A Scalable Physical Memory Allocation Scheme For L4 Microkernel

Chen Tian, Daniel Waddington, Jilong Kuang

Computer Science Lab, Samsung Information System America Inc.
{chen.tian, d.waddington, jilong.kuang}@samsung.com

Abstract

L4 microkernel family has become very successful on mobile devices. However, with the rapid shift from uniprocessor to multicore and manycore processor, many critical OS functions including physical memory allocator (PMA) must be re-designed in order to achieve better system throughput. While research and engineering efforts have been made for PMA in monolithic kernels such as Linux, not much work can be found for L4 microkernels. Due to the the design difference, the PMA in L4 microkernels is part of user level page fault handler (a.k.a. pager), which is executed as a stand-alone server in the least privilege mode. Memory allocation and free requests are handled through inter-process communication (IPC) rather than normal system or kernel function calls. In this work, we first study the scalability issue of the PMA implementation in L4 microkernels, and propose our solution in the context of Fiasco.OC, a state-of-the-art L4 microkernel implementation. We also discuss how to leverage the L4 microkernel design advantages to implement a PMA with more advanced features, such as load balancing, customizability and NUMA-awareness. Finally, we conduct experiments to verify the scalability result of our solution. The experiment is conducted on a 48-core AMD magny-cours server.

Keywords microkernel, multicore processor, memory allocation

1. Introduction

L4 microkernel family has become very successful on mobile devices because it can provide virtualization and security for other mobile applications including Android and Linux. For example, L4Android[4] and L4Linux[6] projects have successfully ported Android and Linux on L4 Fiasco.OC microkernel[5]. OK Labs also announced that the OKL4[10] microkernel has been deployed on over 1.5 billion mobile devices. However, with the rapid shift from uniprocessor to multicore processor that is common in both server and mobile devices, many critical OS functions including physical memory allocator (PMA) must be re-designed in order to achieve better system throughput. While research and engineering efforts have been made for PMA in monolithic kernels such as Linux, not much work can be found for L4 microkernels.

In a monolithic kernel like Linux, PMA always exists in the most privileged level (a.k.a. kernel mode) and services all kinds of physical memory requests. When an application first touches a virtual memory location allocated by a virtual memory allocator (VMA) (e.g. malloc), a page fault (PF) exception will be raised by processor and PMA is called inside PF handler to service a page allocation request. This allocation-on-demand behavior is also known as the *memory overcommit* policy, which is common in modern OS. Besides, when OS needs to dynamically create some kernel objects it directly asks for physical memory from PMA rather than using VMA and page fault mechanism to allocate physical memory. Thus, PMA needs to service direct calls from other part of kernel as well.

In contrast, PMA in L4 microkernels is rather different from that in a monolithic kernel due to the different OS design principles. The main idea of microkernel architecture is to use minimum software, which is usually the only one executed at kernel mode, to provide other software with most basic mechanisms including hardware interrupt/exception handling, process management and inter-process communication (IPC). All other functions that may benefit from more flexible implementation or potentially break down the system, such as physical memory allocation, device drivers and file systems are provided as services executed at user mode. Consequently, a PMA is split into two parts, one still staying in kernel mode, and the other running as a server in user mode. While kernel mode PMA services all dynamic memory allocation requests made by code running in kernel, user mode PMA mainly handles the memory requests made by other applications. Similar to Linux, an application's physical memory requests are delivered to PMA through page fault handler, which is implemented as a server too. Servicing page fault related memory allocation actually reduces the need of considering fragmentation and cache false sharing issue, because such allocations are made at page granularity and typically one physical page is given out upon a page fault.

When an OS is deployed onto multicore and manycore processors, the scalability of PMA becomes critical to system throughput. Many scalable PMA designs have been studied and even implemented in monolithic kernels. For example, Linux kernel has deployed a scalable buddy allocator in version 2.6.24 [17]. The basic idea is to reduce the contention of coalescing and splitting memory regions by using per-cpu based list. Although prior research [3] reports that Linux still has scalability issue in memory management, decentralization must still be performed for improving the scalability in general. In this paper, we show that creating a PMA for each core can achieve good scalability in L4 microkernel, because it not only reduces the contention, but also effectively reduces IPC overhead that is critical to microkernel's performance. It should be noted that this work focuses on user mode PMA, which manages over 90% physical memory of the system, because the portion of physical memory reserved and managed by kernel mode PMA is very small (usually less than 8%).

