Characterizing OS behavior of Scale-out Data Center Workloads

Chen Zheng, Jianfeng Zhan, Zhen Jia and Lixin Zhang
Advanced Computer Systems Laboratory, Institute of Computing Technology
Chinese Academy of Sciences, Beijing 100190, China
Email: {zhengchen, zhanjianfeng, jiazhen, zhanglixin }@ict.ac.cn

***Abstract*—Data center workloads dominate today's most popular applications, such as search engine, media streaming, and online big data analysis. Continuing efforts have been taken to characterize the micro-architectural characteristics of scale-out workloads. The most striking conclusion from previous work is that the scale-out workloads suffer from a notable front end stalls when compared with other traditional workloads, which leads to inefficiency of pipeline execution. However, they fail to consider the effect of OS execution.**

As an attempt to shed some lights on the effectiveness of OS kernel on scale-out data center workloads, we quantitatively break down the OS behavior on scale-out data center workloads characterized by CloudSuite, and take into account the interferences between micro-architecture and OS execution. For the first time, we investigate how the pipeline front end is affected by OS activities, which are more expensive than application-level operations. The insights derived from OS evaluation help us identify key limits of current OS paradigms. Our studies on OS behaviors also naturally lead to several OS evolutionary recommendations to efficiently manage the diversity of scale-out data center workloads on future hardware architectures.

I. Introduction

Data centers are emerging as a dominant computing platform for processing big data and providing global online services. The growing ubiquity of data center workloads in turn has become a driven force behind the innovative design of operating system and manycore hardware. Many efforts have been taken to optimize the execution of scale-out data center workloads. Some researches focus on energy efficiency [13, 14, 16], and others focus on micro-architecture level characteristics of scale-out workloads [6, 8]. These efforts revealed the limitations of modern computers architecture and explain the inefficiency part of modern processors, such as the front end inefficiency preventing the instruction from fetching into the pipeline. Ferdman et al. [6] further concluded that the needs of scale-out workloads and modern processors are mismatched. Noteworthily, unlike the high performance computing (HPC) application, of which long time tasks run independently in static resource partitions, data center application framework needs to tackle massive amount of dynamic parallel tasks with frequent kernel-intensive operations. Unfortunately, because of the gap between OS and architecture communities [9], the interferences between micro-architecture and OS execution have rarely been taken into account.

In this context, we study OS behavior with representative data center workloads from CloudSuite [6] in a real deployment. To locate the insufficient OS design that triggers scalability and performance problem, we classify the impacts of OS on the continuous execution of applications in two parts: the overhead of essential kernel execution desired by application, and application execution delay due to OS interference. We then decompose these two kinds of events with regard to data center applications, and mainly trace behaviors of the latter one. Moreover, in addition to the quantitative study of OS kernel interference, such as timer interrupts, locks, context switches, and page faults, we also take into account the micro-architectural characteristics due to OS execution, including OS execution stall, ITLB miss and cache miss. So as to characterize unique characteristics of data center workloads, we also compare them with HPCC–a traditional HPC benchmark suite. In addition to the investigation, we give several recommendations for future OS design. The major insights derive from our evaluation are listed as follow:

Front end stalls due to OS activities significantly drop down the execution performance.

Context-switches in scale-out workloads are massive and time-consuming.

Periodic timer interrupt could be critical in performance.

Page-faults overhead in scale-out workloads are high.

Fine-grained locks raise latencies. Fine granularity means more critical section and lock invocation frequency.

The rest of this paper is organized as follows. In Section II, we provide an overview of the stat-of-the-art research. Section III details our evaluation methodology, our experiment environment, and the workloads we use. We present our measurement results in Section IV, concentrating on the gap between data center workloads and current operating system. Accordingly, we summarize the characteristics and offer evolutionary recommendations for future OS design. Section V concludes the full paper.

II. Related Work

Ferdman et al. [6] investigated six popular scale-out workloads, and found the scale-out workloads owns inefficiency front end, which preventing the instruction entering the pipeline. Zhen et al. [8] characterized more data center workloads and found similar phenomenon with [6]. The scale-out workloads are always written in high level languages with third-party libraries, which cause a large instruction working set. The high level languages with third-party libraries used by scale-out workloads also complicate application execution as they are more dependent on the operating system or certain framework, e.g. Hadoop. However, they seldom take into account the interference between micro-architecture and OS execution.

Petrini et al [12], showed how application's scalability is limited by OS noise, severely reducing the performance on

large scale computing scenario. Morari [10] offers a quantitative analysis of OS noise to break down the OS performance bottlenecks in detail. Most of these studies focus on the effects of OS interference on application's scalability, rather than kernel execution overhead in terms of the direct effects on applications' execution time.

The OS scalability has also become a hot topic. Some researchers believe the scalability of traditional OS can be improved using mostly standard parallel programming techniques [4], while other believe that only revolutionary changing the OS structure can achieve that goal [2, 15]. Although they had all proved the feasibility of their OS prototypes to manycore resources and scalable computing scenario, few of them tend to analyze the detail of OS inefficiency on scalable workloads.

