Performance Characteristics of Explicit Superpage Support

Mel Gorman, Patrick Healy

Performance Characteristics of Explicit Superpage Support

Mel Gorman
IBM / University of Limerick
mel@csn.ul.ie

Patrick Healy
University of Limerick
patrick.healy@ul.ie

Abstract

Many modern processors support more than one page size. In the 1990s the larger pages, called superpages, were identified as one means of reducing the time spent servicing Translation Lookaside Buffer (TLB) misses by increasing TLB reach. Transparent usage of superpages has seen limited support due to architectural limitations, the cost of monitoring and implementing promotion/demotion, the uncertainity of whether superpages will be a performance boost and the decreasing cost of TLB misses due to hardware innovations. As significant modifications are required to transparently support superpages, the perception is that the cost of transparency will exceed the benefits for real workloads.

This paper describes how processes can explicitly request memory be backed by superpages that is cross-platform, incurs no measurable cost and is suitable for use in a general operating system. By not impacting base page performance, a baseline metric is established that alternative superpage implementations can compare against. A reservation scheme for superpages is used at `mmap()` time that guarantees faults without depending on pre-faulting, the fragmentation state of the system or demotion strategies. It is described how to back different regions of memory using explicit superpage support without application modification and present an evaluation of an implementation running a range of workloads.

Categories and Subject Descriptors D.4 [*Storage Management*]: Virtual Memory

General Terms Algorithms, Measurement, Performance

Keywords superpage, fragmentation, replacement policy, TLB

1. Introduction

In the 1990s, it was observed that the Translation Lookaside Buffer (TLB) reach was shrinking as a percentage of physical memory and the relative amount of program time spent servicing TLB misses was increasing. Superpages were identified as one means of reducing TLB misses. Preferably it would be fully transparent to the application but there are significant obstacles that prevent transparent superpage support being properly adopted in mainstream operating systems for general applications. This paper begins by describing architectural limitations when translating addresses for multiple page sizes that have hindered transparent superpage support. As superpages typically require physically contiguous pages, the paper summarises some earlier work on controlling external fragmentation and page reclaim [GOR08] as they are important pre-requisites for supporting superpages [TALLURI92].

This paper then progresses this earlier work by describing a superpage reservation scheme that can be applied at `mmap()` time to guarantee future faults. This eliminates the need for pre-faulting or demotion strategies that are problematic on a number of architectures. These are prerequisites that must be addressed for reliable Explicit Superpage Support; a mechanism that exposes interfaces directly allowing loaders, libraries or applications to directly request superpages. It is shown that explicit superpage support can back memory regions without direct application modification through the use of libraries.

Over the course of the research, superpage support was implemented on Linux and merged into the mainline releases. This paper presents an evaluation of an implementation based on Linux using AMD® Phenom (X86-64) and POWER® (PPC64) processors. By tackling the fundamental problems of superpage support and enabling usage without application modification, this implementation represents a comparison point for transparent superpage support implementations on the same platform. There is an expectation that similar mechanisms could be implemented on other platforms.

2. Limitations of Transparent Support

There are architectural limitations to consider when implementing superpage support that is cross-platform. Limiting the number of architecture-specific paths reduces maintenance and test costs so the common architectural features must be supported before considering architecture-specific options. This section considers three architectures that make up 99.6% of the Top 500 Supercomputers [MSSD08] list as being a reasonable cross-section of mainstream architectures. Understanding the limitations of two of these architectures in particular illustrate why fully transparent superpage support be problematic for cross-platform operating systems.

2.1 POWER®

POWER® occupies 3 of the top 10 spots in the Top 500 list, has a 12% share overall and occupies the top spot as part of a hybrid system [MSSD08]. The current Power Architecture specification allows multiple page sizes but initially it only allowed two; a base and an implementation-specific large page size between 8KB and 256MB [POWER03]. With POWER4, the page sizes were 4K and 16MB with POWER5+ adding 64K and 16GB superpages. Unfortunately, in general[1] the Memory Management Unit (MMU) requires the same superpage size be used within segments of either 256MB or 1TB virtual address range, thus hindering transparent superpage support. Promotion or demotion would require modifying the entire segment at once with is prohibitively expensive and makes transparent superpage support on POWER impractical.

[1] The specification allows 4K and 64K pages to mix within a segment but support is not universal.

WIOSCA '10 June 19, 2010, Saint-Malo, France.

2.2 Itanium® (IA-64)

Itanium maintains a 1.8% share of the Top 500 list and is used on very large single-image machines such as the SGI Altrix 4700 supporting up to 1024 processors on a single machine. The Itanium supports 11 different page sizes but the Itanium port of Linux has similar problems to POWER with respect to address translation that hinder transparent support.

Two mutually exclusive hardware-defined formats usable by page table walkers exist for Itanium called short- and long-format Virtual Hashed Page Table (VHPT); SF-VHPT and LF-VHPT respectively. SF-VHPT has a smaller cache-footprint but can only use one page size per region. As the 64-bit address space is divided into eight regions, this is a significant restriction as multiple pages can only be used within a region if the hardware page table walker is disabled - a significant penalty. The LF-VHPT has larger PTEs and is less cache friendly but potentially has a smaller TLB footprint and has no restrictions on superpage placement.

Linux favours SF-VHPT for two reasons that may be important to other platforms. Firstly, TLB-and-context-switch-intensive workloads are relatively rare and Linux has a policy of minimising cache usage. Secondly, the "hardware-independent" representation of page tables used by Linux is a 4-level hierarchical page table format which, if mapped in the correct place, can be used directly by the SF-VHPT hardware walker. LF-VHPT can still be used on Linux [WIENARD08] but with more expensive TLB reloads. As this can only be potentially offset by superpage usage, the move to the long-format is currently unjustified on that system. This severely limits the possibility of transparent support on the Itanium platform.

2.3 X86 Variants (Intel® EM64T, AMD® X86-64)

X86 variants dominate the Top 500 list with a share of 73.8% for Intel EM64T and 12% for AMD X86-64. 2MB or 4MB superpages are supported depending on the address translation used. In 32-bit mode, 4MB superpages are typically available unless Physical Address Extension (PAE) is enabled. If PAE or 64-bit mode is used, the superpage size is generally 2MB but some recent AMD processors also add a 1GB superpage size.

Due to how addresses are translated, there are limits on how many superpage sizes can be supported but no significant limit on where the superpages are placed. The difficulties for transparent support are two-fold, one hardware and one software. The TLB characteristics differ significantly between processor implementations. The Intel Pentium D, including Xeon variants, have fully-associative TLBs. Others such as the Core Duo have set-associative TLBs with different numbers of entries for different page sizes. The AMD Phenom 9950 processor reports multiple TLBs with the L1 DTLB fully-associative and L2 DTLB set-associative. This degree of variation significantly complicates cost/benefit calculations regarding superpages.