The remainder of this paper is organized as follows: section 2 introduces general L4 microkernel architecture and describes the existing scheme of physical memory allocation. Section 3 presents the design and implementation of a scalable PMA under Fiasco.OC. Section 4 describes how to implement more advanced features on top of the basic implementation. Experimental results are presented in section 5 followed by related work in section 6. We conclude in section 7.

2. Background

2.1 Microkernel Architecture

Traditionally, an OS kernel is responsible for managing hardware resources and provide necessary services to applications. Those services include physical memory and virtual memory management, thread scheduling, file system management and device drivers and so on. Most famous kernels such as *Linux*, *Mac OS*, *Windows* maintain the service routines in kernel mode as shown on the left of Figure 1, and therefore called monolithic kernels. While they have been predominately deployed on different computing platforms, the increasing complexity due to diversified features and reliability issues due to incorporating more device drivers have demanded tremendous efforts in maintenance and upgrading.

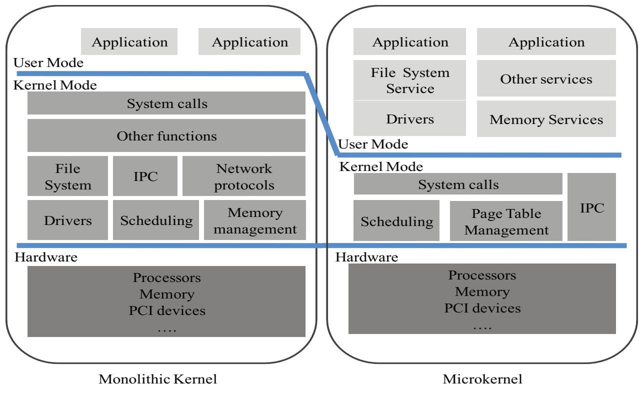

Figure 1. Design Difference between Monolithic kernel and microkernel.

As an alternative, microkernel design has much less complexity as shown on the right of Figure 1. The key difference is that most OS-provided services are moved from kernel mode to user mode. Only essential functionalities such as IPC, thread scheduling and page table maintenance are still kept in kernel mode. As a result, any service request from an application needs to be directed to the service provider, which is now a stand-alone process, through IPC calls. This design makes microkernel much simpler and more robust compared to monolithic kernel. Due to fewer functionalities, a microkernel can be implemented in less than twenty thousand of lines of source code, and thus is not difficult to maintain. The executable binary is usually as small as a few hundreds Kilobytes, which can easily fit into the L2 Cache of many types of processors.

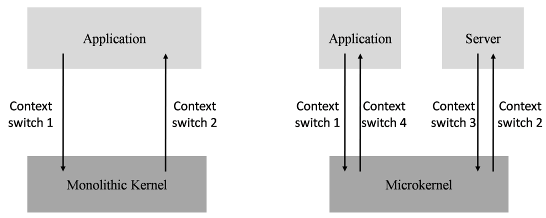

Figure 2. Handling services in Monolithic kernel and microkernel.

The study on microkernel architecture was intensively conducted two decades ago. Although many different examples, including *Mach* [1], the earlier version of monolithic kernel *Mac OS* and GNU microkernel *Hurd* [9], have been presented, the performance was very disappointing due to considerable overhead. The reason is that a service request that would normally need two context switches now require four context switches to be completed as shown in figure 2. Although IPC performance seems to be a fundamental problem to microkernel, the research results from Liedtke et al. in 1997 have revealed that IPC designs and implementations can be highly optimized [14]. As a result of a sequence of research efforts, second generation microkernels featured with high IPC performance have been designed and implemented. Most notable example is L4 microkernel family[13], including *L4/Fiasco*, *L4Ka::Pistachio*, and *NICTA::L4-embedded* and so on.

