Fast Address Translation for Large In-Memory Filesystems

Joshua Landgraf
University of Texas at Austin
jland@cs.utexas.edu

Huy Doan
University of Texas at Austin
huydoan@utexas.edu

Tim Stamler
University of Texas at Austin
tstamler@cs.utexas.edu

Vijay Chidambaram
University of Texas at Austin
vijay@cs.utexas.edu

Abstract

In this paper, we discuss and evaluate the benefits of using segmentation and huge pages to address RAM devices with large memory capacities. In these kinds of devices, the overhead of page faults caused by insufficient TLB coverage becomes much more severe. We seek to alleviate this overhead by using segmentation and huge pages, which will both increase TLB coverage and reduce the overhead caused by page faults. Through some simple benchmarks, we show how the negative effects of a page fault manifest on a machine with 8GB of RAM, which should already have a much lower page table overhead than the devices we are targeting with this work.

In particular, we explore how DAX, a Linux tool for providing direct access to files in memory, can be used to mount these devices and how huge pages can be enabled within DAX to improve their performance. We evaluate a version of the Linux kernel with a DAX device that uses 2MB huge pages, as well as present our work toward getting 1GB huge pages to work with DAX, something that could greatly benefit systems that require 100's of gigabytes up to terabytes of either volatile or persistent RAM.

Categories and Subject Descriptors C.4 [*Performance of Systems*]; D.4.8 [*Operating Systems*]: Performance; H.3.4 [*Information Storage and Retrieval*]: Systems and Software

Keywords Linux, operating systems, TLB, virtual memory, segmentation, paging, DAX, huge page, trace, x86

1. Introduction

With the advent of non-volatile RAM on the horizon and with large-scale web servers using more and more RAM for caching, a solution for addressing large regions of memory efficiently is becoming a pressing need. In a world where operating systems and computer hardware are designed to handle memory on the order of gigabytes, we might soon be finding ourselves in a position where we need to handle terabytes of memory. If this becomes the case, a lot of additional pressure will be placed upon page tables to map and resolve the virtual-to-physical address translations for such a massive amount of physical memory. The concept of paging has been around since the original Atlas machine[14] that MULTICS was implemented around. However, this paging was used to address mere kilobytes of memory, and it hasn't changed much since. With the growing need to find performance boosts from every aspect of the operating system and hardware, 4KB pages aren't going to cut it anymore.

As far as other methods of virtual addressing, we also briefly explored the possibilities of using segmentation to address large memory devices. Segmentation would allow for a total bypassing of all page table overhead and would make address address translation extremely fast. If segmentation was limited to just a single device, then the security concerns of stripping away virtual addresses could also be addressed in a relatively easy manner. However, the support for segmentation on modern hardware architectures is dwindling, with only two general segmentation registers (in x86_64 architecture in 64-bit mode) being made available to the operating system at all[5]. This, combined with the fact that all paging would be need to be disabled for the whole system in order to use segmentation in this way, led us to the conclusion that this would not be a viable option.

In lieu of a pure segmentation approach, the next logical step for getting around 4KB pages was simply to increase the page size. Huge pages have been around for a while[2], but work to efficiently integrate them into modern operating systems is still ongoing[1]. Because of this, despite the fact that Intel hardware has supported huge pages for quite some time, they are infrequently used by Linux. In particular, Intel supports two levels of huge pages, 2MB and 1GB, corresponding to different tiers in their page table hierarchy. However, tools such as DAX, which will be crucial for accessing non-volatile RAM devices, do not yet fully support huge pages. In this paper, we seeks to show the benefits of 2MB huge pages, which are marginally supported in DAX, and attempt to provide support for 1GB pages, which will be useful for NVRAM devices that may be terabytes in size.

We evaluate this with a series of micro-benchmarks that show the performance hits that are sustained when a page fault is triggered, as well as showing how the use of huge pages gets around the majority of these page faults. This is done by measuring pure page fault overhead and comparing it to the speed of a TLB hit, as well as performing random access benchmarks that show how higher TLB coverage statistically increases your chances of hitting the TLB instead of causing a page fault. In addition to this, we describe the process by which huge pages can be used with DAX so

that existing applications can be ported to use them, and so that new applications can be written in such a way to make best use of them.

