Multiple Page Size Support in the Linux Kernel

Simon Winwood [‡] [§]

[‡]IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598, USA
{swinwoo, yefim, frankeh}@us.ibm.com

Yefim Shuf [‡] [¶]

[§]School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia
sjw@cse.unsw.edu.au

Hubertus Franke [‡]

[¶]Computer Science Department
Princeton University
Princeton, NJ 08544, USA
yshuf@cs.princeton.edu

Abstract

The Linux kernel currently supports a single user space page size, usually the minimum dictated by the architecture. This paper describes the ongoing modifications to the Linux kernel to allow applications to vary the size of pages used to map their address spaces and to reap the performance benefits associated with the use of large pages.

The results from our implementation of multiple page size support in the Linux kernel are very encouraging. Namely, we find that the performance improvement of applications written in various modern programming languages range from 10% to over 35%. The observed performance improvements are consistent with those reported by other researchers. Considering that memory latencies continue to grow and represent a barrier for achieving scalable performance on faster processors, we argue that multiple page size support is a necessary and important addition to the OS kernel and the Linux kernel in particular.

1 Introduction

To achieve high performance, many processors supporting virtual memory implement a Translation Lookaside Buffer (TLB) [8]. A TLB is a small hardware cache for maintaining virtual to physical translation information for recently referenced pages. During execution of any instruction, a translation from virtual to physical addresses needs to be performed at least once. Thereby, a TLB is effectively reducing the cost of obtaining translation information from page tables stored in memory.

Programs with good spatial and temporal locality of reference achieve high TLB hit rates which contribute to higher application performance. Because of long memory latencies, programs with poor locality can incur a noticeable performance hit due to low TLB utilization. Large working sets of many modern applications and commercial middleware [12, 13] make achieving high TLB hit rates a challenging and important task.

Adding more entries to a TLB to increase its coverage and increasing the associativity of a TLB to reach higher TLB hit rates is not always feasible as large and complex TLBs make it difficult to attain short processor cycle times. A short TLB latency is a critical requirement for many modern processors with fast physically tagged caches, in which translation information (i.e., a physical page associated with a TLB entry) needs to be available to perform cache tag checking [8]. Therefore, many processors achieve wider TLB coverage by supporting large pages. Traditionally, operating systems did not expose large pages to application software, limiting this support to the kernel. Growing working sets of applications make it appealing to support large pages for applications, as well as for the kernel itself.

A key challenge for this work was to provide efficient support for multiple page sizes with only minor changes to the kernel. This paper discusses ongoing research to support multiple page sizes in the context of the Linux operating system, and makes the following contributions:

- it describes the changes necessary to support multiple page sizes in the Linux kernel;
- it presents validation data demonstrating the accuracy of our implementation and its ability to meet our design goals; and

- it illustrates non-trivial performance benefits of large pages (reaching more than 35%) for Java applications and (reaching over 15%) for C and C++ applications from well-known benchmark suites.

We have an implementation of multiple page size support for the IA-32 architecture and are currently working on an implementation for the PowerPC[1] architecture.

The rest of the paper is organized as follows. In Section 2, we present an overview of the Linux virtual memory subsystem. We describe the design and implementation of multiple page size support in the Linux kernel in Section 3 and Section 4 respectively. Experimental results obtained from the implementation are presented and analyzed in Section 5. Related work is discussed in Section 6. Finally, we summarize the results of our work and present some ideas for future work in Section 7.

2 The Virtual Memory Subsystem in Linux

In this section, we give a brief overview of the Linux Virtual Memory (VM) subsystem[2]. Unless otherwise noted, this section refers to the 2.4 series of kernels after version 2.4.18.

2.1 Address space data structures

Each address space is defined by a `mm_struct` data structure[3]. The `mm_struct` contains information about the address space, including a list of *Virtual Memory Areas* (VMAs), a pointer to the page directory, and various locks and resource counters.

A VMA contains information about a single region of the address space. This includes:

- the address range the VMA is responsible for;
- the access rights — read, write, and execute — for that region;
- the file, if any, which backs the region; and
- any performance hints supplied by an application, such as memory access behaviour.

A VMA is also responsible for populating the region at page fault time via its `nopage` method. A VMA generally maps a virtual address range onto a region of a file, or zero filled (anonymous) memory.

A VMA exists for each segment in a process's executable (e.g., its text and data segments), its stack, any dynamically linked libraries, and any other files the process may have mapped into its address space. All VMAs, except for those created when a process is initially loaded, are created with the `mmap` system call[4]. The `mmap` system call essentially checks that the process is allowed the desired access to the requested file[5] and sets up the VMA.

The *page directory* contains mappings from virtual addresses to physical addresses. Linux uses a three level *hierarchical page table* (PT), although in most cases the middle level is optimised out. Each leaf node entry in the PT, called a *page table entry* (PTE), contains the address of the corresponding physical page, the current protection attributes for that page[6], and other page attributes such as whether the mapping is dirty, referenced, or valid.

Figure 1 shows the relationship between the Virtual Address Space, the `mm_struct`, the VMAs, the Physical Address Space, and the page directory.

2.2 The page data structure and allocator

The `page` data structure represents a page of physical memory, and contains the following properties:

- its usage count, which denotes whether it is in the page cache, if it has buffers associated with it, and how many processes are using it;

[1] This is for the PPC405gp and PPC440 processors, both of which support multiple page sizes.

[2] This section is meant to be neither exhaustive or complete.

[3] Note that multiple tasks can share the same address space

[4] This is not strictly true: the `shmat` system call is also used to create VMAs. It is, however, essentially a wrapper for the `mmap` system call.

[5] This check is trivial if the mapping is anonymous.