Lately, more research efforts are made to further improve microkernel's performance and enrich its features. These efforts lead to third generation microkernels that have drawn intensive attention from both academia and industry. Representative designs include *Fiasco.OC*[5], *Nova*[16], *seL4* [12] and *OKL4* [10]. These designs not only have further optimized IPC, but also offer different attractive features including real-time support, resource access control mechanisms, security and virtualization and so on. In this work, we choose *Fiasco.OC* as the kernel to verify our idea.

2.2 Physical Memory Management in Fiasco.OC

Physical memory management is one of the most basic functions an OS must provide. When an OS is booted, both OS kernel and applications running within the OS need to make physical memory allocation requests dynamically. Different from monolithic kernels (e.g., Linux) where all these requests are handled in kernel mode, microkernels such as Fiasco.OC have two PMAs, one in kernel mode using buddy allocation algorithm, the other in user mode using an AVL tree based allocation algorithm.

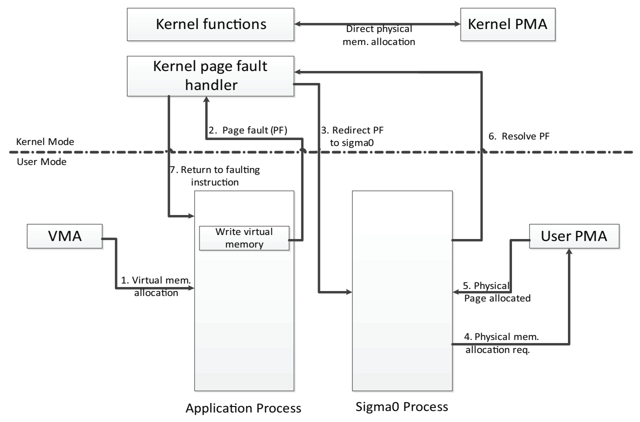
Kernel functions
Direct physical mem. allocation
Kernel PMA
Kernel page fault handler
2. Page fault (PF)
3. Redirect PF to sigma0
6. Resolve PF
Kernel Mode
User Mode
7. Return to faulting instruction
VMA
Write virtual memory
User PMA
1. Virtual mem. allocation
5. Physical Page allocated
4. Physical mem. allocation req.
Application Process
Sigma0 Process

Figure 3. Physical Memory Allocation in Fiasco.OC.

Figure 3 shows how physical memory allocation requests are handled in Fiasco.OC. First of all, kernel functions can directly allocate physical memory though kernel PMA when a kernel object is created. The amount of memory managed by kernel PMA however is only 8% of the total by default. The rest of the memory is managed by a user level PMA and used by applications. This work focuses on this PMA. From the figure, we show a very typical sequence of operations that eventually trigger a page allocation and make uses of an allocated page. A process first allocates a stack variable or heap data through a virtual memory allocator such as *malloc* (step 1). When the virtual address is touched (either read or write), a page fault exception is raised by the processor, which transfers the execution from user code to kernel PF handler (step 2). The handler further forwards the request to a special user-level application, *Sigma0* (also known as *pager*), which by default handles all page faults in L4 microkernel design including Fiasco.OC (step 3). In a multicore environment, it is possible to have multiple page faults take place on different cores simultaneously. In this case the kernel PF handler serializes them and forwards the request to Sigma0 one by one. When received a PF through an IPC call, Sigma0 requests a physical page from PMA (step 4 and 5), which is typically implemented as part of Sigma0, and send it back to kernel (step 6), which then populates the page table for the faulting process. After that, kernel PF handler switches back to the faulting instruction so the application process can continue (step 7).

While the entire process is transparent to applications, it involves four context switches (step 2, 3, 6 and 7), two of them being IPC calls (step 3 and 6). This is the cost that L4 microkernel design must pay for security and reliability.

3. A Scalable PMA Design For L4 Microkernel

3.1 Scalability Issue

Most monolithic OS kernels support multi-tasks running on multicore and manycore processors, and therefore a PMA must be able to handle concurrent requests made by different processes running on different cores. As a result, lock based synchronization is typically used to ensure the PMA (or the data structure in PMA) is exclusively used by only one process at a given time. This solution, however, may introduce severe scalability issue due to lock contention. Several solutions have been proposed in Linux to remove such *big locks*, i.e., locks that cause high contention. For example, the buddy allocator in Linux kernel 2.6.24 uses per-core lists, which are protected by per-core locks, to track free pages. In the presence of NUMA architecture, Linux kernel 2.6.30 employs another allocation scheme that assigns one allocator for each NUMA zone.