2. Motivation and Background

Almost all modern computers and operating systems use virtual addressing to manage memory. Virtual addressing separates virtual address, which are used by programs, from physical addresses, which are used by the computer hardware. This adds a layer of indirection that various operating systems depend on for implementing security, managing memory contention, and providing useful abstractions to programmers.

There are two major forms of virtual addressing: segmentation and paging. These are not mutually exclusive and can be used together (usually with segmentation built on top of paging) or can be used on their own. In segmentation, memory is managed in variable-length segments. Segment addresses consist of a segment identifier and an offset into that segment. Segmentation allows for minimal overhead when translating addresses as a single segment can be responsible for large portions of memory, so the operating system does not have many segments to keep track of. However, it can also result in very coarse-grained control over memory as a segment must exist in its entirety or not at all. In paging, memory is managed in fixed-size chunks, called pages. Paging addresses are usually just one number, with higher bits corresponding to the page number and lower bits corresponding to the offset into the page. Paging allows for much more fine-grained control over memory at the cost of much higher address management and translation overhead. When segmentation is built on top of paging, segments are broken down into pages, allowing for both fine-grained control of memory while retaining the segmentation interface. However, this incurs additional overhead as the segment address must first be converted to a page address before the page address can be converted to a physical address.

A number of systems have been built specifically to mitigate the overhead of paging [4]. One of these is paging with "huge" pages (pages larger that the default size of 4KB). Huge pages allow for larger portions of the address space to be translated at a time. Huge pages are also generally faster to translate because fewer levels of the page table need to be traversed to find the corresponding entry. Most modern CPUs and operating systems support huge pages. However, managing huge pages alongside 4KB pages can be problematic due to memory fragmentation and bloat. Optimal use of huge pages is an ongoing area of research [2][1]. Like segmentation, huge pages also reduce the granularity at which the operating systems can manage memory.

Another approach to reducing paging overhead is the Translation Lookaside Buffer (TLB) [3]. TLBs are hardware caches for virtual addresses that allow for quick translation of frequently-used addresses. However, due to the memory used to implement TLBs, they are often very limited in size. For instance, modern CPUs are often limited to 64 L1 TLB entries, which can only translate addresses for up to 256KB of data (Table 1). Many CPUs also have L2 TLBs, which can translate more addresses, but at a significantly slower rate. However, these issues are less relevant when huge pages are in use as the TLB reach for huge pages is orders of magnitude greater than with 4KB pages. This allows for applications to have much larger working sets without having to fall back on slow page translation for memory accesses. It is also worth noting that TLBs can accelerate segment address translation as well. Since segments are variable-length, TLB reach is not an issue. The only major limitation the TLB imposes is the number of segments an application can use at the same time before falling back to normal address translation, which is generally not an issue as most applications do not use many segments.

Table 1. TLB Entries and Reach for Intel Skylake i7-6700 CPU

Page Size	L1 Entries	L1 Reach	L2 Entries	L2 Reach
4KB	64	256KB	1536	6MB
2MB	32	64MB	1536	3GB
1GB	4	4GB	16	16GB

Fast page translation and TLB reach will only continue to become more important as DRAM continues to get faster and larger. However, several new non-volatile memory technologies, including Intel's 3D XPoint, are expected to put even more pressure on TLB reach by greatly expanding the amount of physical memory in computers. When combined with Linux's new DAX feature, computers will be able to use this memory to store files, blurring the line between disk and memory. We believe it is important to explore means of accelerating address translation for these devices so that it does not severely limit performance.

3. Related Work

In order to provide fast access to large memory, [10] proposed a solution called **redundant memory mapping** (**RMM**) that took advantage of the natural contiguity in address space [10]. This approach works similarly to segmentation and maps a contiguous region in the virtual address space to the physical address space, using the page table as the fallback mechanism. This solution requires both software and hardware support. A component called **range TLB** was added in parallel with the last-level TLB to accelerate the address translation [10].

