# Towards a Scalable Microkernel Personality for Multicore Processors

Jilong Kuang, Daniel G. Waddington, and Chen Tian

Computer Science Lab, Samsung Research America - Silicon Valley
{jilong.kuang,d.waddington,chen.tian}@samsung.com

**Abstract.** With a steady trend from singe-core to multicore processors, scalability has become a significant design issue for the Operating Systems (OS), as many critical OS functions must be re-designed in order to achieve scalable performance. While numerous efforts have been made to improve scalability of monolithic OS kernels, comparatively little work has been done for microkernels.

In this paper, we begin by studying the scalability of Fiasco.OC, a state-of-the-art microkernel implementation. We then present OmniRE, a new personality for the Fiasco.OC microkernel that is aimed at being multicore scalable. Compared to L4Re (the vanilla "off-the-shelf" Fiasco.OC personality), OmniRE aims to eliminate contention by decentralizing resource management, scheduling, and kernel access. The design also aims to minimize inter-process communication (IPC) across CPUs by localizing resource functionality such as page-fault handling. We conduct experiments to compare OmniRE against L4Re as well as Linux on a 48-core AMD server and a 6-core Intel workstation. Our results indicate that OmniRE provides better scalability than L4Re and can in fact exceed absolute performance of Linux in memory page management at higher core counts.

## 1 Introduction

Compared to monolithic kernels such as *Linux* and *Windows*, microkernel architectures have unique advantages in simplicity, security, robustness and customization. To date, research has predominantly focused on improving microkernel uniprocessor performance and enriching the feature set. These efforts have lead to third generation microkernels, such as *Fiasco.OC*[7], *NOVA*[15], and *OKL4* [6]. As microkernel technology has matured, it has begun to get traction in both mobile platforms (e.g., L4Android [2], OKL4 [6]) as well as embedded and safety-critical systems. More recently microkernels have been explored as a more effective platform for HPC-like applications [14] [17].

The move towards multicore processors has driven many OS communities to reexamine fundamental internal design decisions in order to improve multicore scalability. For example, in the Linux kernel, little use of coarse-grained locks remained by version 2.6; most notably code locking had been converted to data locking, advanced "lockless" data structures had been applied (e.g., Read-Copy-Update [12]) and schedulers had been re-implemented to support per-core scheduling queues.

Nevertheless, as of now, not much work has been done for microkernels. Although a number of research OSes, including Barrelfish [4], Helios [13], FOS [18], and Corey [20], have taken an approach based on the concept of *multi-kernels* to improve OS scalability on multicore processors, they have shied away from shared-memory kernels due to the need to partition resources. However, shared-memory kernels provide obvious advantage with respect to integration and resource sharing across cores. It is for this reason that we have chosen a shared-memory microkernel, Fiasco.OC [7,10], as the basis of our work.

In this paper, we propose OmniRE, a scalable runtime personality (user-land) for the Fiasco.OC microkernel. OmniRE is a direct replacement for the L4Re personality from the Fiasco.OC group [7]. OmniRE incorporates a hierarchical resource management design that eliminates central points of contention and provides sufficient flexibility to allow tailoring of the OS to underlying processor, memory and IO topologies. Compared to L4Re, OmniRE offers the following differentiation: 1) It eliminates contention on resource management by decentralization of memory management, scheduling, and access to kernel services. 2) It minimizes cross-core IPC on multicore architectures by forcing resource management, resource access and page-fault handling to be localized to the same core whenever possible.

The principal contributions and structure of the paper are as follows:

- We study and investigate the scalability potential of the Fiasco.OC microkernel and examine performance scaling for the L4Re personality (Sections 2 and 3).
- We present a design and implementation of OmniRE, a scalable multicore user-land based on the Fiasco.OC microkernel that uses a hierarchical arrangement of multi-threaded services and decentralized resource management to successfully achieve scalability (Section 4).
- We conduct experiments to empirically compare Fiasco.OC/OmniRE against Fiasco.OC/L4Re as well as Linux 3.0 running on a 48-core AMD server and a 6-core Intel workstation (Section 5).

## 2 Fiasco.OC and L4Re Overview

As a representative third generation microkernel, Fiasco.OC and its user-land runtime environment (L4Re), have become increasingly popular due to the availability of advanced features (e.g., capabilities, multicore support, multi-ISA support, Linux para-virtualization) and general maturity.