Although the scalability issue of handling physical memory requests in monolithic kernels has drawn enough attention, it has not been fully studied in L4 microkernels. To address this issue, we first measured the scalability of memory allocation and free using L4Re, the native runtime environment of Fiasco.OC [7]. As shown in Figure 4, as the number of cores that sends memory allocation and free requests increases, the time of servicing requests is significantly increased. The degradation across 48 cores is 25x for allocation and 2x for free respectively.

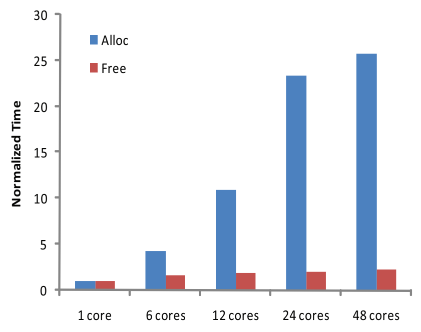

Figure 4. Scalability of Allocation and Free in L4Re.

While the degradation of physical memory allocation in Linux is caused by locks, the reasons for that in the context of L4 microkernel architectures are different. First, the contention of sending IPC to Sigma0 causes the PMA in L4 microkernel not to scale. Sigma0 by default is the only pager in the system. When multiple requests from different cores need to be handled at the same time, the kernel PF handler needs to serialize the requests and send them to Sigma0. This is analogous to the lock contention in monolithic kernel.

Second, cross-core IPC (i.e., two parties of an IPC on different cores) can have significant negative impact on the scalability of PMA. IPC mechanism in L4 microkernel like Fiasco.OC is fundamental for the entire OS architecture. For example, a physical page request requires an IPC with Sigma0 as shown in step 3 and 6 in Figure 3. Although the IPC implementation has been highly optimized, cross-core IPC is still 10x-12x slower than same-core IPC (i.e., two parties of an IPC on the same core).

Figure 5 has shown the experimental result collected using a modified ping-pong benchmark running on a 2.0GHz 48-core AMD server. Despite the message size, we can see that same-core IPC takes a few hundreds CPU cycles while a cross-core IPC takes about 12 thousands cycles. This is because cross-core IPC involves inter-processor interrupt (IPI) handling, which causes serialization in kernel code and hardware.

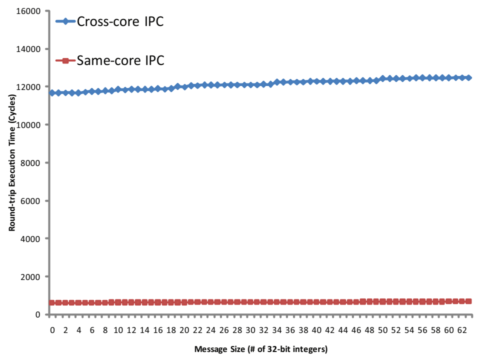

Figure 5. Performance Comparison Between Cross-core IPC And Same-core IPC.

3.2 A Scalable PMA Design

Design PMA can be deployed globally, for each zone, for each core or even for each process. Based on the reasons for the PMA scalability issue in L4 microkernel, i.e., the contention of concurrent page requests and poor cross-core IPC performance, we believe the per-core PMA is the best design for microkernel. Since in L4 microkernel architecture, a PMA is part of a pager, we essentially need to deploy a pager for each core as well. Figure 6 demonstrates the idea.

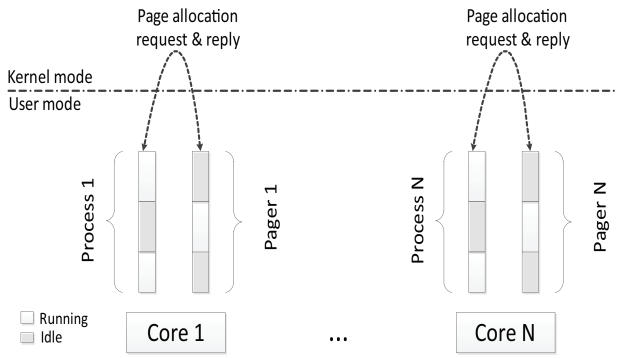

Figure 6. Creating One Pager For Each Core Improves Scalability.