Optimizing operating system for system-intensive workloads is an everlasting topic. Like the micro-kernel [3], lightweight kernel (LWK) [11], Exokernel (library operating system) [5], and the runtime OS, they are designed to eliminate system services or move out of privileged kernels and servers into untrusted application libraries. In this paper, we discuss the possible OS optimization approaches from both OS level and the micro-architectural scope, aiming to performance of data center applications. Micro-architectural behaviors of OS execution, such as OS execution stalls and preemptive activities, reveal hidden effects behind the normal execution flows of OS-intensive data center applications.

III. Evaluation Methodology

This section describes the workloads we evaluated: a set of workloads from CloudSuite benchmarks and traditional HPC workloads. We begin with a description of our operating system environment at both the hardware and software levels. Finally, we describe the details of the workloads configuration and performance profiling tools we used.

A. Benchmark Choose and Setups

In order to find the characteristics of OS behavior on real-world data center workloads, we selected three representative workloads from CloudSuite in different application domains, including data analysis, media streaming, and web search. In addition to data center workloads, we deployed two typical HPC workloads from HPCC, and compared them with the data center workloads.

The details of compared workloads are as follows:

Naive Bayes is a representative data analysis workload which applies Bayes' theorem with naive independence assumptions. We deployed it in a 3-node Hadoop cluster with 30GB input dataset. The Media streaming server is deployed on a single node. And we set 20 clients threads by using a Faban driver with GetMediumlow 70 and GetshortHi 30. The frontend and backend of Web Search are distributed into two nodes, respectively. The data segment size is 35GB, and the index size is 17GB. We also set the client with 140 requests per second to gain the best 138 transaction per second. HPCC is a representative HPC benchmark suite. We deployed two compute-intensive benchmarks from HPCC respectively, including HPL and FFT.

B. Experiment Platforms

Our experiments are performed on a three-node cluster to run all the data center workloads. The nodes in our cluster are connected through 1Gb Ethernet network. Each node has two Intel Xeon E5645 processors, 16GB memory and 8TB disk. A Xeon E5645 CPU includes six physical cores with speculative pipelines. We choose commodity Linux as basic OS, whose distribution version is Centos 6.4 with Linux kernel 2.6.38.6. The detail configuration parameters of each node are listed in Table I.

TABLE I. Hardware Configurations.

CPU type	
Intel Xeon E5645	6cores@2.4G, 12 threads
ITLB	64 entries
DTLB	64 entries
L2 TLB	512 entries
L1 DCache	6×32KB
L1 ICache	6×32KB
L2 Cache	6×254KB
L3 Cache	12MB

C. Experimental Methodology

In this work, we collect OS-level performance data by accessing the proc file system, such as the time spent by each thread. In addition, we get the micro-architectural data by using hardware performance counters. We use Perf and VTune [1] to collect cache event and front end stall respectively. In order to get the precise data, we replay each benchmark using different tools to collect performance data. We also perform a ramp-up period for each benchmark, and collect the performance data after they become steady. For Hadoop data analysis workloads, we collect performance data from all child nodes and compute the mean value.

IV. Quantitative Study of OS Behavior on Data Center Workloads

In this section, we investigate OS behavior on real-world workloads across three major data center application domains. Quantitative studies of these behaviors are utilized to illustrate how they prevent data center workloads from efficiency. According to the insights derived, we give several recommendations for future operating system design.

A. OS Behavior Breakdown

Generally, kernel activities take execution resources from application preemptively or on demand. In our study, we mainly focus on the preemptive kernel activities with larger contribution to OS performance degradation.

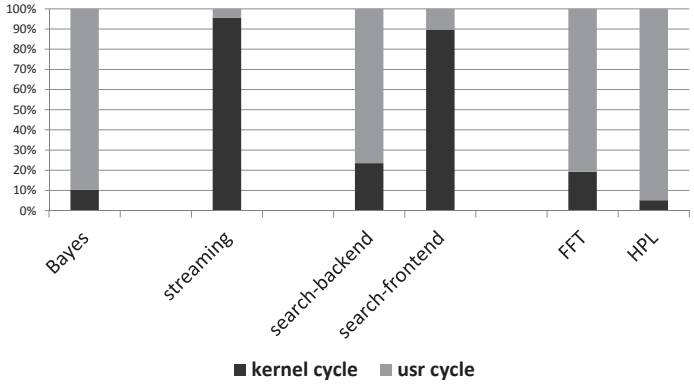

Fig. 1. Execution time breakdown.

We begin exploring the OS behaviors on data center workloads through examining the execution-time in Figure 1. We notice that the media streaming and search service spend most of their execution time in kernel mode, while data analysis workload experiences an average of 10% kernel-time percentage. This behavior is in contrast to compute-intensive HPCC benchmarks, which spend most of their execution time in user mode. Furthermore, although the overall kernel-time percentage of some data center workloads, e.g. bayes and search backend, appear similar to the FFT benchmark, they experience significant different causes. Unlike the frequent inter-process communication in FFT benchmark, bayes and search backend experience most of their kernel execution time due to larger amount of I/O activities and frequent memory access. Notably, although a large part of the presence in kernel mode arises due to system calls invoked by application, the elapsed kernel time when application is forced into kernel by OS activities also takes a large part and has great impact to application performance.