[6] The pages protection attributes may change over the life of the mapping due to copy-on-write and reference counting

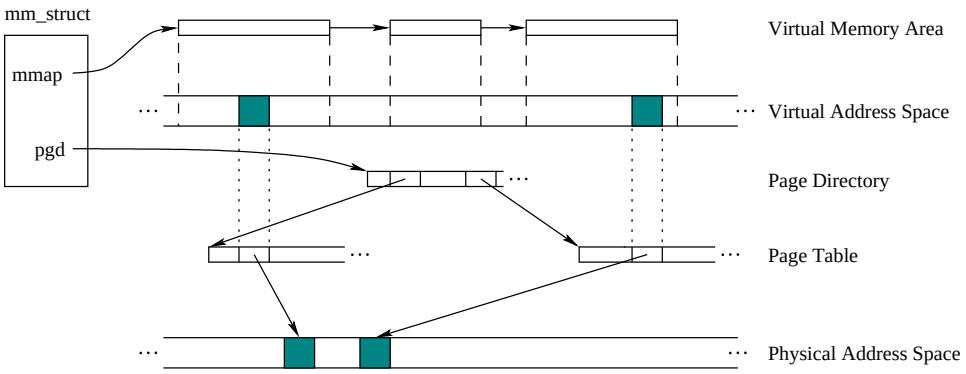

Figure 1: Virtual Address Space data structures

- its associated mapping, which indicates how a file is mapped onto its data, and its offset;
- its wait queue, which contains processes waiting on the page; and
- its various flags, most importantly:

  locked This flag is used to lock a page. When a page is locked, I/O is pending on the page, or the page is being examined by the swap subsystem.

  error This flag is used to communicate to the VM subsystem that an error occurred during I/O to the page.

  referenced This flag is used by the swapping algorithm. It is set when a page is referenced, for example, when the page is accessed by the `read` system call, and when a PTE is found to be referenced during the page table scan performed by the swapper.

  uptodate This flag is used by the page cache to determine whether the page's contents are valid. It is set after the page is read in.

  dirty This flag is used to determine whether the page's contents has been modified. It is set when the page is written to, either by an explicit `write` system call, or through a store instruction.

  lru This flag is used to indicate that the page is in the LRU list.

  active This flag is used to indicate that the page is in the active list.

  launder This flag is used to determine whether the page is currently undergoing swap activity. It is set when the page is selected to be swapped out.

Pages are organised into *zones*; memory is requested in terms of the target zone. Each zone has certain properties: the *DMA* zone consists of pages whose physical address is below the 16MB limit required by some older devices, the *normal* zone contains pages that can be used for any purpose (aside from that fulfilled by the *DMA* zone), and the *highmem* zone contains all pages that do not fit into the kernel's virtual memory: when the kernel needs to access this memory, it needs to be mapped into a region of the kernels address space. Note that the *DMA* zone is only required for support of legacy devices, and the *highmem* zone is only required on machines with 32 bit (or smaller) address spaces.

Each zone uses a buddy allocator to allocate pages, so pages of different orders can be allocated. Although a client of the allocator requests pages from a specific list of zones and a specific page order, the pages that are returned can come from anywhere within the zone. This means that a page for the slab allocator[7] can be allocated between pages that are allocated for the page cache. The same can be said for other non-swappable pages such as task control blocks and page table nodes.

[7]The *slab allocator*[5] provides efficient allocation of objects, such as `inodes`. Pages allocated to the slab allocator cannot be paged out.

2.3 The page cache

The page cache implements a general cache for file data. Most filesystems use the page cache to avoid re-implementing the page cache's functionality. A filesystem takes advantage of the page cache by setting a file's `mmap` operation to `generic_file_mmap`. When the file is `mmap`ed, the VMA is set up such that its `nopage` function invokes `filemap_nopage`. The file's `read` and `write` operations will also go through the page cache.

The page cache uses the page's mapping and offset fields to uniquely identify the file data that the page contains — when an access occurs, the page cache uses this data to look up the page in a hash table.

2.4 The swap subsystem

Linux attempts to fully utilise memory. At any one time, the amount of available memory may be less than that required by an application. To satisfy a request for memory, the kernel may need to free a page that is currently being used. Selecting and freeing pages is the job of the swap subsystem.

The swap subsystem uses two lists to record the activity of pages: a list of pages which have not been accessed during in a certain time period, called the *inactive* list, and a list of pages which have been recently accessed, called the *active* list. The active list is maintained pseudo LRU, while the inactive list is used by the one-handed clock replacement algorithm currently implemented in the kernel. Whenever a page on the inactive list is referenced, it is moved to the active list.

The kernel uses a swapper thread to periodically balance the number of pages in the active and inactive lists: if a page in the active list has been referenced, it is moved to the end of the active list, otherwise it is moved to the end of the inactive list.

Periodically, the swapper thread sweeps through the inactive list looking for pages that can be freed. If the swapper thread is unable to free enough pages, it starts scanning page tables: for each PTE examined, the kernel checks to see whether the page has been referenced (i.e., whether the *referenced* bit is set in the PTE). If so, the page is moved to the active list, if it is not already a member. Otherwise, the page is considered a candidate for swapping. In this manner reference statistics are gathered from the page tables in the system, and used to select pages to be swapped out and freed.

The swapper thread may be woken up when the amount of memory becomes too low. The swapper functions may also be called directly when the amount of free memory becomes critical: when memory allocation fails, a task may attempt to swap out pages directly.

2.5 Anatomy of a page fault

When a virtual memory address is accessed, but a corresponding mapping is not in the TLB, a *TLB miss* occurs. When this happens, the *fault address* is looked up in the page table, by either the hardware in systems with a hardware loaded TLB, or via the kernel in systems with a software loaded TLB (note that this implies an interrupt occurs).

If a mapping exists in the page table, is valid, and matches permissions with the type of the access, the entry is inserted into the TLB, the page table is updated to reflect the access by setting the referenced bit[8], and the faulting instruction is restarted.

If a valid mapping does not exist, the kernel's page fault handler is invoked. The handler searches the current address space's VMA set for the VMA which corresponds to the fault address, and checks whether the access requested is allowed by the permissions specified in the VMA.

The kernel then looks up the PTE corresponding to the fault address and allocates a page table node if necessary. If the fault is a write to a PTE marked read-only, the address space requires a private copy of the page. A page is allocated, the old page is copied, and the dirty bit is set in the PTE. If the PTE exists but isn't valid, the page needs to be swapped in, otherwise the page needs to be allocated and filled.

If the VMA does not define a `nopage` method, the memory is defined to be anonymous, i.e., zero-filled memory that is not associated with any device or file. In this case, the kernel allocates a page, zeroes

[8] Note that architectures with a hardware loaded TLB whose page table doesn't map directly onto Linux's need to simulate this bit

it, and inserts the appropriate entry into the page table. If a valid `nopage` method exists, it is invoked and the resulting page is inserted into the PTE.

In the majority of filesystems, the `nopage` method goes to the page cache. The mapping and offset for the fault address are calculated — the information required for this calculation is stored in the VMA — and the page cache hash table is searched for the file data corresponding to the mapping and offset.

If an up-to-date page exists, then no further action is required. If the page exists but is not up-to-date, it is read in. Otherwise, a new page is allocated, inserted into the page cache, and read in. In all cases, the page's reference count is incremented, and the page is returned.