The benefit of this design is it addresses both problems that lead to scalability issue. First, it reduces the contention by N times on average where N is the number of cores, because there exist N pagers in the system now. Second, it eliminates all cross-core IPCs of page allocations, as each core now has a pager and every process can use same-core IPC to communicate with the local pager. All page requests can be handled on the same core rather than a different core, which is similar to monolithic kernel.

As a result, The performance and scalability of page allocation can be largely improved. Another good side-effect of this design is that the CPU utilization can also be maximized, because no local physical memory request interrupts the application executed on a remote core.

Implementation The general idea of implementing the per-core PMA scheme in Fiasco.OC is to first partition physical memory, and then construct per-core pagers, which initializes PMAs using the right partition. Figure 7 illustrates the implementation details.

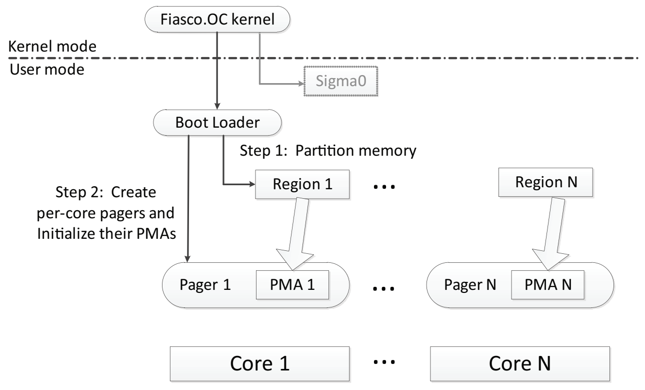

Figure 7. Implementing Per-core Pager and PMA On Fiasco.OC.

When Fiasco.OC kernel is booted, it loads another two modules, *Sigma0* and *moe*. Sigma0 is the default pager and moe is the default loader (like *init* process in Linux). To construct per-core pagers, we developed our own loader. As shown in Figure 7, the loader first obtains all physical memory that are made available by kernel and then partitions them according to a predefined policy. In our basic implementation, memory is split evenly across all cores.

After that, the loader spawns N new pager processes where N is the number of cores, each running on a different core. Each pager obtains initialization information through an IPC with the loader and then initializes itself. During the initialization, a pager creates a PFA that manages the memory region assigned to the core the pager resides. There exist many different data structures and algorithms that are suitable for physical memory management. In our basic implementation, we use an AVL tree based algorithm, which can also be found in [7, 8].

Finally, the boot loader needs to specify the pager for each application based on which core an application is loaded. In Fiasco.OC, the kernel data structure of a process (a.k.a task) contains a filed that can point to any valid pager process. A system call is also provided to set this field in Fiasco.OC. The original idea of pager customization in L4 microkernel is to implement nested pager, which is important for achieving resource isolation. We leverage this feature to ensure each process is connected with the pager that runs on the same core as the application does. A page fault on each core now can be successfully resolved by the same core pager as proposed in Figure 6. As a result, this design and implementation lead to significant improvement on the scalability of physical memory management.

TLB Issue During the implementation, we noticed there is a TLB flush issue that causes performance degradation when pager handles free request. Specifically, when a physical page in a process is released by pager, the mapping between this page and its corresponding virtual page needs to be removed from page table. Since TLB caches page table entries, it must be updated.

The implementation in Fiasco.OC is straightforward, that is, to flush the entire TLB in the core on which the application is running. Since the flush call is made in the pager process, and the pager and application now are on the same core, flushing TLB actually causes the pager process to re-populate the TLB before it yields. However, this flush can be safely removed in our design because when pager is context-switched to the application, which is running on the same core, TLB is flushed anyway. Alternatively, one can also leverage hardware features such as Address Space ID(ASID) associated TLB or instructions that support individual TLB entry invalidation.

4. More Advanced Features

While per-core PMA design in Fiasco.OC achieves better performance, many more advanced features can be added by leveraging the microkernel architecture advantages. In this section, we describes how to implement three additional features, namely *load balancing*, *customizability* and *NUMA-awareness* as shown in Figure 8, on top of the basic implementation described in previous section.