The software problem is specific to Linux "open-coding" 4-level page table traversal without checking the underlying page size instead of using an abstracted API. Transparent support is possible but implementing it incurs a performance cost. Before that cost is incurred, it must be established what the limits of explicit superpage support are.

3. Related Work

This section summarises some earlier work [GOR08] that is a prerequisite for explicit superpage support. Architectural limitations generally require that superpages be naturally aligned and physically contiguous forcing the system to handle external fragmentation. Memory defragmentation is not a complete solution as wired pages exist that cannot move and may be scattered throughout the physical address space. Being able to move arbitrary pages is costly due to locking requirements and copying, both of which vary between architectures.

3.1 Page Allocation

Grouping Pages By Mobility [GOR08] divides the physical address space into superpage-sized arenas. Different arena types exist based on a page's ability to move, be it by migration or reclamation noting that the number of arenas for each type changes based on demand. Each arena has its own free-lists so the search cost is similar to that of the standard allocator. The placement policy does not guarantee that a specific number of superpages can be allocated so the system's memory may also be partitioned. Arenas for wired pages are only ever created on one side of the partition guaranteeing that one partition can be filled with superpages by reclaiming or moving pages. On Non-Uniform Memory Access (NUMA) systems, the partition for wired pages is evenly divided on all nodes to avoid kernel accesses being frequently remote. On batch systems with workloads of varying requirements, the partition would be sized to the largest estimated number of superpages required. If superpages are not allocated, movable base pages still use the partition.

3.2 Page Reclaim

On allocation failure, Age-Biased Arena Reclaim [GOR08] uses the normal page reclaim algorithm to select a reference page and reclaim the naturally aligned arena if all pages are free or reclaimable as likely will be the case due to the placement policy. This minimises disturbance of the normal page reclaim decisions made by the operating system

4. Superpage Reservation

Systems normally ensures successful faults by recording the number of pages required for existing mappings and refusing to create new mappings that exceed available memory and swap space. This is insufficient for superpages as contiguity requirements can fail a page fault due to external fragmentation. Faulting superpage-backed mappings at `mmap()` time would increase the cost of `mmap()` and perform poorly with Non-Uniform Memory Access (NUMA). Optimally on NUMA, CPUs always access pages local to the CPU without copying data between nodes. Pre-faulting places pages local to the thread calling `mmap()` and not necessarily local to the threads accessing the data, incurring costly cross-node accesses.

A better alternative is to reserve the superpages necessary to guarantee future faults at `mmap()` time. Counters track the current state of the system, some of which are tracked on a per-NUMA-node basis so that NUMA policies can be applied correctly.

When creating a new mapping, faulting a page within an existing mapping or unmapping an existing mapping, it is determined how many pages have already been allocated or reserved. For example, pages for shared mappings may already have been allocated or a reserve created by another process. Searching the page cache or page tables for references to allocated pages would be unsuitably expensive. Instead, each mapping is treated as a file. Shared mappings of data use one file whereas private mappings have their own unique file. Each file is associated with one or more `struct file_region` stored on a linked list and defined as follows.

```
struct file_region {
        struct list_head link;
        long from, to;
};
```

The size of the file in superpages is the initial required reservation to succeed all future faults. Each `file_region` on the list implies that `to - from` reserved pages have been used.

Due to the fact that shared and private mappings are accounted differently, there are differences in the run-time behaviour of an application using shared or private mappings backed by superpages.

4.1 Shared Mapping Accounting

Shared mappings reservations for the same data are based on the largest mapping created by any process i.e. the size of the file. The `file_region` is associated with one shared structure such as an inode. On fault, the page cache is searched and the page inserted into the processes page tables if it exists, otherwise the page is allocated, inserted into both page cache before updating the reservations, `file_region` structures and inserting into the page table.

4.2 Private Mapping Accounting

Private mappings always reserve superpages for the entire mapping as even existing pages require new pages for a Copy-On-Write (COW) fault. Read-only private mappings could avoid reserving until `mprotect()` was called but applications may not correctly handle `mprotect()` errors if the reserve could not be made. Due to the per-mapping nature of the reservation, the `file_region` is linked to the virtual area management structure (`struct vm_area_struct` in Linux).

`fork()` presents a difficulty. Shared mappings can use the same pages and reservations between parent and child but it is unknowable how many superpages a child will need for private mappings. Duplicating reservations on `fork()` is expensive, may be impossible and impacts applications calling `exec()` after `fork()`.

An alternative is to only guarantee faults within the address space that created the mapping. A child process's mappings have no reservations but on fault, a superpage allocation without a reservation is attempted. If successful, the fault completes or else the child gets killed. A COW fault from the parent similarly attempts to allocate a new superpage. On allocation failure, the page is unmapped from any children and the mappings are flagged. The parent safely uses the original page but child gets killed if it faults within the flagged mapping. Hence, a programming model based on superpage should not assume reliable superpage behaviour for private mappings in child processes. Either private mappings should be `madvise()` with `MADV_DONTFORK` or child processes should take care not to access the mappings. This model guarantees the address space owner can successfully fault without depending on in-kernel demotion strategies that cannot be implemented for all architectures due to limitations on superpage placement. In the event a child is killed, a message is logged stating that "PID N killed to to inadequate hugepage pool" to make it clear what occurred.

5. Explicit Superpage Support

A simple interface for superpages would be similar to `brk()` or `malloc()` but it is difficult to create a shared mapping without extending the interface with unique semantics. The System V shared memory interface exists but application error can leak memory and private mappings are impossible to create. This section describes an interface for an application to map superpages without transparent support.

5.1 RAM-Based Filesystem

The Virtual File System (VFS) API for the creation, mapping and unmapping of files, both shared and private is well understood by developers. A RAM-based filesystem can create files that are backed by superpages when memory mapped. The filesystem API does not allow the passing of additional information but each mount of the filesystem can support different page sizes. Alternatively, an `fadvise()`-like interface could specify the page size before `mmap()` is called. The filesystem is a low-level interface and more suitable for use by libraries than directly by applications.

5.2 System V Shared Memory

The semantics of shared memory created and attached with the System V interfaces `shmget()` and `shmat()` are similarly well understood. By passing a new flag, `SHM_HUGETLB`, the kernel is instructed to back the region with superpages. Internally, files of the size requested by `shmget()` can be created on a private instance of the filesystem. `shmat()` maps the file similar to a normal file mapped as `mmap(MAP_SHARED)`, thus sharing code between the filesystem and shared memory implementations. An error should be returned if the size is not superpage-aligned or sufficient superpages, leaving the decision on whether to use base pages to the library or application.