Several small research groups have made progress in getting DAX working with huge pages. Some groups at Intel have particularly stood out. Zwisler and his group developed the DAX support for huge pages at the 2MB level. Due to the introduction of the DAX radix tree locking mechanism, the huge page support from Zwisler was disabled. Zwisler successfully re-enabled the huge page support by making it compatible with the DAX radix tree locking. At the same time, another developer from Intel, Wilcox, worked on the support for huge pages at the 1GB level. Wilcox claimed some achievements on the support for 1GB huge pages. However, he reported that the XFS file system was not able to provide him with a 1GB contiguous region and suspected that error was due to the limit in his machine's memory.

4. Address Translation Methods

4.1 Segmentation

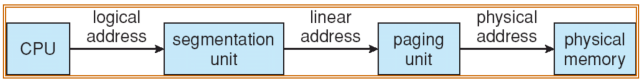

Figure 1. x86 Address Translation

The motivation for segmentation is that in x86 architecture, segmentation cannot be disabled and paging is the source of the large overhead in address translation. Paging is a costly operation because the machine needs to traverse through the paging hierarchy and that can take hundreds of CPU cycles. Segmentation was originally used to divide the memory to different regions for protections. There are regions for code, stack, data, etc. IBM once had a system that used segmentation to divide the memory into regions for different processes or purposes. Modern computers also contain a few general purpose segmentation registers, which we tried to use to support large memory in this research.

The segmentation descriptor (base and limit) are stored in the Global Descriptor Table (GDT). The GDT has 8192 entries but,

as we observed, only about 128 entries are used by our test with Ubuntu 16.04. Therefore, there should be plenty of space for our needs. The information about the GDT could be obtained by using the $sgdt$ instruction from the x86 ISA. It is also worth noting that there is also support for a Local Descriptor Table (LDT) for program-specific segments. Which table is used is determined by a bit in the segment register. For our purpose, we would use segmentation as the way IBM did. We would like to have a special segment for accessing large memory regions. With the special segment and paging turned off, we could go directly from logical address to physical address at a much faster speed and the range of the memory we could address depend on the base and limit of the segment.

Segmentation in the x86_64 architecture is almost disabled by operating in flat mode, setting the bases and limits of most segments to 0. However, the x86_64 architecture has some extra "general-purpose" data segment registers such as FS and GS that are used by the operating system and threading libraries and the limit of those segments are 2^{64}. We wanted to use one of such registers to get the right segment in the GDT for our need. In addition, modifying the content of the GDT needs kernel privilege. To minimize modification to the kernel, we decided to develop a kernel module to gain the privilege and communicate with our program through system calls.

However, this approach has a dead spot. The x86_64 architecture enforces paging in 64-bit mode. It still allows turning off paging in 32-bit mode but we did not want to operate our system in 32-bit mode. Therefore we abandoned this approach and switch to huge page with DAX support. Different architectures may have different designs and policies around segmentation and paging. Therefore, the segmentation approach may still be considered in those architectures which are not in the scope of our project.

4.2 2MB Pages

Linux currently supports a variety of ways to allocate and access data via huge pages, including `mmap` with the `MAP_HUGETLB` flag, the Posix shared memory (shm) system, and the libhugetlbfs interface. However, all of these systems target data that is allocated and stored in RAM, which had already been the subject of a significant amount of research [1][2] and will not be compatible with new non-volatile DIMM (NVDIMM) memory technologies, like Intel's 3D XPoint. We believe that these new NVDIMM technologies pose a new and significant research problem as they are managed separately from traditional system memory, can hold filesystems persistently, and can be an order of magnitude larger than the memory in most computers.