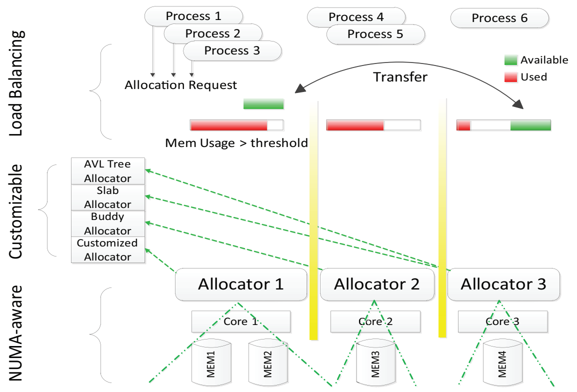

Figure 8. A Scalable PMA With More Advanced Features.

4.1 Load balancing

Having a de-centralized design is usually the key to solve scalability issues. However, highly de-centralization may lead to a load unbalance problem. Particularly, while the per-core PMA can reduce contention caused by multiple concurrent requests, system throughput may still stay poor when the amount of memory requests on each core is unbalanced. Thus, a balancing scheme needs to be implement to maintain a good load balance among all PMAs.

One simple design of the load balancing feature, as shown on the top of Figure 8 is to make each PMA track the current memory usage and use two threshold values *MEM_HIGH* and *MEM_LOW* to maintain the balance of unused memory across all PMAs in the system. The detailed implementation are as follows. After servicing an allocation request, a PMA compares its available memory size with *MEM_LOW*. If the size is too small, a *memory low* request is sent out to other PMAs. The PMAs whose memory usage is low should response to the request and transfer some memory to the requester. Similarly, after each free, a PMA recalculates the memory usage and compares the available memory size with *MEM_HIGH*. If the usage is very low and some of the memory is donated by another PMAs, a *memory return* request should be used so that the donated memory is sent back to the donors.

Since each PMA is part of a pager process, the load balancing requests among PMAs can piggyback on the IPCs among pager processes. When multiple requests are sent to the same pager, they are serialized by kernel and therefore no additional synchronization is needed. However, these requests are essentially cross-core IPCs, so threshold values must be carefully tuned to avoid excessive IPC overhead.

4.2 Customizability

A PMA can be implemented with various algorithms such as tree based algorithm, buddy algorithm and slab allocators and so on. Depending on the memory access pattern of an application, different applications may favor different algorithms in terms of performance. Thanks to the design of L4 microkernel, we can modify the basic per-core PMA implementation so that each PMA is customizable as shown in the middle of Figure 8.

To allow an application to customize its local PMA, a set of explicit pager API functions should be implemented because the pager has the control of the PMA it owns. The pager API functions should be implemented on top of IPC and control the behavior of PMA. For example, one API can specify what PMA algorithm is likely to yield better performance for the running application. Since the application and pager are placed on the same core, and these pager APIs are expected to be called at very low frequency, performance gain of this feature will outweigh the extra overhead caused by API invocation and PMA reshaping . To ensure this, the relationship between application performance and PMA algorithm should be known by developers or obtained through profiling or compiler analysis.

4.3 NUMA-awareness

As the number of on-chip cores increases, non-uniform memory access (NUMA) design becomes prevalent. Under NUMA, a core accessing its local memory is much faster than accessing remote memory. Therefore, when receiving a page allocation request, a PMA needs to allocate a local page whenever possible to achieve better performance.

To make the PMA design aware of NUMA , we only need to change the memory partition policy to recognize NUMA zones (step 1 in Figure 7). Specifically, the boot loader needs to first extract NUMA information from ACPI table and partition the memory according NUMA zones. Within a NUMA zone, the memory should be further partitioned among cores if more than one core exist. The rest steps of constructing per-core PMA remain the same and each PMA now manages its local memory region. As a result, an application can automatically take advantage of hardware benefits.

It should be noted that when load balancing scheme as discussed in earlier section is implemented, memory transfer policy needs to be NUMA aware as well. In particular, the memory transfer should take place within a NUMA zone first so architecture's negative impact on performance can be minimized.