3 Design

This section discusses the approaches we considered and justifies our final design. This section is organised as follows: Section 3.1 discusses the goals that guided the design and the terminology used throughout this and future sections. Section 3.2 discusses the semantics of large pages: what aspects of the support for large pages the kernel exports to user space, the granularity at which page size decisions are made, and the high-level abstractions the kernel exports to the user.

It should be noted that this is an ongoing project, so the approaches describe here may have been improved upon by the time of publication.

3.1 Goals

This section discusses the design goals and guidelines which we attempt to adhere to in the design of our solution. We consider a good design to have the following properties:

Low overhead We do not wish to penalise applications that will not benefit from large pages, so we aim to minimise the performance impact of our modifications for these applications.

Generic The Linux kernel runs on numerous different architectures[9] and is usually ported quickly to new architectures. Any kernel enhancements such as ours should be easily adaptable to support existing and future systems, especially considering that many modern architectures feature MMUs which support multiple page sizes.

Flexible While a generic solution allows for easy portability, it does not indicate how well such a solution takes advantage of an architectures support for multiple page sizes. The design should be flexible enough to encompass any support.

Simple The more complex a solution is, the more likely it is to have subtle bugs, and the harder it is to understand. While we can foresee a point at which a more complex solution may be necessary, the initial design should be as simple as possible.

Minimal The Linux kernel is a large and complex system, so a minimalist approach is required: subsystem modifications that are not absolutely required may result in a solution that is overly complex and unwieldy. Therefore, we try to limit our changes to the VM subsystem only.

3.2 Semantics

This section discusses the semantics associated with supporting multiple page sizes: how the page size for a range of virtual addresses is chosen and whether the kernel considers this page size mandatory or advisory.

The following terms are used throughout this and later sections:

Base page A *base page* is the smallest page supported by the kernel, usually the minimum dictated by the hardware.

Superpage A *superpage* is a contiguous sequence of 2^n base pages.

Order A superpage's *order* refers to its size. A superpage of order n contains 2^n *base pages*.

[9] A count of the number of architectures in the mainline kernel reveals 15 implementations that are more or less complete

Sub-superpage A *sub-superpage* is a superpage of order m, contained in a superpage of order n, such that $n \geq m$. Note that a base page is a sub-superpage with order $m = 0$

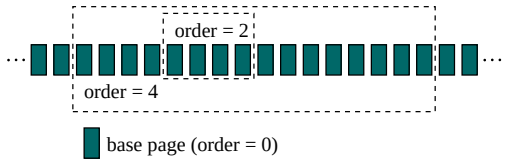

Figure 2: A superpage and sub-superpage

These concepts are illustrated in Figure 2 which shows a superpage of order 4 containing a sub-superpage of order 2.

3.2.1 Visibility

There are two basic approaches to supporting multiple page sizes: restrict knowledge of superpages to the kernel or export page size decisions to user space.

In the former approach, the kernel can create superpage mappings based on some heuristic, for example, a dynamic heuristic based on TLB miss information, or a static heuristic based on the type of mapping such as whether the mapping is for code, data, or whether it is anonymous. This approach is transparent to applications, and should result in all applications benefiting. It is, however, more complex, and would rely on effective heuristics to map a virtual address range with large pages.

In the latter approach, an application explicitly requests a section of its address space be mapped with superpages. This request could come in the form of programmer hints, or via instrumentation inserted by a compiler. While this approach requires applications to have specific knowledge of the operating system's support for large pages, it is much simpler from the kernels perspective. The major problem with this approach is that it requires the application programmer to have a good understanding of the applications memory behaviour.

We have decided on the latter approach, due to its simplicity: the former approach would necessitate developing heuristics that require fine-tuning and rewriting.

3.2.2 Granularity

This section discusses the granularity of control that the application has over page sizes. The approaches considered were:

per address space While making page sizes per address space would simplify some aspects of the implementation, it is too restrictive. We expect applications to have regions of their address space where the use of large pages would be a waste of memory;

per address space region type [10]
This approach also has its drawbacks: there is no clear set of types, although the region's attributes (e.g., executable, anonymous) could be used, so again this approach is limited without any clear gains;

per address space region This approach is more flexible than either of the above approaches, however it does not allow for hotspot mapping within a region; or

over an arbitrary address space range. This is the most flexible approach, however, there are implementation issues: the kernel would need to keep track of the applications desired page sizes for the entire address space.

To allow maximum flexibility while minimising implementation overhead, we have decided upon a combination of the last two options: an application can dictate the page size for an arbitrary address range only if that range belongs to an address space region. This means that an application can map a region hotspot with large pages, but leave the rest of the region at the system's default page size.

3.2.3 Interface

This section discusses the guarantees given about the actual page size used to map an address space range.

The kernel can take a best-effort approach to mapping a virtual address with the applications indicated page size, falling back to a smaller page size if

[10] A region is a defined part of the address space that created by the `mmap` system call, for example.

the larger page is not immediately available. Alternatively, the kernel can block the application until the desired page size becomes available, copying any existing pages to the newly allocates superpage.

Rather than mandating either behaviour, we have elected to allow the application to choose between the two alternatives. In situations where selecting a larger page size is merely an opportunistic optimisation for a relatively short running application, the first behaviour is desirable. In cases where the application is expected to execute for an extended period of time, however, the expected performance improvement may be greater than the expected wait time, and so waiting for a superpage to become available is justified. If an application is expected to re-use a large mapping over a number of invocations (a text page or a data file, for example), the application will benefit by waiting for the large page to be constructed.

4 Implementation

This section discusses the implementation of the design in Section 3.

4.1 Interface

An application requires some mechanism to communicate a desired page size to the kernel. A system call is the conventional mechanism for communicating with the kernel. In this section, we discuss our implementation of a system call interface for setting the page size for a region of the address space.

We considered three options: add a parameter to the `mmap` system call which specifying the page size for the new mapping; implement a new system call, `setpagesize`; and add another operation to the `madvise` system call.

Using the `mmap` system call would appear to be an obvious solution. It has, however, several negative aspects: firstly, the `mmap` system call is complex and is frequently used. Modifying `mmap`'s argument types would break existing code, as would adding extra parameters. Secondly, the application would be restricted to the one page size for that mapping, for the life of the mapping.

Using a new system call would be the cleanest alternative, however this requires significant modifications to all architectures, and is generally frowned upon where an alternative exists.

Using the `madvise` system call would allow an application to modify the page size at any point during its execution and would not affect existing applications, as any modification would be orthogonal to current operations.

We therefore added a *setregionorder*(n) operation to the `madvise` system call, where n is the new page order. We implemented this using the `advise` parameter of the `madvise` system call. The upper half of the parameter word contains the desired page order, while the lower half indicates that a *setregionorder* operation is to be performed.