5.3 Anonymous `mmap()` Mappings

Memory mappings created with `mmap()` take a number of flags such as `MAP_PRIVATE`, `MAP_ANONYMOUS` and others described in the manual page. Similar to the previous interfaces, `mmap()` can be extended to support a `MAP_HUGETLB` flag which automatically backs the mapping with superpages.

The difficulty with this interface is that setting the page size after `mmap()` is not an option due to possible architectural limitations and setting a default page size to be used for `mmap()` can race when threads are in use. If the architecture does not limit the placement of superpages, then `madvise()` can be extended after `mmap()` and before the pages are faulted. Otherwise, using `MAP_HUGETLB` implies that a default superpage size is used.

5.4 Explicit Programming API

Three functions are defined that are useful to applications or libraries with superpage-awareness. The naming reflects the Linux implementation and uses the term "huge page" for superpages.

`hugetlbfs_unlinked_fd()` creates a file on the RAM-based filesystem for the requested superpage size and unlinks, rendering it invisible to other processes. This is useful when a process needs complete control over the size of the mapping placement within the address space.

`get_huge_pages()` takes a length and flags parameter. The length must be superpage-aligned and return a buffer to a superpage-aligned length of memory or NULL on failure. It is suitable for use in the implementation of custom allocators. It is not suitable for use as a drop-in replacement for `malloc()` as it is a raw interface to superpages that does not cache-color buffers.

`get_hugepage_region()` similarly takes a length and flags parameter. The length need not be aligned and wastage is used to cache-color the buffer. On allocation failure, a buffer backed by base pages may be returned if requested by the caller. This is suitable as a drop-in replacement requiring that an application convert call-sites that create large buffers to use `get_hugepage_region()` instead.

6. Backing Memory Sections with Superpages

This section describes how different types of mappings of a process can be backed by superpages using explicit superpage support.

6.1 Heap

Allocations increasing the size of a heap currently use either `brk()` or `mmap()` to create memory regions. A custom allocator should use `get_hugepage_region()` and divide the superpage up into

buffers as appropriate. This requires source modification which may be undesirable but it is a relatively straight-forward exercise.

If the allocator provide hooks for applications to create more heap then `get_huge_pages()` can be used by pre-loading and registering the hook before the application starts. GNU libc uses `sbrk()` for small allocations and `mmap()` for blocks larger than `MMAP_THRESHOLD` which is 128kB by default. A morecore hook allows registry of a custom function that allocate more heap.

6.2 Mapping Text/Data/BSS

The effectiveness of backing data section with superpages depends on the compiler and the application. If the data section contains pre-initialised data, then backing it with superpages will be effective. If the data section is zeros but stored on disk, then the copy will be effective. If it uses `malloc()` with a hook, then it can be backed similar to the heap. Otherwise, the loader must be modified.

Text and data are mapped with different permissions and the protection granularity depends on the page size. Applications should be relinked with the sections superpage-aligned which requires a suitable linker tool chain. Otherwise, architectural limitations determine if sections can be partially backed by superpages.

On `mmap()` of a section, the kernel can check the page cache. If the data exists but is stored in an unaligned base page, then the data should be copied to a superpage and the existing page cache entry replaced so that only one copy exists. If a superpage cannot be allocated, then the base pages are used. As the checks are made before the insertion in the process page tables, architectural limitations with address translation are avoided. Alternatively the loader can use `get_hugepage_region()` or similar to allocate a suitably large region, copy the data from disk and remap it to a suitable place. It is implementation-specific as to whether text is shared between processes or not.

6.3 Stack

The stack area of memory has similar requirements to `brk()` but is not as easily handled with hooks like `malloc()` or copied and remapped like text and data sections. The maximum stack size is unknown at application start time making it inherently unreliable to back with superpages without a demotion strategy. Consequently, explicit superpage support requires the stack be a fixed size.

7. Evaluation

Explicit superpage support as described in this paper has been implemented over the course of a number of kernel releases. This evaluation was based on Linux Kernel 2.6.31, downloaded from http://www.kernel.org and configured using the distribution's kernel configuration file. `libhugetlbfs 2.7` was used to transparently access superpages without application modification. The distribution used was Debian GNU/Linux Lenny current as of October 2009. This distribution is freely available for independent verification and is commonly used as a server operating system. The compiler used was gcc 4.3.2.

Performance results are based on test-runs without profiling as it can distort results due to increased interrupt handling from the PMU and the profiling daemon gathering data. Profile information was gathered during a separate using OProfile[2]. Due to limitations of the PMU, DTLB and cache misses data were collected during separate runs. In all cases, the relative performance of base pages and superpages matched so data collected during profiling should be indicative of what happened during no-profile runs.

During the base page runs, no superpages were used. During the superpage runs, the maximum number of superpages that was possible to use were allocated. Depending on the workload, this varied from negligible amounts to almost the size of physical memory. As the underlying implementation does not support the paging of superpages, there are no eviction rates to report but to make the comparison fair, it was confirmed that there was no meaningful paging activity while the tests were running with base pages.

The two machines were chosen based on the parts or models being readily available and are described in Table 1. Each workload was first run with base pages, then with superpages. They were rerun with profiling before moving onto the next workload. All superpages were released for each base page run and reallocated for superpage runs. All tests ran without rebooting without any failures to allocate superpages.

[2] http://oprofile.sourceforge.net/news/

7.1 STREAM (Memory Throughput)

STREAM [MCCALPIN07] is a synthetic memory bandwidth benchmark that measures the performance of four long vector operations: Copy, Scale, Add and Triad. It can be used to calculate the number of floating point operations performed during the benchmark to estimate the cost of the "average" memory access. Simplistically, more bandwidth is better.

The C version of the benchmark was selected and used three statically allocated arrays for calculations. Modified versions of the benchmark using `malloc()` and `get_hugepage_region()` were found to have similar performance characteristics.

The benchmark has two parameters: `N`, the size of the array and `OFFSET`, the number of elements padding the end of the array. A range of values for `N` were used to generate workloads between 128K and 2GB. For each size of `N` chosen, the benchmark was run 10 times and an average taken. The benchmark is sensitive to cache placement and optimal layout varies between architectures as noted on the benchmark author's homepage. Where the standard deviation of 10 iterations exceeded 5% of the throughput, `OFFSET` was increased to add one cache-line of padding between the arrays and the benchmark for that value of `N` reran. High standard deviation were only observed when the total working set was around the size of the L1, L2 or all caches combined.

The benchmark avoids data re-use, be it in registers or in the cache. Hence, benefits from superpages would be due to fewer faults, a slight reduction in TLB misses as fewer TLB entries are needed for the working set and an increase in available cache as less translation information needs to be stored.