5. Experimental Results

To show the effectiveness of the per-core PMA scheme, we conduct a set of experiments and use L4Re based implementation as a comparison where sigma0 is used by all processes.

5.1 Setup

Software We developed a memory allocation benchmark to stress the physical memory allocator. In particular, the benchmark performs 100 times of memory allocation task. Each task first allocates certain number of memory pages, then touches the first byte of each page, which invoke the physical memory allocation, and finally frees all pages. We also developed a L4Re version equivalent benchmark for comparison.

To eliminate the effect of virtual memory allocation, we write our own version of *malloc* that does not perform any optimization, but simply manages a 2GB virtual memory range. In addition, the *free* call not only reclaims the virtual memory, but also sends the free request to PMA so that physical page can be freed and no physical to virtual mapping is cached.

The Fiasco.OC kernel used in the experiment is *Revision 36. x86 32-bit* build without kernel debugger. The kernel, loader, pager and benchmark are all compiled by GNU GCC 4.4.6 with *-O2* optimizations.

Hardware The experiments are conducted on an AMD Magny-cours server equipped with four AMD Opteron 6174 2.2MHz processors (also called multi-chip module packages. Each processor combines 2 dies of 6 cores, so there are 48 cores in total. Each core has a 64KB L1 D-Cache, 64KB I-Cache and 512KB L2 cache. Each die shares a 12MB L3 cache. During the experiment, 4GB DRAM is used due to the limitation 32-bit address space.

5.2 Results

Scalability The first experiment we conducted is to compare the scalability between our per-core PMA design and L4Re native PMA. In the experiment, we create an instance (process) of the benchmark on each core. A coordinator process is also created to ensure that all processes start at the same time. Each process allocates 10,000 pages in total. The L4Re memory allocation benchmark is configured and deployed in the same way.

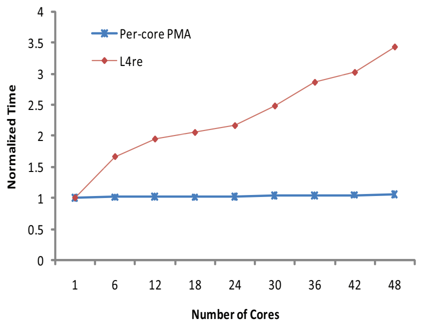

Figure 9. Scalability Comparison.

Figure 9 shows the normalized results in terms of execution time. As the number of cores increases from 1 to 48 at step 6, the normalized time of using our PMA stays flat and the degradation is only about 5% when 48 cores are used. In the case of using L4Re native PMA, however, the performance degrades over 240%.

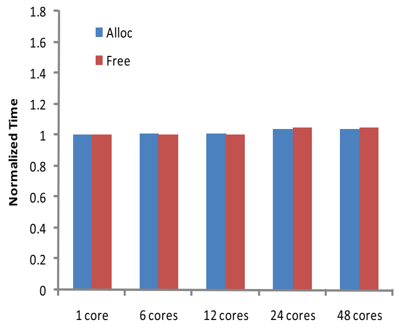

Figure 10. Scalability of Allocation and Free.

We also measured the performance of memory allocation and free separately and Figure 10 shows the result. Although a small degradation i.e., 4% for alloc and 5% for free, can be noticed when 48 cores are used, the scalability of allocation and free call is much better compared to L4Re native PMA (as shown Figure 4) where the degradation for alloc and free are over 2500% and 230% respectively.

Performance Sensitivity While our main goal in this work is to obtain good scalability, we noticed that the performance is sensitive to the number of outstanding pages (i.e., the number of pages that have been allocated before free starts). As shown in Figure 11, when the number of outstanding pages increases from 50(200KB memory) to 1000(4MB memory), we observed a 1.8x slowdown. This is because we are using AVL tree algorithm to maintain allocated and freed memory chunks in each PMA. Thus, the more outstanding pages, the more nodes in the tree, which increases the time of lookup, re-balancing, insertion and deletion.

Despite the performance degradation, we still observe good scalability of our implementation. The most degradation across 48 cores we observed is only about 8%, which happens when we keep 1000 outstanding pages before any page is freed.