The basic components of an L4Re-based system are:

- *Microkernel* - provides primitives to execute programs in tasks, to enforce isolation among them, and to provide means of secure communication in order to let them cooperate. As the kernel is the most privileged, security-critical, software component in the system, it provides only a minimal set of mechanisms that are necessary to support applications.
- *L4 Runtime Environment* - L4Re comprises low-level software components that interface directly with the microkernel. The root pager *Sigma0* and the root task *Moe* are the most basic components of the runtime environment. Other services (e.g., for device enumeration) use interfaces provided by them.

– *Applications* - run on top of the system and use services provided by the runtime environment or other applications. Applications include conventional user-local programs that face the end-user, as well as virtual machine monitors, device drivers and other system services.

## 3 Study of Off-the-Shelf L4Re Scalability Characteristics

Although the Fiasco.OC kernel is explicitly designed to support multicore processors [9], there are scalability limitations in the current L4Re platform. To investigate the L4Re scalability characteristics, we have developed micro-benchmark applications to test memory management and thread creation (corresponding to kernel object creation) as key indicators of system scaling. The test platform is a 48-core AMD Magnycours server (see Section 6 for more details).

Figure 1 shows the memory management results, where each iteration performs a single page (4K) allocation (via std::malloc API), writes to each integer element in the page and then frees the memory (via std::free). The results show that memory management (allocation, physical-to-virtual mapping and paging) on the L4Re-based platform degrades significantly from only two cores. Figure 2 shows the thread creation results, where each iteration creates a new child thread which executes an empty function. The parent thread (per-core worker) waits for a child thread to complete before starting the next thread. Similarly, it is evident that L4Re does not scale well with respect to thread creation.

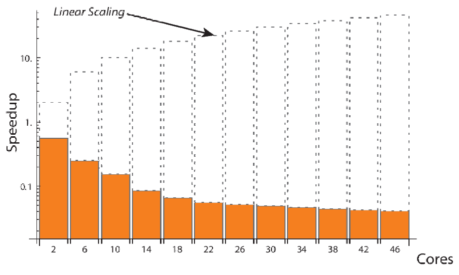


**Fig. 1.** L4Re Memory Allocation Scaling

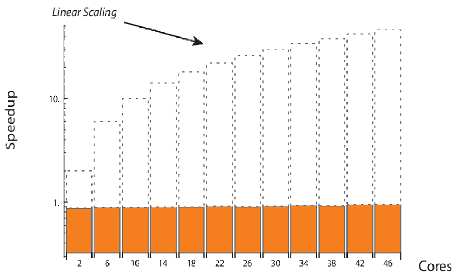


**Fig. 2.** L4Re Thread Creation Scaling

We believe that the current scalability limitations in L4Re-based system are predominantly a result of centralized resource management in the L4Re personality.

## 4 OmniRE Design

OmniRE is a new personality for the Fiasco.OC microkernel [7,10]. OmniRE directly replaces the L4Re personality [3]. Key design elements are:

– Decentralized management of memory (physical and virtual), thread/process, IO and IRQ resources.

– Minimization of cross-core IPC by localization of page-fault handling and service access.
– Explicit resource management and quota control for all resources in the system. Secure access control to resources realized through the microkernel's capability feature.

## 4.1 OmniRE Detailed Design

OmniRE is responsible for managing all of the resources in the system. This includes controlling allocation of kernel objects (e.g., threads, semaphores, IPC gates) as well as resources directly used by the application (e.g., memory, I/O ports). The fundamental basis of OmniRE's design is that resource management (e.g., allocation, freeing), resource access (e.g., invocation on an IPC-gate), and page-fault handling should all be localized to the same processor core whenever possible. Permissions and quotas are arranged hierarchically and managed locally. Reallocation and resource balancing is performed at a coarser granularity. The rationale for core-localization is to both minimize cross-core communications and decentralize resource management (reducing contention). Cross-core IPC is approximately ten times slower (see Section 6.1) than same core IPC. Cross-core data sharing leads to unpredictable levels of degradation due to serialization on locks and underlying side-effects such as false-sharing.