However, the Linux I/O interface poses significant issues for accessing these files. Traditional Linux I/O calls invoke the Linux I/O stack, which imposes significant overhead on these relatively fast devices. Alternatively, `mmap` allows for files to be mapped into virtual memory, which is redundant given that the files are already available on in-memory devices. For this reason, kernel developers created a new feature for accessing them called DAX. DAX-enabled filesystems work with the Linux virtual memory system to map the virtual addresses returned by `mmap` directly to the physical addresses of the file on the NVDIMM device. This avoid I/O stack overhead as well as the double copy problem from standard `mmap`.

Unfortunately, huge page support for DAX has been unreliable. Kernel developers in the Linux NVDIMM group[1] had previously developed support for DAX huge pages only for it to be wiped out by updates to the Linux virtual memory system. Since then, they have just managed to get support for 2MB huge pages working again. We chose to focus on Ross Zwisler's particular kernel modifications as this was one of the easier kernels to get working. We then experimented with it to see what was necessary to get huge pages working with user applications.

Most of the requirements of DAX with huge pages come from how Linux's virtual memory system works. In order to map a region of physical memory to virtual memory, two things are required: the starting addresses (both virtual and physical) must be aligned to the desired page size and both regions of memory must be at least as big as the page size. While these requirements may seem quite trivial, one must remember that Linux has only recently started to take up significant support for huge pages. As of right now, the system rarely guarantees alignment and continuity beyond 4KB. Instead, the task of meeting these requirements is almost completely left to the user. For this reason, we have compiled the follows series of steps users can take to configure their system and applications for meeting the requirements of DAX huge pages:

1. Before anything, the user will need a kernel capable of using DAX with huge pages. The latest kernel that we tested was Zwisler's dax_pmd_v9 branch[2], which is based on Linux 4.9-rc3. When building the kernel, the following options must be enabled[8]:
   - All options under Device Drivers → NVDIMM (Non-Volatile Memory Device) Support
   - Processor type and features → Support non-standard NVDIMMs and ADR protected memory
   - Processor type and features → Device memory (pmem, etc...) hotplug support
   - File systems → Direct Access (DAX) support
2. (Optional) If the user's system does not currently contain a DAX-capable device, one can be emulated for testing purposes. This can be done with the `memmap` kernel argument[8], which reserves portions of normal memory and treats it like a NVDIMM device. In our experience, specifying where to place the region in physical memory can take some guesswork, though higher addresses should generally be better than lower ones. Even with the right placement, the region may not be fully reserved and the system may need to be rebooted once or twice for it to work properly. If a user wanted to create a 1GB NVDIMM testing device on a system with 8GB of RAM, the following kernel argument is a good place to start: `memmap=1G!7G`. From here on, we will assume the user has an NVDIMM device available within Linux as `/dev/pmem0`.
3. Currently only two filesystems should work with DAX and huge pages: XFS and EXT4. We focused on working with XFS as EXT4 does not support 1GB huge pages. When creating an XFS file system on the NVDIMM device, the user is responsible for enabling support for 2MB extents, which will provide the alignment and continuity needed for 2MB pages. This can be done by overriding XFSs built-in RAID settings, which allow the user to specify a stripe unit and width. By setting the stripe unit to 2MB and the width to 1, XFS will now support 2MB extents. A NVDIMM device could be formatted with XFS as follows:
   ```
   mkfs.xfs -f -d su=2m,sw=1 /dev/pmem0
   ```
4. Once the filesystem is created, DAX will have to be manually enabled since this is not done by default. This is done at mount time as demonstrated by the following command:
   ```
   mount -o dax /dev/pmem0 /mnt/pmem0
   ```

[1] `https://nvdimm.wiki.kernel.org`

[2] Zwisler's dax_pmd_v9 is available at `https://git.kernel.org/cgit/linux/kernel/git/zwisler/linux.git/?h=dax_pmd_v9`