Within the kernel, the `madvise` system call verifies that the requested page order is actually supported by the processor, and sets the VMA's `order` attribute accordingly.

4.2 Address space data structures

This section discusses the modifications made to the kernel's representation of a virtual address space. The application can modify the page size used by a VMA at runtime, either by an explicit `madvise` system call or by instructing the kernel to fall back to a smaller page size if a larger is not available. Consequently, the kernel needs to keep track of the following: firstly, the page size indicated by the application, which is associated with the VMA; secondly, the actual page size used to map a virtual address.

To communicate the requested page order to the VMA's `nopage` function, another parameter was added. This parameter indicates the desired page order at invocation, and contains the actual page size upon return. We rely upon the fact that subsystems which have not been modified will only return base pages.

To store the superpage size that actually maps the virtual address range, the PTE includes the order of the mapping. To achieve this, we associated unused bits within the PTE with different page sizes, although the actual bits and sizes may be dictated by hardware.

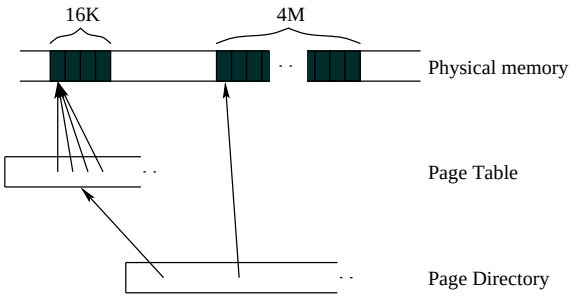

Figure 3: The modified page table structure

The page table structure was also modified: superpages which span a virtual address range greater or equal to that of a non-leaf page directory entry are collapsed until they fit into a single page table node (see Figure 3). This means that we can now have valid page table elements at each level of the address translation hierarchy. This affects kernel routines which scan the page table, for example, the swap routine.

Although the main reason behind this was to conform to the page table structure defined by the x86 family, it also has other advantages: the kernel can use positional information to determine the page size, rather than relying solely on the information store in the PTE. This means that the number of page sizes supported by the kernel is not restricted by the number of unused bits in the PTE (which can be quite few). There may also be some performance advantage as the TLB refill handler does needs to traverse fewer page table levels.

4.3 Representing superpages in physical memory

This section discusses the representation of superpages in the `page` data structure. The kernel needs to keep track of various properties of the superpage, such as whether it is freeable, whether it needs to be written back, etc. The superpage can include sub-superpages which are in use: any superpage operation that affects the sub-superpage also affects the superpage, and this needs to be taken into consideration.

We considered the following representations of superpages: firstly, an explicit hierarchy of `page` data structures, with one level for each possible order. A superpage would then be operated on using the `page` data structure at the appropriate level. This implies that each operation would only have to look at a single instance of the `page` data structure.

This approach is the cleanest in terms of semantics. Unfortunately, the kernel makes certain assumptions about the one-to-one relationship between the `page` data structure and the actual physical page. Implementing this design would violate those assumptions and also involve significant modifications to the lower levels of the kernel.

The alternative involves a modification to the existing `page` data structure, such that each page contains the order of the superpage it belongs to. A superpage of order n would then be operated on by iterating over all 2^n base pages. This approach conforms to the kernels existing semantics. It is, however, subject to various race conditions, and is inelegant.

We implemented a combination of the two approaches presented: while we do not have an explicit hierarchy, there is an implicit hierarchy created by storing the superpage's order in each component base page. We logically partition the properties of a page into those associated with superpages, or with base pages.

This partitioning was guided by the usage of these properties: if the property was used in the VM subsystem only, it was usually put in the superpage partition. If the property was used for I/O, it was put into the base page partition. The properties were then partitioned as follows:

- the page's `usage count` is per superpage. As all allocation are done in terms of superpages, it follows that a superpage is only freeable if no sub-superpage is being used. This means that whenever a sub-superpage's usage count is modified, the actual modification is applied to the superpage;
- the `mapping` and `offset` properties are per base page, as they are only used to perform I/O on the page;
- the `wait queue` is per base page, as it is used to signal when I/O has completed;
- the `flags` are partitioned as follows:

  locked is per base page, as it is used primarily to indicate that a page is undergoing I/O;

error is per base page, as it is used to indicate an I/O error in the page;

referenced is per superpage, as it is used by the VM subsystem only;

uptodate is per base page, as it is set when I/O successfully completes on a page;

dirty is per superpage, as it is primarily used in the VM subsystem;

lru is per superpage, as it indicates whether a page is in the LRU list, and the LRU list is now defined to contain superpages;

active is per superpage, as it indicates whether a page is in the active list, and the active list is now defined to contain superpages;

launder is per superpage, as it is only used in the swap subsystem, and the swap subsystem has to deal with superpages.

All other flags are per base page, as they reflect static properties of the page, (for example, whether the page is in the highmem zone).

Operations that iterate over each base page in a superpage are required to operate in ascending order to avoid deadlock or other inconsistencies.

4.4 Page allocation

The current page allocator supports multiple page sizes, however it has 2 major problems: firstly, non-swappable pages can be spread throughout each zone, causing memory fragmentation; secondly, if a large page is required, but a user (i.e. swappable) page is in the way, there is no efficient way to find all users of that page.

While the latter problem can be solved by Rik van Riel's reverse mapping patch[18], the former is still an issue. For this implementation, we have created another *largepage* zone, which is used exclusively for large pages. While this is not a permanent solution, it does aid in debugging, and solves the immediate problem for specialised users. The size of the *largepage* zone is fixed at boot time.

For maximum flexibility, the current allocator should be modified so that pages which are not pageable are allocated in so that they do not cause fragmentation. Also, pages which are allocated together will probably be freed together, so clustering pages at allocation time may also reduce fragmentation.

4.5 The Page Cache

To support mapping files with superpages, the page cache needs to be modified. The bulk of these modifications are in the `nopage` and affiliated functions, which attempt to allocate and read in a superpage of the requested order. To avoid any problems due to overlapping superpages, we require a superpage of order n also have file order n — that is, the alignment of the superpage in the virtual, physical, and file space is the same. For example, a 64K mapping of a file should be at a file offset that is a multiple of 64K, a virtual offset that is a multiple of 64K, and a physical offset of 64K[11].

The changes to the `nopage` function are essentially straightforward. If an application requests a superpage which contained in the page cache, it get back a sub-superpage whose order is the minimum of the requested order and the superpage's order. If a superpage does not exist, a page of the requested order is allocated, each base page is read in, and the superpage is added to the LRU and active queues.