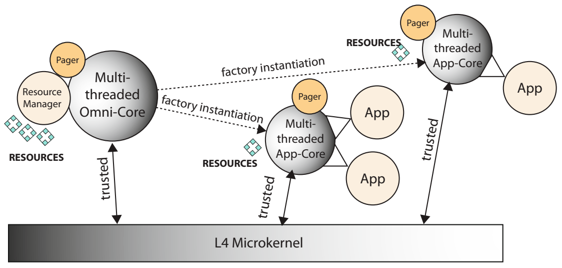


**Fig. 3.** OmniRE High-level Architecture

Our design includes two key elements: 1) *Omni-Core* and 2) a set of *App-Cores*. Omni-Core forms the root of the resource management tree; it manages the highest level of resource partitioning. App-Cores are localized delegates that are instantiated by Omni-Core (see Figure 3). Each App-Core is isolated in a separate process. They are assigned a coarse-grained allocation of resources from Omni-Core, which is dynamically load-balanced across App-Cores as needed. The detailed architecture is given in Figure 4.

App-Cores are the direct representative of the runtime environment for applications. They are instantiated by Omni-Core (either at boot time or dynamically) and indirectly used to load applications. The logical resource partition (i.e., set of quotas) for an application is managed by an *App-Env* (Application environment) that is instantiated inside the App-Core. Application requests to the App-Env are associated with *Service Points* that can be used to further partition the application's resources on a per-thread basis.

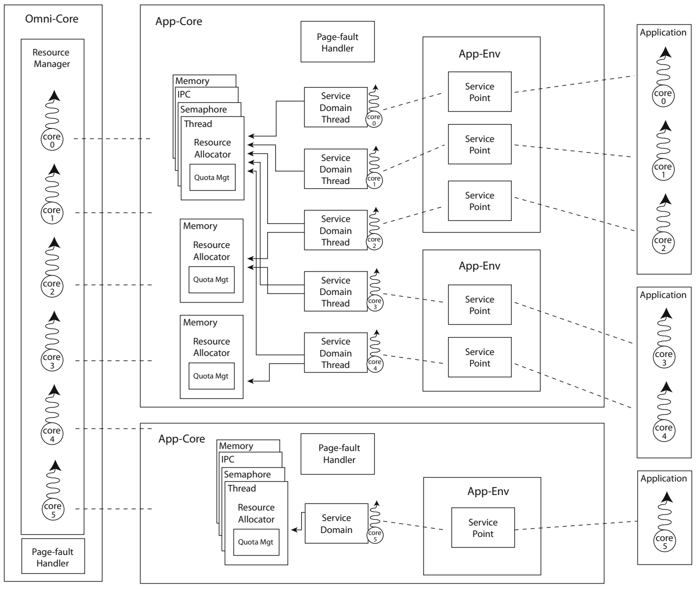


**Fig. 4.** Detailed OmniRE Architecture

Service Points are also useful (as a level of indirection) for transferring resources as a thread migrates between cores or applications.

Resource management in each App-Core is decentralized so that resources that have non-uniform access properties (e.g., memory, CPUs) can be separated out. To facilitate this, each App-Core maintains a number of *Service Domain Threads* that redirect resource requests to different *Resource Allocators*. Resource Allocators exist for each type of resource in the system (thread, process, memory, IPC gate, semaphore, IRQ object). They manage a strict quota of resources, defined by a secure system specification, that is assigned to the App-Core by Omni-Core during start-up. A key use of Resource Allocators is to support NUMA memory allocation across multiple memory controllers.

Essential to the design is that both Omni-Core and the App-Cores are part of the Trusted Computing Base (TCB). This means that the App-Core is trusted to, 1) prevent violation of quotas agreed with Omni-Core, and 2) not abuse its privilege to directly invoke the kernel and request kernel-level resources (e.g., threads). It is the responsibility of the App-Core to manage the application's access to resources according to defined quotas. Applications do not have direct access to either the kernel or to the Omni-Core process. Doing so would break the security model and open up potential for QoS interference between applications.

## 5 Case Study: Physical Memory Management

This section addresses in more detail the hierarchical resource management scheme in the context of physical memory management, which is one of the most basic functions any OS must provide. Different from monolithic kernels (e.g., Linux), where all page-level memory requests are ultimately handled in kernel mode, microkernels such as Fiasco.OC have two Physical Memory Allocators (PMA), one in the kernel and another in user-land.