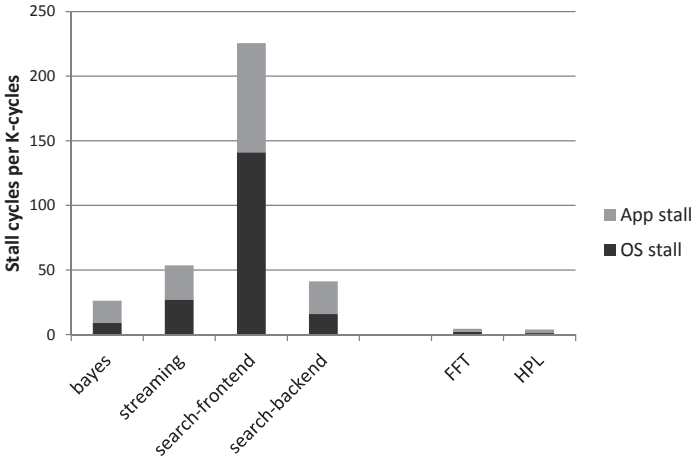

Fig. 2. Front end stall cycles per thousand cycles.

Figure 2 presents respective front end stalls due to application codes and OS activities, in which each bar represents the average stall cycles per thousand cycles. Since data center workloads suffer from notable front end stalls [6, 8], we only take front end stall as a criterion by accessing hardware performance counters. Pipeline front end fetches instructions from L1 Instruction cache, and then decodes and issues instructions to back end. The instruction fetch stalls will induce a lack of instruction to execute. Previous work [6, 8] founds that the data center workloads suffer from a notable front end stalls when compared with other traditional workloads. Our observation in Figure 2 presents the similar phenomenon and further infers that stalls due to kernel instruction execution greatly influence the front end efficiency across data center workloads. The histogram shows that data center workloads ranges from 30 to 230 stall cycles per thousand cycles, which are larger than the stall cycles in HPCC workloads. Meanwhile, we find that the front end stalls due to kernel code execution account for the major part. The higher OS stalls in data center workloads are mainly caused by architecture inefficiencies due to frequent OS activities, which poorly exploit the memory hierarchy. We report the comparison diagram of kernel L1 instruction cache miss and ITLB (Instruction Translation Lookaside Buffer) miss in Figure 5 and Figure 8 respectively, both of which are the major causes to OS stalls and corroborate our observation.

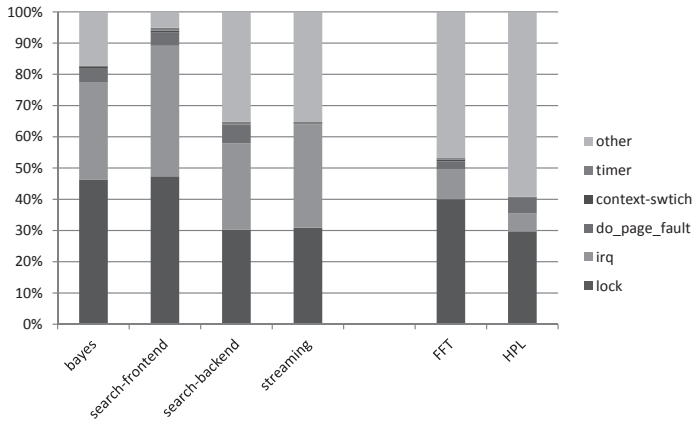

Fig. 3. OS stall breakdown.

We then break down stalls by relevant OS behaviors, and normalized the stall cycles in Figure 3. By inspecting the graph, several lessons can be learned. First, the execution efficiency of data center workloads is most affected by different OS activities. Hence, it provides huge opportunities in this area to design dedicated kernel for certain workload to gain performance benefit by optimizing certain OS activity. Second, we observe that preemptive kernel activities are responsible for the majority of OS stall, including timer interrupt, interrupt handler, context switch, page fault, and lock. The inherent inefficiency of these basic activities and OS abstractions motivates the evolution of current OS design. Thirdly, we find that, OS stalls due to interrupt handler (irq) in data center workloads account for a larger proportion than HPCC workloads. Because data center workloads have more I/O activities and resource contention. Lock invocation has also caught an average of 40% OS stalls across data center workloads. Lastly, although the OS behavior with data center workload differs considerably with each other, the class of data center workloads as a whole has a significant demand for OS than other workloads. In contrast to the light weight kernel (LWK) designed for HPC workloads, data center workloads need a general-purpose OS kernel which could be able to multiplex resources efficiently among diverse workloads. So as to reduce the OS overhead and resource contention, application-aware resource management could be a promising approach. Data center workloads may benefit significantly from more exposed low-level resources.