5. In order to fully enable 2MB extents in XFS, the default extent size of the filesystem must be changed. This can be done by setting the extent size of the folder at which the filesystem is mounted, causing all future files to inherit the new extent size. This can be done with the following command:
`xfs_io -c "extsize 2m" /mnt/pmem0`
6. The last problem that needs to be addressed is that `mmap` does not align addresses to 2MB by default. The Linux NVDIMM kernel devs were aware of this issue and had developed a patch to resolve this problem from the kernel size. However, this was before the changes to the virtual memory system and it seems this feature has not yet made it into Zwisler's kernels. While this issue is addressed, we found that we could force `mmap` to use a custom, 2MB aligned addresses with the `MAP_FIXED` flag. There are two ways this can be done. Zwisler has suggested the use of the virtual address `0x10200000`, which is generally free and 2MB aligned. However, applications performing multiple calls to `mmap` using this address must be careful as a future call to `mmap` could result in the previous map being replaced with a new once. Alternatively, we also found that pointers generated by `posix_memalign` could also be used as valid address to textttmmap when 2MB aligned, but this wastes system memory and is generally not recommended. An example of this fix for a application written in C could look like this:
```
file_ptr = (char*)mmap((void*)0x10200000, file_size,
PROT_READ|PROT_WRITE, MAP_SHARED|MAP_FIXED, fd, 0);
```

The above steps should be sufficient for getting Zwisler's kernel working with 2MB huge pages from a clean Linux install. We would like to call special attention to the last step, which was particularly important when converting `mmap`-based benchmarks to use DAX huge pages. To make sure an application is using huge pages, a user can invoke the Linux Ftrace debugging interface and make sure that page faults are not falling back to the default page fault handler. Ftrace is covered in more detail the next section. We believe that following these steps will allow others to reproduce our results with getting huge DAX pages working with their applications.

4.3 1GB Pages

With the current working source code that supports 2MB huge pages from Zwisler, we tried to add support for 1GB huge pages on top. The modification was expected to be trivial since support for 1GB huge pages should have been similar to that for 2MB huge pages. We started with the most recent stock Linux kernel at the moment (version 4.8.4) because Zwisler had taken a similar approach. Adding 1GB huge page support posed many challenges. First, the DAX code only supported only 4KB and 2MB pages and used a lot of Boolean logic to toggle between the two. The logic ranged from constant definitions to functional logic. We had to modify the logic to support to the 1GB huge pages. While working on this process, we found out that, while the code was there, 2MB huge page support for DAX was turned off in the stock Linux kernel. Therefore, we went back to Zwisler's branch to get the latest working kernel code that served our purpose.

Tracing the calls : one of the major challenges we faced was to find out which pieces of code are relevant and how they got called. Tracing function calls was the approach that we took in order to find out which functions we needed to touch to add support for 1GB huge pages. We used Ftrace to generate the graph of function calls. In order to use Ftrace, the kernel must be compiled with the tracing functionality turned on. There are two ways to do this.

- Using menuconfig: navigate to Kernel Hacking → Tracers and turn on the options "Kernel Function Tracer", "Kernel Function Graph Tracer", "Trace syscalls", and "enable/disable function tracing dynamically."
- Setting flags:
  - CONFIG_FUNCTION_TRACER
  - CONFIG_FUNCTION_GRAPH_TRACER
  - CONFIG_STACK_TRACER
  - CONFIG_DYNAMIC_FTRACE

After building the kernel and reboot, all of the tracing information as well as its utilities can be found at /sys/kernel/debug/tracing. The function graph tracer is the most useful utility for this purpose. To turn it on, run the following command while in the tracing folder:
`[tracing]# echo function_graph > current_tracer`
The setting to get the trace is simple, we only needed to turn on and off tracing right before and after running our benchmark program:
`echo 1 > /sys/kernel/debug/tracing/tracing_on; ./bench;`
`echo 0 > /sys/kernel/debug/tracing/tracing_on`
The trace can be found at `/sys/kernel/debug/tracing/trace` and includes many things happening inside of the kernel, including scheduling and syscalls. Fig 2 shows a small, relevant portion of the trace. We were only interested in the kernel activities during and after the function dax_pmd_fault which occurs when there is a 2MB DAX page fault. The work on supporting 1GB pages is still in progress. Due to the complexity of the current code and time restrictions, we were not able to make it work at the time of writing this paper. The focus was at the file `fs/dax.c` and spans through `include/linux/mm.h` and other paging header files in the x86 architecture section.

```
CPU  DURATION                  FUNCTION CALLS
|     |    |                    |   |    |    |
2)               |        __dax_pmd_fault() {
1)   0.035 us    |                  __d_lookup_rcu();
2)               |          down_write() {
2)   0.027 us    |            _cond_resched();
1)   0.031 us    |                  __lookup_mnt();
2)   0.277 us    |          }
1)   0.611 us    |        }
```