Kernel functions can directly allocate physical memory through the kernel PMA. However, by default, the amount of memory managed by the kernel is less than 10% of the total. The rest of the memory is managed by a user-level PMA, which allocates memory to applications. We therefore only focus on the user-level PMA design.

**Existing PMA Design.** Figure 5 shows a typical sequence of operations for allocating a page in Fiasco.OC. First, a process allocates a stack variable or heap data through a virtual memory allocator such as *malloc* (step 1). When the virtual address is touched a page-fault exception is raised by the processor, which transfers execution to the kernel's page-fault handler (step 2). The handler forwards the request to a special user-level application, *Sigma0* (also known as the *pager*), which by default handles all page-faults in the system (step 3). In a multicore environment, it is possible to have multiple page-faults taking place on different cores simultaneously. In this case the kernel page-fault handler serializes them and forwards the request to *Sigma0* one by one. When *Sigma0* receives a page-fault notification IPC call, it requests a physical page from its PMA (steps 4 and 5). The result is then sent back to the kernel (step 6), which then populates the page table for the faulting process. After that, the kernel page-fault 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 (steps 2, 3, 6 and 7), two of them being IPC calls (step 3 and 6). This is the cost that a microkernel design must pay for security and reliability.

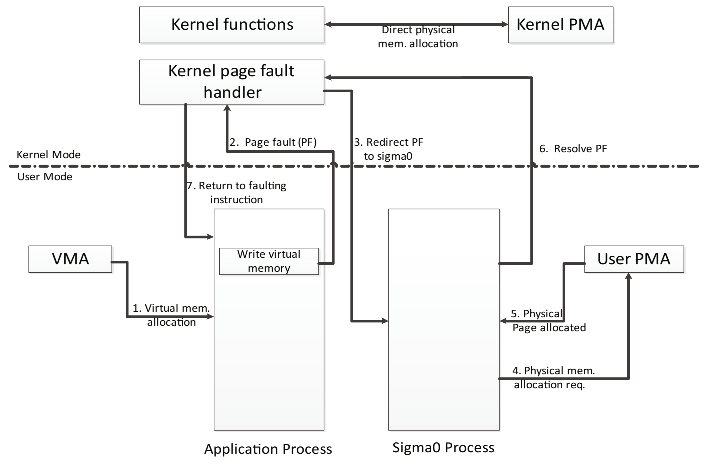

Kernel functions
Direct physical mem. allocation
Kernel PMA
Kernel page fault handler
2. Page fault (PF)
3. Redirect PF to sigma0
6. Resolve PF
Kernel Mode
User Mode
7. Return to faulting instruction
VMA
Write virtual memory
User PMA
1. Virtual mem. allocation
5. Physical Page allocated
4. Physical mem.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

**Fig. 5.** PMA Process in Fiasco.OC Kernel

**Scalable PMA Design in OmniRE.** The user-level PMA can be arranged globally, for each NUMA zone, for each core or even for each process. In order to minimize cross-core IPC we chose a per-core PMA and per-core pager design (see Figure 6).

The benefit of this design is that it addresses two prominent scalability inhibitors. First, it reduces pager contention by distributing page handling across cores. 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 - making the behavior comparable to that of a monolithic kernel design. The result is that general performance and scalability of page allocation is largely improved. Furthermore, CPU utilization can also be maximized because no local physical memory requests interrupt applications executing on a remote cores.

The implementation of the per-core PMA scheme in OmniRE is to first partition physical memory and then construct per-core pagers as shown in Figure 7. When OmniRE is booted, it first loads *Sigma0* as the default pager and then hands off the paging for applications to Omni-Core. It is necessary to load *Sigma0* as this provides paging for the kernel and Omni-Core itself. As illustrated in Figure 7, Omni-Core first obtains all physical memory that is made available by the kernel. Management of this memory is then delegated to App-Cores which are localized with the applications.

To "link" the associated App-Core to each application, the kernel Process Control Block (PCB) pager field is modified. This effectively enables per-core paging (see Figure 6).