To use superpages, the benchmark was first compiled with the `libhugetlbfs` **ld** wrapper to align the text and data sections to a superpage boundary [LIBHTLB09]. It was then loaded with `hugectl --text --data --no-preload`.

Figure 1 shows the comparison for the different STREAM operations when using base pages and superpages. On the test machines, there were massive reductions in DTLB misses. In the case of PPC64, it is reported as a 100% reduction but due to limitations to OProfile, TLB misses were not reduced to zero but there were so few that OProfile did not sample a miss.

X86-64 gained 3.5% in the Add operation but was offset by a 3.31% regression in the Triad operation. Again, DTLB misses were significantly reduced and the time spent servicing misses was reduced from 1.84% to close to 0%. Cache misses were also reduced overall. The differences may again be potentially explained by cache coloring with Triad suffering significant conflicts.

On STREAM, data is not often re-used so increased cache capacity does not necessarily help STREAM unless the data is pre-fetched and cache conflicts are not an issue. As cache misses were not significantly reduced on X86-64, it implies that data was not being accurately pre-fetched or there were significant cache conflicts. Further investigation showed that there were 60 times more cache misses than TLB misses implying that cache conflicts

CPU	AMD Phenom® 9950 Quad-Core
CPU Frequency	1.3GHz
# Physical CPUs	1 (4 cores)
L1 Cache	64K Data, 64K Instruction per core
L2 Cache	512K Unified per core
L3 Cache	2048K Unified Shared per chip
Main Memory	8 GB
Mainboard	Gigabyte GA-MA78GM-S2H
Superpage Size	2MB (1GB available but unused)
Est. TLB Miss Cost	51 cycles
Machine Model	Custom built

X86-64

CPU	PPC970MP, altivec supported
CPU Frequency	2.5GHz
# Physical CPUs	2 x dual core (4 cores in all)
L1 Cache	32K Data, 64K Instruction per core
L2 Cache	1024K Unified per core
L3 Cache	N/a
Main Memory	10GB
Mainboard	Specific to the machine model
Superpage Size	16MB
Est. TLB Miss Cost	563 cycles
Machine Model	Terrasoft YDL Powerstation

PPC64

Table 1. Machine Configurations Used For Evaluation

were a greater problem than TLB misses for this workload in the chosen configuration.

In contrast, PPC64 has a very high TLB miss cost and the reduction in misses resulted in large performance gains of between 11.27% for the Copy operation to 16.57% for Triad. Further, an estimated 18.23% of time was spent servicing TLB misses for the STREAM benchmark. As cache misses only occurred 3-4 times more than TLB misses, it made TLB misses a more dominant factor than the other test machines. Cache misses were also significantly reduced, possibly helped by the fact that the arrays were rarely aligned to a superpage boundary.

The results show that TLB misses are not always a significant bottleneck for workloads with poor cache locality. In the situation where data structures are aligned to a superpage size, superpages can in fact impair performance due to increased cache conflicts depending on the CPU implementation. For the STREAM benchmark in particular, there is a potential for between 1.8% and 5% of performance to be gained by using superpages on X86-64 machines but care should be taken in future tests to avoid aligning arrays to the superpage size. In the PPC64 case, significant performance increases can be achieved with superpages due to the high cost of a TLB miss.

7.2 SysBench (Database Workload)

OnLine Transaction Processing (OLTP) is a general class of workload where clients perform a sequence of operations whose end result must appear to be an indivisible operation. TPC-C[3] is considered an industry standard for the evaluation of OLTP but requires significant capital investment and is extremely complex to setup. Sysbench[4] is a system performance benchmark comprising file IO, scheduler, memory allocation, threading and OLTP benchmarks. The setup requirements are less complicated and supports database back-ends for MySQL, Postgres and Oracle. Postgres[5] was used for this experiment on the grounds that it uses a shared memory segment similar to Oracle making it a meaningful comparison with a commercial database server. Both Sysbench 0.4.8 and Postgres 8.3.4 were built from source.

Postgres was configured to use a 756MB shared buffer, an effective cache of 150MB, a maximum of 6*CPUs were allowed to connect and the "update_process_title" turned off. Options that checkout, fsync, log or synchronise were turned off to avoid interference from IO. **pg_ctl** was invoked with `hugectl --shm pg_ctl` when backing the shared memory segments superpages.

The system was configured to allow the postgres user to use superpages with `shmget()` (`hugetlb_shm_group`). Postgres uses System V shared memory but it has no superpage awareness. To transparently use superpages, **pg_ctl** was invoked with `hugectl --shm pg_ctl` to automatically back shared memory with superpages.

The Sysbench client did not use superpages as it would be expected that a database administrator has limited control of the clients. The table size was 10 million rows, read-only to avoid IO and the test type was "complex" described making each operation by the client a database transaction. Tests were run varying the number of clients accessing the database from one to four times the number of CPU cores in the system, a total of 16 clients for both systems.

Figure 2 shows the comparison of the number of transactions per second when using base pages and superpages. Overall, X86-64 shows a 2.81% improvement and 4.57% on PPC64 with reliable reductions in TLB, although not massive reductions on X86-64. Broadly speaking, the performance gains were close to predicted maximums of 2.3% for X86-64 and 3.63% for PPC64. Superpages performed slightly better than predicted due to the slight reduction in cache misses and remaining differences can be accounted for by small performance differences in performance when profiling. The difference between predictions and reality are so slight that there is a high degree of confidence that all potential performance improvements due to superpages are being achieved.

Anecdotal evidence in private communication with performance analysts presume that database workloads benefit from superpages by between 2% and 7% and these results would appear to correlate.

7.3 SPECcpu 2006 v1.1 (Computational)

SpecCPU 2006 v1.1[6] is a standardised CPU-intensive benchmark used in evaluations for HPC that also stresses the memory subsystem. A `--reportable` run was made comprising "test", "train" and three "ref" sets of input data. Three sets of runs compare base pages, superpages backing just the heap and superpages backing text, data and the heap. Only base tuning was used with no special compile options other than what was required to compile the tests. The benchmark suite classifies the run as "not publishable" on the grounds it did not recognise the `-O2 -m64` compiler switches but this is not considered significant.

Figure 3 shows the comparison of the time taken to complete the Integer portion of the benchmark when using base pages and superpages.

