Memory Management for Heterogeneous Multicores

Simon Gerber
Systems Group, ETH Zürich
simon.gerber@inf.ethz.ch

April 4, 2013

1 Introduction

While memory management in single-core systems is a very-well researched [6, 10, 11] – one might go so far as to say solved – problem, nobody has really come up with a conclusive solution for scalable memory management on a system with more than a handful of cores, especially if those cores have different views of the physical address space, actual disjunct physical address spaces, different page table formats and MMU architectures, or different NUMA regions (we call such systems *heterogeneous*).

Existing operating systems are ill-equipped to handle these kinds of heterogeneous multicore systems. Even though there have been attempts to make the memory management system of Linux scale better on large multicore systems, e.g. Corey [2] and the Tornado and K42 operating systems have shown that memory management (among other OS services) scales better on larger multicore systems when implemented using distributed objects [4, 12] there is currently no readily available solution for heterogeneous systems.

We believe that by employing the multikernel design principles shown in *Barrelfish* [1], we can create a flexible, scalable, and efficient memory management system for both homogeneous and heterogeneous multicore systems.

2 Background

There has been some work on making existing memory management systems scale on large multicore systems. One approach by Boyd and Wickizer [2] shows how new interfaces can help with making Linux' memory management scale better by being able to specify which memory regions are shared where explicitly.

Another approach is shown by Gamsa et al. in the Tornado operating system [4] and later by Da Silva et al. in K42 [12]. By creating distributed objects for data structures that are bottlenecks on a multicore system it is possible to alleviate some of the unnecessary serialization that is present in a classical centralized system.

However, both those approaches do not solve the more general problem of managing memory in a heterogeneous multicore system or a system with non-coherent physical physical memory, disjoint or even worse, partially disjoint physical address spaces. They also do not tackle all the issues that arise from having NUMA architectures.

Another important piece of previous work, although not directly related to memory management by default, is the *multikernel* operating system design and implementation (we are focusing on the design presented by Baumann et al. [1]) which defines the multikernel design principles of a) no sharing of memory between the operating system instances on different cores in a multicore system and b) employing distributed systems techniques to synchronize the copies of the operating system state on all cores.

Additionally, as we want to be able to make strong guarantees about the security of private memory regions and about the distribution of physical memory to applications (cf. Hand [6]), we need a resource management system that can help us enforce access restrictions in a distributed setting. One such system that has recently gotten more attention again are *capabilities* [1, 7, 9]. In particular, Barrelfish uses capabilities to manage and restrict access to any physical resources in the system.

Noteworthy prior art in the context of the current design of the memory management system in Barrelfish is Hand's *self-paging* [6]. The idea of self-paging ties into the exokernel design principle of allowing user applications to manage their resources by themselves in order to get an implementation that is tailored to their needs. Barrelfish also enables applications to self-page by providing low-level access to the memory management hardware through the capability system.

3 History/Preparation

Preliminary work – in the context of my Master's Thesis [5] – has presented one possible building block for efficient distributed memory management: making the resource manager (RM) and the memory manager (MM) aware of each other. The reason behind that coupling is that only the MM can make sure that all the references that were created using a security token (e.g. a capability in Barrelfish) are removed when that token is revoked by the RM. This is important because no access that has been acquired using a token is allowed to survive the revocation of that token.

The implementation presented in [5] keeps some state that represents the current mapping (if any) attached to each copy of a capability. While managing this additional state has some overhead, it is needed by the RM in order to instruct the MM to remove the mappings belonging to a capability that is being revoked.

The overhead can be mitigated in part by allowing user space applications to batch the installing and removing of mappings. Given an architecture that allows large/huge pages[1] and some careful design of the system call interfaces, we can guarantee that an arbitrary mapping will only ever need three system calls at most.

First experiments using a basic implementation of that idea show that keeping in-kernel state for each mapped memory region has a relatively low overhead (see Figure 1) that is

[1] i.e. an architecture that has an equivalent of IA-32 large pages or Intel 64 large and huge pages [8].

offset by the fact that the new implementation only needs a single system call for installing sequential page table entries of the same mapping within a leaf page table.

