An Approach for Managing Large, Heterogeneous Address Spaces

Simon Gerber (ETH Zurich & HP Labs)*, Dejan Milojicic (HP Labs), Timothy Roscoe (ETH Zurich)
{simon.gerber,troscoe}@inf.ethz.ch, dejan.milojicic@hp.com

September 23, 2014

Abstract

In this work, we are addressing how to support very large volatile and non-volatile main memories and how to manage them in OS-exported address spaces. We believe that investigating memory management is an important problem based on current hardware and software trends. The fact that RAM is getting large and non-volatile memory is about to get very large changes the premises under which an OS has to manage memory. The need for improving memory management in OSes is increased even more as the memory management units (MMUs) of current processors have not kept up with the increase in memory size, as their translation-lookaside buffers (TLBs) are not getting bigger, thus causing more page table walks due to conflict misses in the TLB. Additionally page tables for translating these amounts of addresses start taking up non-negligible amounts of physical memory. On the software side, we believe that future applications will be very dynamic in terms of their memory needs with diverse usage patterns, warranting the use of pages of different sizes as well as mixed usage of persistent and volatile memory, and different requirements on the locality of their working sets.

Our initial experiments show that although it is hard to get good benefits from large pages [3], they can make a difference. As mainstream operating systems do not expose managing pages of different sizes well, it is hard to get those benefits. The core question we are addressing is what happens when the application (or runtime system) gets the chance to manage explicitly (but safely) both its own physical memory and its own page tables. Then, if this approach proves to be beneficial, we will investigate how best to package this functionality in the OS and system-level libraries. We believe that the key benefits of this work include the ability to deal with memory heterogeneity, improve the flexibility of OS-level memory management, and make managing emerging memory systems easier by enabling application-based memory management through a capability-based system.

Looking at the support for large pages in Linux we see two methods which applications can use to allocate large pages: transparent hugepages and hugepagetlbfs [4]. However both of these interfaces have drawbacks. Most importantly, large page allocation in Linux is always backed by a system-wide pool of large pages of uniform size. Thus Linux cannot provide applications with different types of large pages at the same time. Our approach eliminates the need for a large page pool by building on self-paging [2], where applications are in charge of managing their own page tables and mappings, and combine self-paging applications with a decentralized resource management system based on capabilities. The capability system allows applications to request arbitrary regions of physical memory which are mappable as large pages.

We base our work on Barrelfish [1], so we can reuse Barrelfish's existing memory management system which is built around managing resources using capabilities and self-paging. Additionally, the core building block for Barrelfish's self-paging is the capability system as well. Barrelfish's capability system safely exposes the MMU hardware to applications by having specialized capabilities for managing page tables.

To make using different page sizes accessible to application programmers, we extend the interface provided by the standard Barrelfish library OS – which is modelled after BSD virtual regions and memory objects – to accept extra flags which indicate the page size that the new mapping should have. This interface allows user-space code to select a page size for each mapping independently of all other mappings.

In our initial experiments we focus on avoiding NUMA interactions and show up to 4x improvements, while in the worst case we do not see any change in performance. Our next step is to investigate the effects of different NUMA-aware allocation policies and to extend the model to large-scale non-volatile memory regions. Additionally we are going to extend our work to investigate ARM-based systems, as well as investigating opportunities created by sophisticated profiling of application memory accesses.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[2] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide.* Number 325384-042US. March 2012.

[4] Kernel Developers. HugeTLB Page Support. `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`. Accessed: 2014-08-26.

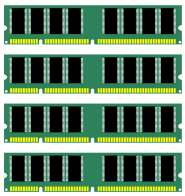
*student

Large, heterogeneous address spaces

Simon Gerber[1,2], Dejan Milojicic[2], Timothy Roscoe[1]
[1]Systems Group, ETH Zürich; [2]HP Labs, Palo Alto

www.barrelfish.org

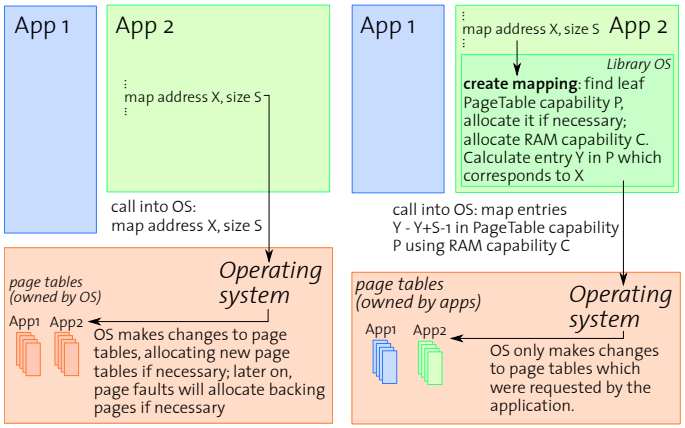
Problem statement



- DRAM sizes growing while TLB sizes remain fixed: large pages can help
- Linux's large page management inflexible
- Non-trivial to get performance benefits from large pages
- Our key idea: put applications in charge of their address space
- Early results show our approach to work well
- Next steps: NUMA-awareness, extend to NVM, improve decisions by leveraging on-line statistics

Key idea

Give applications control over their physical memory and their page tables using capabilities, self-paging and a library OS



Classical memory management system

- OS owns page tables
- Applications cannot access MMU

Self-paging

- Applications own their page tables
- OS provides safe access to MMU

- Self-paging implemented on top of distributed capability system: enable applications to manage their own AS
- Library OS provides familiar interfaces on top of unusual mechanisms
- Experimental implementation in Barrelfish

Conclusion

- Initial experiments promising: up to 4x performance increase
- Self-paging gives applications freedom in choosing page sizes that fit their workloads
- Library OS enables system administrators to configure system characteristics on an application by application basis

Results: Running times

All results were obtained on a HP DL580 gen8 with 1.5TiB of RAM running x86_64 Barrelfish with large page support.