The difficulty in analysing the performance of this benchmark was that profiling introduced an unusually large amount of overhead in comparison to other benchmarks in this study. In comparison to other workloads, the number of PMU events recorded was

[3] http://www.tpc.org/tpcc/

[4] http://sysbench.sourceforge.net/

[5] http://www.postgresql.org/

[6] http://www.spec.org/cpu2006/

X86-86 Test Machine

Size	Tput Add	Tput Copy	Tput Scale	Tput Triad	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
53MB	1.0401	1.0093	1.0016	0.9679	99.9245	1.8677	0.0009	6.8965
85MB	1.0250	1.0027	1.0019	0.9558	99.9279	1.8558	0.0008	6.6690
128MB	1.0491	1.0089	1.0007	0.9776	99.9513	1.8460	0.0006	6.6712
213MB	1.0326	1.0111	1.0135	0.9736	99.9411	1.8541	0.0007	6.1082
341MB	1.0384	0.9995	0.9975	0.9608	99.8270	1.8381	0.0019	5.8182
512MB	1.0454	1.0077	0.9990	0.9734	99.8184	1.8370	0.0020	6.1538
853MB	1.0159	0.9902	0.9919	0.9524	99.7965	1.8285	0.0022	6.0349
1GB	1.0246	0.9943	0.9902	0.9690	99.7841	1.8395	0.0024	5.9608
2GB	1.0445	1.0020	0.9948	0.9723	99.7831	1.8352	0.0024	6.0384
	1.0350	1.0029	0.9990	0.9669	99.8050	1.8446	0.0013	6.0513

PPC64 Test Machine

Size	Tput Add	Tput Copy	Tput Scale	Tput Triad	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
53MB	1.1828	1.1350	1.1179	1.1717	100.0000	18.0229	0.0000	16.5982
85MB	1.1416	1.0841	1.1338	1.1614	100.0000	18.3195	0.0000	8.9795
128MB	1.1588	1.0923	1.0882	1.1544	100.0000	18.3854	0.0000	12.0667
213MB	1.1649	1.1380	1.1515	1.1803	100.0000	17.8887	0.0000	16.8234
341MB	1.1830	1.1338	1.1159	1.1702	100.0000	18.2086	0.0000	16.0503
512MB	1.1547	1.0965	1.0941	1.1564	99.9919	18.4399	0.0015	7.7781
853MB	1.1404	1.0983	1.1409	1.1565	99.9951	18.3760	0.0009	5.2285
1GB	1.1621	1.1366	1.1491	1.1783	100.0000	17.9392	0.0000	16.7136
2GB	1.1534	1.0964	1.0943	1.1550	99.9980	18.4667	0.0004	7.7075
	1.1601	1.1121	1.1204	1.1649	99.9978	18.2262	0.0000	10.8746

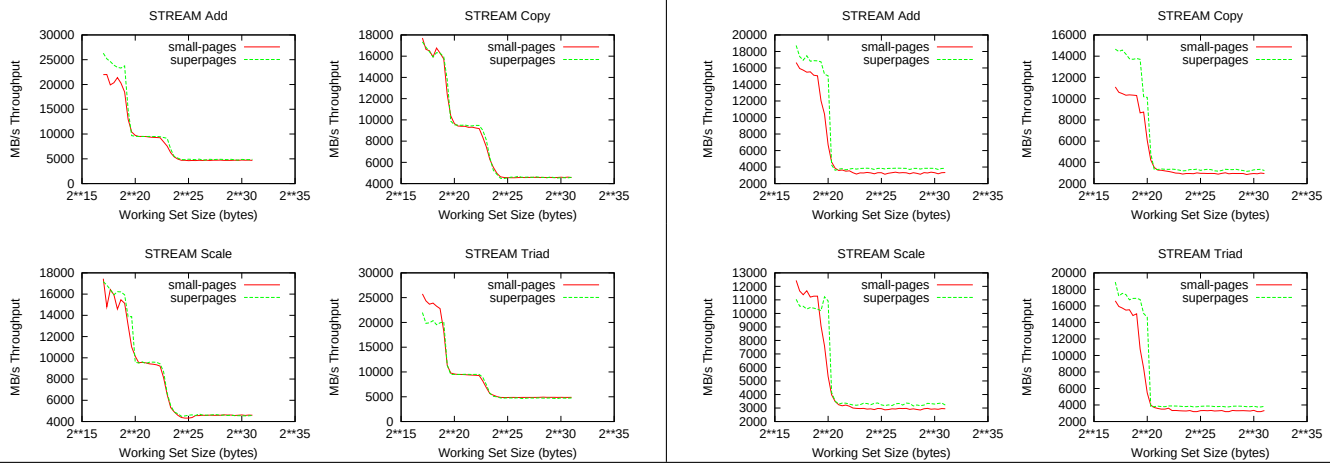
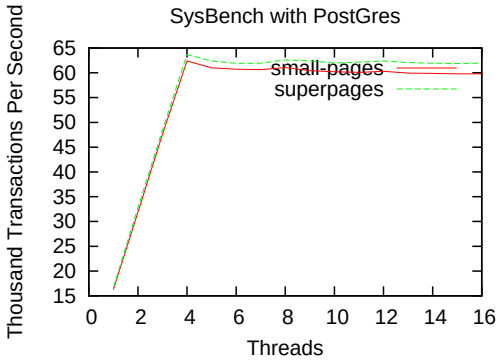


Figure 1. STREAM Comparison with Superpages

X86-64 Test Machine

Threads	T/ps	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
1	1.0230	12.2237	4.6178	2.2812	4.2396
2	1.0286	17.5468	4.0018	1.8473	5.5468
3	1.0166	15.2556	4.4714	2.1289	5.8291
4	1.0204	13.1550	4.5500	2.2249	5.7533
5	1.0230	12.4769	4.5357	2.2289	3.3353
6	1.0193	12.0048	4.5196	2.2263	9.9602
7	1.0217	12.0871	4.5047	2.2174	8.3249
8	1.0255	12.9372	4.5109	2.2043	4.1955
9	1.0326	12.1535	4.4801	2.2106	5.3998
10	1.0294	12.3884	4.5270	2.2276	4.9386
11	1.0341	12.5336	4.5210	2.2224	4.2965
12	1.0347	12.6007	4.5041	2.2101	5.8149
13	1.0358	12.8133	4.5060	2.2075	6.2652
14	1.0342	13.0694	4.5217	2.2089	2.8603
15	1.0350	12.7462	4.5131	2.2127	4.8027
16	1.0365	12.6255	4.5224	2.2220	3.9851
	1.0281	12.8751	4.4860	2.1904	5.3812