Figure 2. A portion of the trace

5. Benchmarks

Most of the evaluation done on DAX and huge pages was in the form of microbenchmarks, mainly due to complications with the macrobenchmarks. However, these benchmarks are still significant for demonstrating the importance of TLB coverage and showing that DAX huge page overhead is minimal.

These tests were performed on a Intel Core i7-3770T processor running at 2.5GHz with 8GB of RAM. This is the third-generation Ivy Bridge processor, which has a TLB with 64 L1 entries and 512 L2 entries for 4KB pages. It has only a single level of TLB for huge pages, which has 32 entries for 2MB pages, the ones we evaluated.

5.1 Microbenchmarks

We create a few microbenchmarks from scratch in order to measure the exact values that we were looking for under the conditions we needed. These benchmarks consisted of a sequential memory access pattern across pages that need to be retrieved from the page table, a sequential access pattern on pages in the TLB, and a random access pattern that would use both. The first sequential benchmark begins at the beginning of whatever region of memory was being tested and accesses the first byte of each page. Since this would be the first access to that page in the program, a page fault would

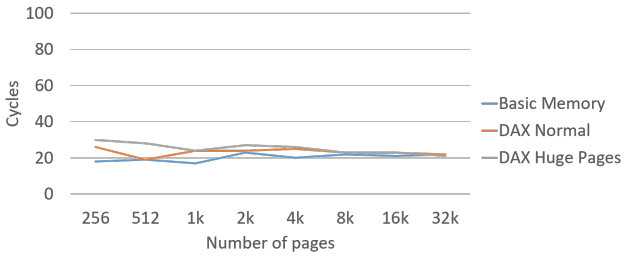

Figure 3. TLB Access Speeds

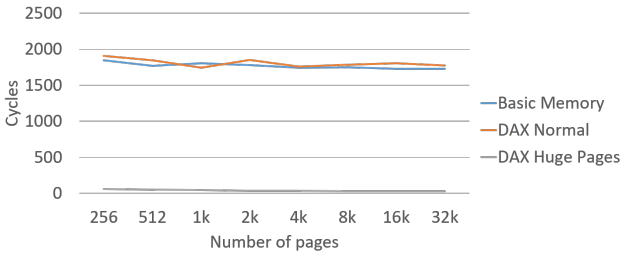

Figure 4. Page Table Access Speeds

Table 2. FIO Results

Page Size	Avg Throughput	Total Run Time
4KB	761.8 MB/s	84ms
2MB	4469.2 MB/s	14ms

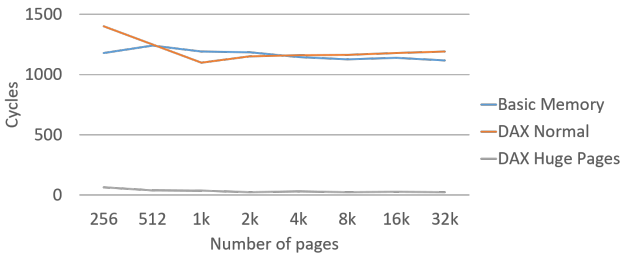

Figure 5. Random Access Speeds

be triggered and the virtual to physical mapping would be stored in the TLB. This access time would be measured for each page fault and averaged across a number of trials. The second sequential benchmark would access a page twice, once to get the page into the TLB, and then a second time on a different cache line within that page. The hope here is that the second access would be able to use the TLB entry, but none of the other levels of the CPU cache. The random benchmark did exactly what it sounds like; it randomly accesses different memory locations in the region. If the page has already been accessed, it will have the benefit of being in the TLB. If not, then the page fault cost will be measured. This was done over a number of trials to get a reasonable performance measurement. In all these cases, the size of the memory region over which these were tested was varied from 256 pages (1MB) up to 32000 pages (128MB). This was done to ensure consistency in the measurements in the first two benchmarks and to show the benefits of increased TLB coverage in the random benchmark.