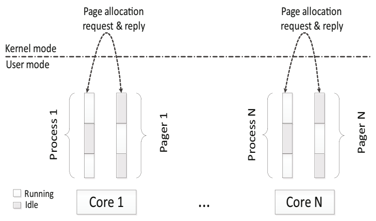


**Fig. 6.** Creating One Pager For Each Core Improves Scalability

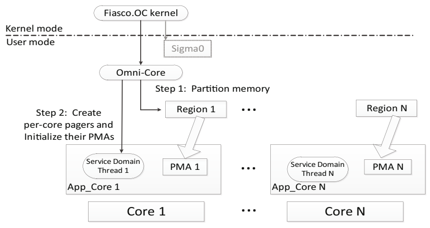


**Fig. 7.** Initialization of Per-core Paging and PMA

## 6 Experimental Results

The current OmniRE prototype is implemented on a 32-bit x86 platform. Performance and scalability results are collected from benchmark executions on both a 48-core (4x12) AMD Opteron-based server platform and a single 6-core Intel Xeon workstation. Timing measurements are taken using the on-chip time stamp counters. Measurements for performance and scalability are taken from a series of micro-benchmark applications. All benchmarks are based on replicated processes (pinned to individual cores) to remove the effects of contention on a shared page table by threads in a single process. Table 1 gives additional detail of the two test platforms.

### 6.1 Fiasco.OC IPC Scalability

In this section, we provides results for the scalability of IPC. We chose to include this data because of the fundamental importance of IPC performance and its broad effect

**Table 1.** Test Platform Specification

| | |
|---|---|
| L4Re | Fiasco.OC Revision 36. x86 32-bit build. |
| Linux | Ubuntu Linux kernel 3.0.0-16 server stock build (x86_64). |
| Compiler | GNU GCC 4.4.6 with optimizations on (O2). |
| AMD Magnycours Server (Dell R815) | CPU: 4x AMD Opteron 6174 2.2MHz CPU. Each multi-chip module package (processor) combines 2 dies of 6 cores. DVFS is turned off. |
| | L1 cache (64KB data per core, 64KB instruction per core). L2 cache (512KB per core). L3 cache (12MB per socket). |
| | 32GB DRAM; integrated DDR3 with support up to 42.7 GB/s memory bandwidth per CPU. |
| | Four x16 Hypertransport links @ up to 6.4GT/s per link. |
| Intel Xeon Workstation | CPU: 1x Intel W3670 3.2GHz 6-core. DVFS is turned off. HT is turned off. |
| | L1 Cache (64KB data per core, 64KB instruction per core). L2 cache (256KB per core). L3 12Mb shared. |
| | 4GB DRAM on single memory controller. |

on scaling on the OmniRE personality. In this benchmark processes are arranged in pairs either on a single core (same-core) or on adjacent separate cores (cross-core). For cross-core, pairs are built up in clusters on the same die. Each pair exchanged 1 million IPC messages in a ping-pong fashion. The implementation has identical semantics to the L4Re functions `l4_ipc_call` and `l4_ipc_reply_and_wait`. Total time to complete the exchange is measured and the mean taken across all cores.

Figure 8 shows the IPC scaling results. Same-core performance is approximately one order of magnitude faster than cross-core. On the AMD platform, mean (per-pair) performance actually improved by 16% over an increase of 36 cores (6-42). We believe that this increase in performance is an artifact of the hardware architecture, specifically the size and design of the Opteron's cache. A similar trend is observed for the memory management benchmark on the AMD platform, which we will describe later. On the Intel platform, the data shows negligible performance change for both cross-core and

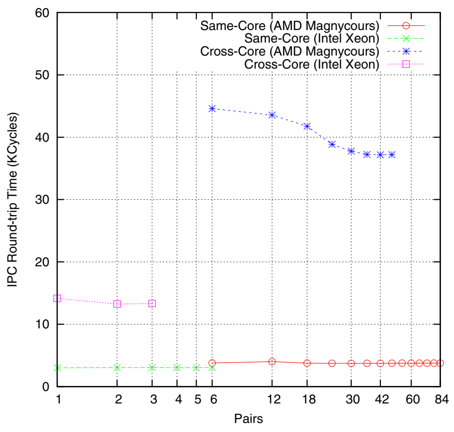


**Fig. 8.** Fiasco.OC IPC Performance

same-core IPC. In summary, Figure 8 validates our design in the following two aspects: 1) Minimizing cross-core IPC on multicore architecture whenever possible has significant performance gain. 2) Decentralizing resource management is preferable as both same-core and cross-core IPC are scalable.