PPC64 Test Machine

Threads	T/ps	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
1	1.0378	85.2803	4.1822	0.5686	-3.7405
2	1.0326	81.0006	4.0372	0.7128	1.3170
3	1.0337	86.5839	4.0756	0.5075	4.0601
4	1.0295	85.6629	4.0587	0.5407	-0.2858
5	1.0356	85.0158	4.2660	0.5934	-2.4747
6	1.0453	76.1603	4.3419	0.9561	-4.3690
7	1.0465	78.5018	4.4753	0.8888	-6.6095
8	1.0456	74.0672	4.4363	1.0566	-6.7646
9	1.0454	74.6075	4.6659	1.0911	-5.6435
10	1.0491	74.9730	4.8548	1.1192	-9.9730
11	1.0501	76.1698	4.7760	1.0484	-1.7432
12	1.0520	78.5839	4.6714	0.9211	-9.2144
13	1.0558	74.8930	4.7346	1.0952	-3.8126
14	1.0591	72.6271	4.8246	1.2126	-3.4165
15	1.0579	74.3348	4.8794	1.1500	-3.6092
16	1.0560	77.2950	4.8517	1.0115	-1.7645
	1.0457	77.7027	4.4980	0.8701	-3.8722

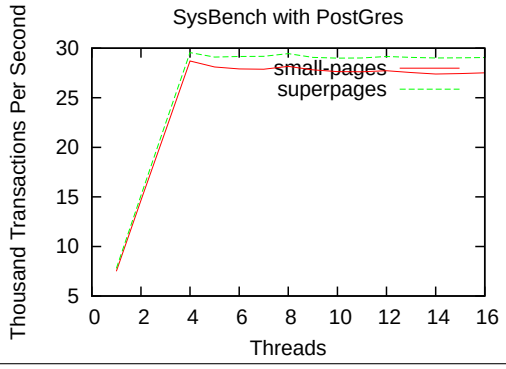


Figure 2. Sysbench Comparison with Superpages

far higher making the overhead of interrupts was correspondingly higher and a greater amount of CPU time was dedicated to the profiler than on other benchmarks. Unfortunately, reducing the granularity impaired the accuracy of the TLB measurements and could not be otherwise worked around on the available hardware. In all case where the actual performance exceeded the predicted performance, the difference was found in the profiling overhead.

X86-64 showed negligible difference in the number of TLB misses due to the erratic nature of the reference pattern for many of the tests. However, as the cost of a superpage TLB miss is often lower on that architecture due to the different number of pagetable levels, there can still be a performance gain in many cases. PPC64 in some cases reduced the number of TLB misses due to taking advantage of a larger superpage size but they were in the minority.

Despite the difficulties in the exact analysis of the reason for the performance gain, X86-64 gained 10.40% and PPC64 gained 15.14%.

Figure 4 shows the comparison of the time taken to complete the Floating Point portion of the benchmark when using base pages and superpages.

On X86-64 there was a negligible difference in the number of TLB misses reduced and gains were due to a lower cost of a superpage TLB miss. Similarly, the profiling overhead was unavoidable and made an accurate analysis of the TLB behaviour problematic but where actual performance gains differed from expected gains, a significant portion of the difference was accounted for by the profiling overhead. Overall, X86-64 gained 5.19% and PPC64 gained 8.20%.

7.4 SPECjvm 2008 (Java)

Java is used in an increasing number of scenarios, including real time systems and it dominates in the execution of business-logic related applications. Particularly within application servers, the Java Virtual Machine (JVM) uses large quantities of virtual address space that can benefit from being backed by superpages. SpecJVM 2008[7] is a benchmark suite for Java Runtime Environments (JRE). According to the documentation, the intention is to reflect the performance of the processor and memory system with a low dependence on file or network I/O. Crucially for HPC, it includes SCIMark[8] which is a Java benchmark for scientific and numerical computing.

The 64-bit versions of IBM Java Standard Edition Version 6 SP 3 were used for both X86-64 and PPC64. Installation on PPC64 required `libstc++-5` from outside of the distribution as the distribution uses `libstc++-6` and the backwards compatibility library was 32-bit only. The JVM was configured to use a maximum of 756MB for a heap. Unlike the other benchmarks, the JVM is superpage-aware and uses superpage-backed shared memory segments when `-Xlp` is specified to the JVM. Superpages were allocated in a static pool as required by the JVM. As tuning parameters were used, the `--peak` parameter was specified as required by the run rules.

Figure 5 shows the comparison of the number of transactions per second when using base pages and superpages. Overall, X86-64 shows a 7.57% performance improvement with 4.54% on PPC64 with reliable reductions in TLB misses. There are a wide range of results in the test categories with `scimarklarge` suffering despite a prediction that tests would gain by between 2% and 21.2% on the three machines. An examination of the sub-tests that make up the `scimarklarge` benchmark showed significantly different TLB behaviour and performance benefits with a mix of gains and losses complicating a general analysis from the context of superpages. In general, the differences in the sub-tests account for the apparent difference between predicted and actual performance.

The disparity of the results show that superpages can help JVM workloads but by how much or little depends on the exact workload being executed within the JVM. As enabling superpages on JVMs is generally so trivial, the best recommendation is to test the workload with and without superpages measuring both absolute performance and profiling CPU and TLB miss counts.

[7] http://www.specbench.org/jvm2008/

[8] http://math.nist.gov/scimark2/

8. Conclusions

This paper noted that transparent superpage support did not take architectural limitations into account and overly simplified cost functions. Explicit support is usable by 99.6% of the architectures making up the Top 500 supercomputer list, was implemented for the operating system running on 87.8% of them and does not necessarily require application modification. Evaluation was performed on two significantly different machines.

Average performance improvements due to explicit superpage use were very roughly in the range of 2% to 10% on X86-64 and 4% to 15% on PPC64 running a variety of workloads without imposing additional overhead on applications using exclusively base pages. PPC64 typically benefited to a greater degree than X86-64 which can be mostly accounted for by two facts. The first is that the cost of a TLB miss on the target machine was very high - 563 cycles versus 51 cycles for a miss on the X86-64 based machine. The second is that the larger superpage size on PPC64 as well as the larger number of TLB entries meant that the TLB reach was increased by a far greater degree on PPC64 than on X86-64.

It was not reported in the figures but superpage use reduced the number of page faults incurred by the system by a factor related to the difference in size between base and superpages. The significance of this is mitigated by the fact that the workloads faulted the bulk of their data early in the process lifetime and did not page heavily. I/O rates are affected during application start-up as superpages are being allocated but as this principally affected application initialisation, it was not found to have made any meaningful performance difference to the target workload.

The performance improvements are less dramatic than predicted in early literature but those estimations were based on software TLB miss handlers and in-order processors. The returns have diminished due to faster hardware pagetable walkers and speculative address translation. However, bottlenecks still exist and superpages can reduce TLB miss rates as well as cache miss rates due to less translation information being stored. This may be of particular importance to virtualised workloads where translation an address in a guest is significantly more expensive than native translation.