In the rest of this section, we will present quantitative analysis of each OS activity and argue the evolving gap between modern kernel abstractions and scale-out data center workloads.

B. Context switch

Contemporary OS kernel implements real time multitasking and multiplexing using complex scheduling policies. Most of the schedulers allocate processes among system resources to load balance OS effectively. Schedulers are designed to achieve throughput, latency and fairness. But in practice, the goals often conflict with each other. Thus for running applications, they are preempted frequently by other tasks or kernel daemons.

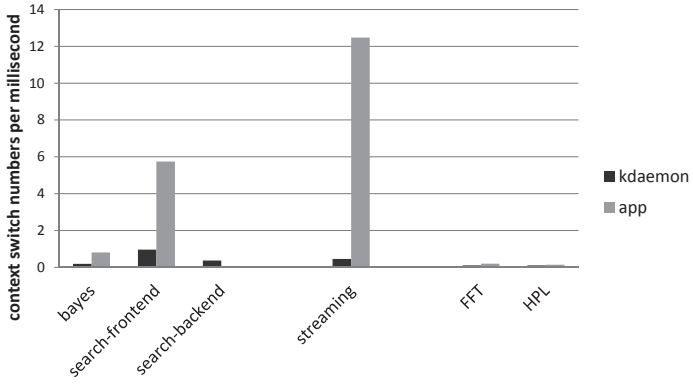

Fig. 4. Context switch frequency. The bar of kdaemon shows the context switch frequency when application processes are switched to kernel daemons. The bar of application shows the frequency when kernel schedules CPU among application threads.

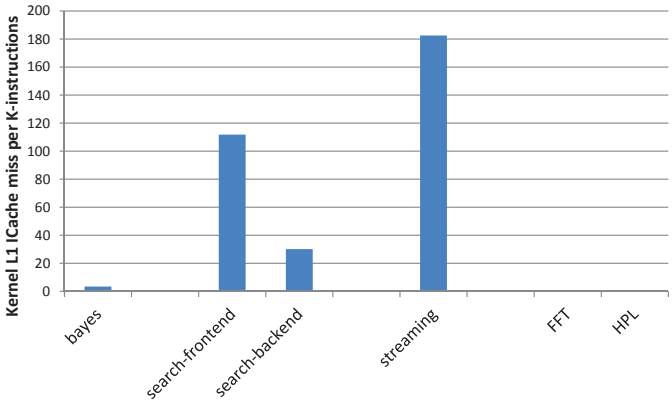

Fig. 5. Kernel L1 ICache misses per thousand kernel instructions.

We investigated OS overhead due to context switch for data center workloads in Figure 4. To avoid confusing with voluntary scheduling in applications, we only track the preemptive switches which inactivate current running tasks or migrate tasks to other processors. We found that data center workloads have higher context switch frequencies on average, in which media streaming achieves the most preemptive context switch frequency. We also find that context switch due to kernel daemons take a higher proportion in streaming workload. Because high concurrent threads are needed to serve for frequent requests in streaming workload and search frontend, which leads to complicated scheduling activities. Frequent context switch and process migration result in calls to process manipulation, and accompanying local cache update accounts for the majority of these calls, which significantly degrades performance. Kernel L1 instruction cache misses rates of diverse workloads in Figure 5 indicate the locality of data center workloads, part of which is caused by context switches. The kernel L1 instruction cache miss also contributes to the OS front end stalls. In Figure 6, we present context switch latencies breakdown, ranging from 10% to 58% of kernel mode time for data center workloads. Unlike media streaming and bayes, we find maximum sensitive to preemptive kernel daemons in search frontend. The kernel daemons contribute almost 20% of all the time delay to search frontend, such as kworker,

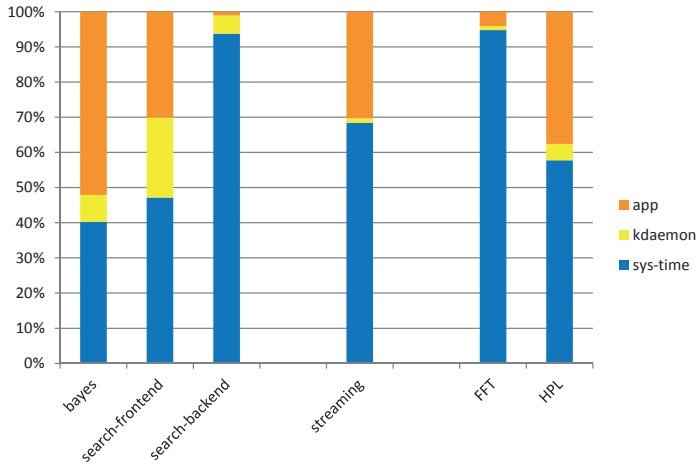

Fig. 6. Context switch latency breakdown. We normalized the kernel mode time of different workloads, and then synthesized the kernel time caused by context switch due to application scheduling and kernel daemons preemption.

ksoftirqd, watchdog, irqbanlance and etc. Despite the enhanced features and flexibility, kernel daemons could be a major cause to context switch latency by preempting the execution resource that follows a cache flush while the workloads are in progress.