Because reading in a large page can cause significant I/O activity (the amount of time required to read in 4MB of data from a disk can be significant), we may need to read in base pages in a more intelligent fashion. One solution is to read in the sub-superpage which contains the address of interest first and schedule the remainder of the superpage to be read in after the first sub-superpage has completed. When the rest of the superpage has completed I/O, the address space can be mapped with the superpage. Note that this is similar to the early restart method used in some modern processors to fetch a cache line.

4.6 The swap subsystem

In our current implementation, a region mapped with superpages will not be swapped out. Swapping a superpage would negate any performance gained by its use due to the high cost of disk I/O. The superpage may need to be written back, however, and

[11] The virtual and physical alignment constraints are common to most architectures.

this is handled in an essentially iterative manner — when the superpage is not being used by any applications, and it is chosen by the swap subsystem to be swapped out (i.e. when it appears as a victim on the LRU list), each base page is flushed to disk, and the superpage is freed.

In the future, a number of approaches present themselves. The kernel may, for example, split up a superpage into smaller superpages over a series of swap events, until a threshold superpage order is met, and then swap that out. Alternatively, the kernel may just swap out the entire page.

4.7 Architecture specifics

This section discusses the architecture specific aspects of our implementation. Although our implementation attempts to be generic, the kernel requires knowledge of the architecture's support for multiple page sizes and the additional page table requirements.

The architecture specific layer in our implementation consists mainly of page table operations, i.e., creating and accessing a PTE. To constructed a PTE, the kernel now uses `mk_pte_order`, which is identical to `mk_pte`[12] except for an additional `order` parameter. This function creates a PTE with which maps a page of order `order`. To allow the kernel to inspect a PTE, a `pte_order` function is required. This function returns the order of a PTE.

On architectures which use an additional page table (usually because it is required by the hardware), the `update_mmu_cache` needs to be modified to take superpages into consideration. The kernel also requires a mechanism to verify that a page size is supported. This is achieved by implementing the `pgorder_supported` function.

4.8 Anatomy of a large page fault

In systems with a hardware loaded TLB, a TLB miss is transparent to the kernel, and so is not different in the case of a large page. In architectures with a software TLB refill handler, the new page table structure needs to be taken into consideration: the handler needs to check whether each level in the page table hierarchy is a valid PTE. The refill handler also needs to extract the page size from the entry and insert the correct $(VA, PA, size)$ entry into the TLB.

If there is no valid mapping in the page table, a page fault occurs. As with the standard kernel, the VMA is found and the access is validated. The PTE is then found, although a page table node is not created if it is required — the page table node is allocated later on in the page fault process. This postponement in allocating page table nodes is required as the kernel does not know what size the allocated page will be: this is determined when the page is allocated.

On a write access to a page marked read-only in the PTE, a private copy is created and replaces the read-only mapping. This involves copying the entire superpage, so it is a relatively expensive operation — as with all superpage operations, there will only be overhead if the operations would not have been done on each base page. For example, writing a single character to a 4Mb mapping will result in the whole 4Mb being copied, which would not have occurred if the region was mapped with 4K pages. Conversely, if most or all of the base pages are to be written to, copying them in one operation may reduce the total overhead due to caching effects and the reduced number of page faults.

If no mapping exists, the VMA's `order` field is consulted to determine the application's desired page size. If there are pages mapped into the region defined by this order and the fault address, and the application has elected to opportunistically allocate superpages, the kernel selects the largest supported order that contains the fault address, no mapped pages, and is less than or equal to the desired order. Otherwise, the application's desired page order is selected.

After the kernel has determined the correct page order, it examines the VMA's `nopage` method. If the `nopage` method is not defined, a zeroed superpage is allocated and inserted into the page table. Otherwise, the `nopage` method is called with the calculated page order, and the result is inserted into the page table.

If the file that backs the VMA is using the page cache to handle page faults, the kernel searches the page cache for the file data associated with the fault

[12] For backwards compatibility, `mk_pte` calls `mk_pte_order` with order 0

I-TLB 4K pages	128 entries, 4-way SA
I-TLB 4M pages	Fragmented into 4K I-TLB
I-L1 cache	12K micro-ops
D-TLB 4K pages	64 entries, FA
D-TLB 4M pages	Shared with 4K D-TLB
D-L1 cache	8K, 64 byte CL, 4-way SA
unified L2 cache	256K, 64-byte CLS, 8-way SA

Table 1: Pentium 4 processor's memory system characteristics (Notation: CL - cache lines; CLS - cache lines, sectored; SA - set associative; FA - fully associative).

address. If a superpage is found, the minimum of the superpage's order and the requested order is used to determine the sub-superpage to be validated. The sub-superpage is then checked to ensure its contents are valid, and if so, it is returned. If the sub-superpage's contents is not valid, each base page is read in, and the sub-superpage is returned.

5 Experimental Results

In this section, we present and analyze the experimental data from our implementation of multiple page size support in the Linux kernel.

All results in this section were generated on a 1.8GHz Pentium 4 system with 512M of RAM. The Pentium 4 processor has separate instruction and data TLBs and supports two different page sizes: 4K and 4M[13]. Table 1 shows the parameters of the memory system of Pentium 4.

5.1 Validating the Implementation with a Micro-benchmark

This section presents and discusses the data validating the accuracy of our implementation and demonstrating the benefits of multiple page size support for a simple microbenchmark. The use of a simple benchmark makes it possible to reason in detail about its memory behavior and its interactions with the memory system.

The benchmark allocates a heap and initializes it with data. We vary the heap size from 128K to 32M in 128K increments in order to adjust the working set of the benchmark. The benchmark performs 1000 iterations during each of which it strides through the heap in the following manner: for each 4K page, it accesses one word of data. Assuming that caches and TLBs do not contain any information, each data access brings one cache line of PTEs and one cache line of data into the data L1 cache. To ensure that consecutive accesses do not compete for cache lines in the same cache set, we increment the offset at which we access data within a page by the size of a cache line. We also access every sixteenth page to ensure that we use only one PTE per L1 cache line[14].