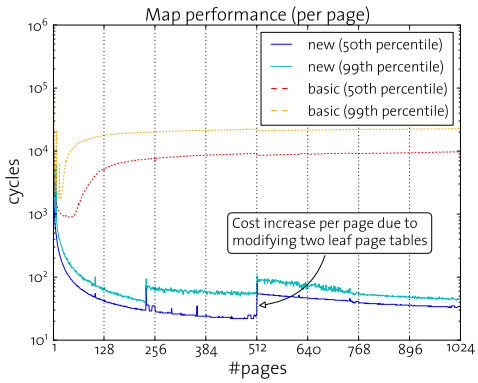


Figure 1: Cycles spent for mapping a single page

4 Goals

Seeing that modern hardware has more and more cores that are not necessarily uniform (a lot of high-end smart-phones have tightly integrated SoCs with heterogeneous ARM cores, e.g. the OMAP44xx chips have two Cortex-A9 and two Cortex-M3 cores that are user-programmable) and have complex physical address space layouts, we feel that by applying the multikernel design principles to memory management we can create a flexible memory management system that is efficient on both homogeneous and heterogeneous multicore systems.

We believe that this research is important for many applications, databases and other large systems that rely on virtual memory operations being fast and that actually need multiple cores because they need an amount of processing power that today's (and most probably tomorrow's) processors are not able to provide with a single hardware thread as well as embedded systems running on heterogeneous multicore SoCs.

The research platform we have chosen for that work is the *Barrelfish* operating system [1]. We choose Barrelfish because some of the most important basic building blocks for a decentralized memory management system like fast IPC and distributed resource management (capabilities) are already available.

Another, somewhat orthogonal, goal is to expose memory management hardware features to applications in a safe manner where possible. While this idea is by no means new [3], it still has not caught on in a lot of systems. One key feature we would like to enable by exposing hardware features is giving user programs the ability to layout data structures in a non-overlapping way in the last-level CPU cache without having to know the machine-specific cache parameters.

5 Project outline

A first step, given the work detailed in section 3, is to design a decentralized and fast memory management system for homogeneous systems. A primary design goal in that step is to making the memory management system easily adaptable to the still quickly changing multicore hardware. This step will also serve to evaluate the soundness of the design in a simpler setting.

A next step would then be to modify our design to make it handle simple heterogeneous systems (e.g. a system containing IA-32 and Intel 64[2] cores or a mix of different ARM cores). One possible architecture for such a system would consist of having an abstract page table format that can efficiently be translated to the hardware page table format by each core.

Another feature of current multiprocessor systems that needs to be considered when designing a memory management system are NUMA[3] domains. The reason NUMA domains are important is that memory accesses inside a NUMA domain are usually faster than memory accesses that cross a NUMA domain boundary. We want our memory system to be able to provide an easy interface for user applications to deal with NUMA domains.

As a bonus, we might be able to answer the question of how to handle shared libraries in the presence of a operating system that is fundamentally share-nothing. The apparent incompatibility of a shared library with a shared-nothing system needs to be addressed as shared libraries are a predominant means of reusing code. One possible approach is to selectively replicate the contents of the shared library to each NUMA region and point processes to the local copy of the shared library.

[2] We are using Intel's terminology for the x86 architecture flavours.

[3] non-uniform memory access

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. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, Dec. 1995.

[4] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association.

[5] S. Gerber. Virtual memory in a multikernel. Master's thesis, ETH Zürich, May 2012.

[6] 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.

[7] N. Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, Oct. 1985.

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

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[10] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[11] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *SIGOPS Oper. Syst. Rev.*, 21(4):31–39, Oct. 1987.

[12] D. D. Silva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann. K42: an infrastructure for operating system research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42, Apr. 2006.

Memory Management for Heterogeneous Multicores

Simon Gerber / Systems Group, ETH Zürich

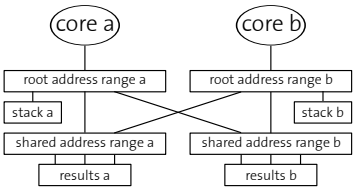
www.barrelfish.org

Problem Statement

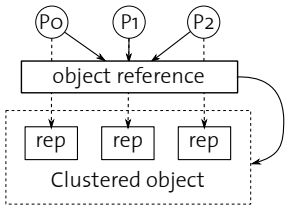
- Memory management on single core well understood.
- Existing work on making memory management systems scale to multicore (e.g. Corey, Tornado, K42).
- No accepted solution for managing memory in heterogeneous systems or systems with non-coherent, disjoint or partially disjoint physical memory.