The time of scheduling activities can be divided into two major parts: the time to execute the kernel scheduler code, and the time spent for process preparation. The latter one enlarges due to updating local cache lines and process state data, when waking up a sleep process or migrating a process to another CPU. So in Table II, we break down in detail about the scheduler activities. We observe massive task migration latencies across data center workloads, which take an average 8% of all the context-switch penalties. From Table II, we find that the time duration of single context switch is very large and differs considerably from one to another. The minimum latency of a context switch is similar across all the workloads, but the maximum latency varies significantly. The kernel activities that vary so much may limit kernel scalability, as shown in previous work [12].

Recommendation: To improve efficiency and reduce context switch latencies, OS kernel designed for data center workloads must take locality of task resources as prime design constraint. Although some priority-based schedulers can be effective for particular processes execution, it disrupts scale-out workloads significantly. Rather than relying on some complex scheduling policy or cache locality technique, which follows the current OS evolutionary approaches, we need to promote new OS paradigms and take a long-term view on the revolutionary problems given the scale-out trends of data center workloads and manycore hardware. In HPC area, LWK is designed as a computing node with majority of OS activities implemented in remote node. Moreover, LWK directly removes kernel daemons to avoid context-switch latency due to daemon preemptions. However, these approaches are ineffective for data center workloads.

There are three key costs that deserve attentions: the cost due to frequent task migration, the varying cost of single context switch, and the cost due to preemption of kernel daemons. We believe that kernel daemons should be deployed on dedicated cores or kernels away from the ones for appli-

TABLE II. CONTEXT SWITCH LATENCY.

	Avg (millisec)	Max (millisec)	Min (millisec)	Migration cost (%)	Total kernel time (%)
Bayes	0.373	2910.154	0.002	9.879	37.126
Search-frontend	0.133	21680.3	0.007	11.781	89.359
Search-backend	0.047	4700.255	0.004	8.812	2.074
Media Streaming	0.032	13.903	0.004	4.322	41.125
FFT	0.030	17.172	0.005	10.261	4.572
HPL	0.082	1636.752	0.006	8.056	0.143

cations, which implicates that no preemption can pollute the cache locality among applications and kernel activities. Co-operation tasks in single address space could be attractive to significantly reduce the context switch cost. In some extreme methods, without the preemption of tasks, we could build multiple cooperative multitasking kernels wherein tasks yield actively when they do not need system resource.

C. Page Fault

Most of current OS kernels implement virtual memory techniques for multitasking and address space isolation. Virtual memory combines active main memory and other secondary memory as a whole, to make a direct and contiguous address space for each task. Additionally, virtual memory delegates to OS kernel the management of process memory, such as page in, page out, and page fault handler. When a process requests a page that is not ready or not even in the current memory resident set, hardware raises interrupt to halt the running task. Kernel then invokes page fault handler to determine validity of the address and swap in the desired page. Page faults, by their very nature, degrade the performance of running tasks and in the degenerate case may cause kernel thrashing.

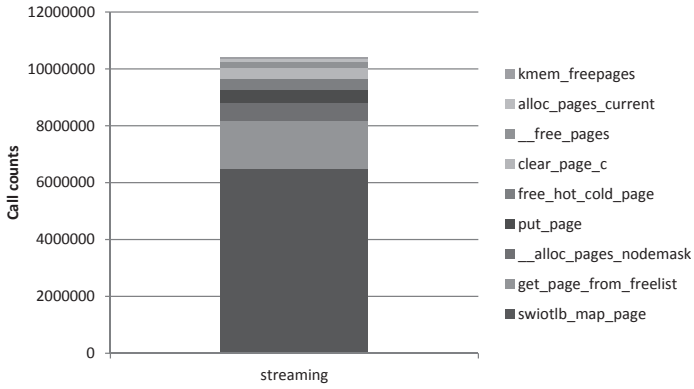

Fig. 7. Invocations into kernel memory management code.

Modern kernel uses complex page replacement algorithms which dedicate to maximize the page references, and heuristic algorithms such as Least Recently Used (LRU) algorithm are also proposed. Unfortunately, in practice, page reference is hard to reach for fair scheduling among different tasks. Table III presents the page fault statistics in general. The diversity of page fault latency is due to different calls raised by do_page_fault. We find that, HPCC workloads suffer far more page faults frequency surpass the ones in data center workloads. This is due to the short execution time of two HPCC workloads, which highlights the memory accessing behaviors when ramp-up period takes large part of the overall execution time. In contrast, the ramp-up period of data center workloads just accounts for a minority part, and we