## 6.2 Memory Page Management with L4Re

In this Section, we measure the basic memory page allocation, mapping and freeing in both OmniRE and L4Re. Each benchmark is executed as a single process running on a dedicated core. The benchmark performs 100 iterations of a memory allocation sequence. Each task first allocates a batch of 100 pages (1 page per allocation), then touches the first byte of each page, which invokes the physical memory allocation, and finally frees all 100 allocated pages. A *coordinator* process is used to synchronize the launch of benchmarks after loading to ensure as close as possible start-times. Similarly, L4Re benchmark uses a sequence of `alloc(..)`, `attach(..)` followed by `detach(..)` and `free(..)` during each iteration for fair comparison.

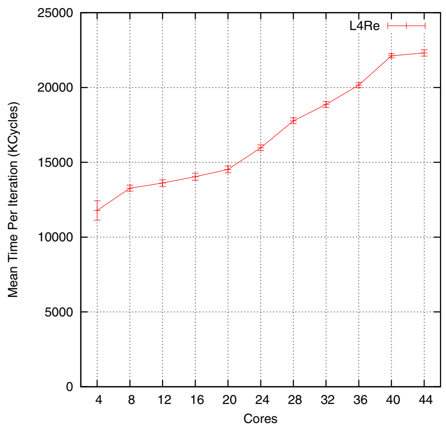


**Fig. 9.** Scalability of L4Re Memory Page Management (AMD Platform)

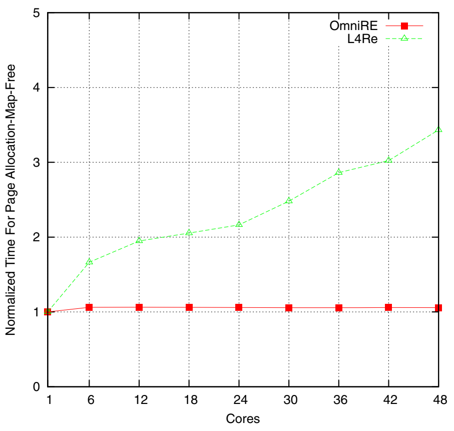


**Fig. 10.** Normalized Scalability Comparison with L4Re (AMD Platform)

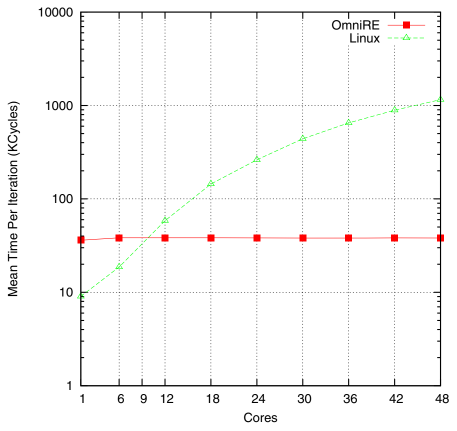
Figure 9 shows the absolute mean time per iteration for L4Re as the number of cores increases. From this figure we can clearly see that degradation of L4Re for the equivalent benchmark using the `alloc`, `attach`, `free`, `detach` APIs is measured at 89% over 40 cores (2.2% degradation per core). We believe that the cause of the degradation in L4Re is due to contention of the page-fault handler (*Sigma0*) which is, in this implementation, single-threaded.

Figure 10 shows a normalized comparison of OmniRE and L4RE memory scaling. As the number of cores increases up to 48, the normalized time of OmniRE remains effectively flat; the degradation is only about 5% when 48 cores are used. In the case of L4Re, however, the performance degrades over 240%. Due to the multi-threaded page-fault handler in OmniRE, this resource contention is largely eliminated.

## 6.3 Memory Page Management Compared with Linux

In this section, we present experimental comparison data for Linux. The benchmark for OmniRE is the same as described in the previous section, except that we allocate a total number of 100K pages in batches of 600 pages (2.4MB). Each allocation is still one 4K page. The Linux implementations of this benchmark use `mmap` and `malloc` based APIs. The `mmap` API provides a means to *eagerly* map physical pages so that a page-fault is not generated. The OmniRE benchmarks also use eager mapping for fair comparison.

