View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2011-035 July 27, 2011

# Vote the OS off your Core

Adam Belay, David Wentzlaff, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

# Vote the OS off your Core

Adam Belay, David Wentzlaff, and Anant Agarwal
CSAIL, MIT, Cambridge, MA 02139
{abelay, wentzlaf, agarwal}@csail.mit.edu

## ABSTRACT

Recent trends in OS research have shown evidence that there are performance benefits to running OS services on different cores than the user applications that rely on them. We quantitatively evaluate this claim in terms of one of the most significant architectural constraints: memory performance. To this end, we have created CachEMU, an open-source memory trace generator and cache simulator built as an extension to QEMU for working with system traces. Using CachEMU, we determined that for five common Linux test workloads, it was best to run the OS close, but not too close — on the same package, but not on the same core.

## 1. INTRODUCTION

The number of cores per package has been growing dramatically in recent years, with as many as 16 cores in AMD's recently announced "Bulldozer" processor, 48 cores in Intel Lab's "Single-chip Cloud Computer" [9], and 64 cores in Tilera's TilePro64. Multicore chips commonly allocate increasing transistor budgets from Moore's Law to increasing the number of cores, while keeping the sizes of per-core caches more or less constant. In this environment, one issue that is especially pressing is the conservation of off-chip bandwidth [7]. Utilizing the bountiful multicore resources in the face of new cache tradeoffs and memory-bandwidth constraints will require changing the way we create our software systems. Specifically, we challenge the traditional notion that an OS should run on the same core as the application that invokes its services, and instead posit that cache effects in modern multicores justify moving the OS to a separate core.

We show that *competition* between the OS and the application for limited cache resources is a significant source of cache misses and does not efficiently make use of on-chip resources. We further demonstrate that the benefits of data sharing, or *cooperation*, between the application and the OS is not sufficient to mitigate the ill-effects of cache interference.

Emerging work in operating system scalability [5, 14, 2] has suggested that OSes should be restructured so that some or all OS services run on dedicated cores rather than multiplexing with the application. FlexSC [13] further confirms this notion and generalizes it to conventional monolithic OSes by showing that the asynchronous rerouting of Linux system calls to a different core than the application that invoked them can improve overall performance.

However, it has been difficult to isolate quantitatively the advantages of dedicating cores to OS services since prior studies tend to provide full system measurements that encompass several factors such as competition, cooperation, concurrency and contention, many of which are unrelated to our central question of whether dedicating OS cores makes sense from the perspective of cache effects. Although one might make the engineering argument that in the final analysis the user only cares about ultimate performance without regard to where that performance comes from, we do believe it is important from a science viewpoint to understand the contributions of each of the factors.

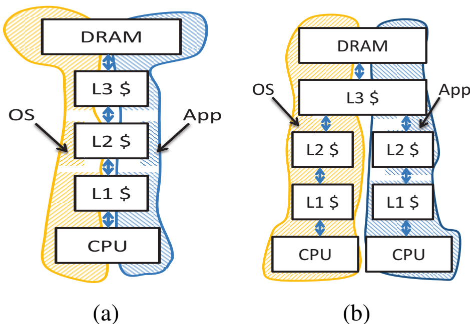
The chief factors which determine whether OS services should be run on dedicated cores include competition, cooperation, concurrency, and contention, and are discussed below.

- **Competition:** When the OS and the application run on the same core, they are fighting over the processor's tightly constrained architectural resources. This includes multiple layers of memory cache, TLBs, branch prediction state, write back buffers, register state, and CPU execution engine state.
- **Cooperation:** When the OS and the application share data, e.g., common memory buffers, they can share more efficiently if they run on the same core because such data is likely to be hot in a nearby cache level. Although sharing is less efficient on different cores, it is still more efficient than if the OS service ran on a different chip.
- **Concurrency:** When the application can do work that does not depend on the result of a pending system call on a different core, the OS and the application can make forward progress in parallel.
- **Contention:** Running the OS on a single core or on a small number of cores can reduce OS lock contention and OS data sharing costs compared to running the OS

on a large number of cores. For highly parallel applications, when the OS runs on the same core as the calling application, the number of cores on which the OS runs can be large and independent of the number of cores optimally suited to the OS.

In the past, concurrency (or the lack of it) has tended to overwhelm other benefits of running the OS and application on separate cores. However, we note that *concurrency* is only applicable to certain workloads and that contemporary programming techniques have effectively mitigated *contention* in Linux for at least 48 cores [6]. Similarly, new OS designs such as Corey [5], BarrelFish [2] and fos [14], are built with scalability as a key goal and take great pains to reduce *contention*. Given these considerations, *competition* and *cooperation* have the potential to be universally relevant and could determine whether the OS should run on a different core in the general case. We decided to study the effects of memory and cache performance in particular because of the high latency and bandwidth limitations of off-chip memory and the changing nature of on-chip caches. Additional architectural resources could be modeled in the future, and would likely show even greater *competition* effects.

To this end, we built CachEMU, a full-system memory trace generator and cache simulator. CachEMU is being released as open-source and is built as an extension to QEMU's binary translation layer, enabling it to support multiple guest operating systems and processor architectures.