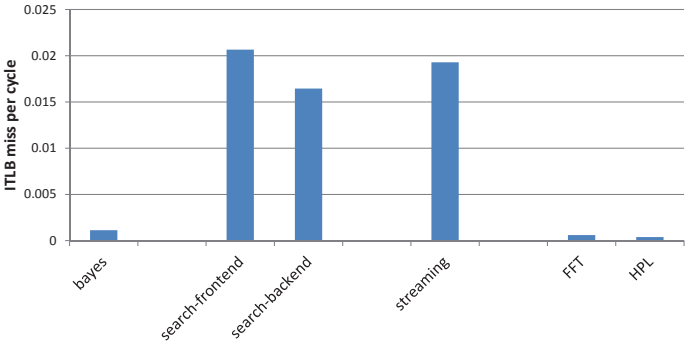

Fig. 8. ITLB miss rate.

only collect performance data when workloads are in steady period. We also find that data analysis workload, bayes, suffers more page fault frequency than other data center workloads. Because bayes workload that bases on Hadoop framework experiences frequent thread creation and data shuffling, which raise massive page fault exceptions once requested pages are not resident in memory. Figure 7 shows the major subsequent kernel activities after page fault exception in media streaming workload, including page mapping, page allocation, page free and etc. In other words, page fault overhead needs to take page creation time, disk rotational time, page seek time, and page transfer time into account. Thus, page fault handler itself will significantly slow down the overall performance. Figure 8 presents the ITLB miss rates of data center workloads. In general, we observe that both web search and media streaming own high ITLB miss rates, when comparing with bayes and HPCC workloads. This is because the process scheduling of these two workloads is far more complicated than the ones in HPCC workloads according to Figure 4. Moreover, it also means that the resource locality for web search and media streaming are not as good as bayes, which brings more severe penalties for web search and media streaming. In general, the frequent page fault occurrences implicate higher ITLB miss rates and cache miss rates. The corresponding page walk and page fault handler bring more severe page fault penalty.

Recommendation: In general, the fundamental solution to the page fault latency is to improve the locality of memory access. Memory management functionalities, like page-on-demand and copy on write, may reduce the impact of page fault. Moreover, drastically simplifying dynamic memory management may also take effect to ensure the locality of each task. Multi-kernel OS design subverts the traditional memory behavior in OS kernel. One OS kernel is divided into multi kernels, which brings much finer granularity in memory accessing and performance isolation. State of the art in multi-kernel design can be classified into the following two categories. One is to design single OS image upon multi-kernel, and the other just

TABLE III. PAGE FAULT LATENCY.

	Frequency (/sec)	Avg (millisec)	Max (millisec)	Min (microsec)	StdDev (millisec)
bayes	104.175	0.009	6.582	0.999	0.082
Streaming	31.9	0.007	0.600	1.999	0.003
Search-frontend	29.561	0.595	14.492	2.010	0.470
Search-backend	27.207	1.411	354.552	2.001	1.347
FFT	60.829	2.215	11068.89	0.999	3.060
HPL	142.368	3.340	16906.492	1.999	7.927

maintains multi-kernel OS as a distributed system with every kernel in an independent context. We believe that the latter is more likely to become evolutionary trend for OS design, because it simplifies interaction and resource multiplexing across kernels. Furthermore, the flexibility and proper isolation also bring us chances to move each kernel forward to gain performance benefits. In more radical approaches, applications with access to low-level resources, such as memory, privileged instructions, devices, and etc, could guarantee the resource locality to considerably reduce the page fault penalty.

D. Interrupt

Apart from exception and scheduling, interrupt is another inevitable kernel function to implement modern multitasking OS, but also a critical source to degrade application performance. To identify the sources of OS interrupt noise, we quantitatively measure the interrupt statics contributed by various interrupts and system daemons.

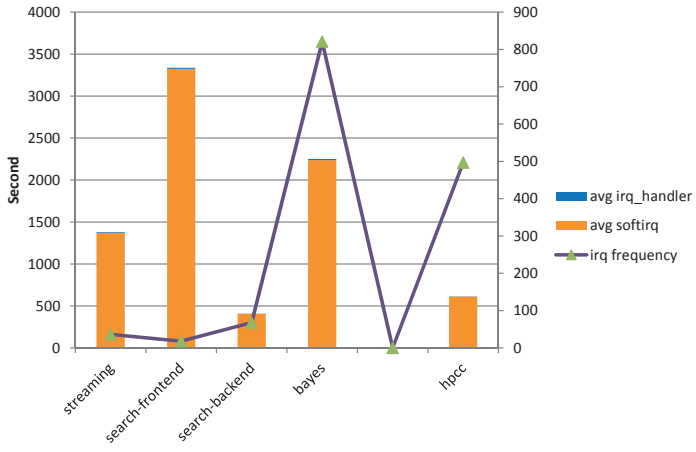

Fig. 9. Interrupt latency and frequency. The irq_handler histogram shows the average latencies of top-half interrupt handlers in different workloads. The sofirq histogram offers the average latencies of bottom-half interrupt handlers . The curve is the interrupt frequency in different workloads.