The results given in Figure 11 show that for single-page allocations on the AMD platform, OmniRE's page management is able to scale almost linearly whilst Linux degrades exponentially going from 9000 cycles on a single core to 1.1M on 48 cores (note the logarithmic y axis scale). Figure 12 shows the single-page allocation data for the Intel Xeon platform using 6 cores. The data shows OmniRE degradation of less than 0.8% over 6 cores and degradation of more than 35.5% for Linux. However, at this low core count absolute performance of Linux is higher than that of OmniRE, as the OmniRE's scalability advantage has yet to offset its inherent microkernel limitations (e.g., doubled IPC communications between kernel and user-land compared to a monolithic kernel). In fact, Figure 11 clearly shows that only when core count exceeds 9 does OmniRE outperform Linux.



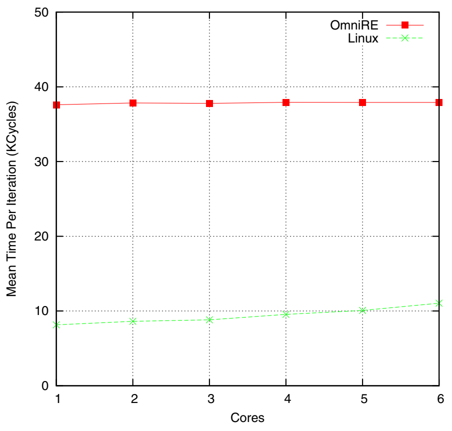


**Fig. 11.** Linux vs. OmniRE Comparison of Memory Page Management Scalability (AMD Platform)

**Fig. 12.** Linux vs. OmniRE Comparison of Memory Page Management Scalability (Intel Platform)

The poor scalability in Linux across 48 cores demonstrates that some critical functions of today's Linux OS do not scale well for even a few cores. Both Figure 11 and Figure 12 show significant degradation of memory management as the number of cores increases. Additional in-house experiments indicate that the cause of this serialization likely relates to the locking strategy on the LRU (Least-Recently Used) page replacement list. It is worth noting that although our Linux data is congruent with data collected by other projects (FOS [19,5], Corey [20] and Barrelfish [4]), we speculate that any difference in the h/w platform (e.g., BIOS, memory) and/or the kernel build may affect the performance. The reader should not focus on this work as a criticism of Linux but simply as a comparison point.

## 7 Related Work

OmniRE is based on the Fiasco.OC microkernel [7] developed by the Operating Systems group of TU Dresden. As part of the Fiasco.OC work the team developed the L4Re user-level personality. However, although the Fiasco.OC kernel has been explicitly designed to support multicore processors [9] there are scalability limitations in the current L4Re mainly relating to the pager *Sigma0*, the IO server and cross-core IPC. Furthermore, L4Re does not provide a secure resource management model but allows applications to directly interact with the kernel and pager, which can potentially result in QoS crosstalk and denial-of-service issues.

The resource management philosophy of OmniRE is inspired by work done by Feske et al. in their Bastei Architecture [8]. The basic premise of their approach is to explicitly manage all resources that are required by both applications and sub-systems. Resources are securely managed through a parent-child trust relationship. The original concepts developed in this work have now been carried through into the commercially supported Genode OS Framework [1]. However, multicore scalability is not currently a primary concern for the Genode Labs group. The current design incurs many cross-core IPC invocations and scalability is limited by single-threaded contention points. OmniRE addresses these concerns at the implementation level and also introduces a different trust model from the Bastei architecture. Also worth mentioning is the resource container work done in the K42 OS [16] that also developed approaches to resource management and donation as a means to alleviate denial-of-service attacks.

The Barrelfish OS [4] developed by Microsoft Research and ETH Zurich Systems Group was started in 2008. Barrelfish is a multi-kernel design that uses the notion of User-level RPC (URPC) to facilitate high-performance IPC exchange via shared-memory region without transitions through the kernel. OmniRE uses a URPC-like approach for communications between the App-Cores and Omni-Core. As with Fiasco.OC, Barrelfish uses a capability model to perform access control to different memory regions. The current implementation is based on 64-bit x86. Data given in [4] shows that Barrelfish degradation for the `unmap` memory operation is approximately 80% at 32 cores on an 8x4 (x8 quad Opteron 8350) AMD platform.