These benchmarks were run with a few different cases with different memory regions. The first set of tests was run just with memory allocated from the kernel. This should be a simple test to just see the speed of DRAM allocated from the system normally. In this way, we hoped to isolate any additional overhead that DAX might cause by itself, regardless of page size. Then, the same tests were performed on a DAX device mapped into the address space of the process. These tests ran without any significant modifications or DAX configurations, using normal 4KB pages to compare against the previous test. Finally, the benchmarks were run on a DAX file configured to use 2MB huge pages as described in 4.2. This is where we hope to see the benefits of using the huge pages.

In addition to the microbenchmarks we created, we also used the flexible I/O tester (FIO) within Linux to test the performance of the DAX devices with and without 2MB huge pages. In order to do this, we created some simple FIO benchmarks that tested 128MB DAX files using both reads and writes.

5.2 Evaluation

Figure 3 shows the results of the microbenchmark that measured just the TLB overhead of accessing a page. As should be expected, the cost of this is very low, and does not depend at all on the size of the page being accessed. The lookup speed should be the same. While this does not seem like much of result, these numbers are important to compare against when looking at the results of the next benchmarks, as they show the lower bound of the performance that we can achieve and should strive for.

Figure 4 shows the results of getting the page table overhead using the different configurations and the difference it can make.

It should be noted that the huge page data was also generated by accessing the memory at 4KB increments, so it contains many more TLB hits. By doing this, we hope to show the optimal case where huge pages can provide the largest possible performance gains compared to 4KB page faults.

The results of the final benchmark, the random benchmark, can be seen in Figure 5. The benefits of the huge pages can still be clearly seen, as the TLB coverage is much greater and the number of page faults is much lower for the 2MB pages compared to 4KB pages. In the case of these benchmarks, the 2MB TLB covers almost the entire memory region in most tests, so each huge page would only need to added to the TLB once before providing TLB hits for the rest of the accesses in that range.

One other result worth noting is the very low, almost non-existent overhead of DAX. It barely showed up in the measurements we did, which is good news for any other devices we might want to mount and access with DAX.

Finally, the results of the FIO benchmark can be seen in Table 2. Here, the benefits of 2MB pages over 4KB pages can still be clearly seen, even when the entire region is larger than the 2MB page TLB reach. It is likely that using 4KB pages causes the benchmark to trigger many more page faults, whereas once the 2MB page TLB entries are populated, they are rarely evicted from the TLB.

5.3 Macrobenchmarks

In addition to the created microbenchmarks previously discussed, we tried to explore potential real world applications that could benefit from the increased performance of huge pages. One type of application that we targeted were databases that mapped database files into memory, such as Lightning Memory-Mapped Database (LMDB)[15] and LevelDB[16]. The database files could theoretically be accessed with DAX and, done properly, use huge pages instead of standard page sizes using the modifications we proposed.

However, we ran into problems with actually getting huge pages to work in these applications. The main problem being that both of the databases previously described did not try to map the entire database into memory at once. They both work by mapping in chunks of the database at a time (64KB chunks in the case of

LMDB, and dynamic sizes in the case of LevelDB). These chunks are tracked by system data structures and then can be potentially be unmapped to free space. Because of this, enabling huge pages was very difficult to integrate into these applications, especially LevelDB, which also used multiple temporary mapped files for performance benefits as well.

The reasoning for not mapping in the entire database at once can be potentially be attributed to the high cost of syncing the mapped memory with the backing file, as well as the potential for large amounts of wasted memory. There is also the problem of unmapping and remapping the entire database any time the size of it changes. In order to work around these problems, but still get the performance benefits of huge pages, the applications could be rewritten extensively to map in chunks of memory of a larger magnitude (perhaps 16MB chunks instead of 64KB chunks) so that huge pages would be most beneficial. Since this would require extensive changes to the already large code bases of these applications, it might be easier to create an entirely new memory mapped database designed to take full advantage of huge pages.