Background

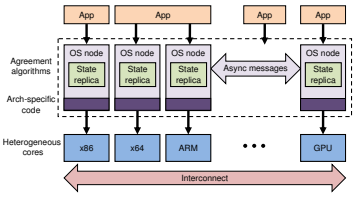
- Corey: Makes shared regions more explicit with new interfaces for Linux.
- Tornado and K42: Clustered objects on shared memory multicore systems eliminate bottlenecks on data structures
- Multikernel design principles: no sharing of memory between cores and employing distributed systems techniques to synchronize copies of the OS.
- Barrelfish as multikernel implementation: Capabilities as resource management system, self-paging for memory management.



Corey's address ranges



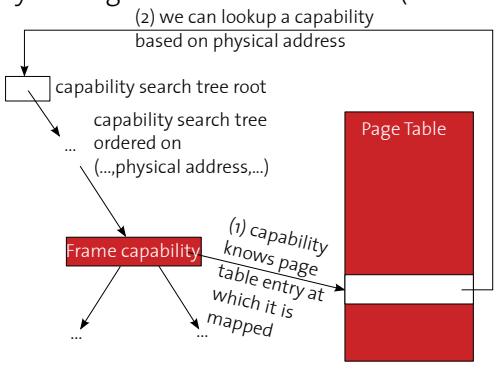
K42's clustered object design



The multikernel architecture

Preparation and History

- Master's thesis on making resource manager and memory manager aware of each other (results below).



- Context: Barrelfish research OS.

Goals

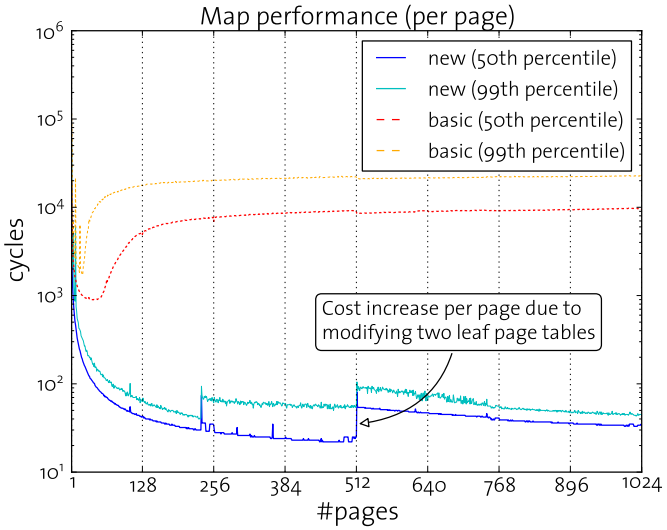
- Create flexible memory management system by applying multikernel design principles to MM.
- Expose MM hardware features securely and efficiently.
- Find generic representation for physical addresses on systems with arbitrary physical address spaces.

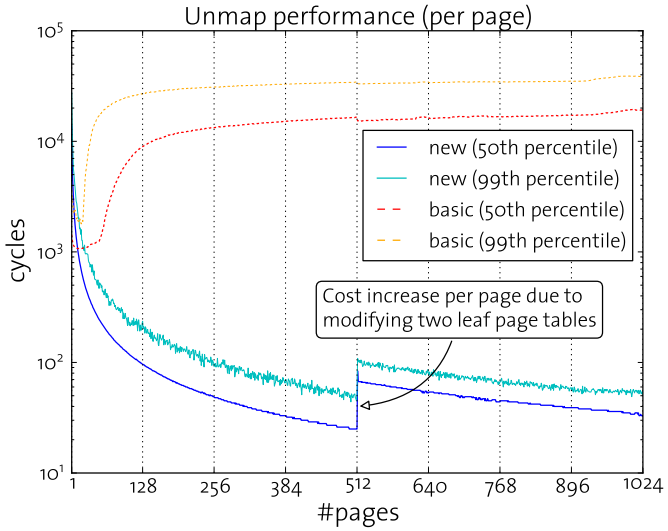
Project

1. Create decentralized and fast MM for homogeneous systems.
2. Extend system to handle simple heterogeneous systems (e.g. a system with mixed IA-32 and x64 cores).
3. Take NUMA domains into account.
4. Generalize system for situations where we have unusual physical address space(s).

Early Results

Performance improvement for new design: batch updates for sequential entries in the same leaf page table.





ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich
inf Informatik Computer Science
Systems@ETH Zürich