_ops` is checked for a NULL value.

6 Using the Abstraction Interface

With a small amount of simple abstraction code, the kernel has become more flexible and is suitable for further superpage development. HugeTLBFS and its existing users remain undisturbed and performance of the system has not been affected in any measurable way. We now describe some possible applications of this new extensible interface.

A character device is an attractive alternative to the existing filesystem interface for several reasons. Its semantics provide a more natural and secure method for allocating anonymous, private memory. The interface code is far simpler than that of hugeTLBFS which makes it a much more agile base for the following extensions.

The optimal page size for a particular memory area depends on many factors such as: total size, density of

```
struct pagetable_operations_struct page_pagetable_ops = {
     .copy_vma           = copy_hugetlb_page_range,
     .pin_pages          = follow_hugetlb_page,
     .unmap_page_range   = unmap_hugepage_range,
     .change_protection  = hugetlb_change_protection,
     .free_pgtable_range = hugetlb_free_pgd_range,
     .fault              = page_fault,
};
```

Figure 4: A sample page table operations structure

access, and frequency of access. If the page size is too small, the system will have to service more page faults, the TLB will not be able to cache enough virtual to physical translations, and performance will suffer. If the page size is too large, both time and memory are wasted. On the PowerPC® architecture, certain processors can use pages in sizes of 4KiB, 64KiB, 16MiB, and larger. Other platforms can also support more than the two page sizes Linux allows. Architecture specific code can be modified to support more than two page sizes at the same time. Fitted with a mechanism to set the desired size, a character device will provide a simple page allocation interface and make it possible to measure the effects of different page sizes on a wide variety of applications.

Superpages are a scarce resource. Fragmentation of memory over time makes allocation of large, contiguous blocks of physical memory nearly impossible. Without contiguous physical memory, superpages cannot be constructed. Unlike with normal pages, swap space cannot be used to reclaim superpages. One strategy for dealing with this problem is *demotion*. When a superpage allocation fails, a portion of the process' memory can be converted to use normal pages. This allows the process to continue running in exchange for a sacrifice of some of the superpage performance benefits.

Selecting the best page size for the memory areas of a process can be complex because it depends on factors such as application behavior and the general system environment. It is possible to let the system choose the best page size based on variables it can monitor. For example, a *population map* can be used to keep track of allocated base pages in a memory region [5]. Densely populated regions could be *promoted* to superpages automatically.

6.1 A Simple Character Device

The character device is a simple driver modeled after `/dev/zero`. While the basic functionality could be implemented via hugeTLBFS, that would subject it to the functional limitations previously described. Additionally, it could not be used to support development of the extensions just described. For these reasons, the implementation uses page table abstraction and is independent of hugeTLBFS.

The code is self contained and can be divided into three distinct components. The first component is the standard infrastructure needed by all drivers. This device is relatively simple and needs only a small function to register with the driver layer.

The second component is the set of device-specific structures that define the required abstracted operations. Figure 4 shows the page table operations for the character device. Huge page utility functions are already well separated from the hugeTLBFS interface which makes code reuse possible. Five of the six page table operations share the same functions used by hugeTLBFS.

The third component is what makes this device unique. It handles page faults differently than hugeTLBFS so a special `fault()` function is defined in the page table operations. This function is simpler than the hugeTLBFS fault handler because it does not need to handle shared mappings or filesystem operations such as truncation. Most of the proposed semantic changes can be implemented by further modifying the fault handler.

7 Conclusions

The page table operations abstraction was developed to enable advanced superpage development and work

around some problems in hugeTLBFS. It does not alter the behavior of the current interface nor complicate the kernel in any significant way. No performance penalty could be measured. Work on the topics previously described can now begin.

The described changes have also led to some general improvements in the way hugeTLBFS interacts with the rest of the kernel. By collecting a set of dispersed hugeTLBFS-specific page table calls into one structure, the list of overloaded operations becomes clear. This API, when coupled with other pending cleanups, removes hard-coded special cases from the kernel. The result is a hugeTLBFS implementation that is even further "on the side" and decoupled from the core memory manager.

8 Future Work

During development of the page table abstraction interface and the character device, a few additional opportunities to clean up the interface between hugeTLBFS and the rest of the kernel became apparent. Every effort should be made to extricate the remaining hugeTLBFS special cases from the kernel. Moving superpage-related logic behind the appropriate abstractions makes for a simpler VM and at the same time enables richer superpage support.

The work presented in this paper enables a substantial body of research and development using Linux. We intend to implement different superpage semantics and measure their performance effects across a variety of workloads and real applications. If good gains can be achieved with reasonable code we hope to see those gains realized outside of our incubator in production kernels.

A separate effort to reduce memory fragmentation is underway [3]. If this body of work makes it into the kernel, it will have a positive effect on the usability of superpages in Linux. When contiguous blocks of physical memory are readily available, superpages can be allocated and freed as needed. This makes them easier to use in more situations and with greater flexibility. For example, page size promotion and demotion become more effective if memory can be allocated directly from the system instead of from the static pool of superpages that exists today.

9 Legal Statement

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] P. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[2] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, NJ, 2004.

[3] M. Gorman and A. Whitcroft. The What, The Why and the Where To of Anti-Fragmentation. *Ottawa Linux Symposium*, 1:369–384, 2006.

[4] J. Navarro. *Transparent operating system support for superpages*. PhD thesis, RICE UNIVERSITY, 2004.

[5] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(si):89, 2002.

[6] T. Selén. *Reorganisation in the skewed-associative TLB*. Department of Information Technology, Uppsala University.