6. Conclusion

In this paper, we explored two major schemes for speeding up address translation for in-memory files: segmentation and DAX with huge pages. We found that segmentation on modern x86_64 CPUs cannot be done directly to physical addresses, eliminating it as a viable option. However, we were able to get 2MB huge pages working with DAX and showed that they can reduce paging overhead by orders of magnitude when compared to DAX with 4KB pages. Finally, we found that getting DAX to work with 1GB huge pages will be a significant amount of work. Although, given the performance improvements brought by 2MB pages, we think the effort will be very worthwhile when working with sufficiently large files.

6.1 Future Work

While we have found that DAX with 2MB pages can significantly speed up address translation for files on NVDIMM devices, there is still a lot of work needed to prepare it for widespread use. Automatic virtual address alignment is probably the most important feature to work on right now, but 1GB page support is also be very important for larger files.

We also believe segmentation may be a viable option on alternative architectures. However, a hybrid virtual memory system that allows for variable-size pages may be a better option going forward. This way programs could continue to use a linear address space while in-memory files could be mapped in with one variable-size page.

Acknowledgments

We would like to thank Ross Zwisler for making his code publicly available and helping us getting it to work.

References

[1] Kwon, Youngjin, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. "Coordinated and Efficient Huge Page Management with Ingens." In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 705-721. USENIX Association.

[2] Navarro, Juan, Sitararn Iyer, Peter Druschel, and Alan Cox. "Practical, transparent operating system support for superpages." *ACM SIGOPS Operating Systems Review* 36, no. SI (2002): 89-104.

[3] Mittal, Sparsh. "A Survey of Techniques for Architecting TLBs." *Concurrency and Computation: Practice and Experience* (2016): 1-35.

[4] Barr, Thomas W., Alan L. Cox, and Scott Rixner. "Translation caching: skip, don't walk (the page table)." In *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48-59. ACM, 2010.

[5] Intel Corporation, "Intel 64 and IA-32 Architectures Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D," Intel Corporation, 2016.

[6] AMD, "AMD64 Architecture Programmer's Manual Volume 1,2 , and 3," AMD, 2013.

[7] "Linux Cross Reference," 2016. [Online]. Available: http://lxr.free-electrons.com/source/Documentation/filesystems/dax.txt. [Accessed 10 December 2016].

[8] M. Maciejewski, "pmem.io: How to emulate Persistent Memory," 22 February 2016. [Online]. Available: http://pmem.io/2016/02/22/pm-emulation.html. [Accessed 10 December 2016].

[9] J. Yang, "W4118 Operating Systems I," 2011. [Online]. Available: https://www.cs.columbia.edu/ junfeng/11sp-w4118/lectures/l05-mem.pdf. [Accessed 10 December 2016].

[10] V. Karakostas et al., "Redundant Memory Mappings for fast access to large memories," 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 66-78.

[11] M. Wilcox, [PATCH v5 00/14] Support for transparent PUD pages for DAX files, list.01.org, 10-Mar-2016.

[12] R. Zwisler, [PATCH v9 00/16] re-enable DAX PMD support list.01.org, 1-Nov-2016.

[13] S. Rostedt, "Debugging the kernel using Ftrace - part 1," 9 December 2009. [Online]. Available: https://lwn.net/Articles/365835/. [Accessed 10 December 2016].

[14] I4 ILBURN, W., EDWARDS, D., LANIGAN, M., AND SUMNER, F., "One level storage system." IEEE Trans. EC-11,2 (April 1962), 223-235.

[15] Symas, "Lightning Memory-mapped Database." [Online]. Available: https://symas.com/products/lightning-memory-mapped-database/. [Accessed 8 December 2016].

[16] LevelDB, "LevelDB." [Online]. Available: http://leveldb.org/. [Accessed 8 December 2016].