We performed two sets of experiments. In the first set, the heap was mapped with 4K pages. In the second set, the heap was mapped with 4M pages. Both the 4K and the 4M cases have several inflection points. The first two inflection points for the 4K case are at 4M and 6M, and the first two inflection points for the 4M case are at 8M and 10M. The first inflection point indicates that the important working set (consisting of data and PTEs) can no longer fit in the fast L1 cache. Up to this point, the benchmark achieves full L1 cache reuse (both data and PTEs fit in the L1 cache)[15]. Between the first and the second inflection points, the benchmark achieves partial cache reuse (some of the data and PTEs remain in L1 across iterations). After the second inflection point, there is no L1 cache reuse (neither data nor PTEs remain in the L1 cache across iterations). The working set, however, still fits in the larger L2 cache. The performance of the 4K case degrades sooner than that of the 4M case due to the space overhead of PTEs[16]. The 4M case does not suffer from this behavior as it uses few PTEs and, hence, significantly less space in the L1 data cache; each cache line can accommodate 16 PTEs mapping a total of 64M of contiguous address space.

By extending the portion of the graph where the benchmark achieves full L1 cache reuse (i.e., past the first inflection point to the right), one can estimate the performance of the benchmark on a system with increasingly larger L1 cache. Similarly,

[13] Note that with large physical memory support (>4GB), the large page size on Pentium 4 processors is 2M.

[14] On our Pentium 4 machine, one 64-byte cache line accommodates sixteen 4-byte PTE entries.

[15] Coincidentally, because we access one cache line of data per 4K page and access every sixteenth page, the 64-entry D-TLB begins thrashing at 4M, too.

[16] Namely, the PTEs occupy the same number of cache lines as the data. Consequently, the number of L1 misses begins to grow once the number of distinct pages we touch exceeds one half the number of cache lines in the L1 data cache.

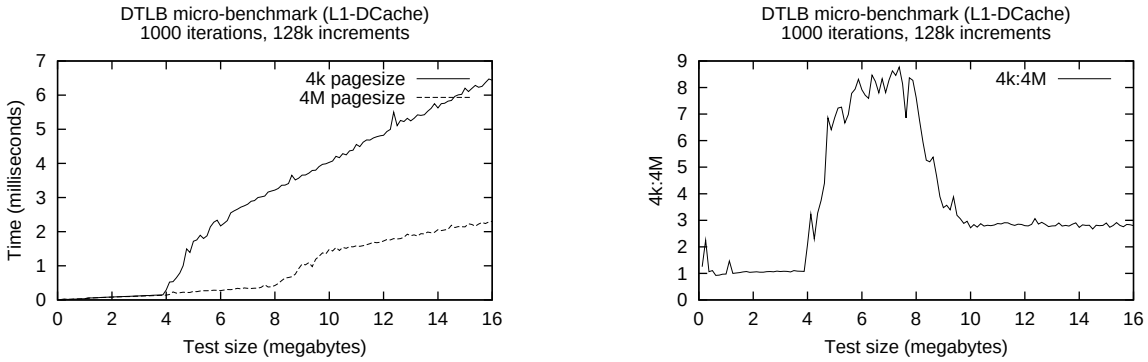

Figure 4: The execution times of the microbenchmark with small 4K pages and large 4M pages (left) and the ratios of execution times (right).

by extending the portion of the graph where the benchmark experiences no L1 cache reuse, one can estimate the performance of the benchmark on a system with a slower L1 data cache (whose access time is equal to the access time of the L2 cache of our configuration). The next inflection point (not shown on the graph) will occur when the L2 cache starts to saturate.

5.2 Assessing Performance for Traditional Workloads

This section discuss the performance of multiple page size support in the context of the SPEC CPU2000 benchmark suite[16], specifically CINT2000, the integer component of SPEC CPU2000.

The CINT2000 benchmark suite was designed to measure the performance of a CPU and its memory subsystem. There are 12 integer benchmarks in the suite. These are the *gzip* data compression utility, *vpr* circuit placement and routing utility, *gcc* compiler, *mcf* minimum cost network flow solver, *crafty* chess program, *parser* natural language processor, *eon* ray tracer, *perlbmk*[17] perl utility, *gap* computational group theory, *vortex* object oriented database, *bzip2* data compression utility, and *twolf* place and route simulation benchmarks. All applications, except for *eon*, are written in C. The *eon* benchmark is written in C++.

We noted that the applications in the CINT2000 suite use the `malloc` family of functions to allocate the majority of their memory. To provide the application with memory backed by large pages via the `malloc` function, we modified the *sbrk* function. The memory allocator uses `sbrk` to allocate memory at page granularity; it then allocates portions of this memory to the application upon request. The `sbrk` function ensures that the pages it gives to memory allocator are valid; i.e., it grows the process's heap using the `brk` system call when required.

We modified the *sbrk* function so that it returns memory backed by large pages. At the first request, `sbrk` maps a large region of memory, and uses the `madvise` system call to map that region with large pages. Whenever the memory allocator requests a memory, `sbrk` returns the next free page in this region.

If the memory request is greater than some threshold (128K), the current memory allocator will allocate pages using the `mmap` system call. To ensure that the memory allocator returned memory backed by large pages, we disabled this feature so that the allocator always uses our `sbrk`.

To allow the applications to use our modified memory allocator and `sbrk` functions, we placed these functions in a shared library and used the dynamic linker's preload functionality. We set the `LD_PRELOAD` environment variable to out library, so the dynamic linker will resolve any `malloc` function calls in the application to our implementation. In this way, no recompilation is necessary for the applications to use large pages.

Table 2 shows the performance results we obtained using large pages. Overall, the results obtained are encouraging, many applications showing approxi-

[17] Due to compilation difficulties, this benchmark was excluded from out results

Benchmark	Improvement (%)
164.gzip	12.31
175.vpr	16.72
176.gcc	9.29
181.mcf	9.43
186.crafty	15.22
197.parser	16.30
252.eon	12.07
254.gap	5.91
255.vortex	22.27
256.bzip2	14.37
300.twolf	12.47

Table 2: Performance improvements for SPEC CPU2000 integer benchmark suite using large pages

mately 15% improvement in run time.

5.3 Assessing Performance with Emerging Workloads

This section discusses the impact of large pages on the performance of Java workloads. Java applications, and SPECjvm98 [15] applications in particular, are known to have to have poor cache and page locality of data references [11, 14]. To demonstrate the advantages of large pages for Java programs, we conducted a set of experiments with the *fast* configuration of Jikes Research Virtual Machine (Jikes RVM) [1, 2] configured with the mark-and-sweep memory manager (consisting of an allocator and a garbage collector) [3, 10].

To get the baseline numbers, i.e., where the heap is mapped with 4K pages, we ran the SPECjvm98 applications with the largest available data size on an unmodified Jikes RVM. The virtual address space in Jikes RVM consists of three regions: the *bootimage region*, the *small heap* (the heap region intended for small objects), and the *large heap* (for objects whose size exceeds 2K). We modified the *bootimage runner* of Jikes RVM[18] to ensure that the *small heap* is aligned to a 4M boundary and is mapped by 4M pages.