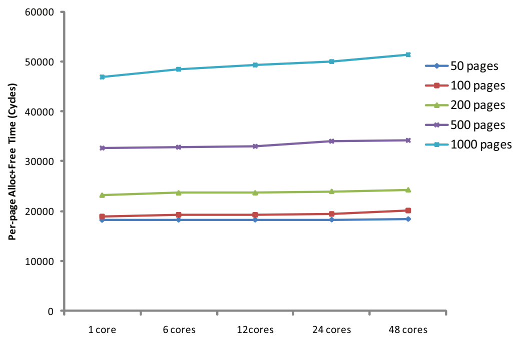

Figure 11. Sensitivity of Outstanding Pages.

6. Related Work and Future Directions

L4Re, the native runtime system of Fiasco.OC, contains many servers that provide OS services [7]. Like other L4 microkernels, the physical allocator is implemented in the default pager, sigma0. As we discussed earlier, having a single pager in the system may lead to severe scalability problem on multicore and manycore platforms. This work aims to solve this problem by enabling per-core PMA, which is complementary to memory allocator in L4Re.

Another physical memory allocation scheme for L4 microkernel is from Genode Labs [8]. Since the goal of Genode is to achieve secure resource management, the scalability is not emphasized. Although the pager in Genode system is multithreaded, all threads are running on core 0. As with older Linux memory allocators, the Genode memory allocator uses a lock to prevent concurrent accesses. This design causes the the scalability issue too.

Similar to Fiasco.OC, several other scalable microkernel designs have also been proposed. For example, Wentzlaff et al. has proposed a scalable OS design that deploys multiple servers for each OS service [17]. Singularity [11] and Barrelfish [2] developed by Microsoft Research and ETH Zurich Systems Group borrow the ideas from distrusted system and apply them to manycore system. HeliOS [15] is also from Microsoft Research that targets heterogeneous platform. Physical memory allocation scheme in these works is not emphasized. Special cares such as load balancing and customizability are not taken. The idea proposed in our work, however, is not only suitable for L4 microkernels, but can also complement the above-mentioned microkernels on physical memory allocation.

The scalability of physical memory allocation has drawn plenty of attention in monolithic kernel community. Although the memory allocator in Linux has evolved from single allocator to a per-cpu-list based buddy allocator, prior work has reported that it still has scalability issue[17]. While Linux continues to make improvement, we believe some advanced features still need to be implemented such as load balancing and customizability.

In the future, we plan to continue our work in two directions. First, we will implement those more advanced features in PMA and use real-world applications to make quantitative analysis. We expect that different applications will get benefits from different kind of features. Second, we will study the physical memory allocation in multi-threaded applications. In this work, we only consider multiple processes running in the system. However, a multithreaded process poses new challenges to the scalability issue because contention may occur on the page table. Thus, a scalability design of page table may need to be considered too.

7. Conclusion

In this work, we investigate the root cause of the scalability issue for the physical memory allocator in L4 microkernels. We presented an implementation of a scalable physical memory allocator in the context of Fiasco.OC L4 microkernel architecture. We also discussed some more advanced features that can be implemented by leveraging L4 microkernel design advantages. Finally, we conducted a set of experiments on a 48-core machine to verify the scalability results of the per-core PMA design in L4 microkernel.

References

[1] M. Accetta, R. Baron, W. Bolosky, R. Golub, Davidand Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, 1986.

[2] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[4] http://l4android.org/.

[5] http://os.inf.tu dresden.de/fiasco/.

[6] http://os.inf.tu dresden.de/L4/LinuxOnL4/.

[7] http://os.inf.tu dresden.de/L4Re/.

[8] http://www.genode.org.

[9] http://www.gnu.org/software/hurd/hurd.html.

[10] http://www.ok labs.com.

[11] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[12] G. Klein, P. Derrin, and K. Elphinstone. Experience report: sel4: formally verifying a high-performance microkernel. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 91–96, 2009.

[13] J. Liedtke. On microkernel construction. In *Proceedings of the Symposium on Operating System Principles*, pages 237–250, 1995.

[14] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 5–6 1997.

[15] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 221–234, 2009.

[16] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, 2010.

[17] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.