The histogram in Figure 9 shows the interrupt breakdown and the curve presents the interrupt frequency. On the average, data analysis workloads yield high level of interrupt latency which worse than HPCC benchmarks. We also find that, the average time softirq spent far surpass the time interrupt itself cost. Figure 10 breaks down the total softirq overhead into several major interrupt handlers. Accordingly, the periodic timer interrupt contributes from 40% to 90% of all the softirq overhead experienced by data center workloads. The rest comes from network interrupt, block softirq, scheduler, and RCU. Periodic timer interrupt consists of timer interrupt handler, run_timer_softirq and process preemption to execute the expired timer events. Furthermore, it may also trigger other activities consuming more CPU time, such as scheduling,

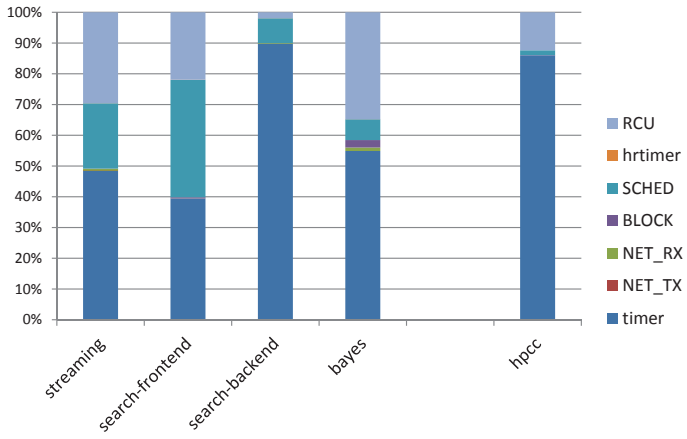

Fig. 10. Softirq latency breakdown of different workloads. The histogram shows the top seven time-consuming bottom-halves interrupt handlers.

process preemption, and system time adjustment. The large part of timer interrupt latency can be explained by the relevant time-consuming softirq and kernel daemon preemption. Unlike HPCC workloads, performance of bayes, search frontend, and media streaming workloads is also dominated by network interrupt due to large among of data shuffling and requests handling. We also observe that most of timer interrupt and network interrupt are provided to CPU 0. Thus, the task running on CPU 0 will significantly been delayed due to preemption and task migration.

Recommendation: Corroborating previous work [7], both the data analysis workloads and the service workloads suffer from timer interrupt and the uneven distribution among processors. However, we note the significant differences between the data center workloads and HPCC workloads in terms of timer impact: the data analysis workloads suffer more time latencies due to subsequent cache inefficiencies and TLB misses caused by timer interrupt, while the HPCC workloads suffers more latencies when interrupts extend the waiting time to perform barrier operation. This observation indicates that dynamic timer mechanism may bring better OS performance in the future. For application executing on different processors, we need independent local software timers to guarantee that their timer interrupt frequency change as the application I/O blocking. This mechanism will significantly decrease the time cost due to meaningless trapping in kernel. The multi-kernel OS design may facilitate the implement of dynamic timer, for the less mutual dependency among kernels.

E. Lock Overhead

Modern kernel has been evolving from uni-processor to support multi-processor, even manycore processors. Unfortunately, locks still dominate OS front end stall and scalability problem. The big kernel lock which prevents multiple kernel

threads has been replaced by much more finer-grained locks in mainland kernel to reach concurrency. Lock of fine granularity is designed to reduce the probability of resource contention. However, fine-grained lock, by its very nature, limits the OS scalability and achievable performance. Since fine granularity raises lock frequency and complicates the kernel design, the relevant overhead could be a disaster for application execution. In some degenerate case, the lock takes more cycles than the critical section. Unfortunately, this can be normal for fine-grained lock.

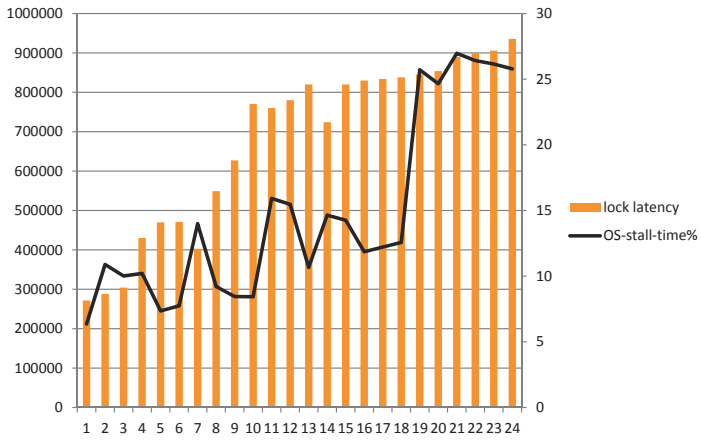

Fig. 11. Lock latency and OS stall percentage. The histogram shows the overall lock latencies of bayes workload with growing CPU core number. And the curve records the OS stall percentages due to lock.