The decision to map only the *small heap* to large pages was based on the observation that, with a few exceptions, most objects created by SPECjvm98 are small. We then repeated the experiments by mapping all three heap regions to large pages. We also varied the size of the *small heap* from 16M to 128M and computed the performance improvements with 4M pages over a configuration that uses only 4K pages.

For each application, Figure 5 shows the minimum, the average, and the maximum performance improvements when the *small heap* is mapped to large pages (left) and when all three heap regions are mapped to large pages (right). It can be seen that for several applications the performance improvements are consistent and range from 15% to 30% even if only the *small heap* is mapped to large pages. The `compress` benchmark is the only one in the suite that creates a significant number of large objects and only a few small objects, and so does not benefit from large pages in this case.

When all three heap regions are mapped to large pages, we observe an additional 5% to 10% performance improvement. For many applications, the performance improvement ranges from 20% to 40% over the base case. It can also be seen that the `compress` benchmark enjoys a significant performance boost.

5.4 Discussion

The observed benefits of large page support can vary and depend on a number of factors such as the characteristics of applications and architecture. In this section, we discuss some of these factors.

A small number of TLB entries covering large pages[19] may not be sufficient for a realistic application to take full advantage of large page support. If the working set of an application is scattered over a wide range of address space, the application is likely to experiencing thrashing of a relatively small 4M page TLB, in some cases to a much larger extent than with the 64-entry 4K page data TLB. This is a problem on processors like Pentium II and Pentium III.

Applications executing on processors with software loaded TLBs are expected to benefit from large

[18] The *bootimage runner* is a program responsible for mapping memory for Jikes RVM and the heap, loading the core of the RVM into memory, and then passing control to the RVM.

[19] In Pentium II and Pentium III microprocessors, there are eight 4M page data TLB entries some of which are used by the kernel.

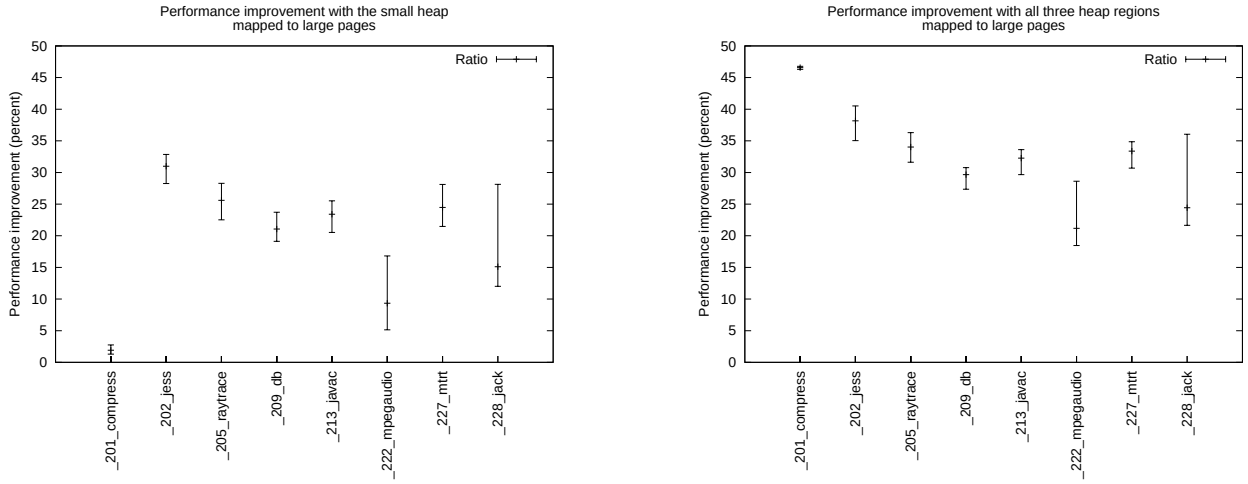

Figure 5: Summary of results for SPECjvm98.

pages. The TLB miss overhead of an application executing on a processor that handles TLB misses in hardware (such as x86 processors) can also be significant unless most page tables of an application can fit in the L2 cache and co-reside with the rest of the working set. This is highly unlikely for applications of interest: assuming that each L2 cache line is 32 bytes and each PTE is 4 bytes, one L2 cache line can cover eight 4K pages (a total of 32k). Hence, a 512K L2 cache can accommodate PTEs to cover only 512M of address space (this does not leave any space for data in the L2 cache). Consequently, for applications with relatively large working sets, it is highly likely that a significant fraction of PTEs would not be found in the L2 cache on a TLB miss. Although hardware makes reloading a TLB from a L2 cache relatively inexpensive, many TLB misses may need to be satisfied directly from memory.

The Java platform [7] presents another set of challenges. For performance reasons, state-of-the-art JVMs compile Java bytecode into executable machine code [1, 2, 9]. In some virtual machines, such as Jikes RVM [1, 2], generated machine code is placed into the same heap as application data and is managed by the same memory manager. It has been observed that the code locality of Java programs tends to be better than their data locality [14]. This suggests that application code should reside in small pages while application data should reside in large pages. In Jikes RVM, generated code and data objects are indistinguishable from the memory manager's point of view and are intermixed in the heap. Because a memory region can only be mapped to either small or large pages, a tradeoff must be made. Mapping the entire heap region to large pages may not be effective since application code may not need to use large pages. What is worse is that some processors have a very small number of 4M page instruction TLB entries[20] which can lead to thrashing of an instruction TLB. Consequently, for best performance results, a JVM should be made aware of the constraints imposed by the underlying OS and hardware, and segregate application code and data into separate well-defined regions.

For Java programs, some performance gains are expected to come from better garbage collection (GC) performance. Much work during garbage collection is spent on chasing pointers in objects to find all reachable objects in the region of the heap that is being collected [4]. Many reachable objects can be scattered throughout the heap. As a result, the locality of GCs is often worse than that of applications [11]. This behavior is representative of systems employing non-moving GCs which have to be used when some objects cannot be relocated (e.g., when not all pointers can be identified reliably by a runtime). Consequently, large pages can improve TLB miss rates during GC (and overall GC performance). Applications that perform GC frequently, have a lot of live data at GC times, or whose live data are spread around the heap can benefit from large page support and achieve short GC pauses. Short pauses are critical for software systems that

[20]There are only two 4M page instruction TLB entries in Pentium II and Pentium III processors.

are expected to have relatively predictable response times.