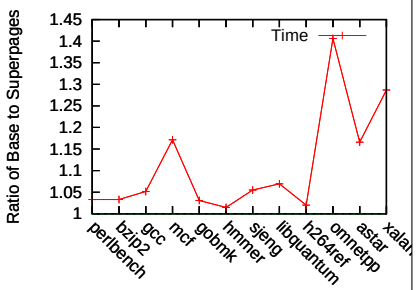
Explicit superpage provides an important tool for predictably mitigating the cost of address translation on a variety of platforms. It establishes a performance baseline for continued research on transparent superpage page and should have comparable performance to specialised OS kernels that enable superpage usage on a per-application basis.

Acknowledgments

The design and implementation of superpages support in Linux has a long history and would not have reached its current state without the sterling work of the Linux community, particularly Andy Whitcroft, William Irwin, Adam Litke, David Gibson, Nishanth Aravamudan, Peter Zijlstra, Kenneth Chen, Hugh Dickins, Nick Piggin, Steve Fox, Jon Tollefson, Andi Kleen and Eric Munson. Parts of the material presented in this work are supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

X86-64 Test Machine

Integer		Heap	TDH	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
perlbench	C	1.0331	1.0101	-0.1132	1.5427	0.8460	0.2181
bzip2	C	1.0331	1.0331	0.1093	4.2693	2.3828	-0.0008
gcc	C	1.0516	1.0075	-0.2129	2.0105	1.1155	-0.0501
mcf	C	1.1715	1.1523	-0.0025	19.1618	11.5114	-0.1049
gobmk	C	1.0307	1.0160	-0.1417	0.7487	0.4139	0.9051
hmmer	C	1.0147	1.0126	1.4300	0.0700	0.0378	-0.2978
sjeng	C	1.0505	1.0551	-0.1963	2.2159	1.2317	0.6472
libquantum	C	1.0547	1.0694	-0.0352	1.4599	0.8076	0.0036
h264ref	C	1.0200	0.9967	0.2062	1.0508	0.5822	0.0263
omnetpp	C++	1.4061	1.3306	0.0910	10.2110	5.8825	0.0526
astar	C++	1.1660	1.1175	-0.0048	13.2004	7.7051	-0.1015
xalancbmk	C++	1.2869	1.1135	-0.2086	6.2355	3.6936	-0.0024
		1.1040	1.0727	-0.0065	5.7429	3.4300	-0.0121



PPC64 Test Machine

Integer		Heap	TDH	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
perlbench	C	1.0101	1.0135	10.0610	3.7612	3.2238	5.0669
bzip2	C	1.0133	1.0216	7.5871	0.7693	0.6592	0.2785
gcc	C	1.1220	1.1209	23.9549	14.3698	13.2266	21.8105
mcf	C	1.4114	1.4183	0.2042	81.7208	75.2578	0.7505
gobmk	C	1.0128	1.0118	6.8243	3.3925	2.9355	2.2177
hmmer	C	1.0517	1.0372	7.6516	0.1102	0.0944	2.8921
sjeng	C	1.0795	1.0666	4.2515	12.1977	10.8760	8.8339
libquantum	C	1.2766	1.2665	0.0225	15.3595	14.1098	-2.5114
h264ref	C	1.0114	1.0031	2.4418	5.5046	5.6642	9.7435
omnetpp	C++	1.3962	1.3974	4.5856	48.2496	44.8656	4.5267
astar	C++	1.3530	1.3542	0.2057	62.8308	58.1426	0.3568
xalancbmk	C++	1.1949	1.1923	9.6174	28.1254	24.7488	2.9312
		1.1514	1.1488	2.8016	25.8631	23.8843	3.4427

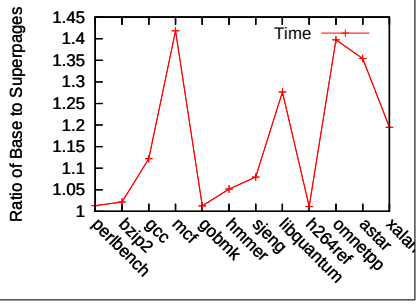
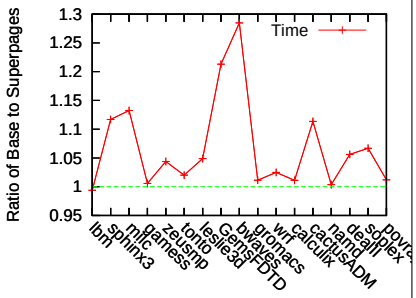


Figure 3. SPECcpu Integer Comparison with Superpages

X86-64 Test Machine

Floating Point		Heap	TDH	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
lbm	C	0.9327	0.9938	0.0064	1.5556	0.8606	-0.0705
sphinx3	C	1.1170	0.7827	-0.0529	2.5732	1.2840	0.0373
milc	C	1.1179	1.1326	-0.0180	4.6696	2.6102	-0.0113
gamess	Fortran	1.0019	1.0056	11.4262	0.0087	0.0042	-1.3196
zeusmp	Fortran	1.0000	1.0438	0.0364	7.4512	4.2452	0.0622
tonto	Fortran	1.0201	1.0085	0.7181	0.2380	0.1305	0.2394
leslie3d	Fortran	1.0488	1.0265	-0.0813	1.1342	0.6262	0.1054
GemsFDTD	Fortran	1.2130	1.1221	-0.0622	7.5406	4.3192	-0.0290
bwaves	Fortran	1.2830	1.2848	0.0032	0.5356	0.2947	0.3994
gromacs	C/Fortran	1.0113	1.0113	0.9641	0.0506	0.0275	1.1282
wrf	C/Fortran	1.0238	1.0248	-0.3859	0.7072	0.3913	-0.0687
calculix	C/Fortran	1.0104	1.0112	0.2856	0.0653	0.0358	0.5416
cactusADM	C/Fortran	1.0413	1.1135	-0.4269	22.2966	13.6379	-0.0667
namd	C++	1.0036	1.0018	0.5012	0.0218	0.0119	-0.0429
dealII	C++	1.0561	1.0386	-0.3481	0.4911	0.2707	0.0464
soplex	C++	1.0406	1.0669	-0.0229	3.5048	1.9580	-0.0296
povray	C++	1.0120	1.0120	-0.6250	0.0042	0.0023	0.5437
		1.0519	1.0355	-0.2543	5.3728	3.2626	0.0696