Another prominent OS for multicore processors that is based on a multi-kernel design is the Factored Operating System (FOS) work from MIT [18] [19]. This work is driven by their work on scalable multicore MIMD processors and focuses on the use of spatial distribution to scale OS services including physical resource management, file systems, network protocols and applications. Each system service is "factored" into a collection of Internet-inspired servers that communicate via user-level message passing. The FOS solution is based on a proprietary microkernel and is currently implemented on 64-bit x86. Results collected from a 48-core AMD platform (quad Opteron 6168) reported in [19] showed that over 20 "clients", which we assume correlates to individual processes, FOS's page allocator performance degraded by 60%.

Finally, Tessellation OS [11] from UC Berkeley is a more recent effort to develop a multicore OS that integrates both space and time partitioning to share resources across system services and applications. The Tessellation OS design uses a hierarchical (two-level) scheduling scheme to manage global and local (partition) resource management. As with OmniRE, Tessellation also aims to provide QoS enforcement and minimization of QoS crosstalk. This OS is still in its early stages and as yet no performance and scaling results have been published.

## 8 Conclusions

In this paper we presented OmniRE, a new OS design based on the shared-memory Fiasco.OC microkernel that uses multi-threaded (per-core) system services and resource management delegation to eliminate points of contention and thus promote scalability. We have shown that the Fiasco.OC kernel's use of per-core data structures and internal separation, coupled with the OmniRE personality, provide a complete scalable solution. We implemented and evaluated OmniRE on both AMD and Intel platforms against L4Re and Linux 3.0. Our experimental data shows that OmniRE is able to successfully remove contention on memory management and kernel object management across 48 cores, which substantially outperforms Fiasco.OC and, at higher core counts, exceeds the scaling performance of Linux.

## References

1. Genode, `http://www.genode.org`
2. L4android, `http://l4android.org/`
3. L4re, `http://os.inf.tu-dresden.de/l4re/`
4. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems. In: Proc. of SOSP 2009 (2009)
5. Boyd-Wickizer, S., Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., Zeldovich, N.: An analysis of linux scalability to many cores. In: Proc. of OSDI 2010 (2010)
6. G. D. Broadband: Okl4 microkernel, `http://www.ok-labs.com`
7. T. U. Dresden: Fiasco.oc microkernel, `http://os.inf.tu-dresden.de/fiasco/`
8. Feske, N., Helmuth, C.: Design of the Bastei OS Architecture. Technical report, Technische Universität Dresden (December 2006)
9. Hohmuth, M., Peter, M.: Helping in a multiprocessor environment (2001)
10. Liedtke, J.: On micro-kernel construction. SIGOPS Oper. Syst. Rev. 29, 237–250 (1995)
11. Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., Kubiatowicz, J.: Tessellation: space-time partitioning in a manycore client OS. In: Proc. of HotPar 2009 (2009)
12. Mckenney, P.E.: Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. PhD thesis (2004)
13. Nightingale, E.B., Hawblitzel, C., Hodson, O., Hunt, G., Mcilroy, R.: Helios: Heterogeneous multiprocessing with satellite kernels. In: Proc. of SOSP 2009 (2009)
14. Schubert, L., Wesner, S., Kipp, A.: Reputing microkernels. In: Proc. of UK e-Science All Hands Meeting 2009 (2009)
15. Steinberg, U., Kauer, B.: Nova: a microhypervisor-based secure virtualization architecture. In: Proc. of EuroSys 2010 (2010)

16. Tam, A., Tam, D.K.-F., Azimi, R.: Implementing resource containers in k42 (2003)
17. Thibault, S., Deegan, T.: Improving performance by embedding hpc applications in lightweight xen domains. In: Proc. of HPCVIRT 2008 (2008)
18. Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev. 43(2), 76–85 (2009)
19. Wentzlaff, D., Gruenwald III., C., Belay, A., Kasture, H., Youseff, L., Miller, J.E., Modzelewski, K., Agarwal, A.: Fleets: Scalable services in a factored operating system. Network (2011)
20. Wickizer, S.B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., Zhang, Z.: In: Proc. of OSDI 2008 (2008)