The availability of large pages can also be beneficial for programs that use data prefetching instructions. Modern processors squash a data prefetching request if the appropriate translation information is not available in the data TLB. Consequently, high TLB miss rates of applications can lead to many prefetching requests being squashed, thereby leading to ineffective utilization of memory bandwidth and reduced application performance[14]. The use of large pages can help reduce TLB misses and take full advantage of prefetching hardware. Further, a hardware performing automatic sequential data and code prefetching stops when a page boundary is crossed and has to be restarted at the beginning of the next page[21]. Large pages make it possible for such hardware to run for a longer period of time and to perform more useful work with fewer interruptions.

6 Related work

Ganapathy and Schimmel [6] discussed a design of general purpose operating system support for large pages. They implemented their design in the IRIX operating system for the SGI ORIGIN 2000 system that employs the MIPS R10000 processors (which handle TLB misses in software).

An important aspect of their approach is that it preserves the format of `pfdat` and PTE data structures of the IRIX OS. The `pfdat` structures represent pages of a base size and contain no page size information (just as in the original system). Large pages are simply treated as a collection of base pages. Consequently, only a few parts of the OS kernel need to be aware of large pages and need to be modified.

The PTEs contain the page size information but the page table layout is unchanged. They use one PTE for each base page of a large page and create a set of PTEs that correspond to all addresses falling withing a large page. As expected, for the large page PTEs, the page frame numbers are contiguous.

To support multiple page sizes, the TLB miss handler needs to set a page mask register in the processor on each TLB miss. To ensure that programs that do not use large pages do no incur unnecessary runtime overhead, a TLB handler is configured per process. The allocation policy is specified on a command line (on a per segment basis) before starting an application. Hence, applications do not need to be modified to take advantage of large pages, and applications that do not use large pages are not put at disadvantage.

The advantage of this design is that it allows different processes to map the same large page with different page sizes. The disadvantages are (i) this approach does not reduce the size of page tables for applications that use large pages and (ii) the information stored in PTEs that cover a large page needs to be kept consistent.

They demonstrated that applications from SPEC95 and NAS parallel suite do benefit from large pages. For these applications, they registered 80% to 99% reduction in TLB misses and 10% to 20% performance improvement. A business application like the TPC-C benchmark (which is known to have poor locality and large working set) was also shown to benefit from large pages. The authors report 70% to 90% reduction in TLB misses and 6% to 9% performance improvement for this application.

Subramanian et al. [17] describe their implementation of multiple page size support in the HP-UX operating system for the HP-9000 Series 800 system which uses the PA-8000 microprocessor.

In their design the VM data structures such as the page table entry, virtual and physical page frame descriptors are based on the smallest page size supported by the processor. A large page is defined as a set of contiguous small base size pages. Hence, this design is conceptually similar to that of Ganapathy and Schimmel [6].

The authors note that an important advantage of this approach is that it does not require changes to many parts of the OS. However, it neither reduces the sizes of data structures for applications that use large pages. In addition, locking, access, and updates of data structures for large pages are somewhat inefficient. In spite of the benefits of space efficiency and the efficiency of updates, they choose not to use variable page size based data structures because, as the authors indicate, such an approach would lead to more changes in the OS and would

[21] This is due to the fact that such automatic prefetching hardware uses physical addresses for prefetching.

have negative performance implications (e.g., a high page-fault latency in certain cases).

In their scheme, applications do not need to be recompiled to take advantage of large pages. The hints specifying large page sizes are region-based and are used at page fault time. In some cases, such as for performance reasons, the OS can ignore these page size hints and fall back to mapping small pages.

They implemented their design in the HP-UX operating system and studied the impact of large pages on several VM benchmarks, SPEC95 applications, and one commercial application. The reported performance improvements range from 15% to 55%.

7 Conclusions and Further Work

Many modern processors support pages of various sizes ranging from a few kilobytes to several megabytes. The Linux OS uses large pages internally for its kernel (to reduce the overhead of TLB misses) but does not expose large pages to applications. Growing memory latencies and large working sets of applications make it important to provide support for large pages to the user-level code as well.

In this paper, we discussed the design and implementation of multiple page size support in the Linux kernel. We validated our implementation on a simple microbenchmark. We also demonstrated that realistic applications can take advantage of large pages to achieve significant performance improvements.

This work opens up a number of interesting direction. In the future, we plan to modify kernel's memory allocator to further support large pages. We would also like to evaluate the impact of large pages on database and web workloads. These types of workloads are known to have large working sets and poor locality. Achieving high performance on commercial workloads is crucial for continuing success of Linux.

The latency of fetching a large 4M page from a disk (as a result of a page fault) can be significant. We consider implementing the "early restart" feature that would fetch and map the critical chunk of data first and complete fetching the remaining data chunks later, thereby reducing pauses experienced by applications.

Some architectures support a number of different page sizes (e.g., 16K, 256K, 4M, and 64M). We would be interested in evaluating the performance of applications on systems that have this architectural support.

Acknowledgements

We would like to thank Pratap Pattnaik and Manish Gupta for supporting this work. We also like to thank Chris Howson for all of his helpful advice.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), 2000.

[2] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño - a compiler-supported Java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 1999)*, Atlanta, GA, May 1999.

[3] C. Attansio, D. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Proc. of Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug. 2001.

[4] H.-J. Boehm. Reducing garbage collector cache misses. In *Proc. of ISMM 2000*, Oct. 2000.

[5] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Summer 1994 USENIX Conference*, pages 87–98, 1994.

[6] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proc. of the 1998 USENIX Technical Conference*, New Orleans, USA, June 1998.

[7] J. Gosling, B. Joy, and G. Steele. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, 1996.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1995.

[9] The Java Hotspot Performance Engine Architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[11] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proc. of SIGMETRICS 2000*, June 2000.

[12] C. Navarro, A. Ramirez, J.-L. Larriba-Pey, and M. Valero. Fetch engines and databases. In *Proc. of Third Workshop On Computer Architecture Evaluation Using Commercial Workloads*, Toulouse, France, 2000.

[13] V. Oleson, K. Schwan, G. Eisenhaur, B. Plale, C. Pu, and D. Aminv. Operational information systems - an example from the airline industry. In *First USENIX Workshop on Industrial Experiences with Systems Software (WIESS)*, San Diego, California, October 2000.

[14] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proc. of SIGMETRICS 2001*, June 2001.

[15] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.

[16] Standard Performance Evaluation Council. *SPEC CPU2000 Benchmarks*, 2000. `http://www.spec.org/osg/cpu2000/`.

[17] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proc. of the 1998 USENIX Technical Conference*, New Orleans, USA, June 1998.

[18] R. van Riel. Rik van Riel's Linux kernel patches. http://www.surriel.com/patches/.