PPC64 Test Machine

Floating Point		Heap	TDH	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
lbm	C	1.0013	0.9962	0.0255	17.5501	16.2041	-0.0984
sphinx3	C	1.1497	1.1410	0.8974	16.5392	15.2007	0.5367
milc	C	1.2701	1.2701	0.3120	30.1260	27.8394	0.1782
gamess	Fortran	1.0000	0.9898	66.0211	0.0083	0.0026	9.4328
zeusmp	Fortran	1.0008	1.0605	33.2100	28.5960	18.1445	2.7259
tonto	Fortran	1.0043	1.0091	32.4662	0.4822	0.3761	30.8421
leslie3d	Fortran	1.0631	1.0659	1.0831	6.2687	5.8422	-0.0215
GemsFDTD	Fortran	1.5143	1.5161	1.3658	33.0646	30.5099	0.2088
bwaves	Fortran	1.1774	1.1770	0.1540	4.2969	4.6088	0.3614
gromacs	C/Fortran	0.9806	0.9798	2.0777	0.2020	0.2005	0.8670
wrf	C/Fortran	1.0382	1.0398	1.4789	3.6582	3.7127	1.4317
calculix	C/Fortran	0.9075	1.0079	3.1309	0.4243	0.3830	-0.0425
cactusADM	C/Fortran	1.0108	1.0167	0.3439	3.1349	2.8760	0.1809
namd	C++	0.9975	0.9963	1.7978	0.0987	0.0895	1.3470
dealII	C++	1.0587	1.0587	3.6721	3.0087	2.8840	1.5291
soplex	C++	1.2005	1.1990	1.8860	21.4962	19.6059	0.4767
povray	C++	0.9923	0.9942	46.2185	0.0069	0.0035	13.4026
		1.0722	1.0820	5.2703	11.5641	10.0068	0.9617

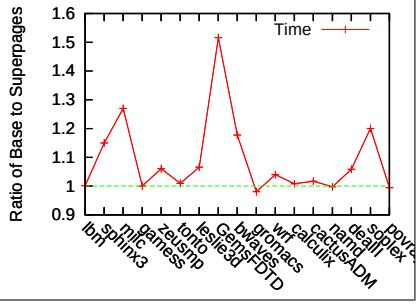
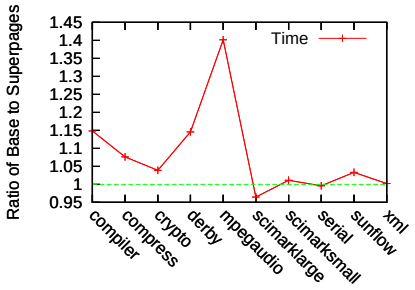


Figure 4. SPECcpu Floating Point Comparison with Superpages

X86-64 Test Machine

	Ops/m	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
compiler	1.1485	98.7754	2.0298	0.0197	33.4153
compress	1.0763	99.4674	0.0613	0.0002	48.8930
crypto	1.0389	98.9939	0.1388	0.0008	60.2028
derby	1.1457	97.1308	3.9328	0.1101	41.3023
mpegaudio	1.4017	99.4224	0.0709	0.0003	66.0157
scimarklarge	0.9645	99.9929	2.9133	0.0001	3.0529
scimarksmall	1.0113	99.6994	0.0389	0.0001	3.0988
serial	0.9959	95.9985	1.1244	0.0570	56.6005
sunflow	1.0333	99.9352	0.5843	0.0003	52.6016
xml	1.0023	98.7256	0.8647	0.0090	42.9273
	1.0757	99.2009	0.4458	0.0016	17.5977



PPC64 Test Machine

	Ops/m	DTLB Miss Rdn	DTLB Time % Base	DTLB Time % Super	Cache Miss Rdn
compiler	1.0618	99.7249	8.4054	0.0302	34.3786
compress	1.0115	99.3789	0.3609	0.0023	39.6647
crypto	1.0112	98.9592	0.7599	0.0080	74.1076
derby	1.1434	99.8811	22.1169	0.0423	48.0293
mpegaudio	1.3311	99.7164	0.4113	0.0012	93.8209
scimarklarge	0.9933	99.9993	21.1900	0.0001	9.1881
scimarksmall	1.0094	99.7218	0.2607	0.0007	95.4315
serial	0.8732	97.5293	2.8358	0.1408	64.0971
sunflow	1.0106	99.8733	3.0512	0.0050	85.5716
xml	1.0655	98.8627	5.0768	0.0837	68.6790
	1.0454	99.8526	2.3478	0.0066	33.8337

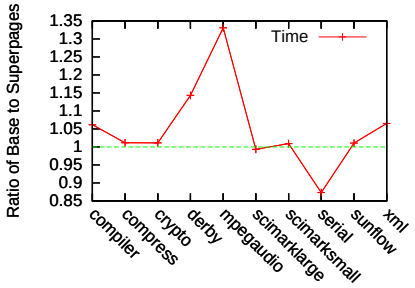


Figure 5. SPECjvm Comparison with Superpages

References

[POWER03] Joe Wetzel, Ed Silha, Cathy May, Brad Frey. PowerPC Operating Environment Architecture Book III v2.01

[MSSD08] Hans Meuer, Erich Strohmaier, Horst Simon and Jack Dongarra. Top 500 Supercomputers List for November 2008. In http://www.top500.org

[GOR08] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support In ISMM '08: Proceedings of the 7th international symposium on Memory management,

[TALLURI92] Madhusudhan Talluri and Shing Kong and Mark D. Hill and David A. Patterson and Sun Microsystems Laboratories. Tradeoffs in supporting two page sizes In Proceedings of the 19th Annual International Symposium on Computer Architecture,

[NAV04] Juan E. Navarro. Transparent operating system support for superpages PhD Thesis, Rice University, Houston, TX, USA, 2004.

[CASCAVAL05] Calin Cascaval and Evelyn Duesterwald and Peter F. Sweeney and Robert W. Wisniewski Multiple Page Size Modeling and Optimization In Proc. 14th International Conference on Parallel Architecture and Compilation Techniques.

[CWH03] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium Page Tables and TLB Technical Report UNSW-CSE-TR-0307, School of Computer Science and Engineering, University of NSW, May 2003.

[WIENARD08] Ian Wienard Transparent Large-Page Support for Itanium Linux Masters Thesis, University of NSW, July 2008.

[MCCALPIN07] John D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers In a continually updated technical report. http://www.cs.virginia.edu/stream/

[GANAPATHY98] Narayanan Ganapathy and Curt Schimmel, General purpose operating system support for multiple page sizes In In Proceedings of the USENIX Conference. USENIX

[LIBHTLB09] Various Authors, libhugetlbfs 2.7 HOWTO In the libhugetlbfs source tarball. http://sourceforge.net/projects/libhugetlbfs

[GOR08a] Mel Gorman, Using the Direct Hugepage Allocation API with STREAM In author's home site, http://www.csn.ul.ie/ mel/docs/stream-api