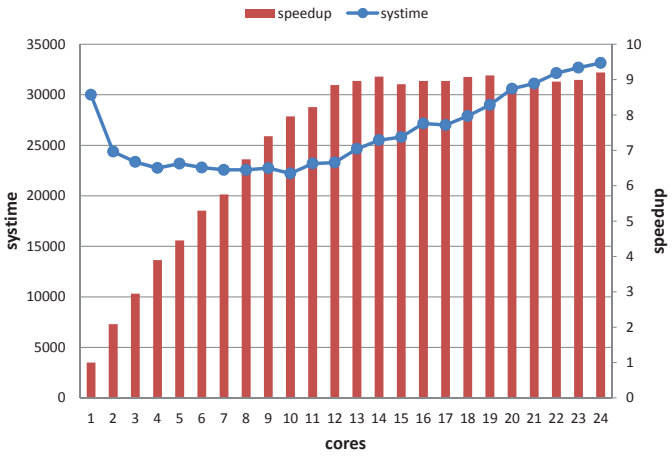

Fig. 12. kernel time and speedup when enlarging core number.

To analyze how OS lock overhead scales with the modern hardware, we investigate the utility by enlarging the core counts for bayes workload in Figure 11. A single lock overhead is measured by both lock acquire time and lock release time. The number of cores evolved from 1 to 24, while the lock overhead begins to dominate the execution time and front end stalls. Because the enlarging core number increases time overhead due to cache coherence broadcasts for lock acquiring. The growth of lock invocation frequency due to fine granularity is also a major cause. We also present OS stalls due to lock, it mainly caused by lock contention which leads to resource contention in cache level. The histogram in Figure 12 presents the speedup over core number, while the curve shows the kernel time when running bayes benchmarks. We find that the application speedup is restricted by OS stall and the approximately linear growth kernel execution time. The abnormal kernel time before three cores could be explained by sharp preemption for scarce CPU resources. The speedup of workload increases linearly until 14 cores, and remains among 9 times. In general, the increasing lock interference and restricted speedup indicate the insufficient of current OS design in scalability.

Recommendation: Improving the lock granularity may prompt the performance of OS kernel. However, although finer-grained lock reduces contention probability, it may increase the frequency of lock invocations, and even enlarges critical sections, which inherently degrades OS performance. Moreover, the programming difficulty and cost to make the lock finer-grained may not be acceptable. Hence we recommend that OS kernel should be designed to avoid the use of lock in the common case or to design lock-free mechanism. Thus multi-kernel OS without single OS image that uses local data structure will definitely reduce the frequency of cross-core locks. We envision that the usage of message passing in OS kernel should become evolutionary trend for OS design, for it may reduce the lock invocation due to memory sharing. Moreover, message-based system call methodology is also a big challenge for OS designer, which will totally break the current approaches.

V. Conclusion

In this study, we performed a quantitative study of OS behavior on scale-out data center workloads, taking into account the interferences between micro-architecture and OS execution. Our investigation shows that performance of data center workloads are significantly dominated by preemptive OS activities in contemporary OS design. We also, for the first time, presented how the front end is affected by OS activities. We then explained the specific cause and impact of each OS activities, and recommended the OS design evolutionary approaches that can lead to shine in the future. Specifically, our analysis showed that efficiently executing data center workloads requires scalable OS activities, independent local data structure, dynamic timer interrupt, and lock-free mechanisms. Accordingly we recommended that multi-kernel structure with more local resource allocation could be a leading trend for future OS design.

Acknowledgment

We are very grateful to anonymous reviewers. This work is supported by the Chinese 973 project (Grant No.2011CB302502) and the NSFC project (Grant No.60933003).

References

[1] "Intel vtune amplifier xe performance profiler." *http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/*.

[2] 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, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[3] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan *et al.*, "Microkernel operating system architecture and mach," in *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992, pp. 11–31.

[4] 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*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[5] D. R. Engler, M. F. Kaashoek *et al.*, "Exokernel: An operating system architecture for application-level resource management," in *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, 1995, pp. 251–266.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 37–48, Mar. 2012.

[7] K. B. Ferreira, P. G. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Piscataway, NJ, USA: IEEE Press*, pp. 1–12, 2008.

[8] Z. Jia, J. Zhan, L. Wang, and L. Zhang, "Hvcbench: A benchmark suite for data center," *The 19th IEEE International Symposium on High Performance Computer Architecture(HPCA 2013) Tutorial Technical Report*, 2013.

[9] J. Mogul, A. Baumann, T. Roscoe, and L. Soares, "Mind the gap: reconnecting architecture and os research," *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pp. 1–1, 2011.

[10] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 852–863, 2011.

[11] A. Morari, R. Gioiosa, R. W. Wisniewski, B. S. Rosenburg, T. A. Inglett, and M. Valero, "Evaluating the impact of tlb misses on future hpc systems," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 1010–1021.

[12] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pp. 55–, 2003.

[13] Z. Ruijin, L. Chao, and L. Tao, "Enabling distributed generation powered sustainable high-performance data center," *The 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[14] Z. Ruijin, L. Zhongqi, and L. Tao, "Exploring high-performance and energy proportional interface for phase change memory systems," *The 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[15] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating SYstems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[16] J. Zhan, L. Wang, X. Li, W. Shi, C. Weng, W. Zhang, and X. Zang, "Cost-aware cooperative resource provisioning for heterogeneous workloads in data centers," *Computers, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2012.