(a) (b)

**Figure 1: In figure (a) the OS and the Application share a core and caches, where as in figure (b) the OS and the Application run on separate cores but share an L3 cache.**

Figure 1(a) shows the traditional OS and application placement of a monolithic OS. We suggest the alternative placement strategy shown in figure 1(b) — Never run the OS on same core as the application, but always run it on a nearby core. Using CachEMU to evaluate five common Linux workloads on an architectural model of Intel's Nehalem processor, we show that this placement strategy tends to be optimal for OS intensive workloads.

# 2. RESULTS

Using CachEMU, we present a variety of experiments that examine the effects of competition and cooperation in the context of OS and application workload placement strategies.

## 2.1 CachEMU

CachEMU is a memory reference trace generator and cache simulator based on the QEMU [3] processor emulator. Through modifications to QEMU's dynamic binary translator, CachEMU interposes on data and instruction memory access. This is achieved by injecting additional micro-operations at the beginning of each guest instruction and data lookup. This work is the first publication describing CachEMU or using its results. CachEMU's cache model has fully configurable cache size, associativity, block size, and type (instruction, data, or unified). Raw memory tracing, however, could easily be directed to additional purposes in the future, such as write-back buffer modeling or simulating TLB costs. Like SimOS [12], CachEMU's modeling can be enabled or disabled dynamically, allowing system boot-up or any other operations that are not relevant to an experiment to run without added overhead.

Several of QEMU's advantages are preserved in CachEMU, including the ability to perform full-system emulation across multiple processor ISA's with realistic hardware interfaces. Although our study is currently limited to Linux on x86-64, full-system emulation makes CachEMU a powerful tool for studying the effects of OS interference on a variety of potential platforms (e.g. Android mobile phones and Windows desktops.) Simics provides similar full-system and memory trace capabilities to CachEMU but is currently proprietary [10].

CachEMU builds upon the work of past memory reference tracers. For example, ATUM used modifications to processor microcode to record memory traces to a part of main memory [1]. Previous studies have established that OS interference can have a significant effect on memory caches [1, 8].

CachEMU brings the ability to study OS interference to new machines and allows for the study of new kinds of applications. We feel that such an effort is now even more relevant than ever because of the dramatic increase in cores on a chip, each with dedicated cache resources, changes in the scale of applications, and the rise in complexity of cache topologies. We plan to release CachEMU as an open source tool for other OS and architecture researchers to utilize and extend.

## 2.2 Methodology

We used CachEMU to evaluate the effects of competition and cooperation, counting kernel instructions toward the OS and user instructions toward the application. Each application was run in a separate virtual machine (for isolation purposes) with a 64-bit version of Debian Lenny installed and a single virtual CPU. Instruction-based timekeeping, where

each guest instruction is counted as a clock tick, was used with QEMU in order to mask the disparity between host time and virtual time. For cases where the user and the OS were using separate caches, a basic cache coherency protocol simulated shared memory contention by having writes in one cache trigger evictions in the other cache. Evictions were performed only for unshared caches. For example, a memory reference could cause an eviction in a separate L2 cache while leaving the entry present in a shared L3 cache. All cache accesses were modeled with physical addresses and each cache line used a standard 64 byte block size.

We chose five common Linux workloads with heavy usage of OS services. They are as follows:

- **Apache:** The Apache Web Server, running an Apache Bench test over localhost.
- **Find:** The Unix search tool, walking the entire filesystem.
- **Make:** The Unix build tool, compiling the standard library 'fontconfig' (includes gcc invocations and other scripts.)
- **Psearchy:** A parallel search indexer included with Mosbench [6], indexing the entire Linux Kernel source tree.
- **Zip:** The standard compressed archive tool, packing the entire Linux Kernel source tree into a zip archive.

## 2.3 Cache Behavior

In order to gain a better understanding of the effects of capacity on competition and cooperation, we tested a spectrum of single-level 8-way associative cache sizes ranging from 4KB to 16MB. For each test, we compared the number of misses occurring under separate OS and application caches with the number of misses occurring in a shared application and OS cache. In general, we observed that competition was a dominant factor that discouraged sharing for small cache sizes, while cooperation was a dominant factor that encouraged sharing for larger cache sizes.

For example, figure 2 shows the cache behavior of the zip workload when the OS and the application share a cache. For this test, competition effects were dominant until the cache size reached 1 MB. Then from 1 MB to 16 MB the reduction in misses because of cooperation — shared data between the application and the OS — overtook the number of cache misses caused by competition. Although the number of misses avoided as a result of cooperation is relatively small, the performance impact is still great because for larger cache and memory sizes (i.e. where cooperation is a dominant effect) there tends to be much greater access latencies. We also note that the zip workload generally had a higher proportion of misses caused by the OS, a common trend observed in our tests.

Figure 3 includes all five test applications and shows the effect of cache size from a different perspective; We calculated the percentage decrease in total misses caused by having separate caches. This normalizes the cache effects to the baseline miss rate of each cache size. For small cache sizes (usually less than 256 KB,) we saw advantages to having separate OS and application caches because of the reduction in cache competition. For large cache sizes (1 MB and above,) data transfers between the OS and application became a dominant factor, and we saw a net advantage to having a shared OS and application cache. The behavior of caches between 256 KB and 1 MB was application specific and depended on working set size.

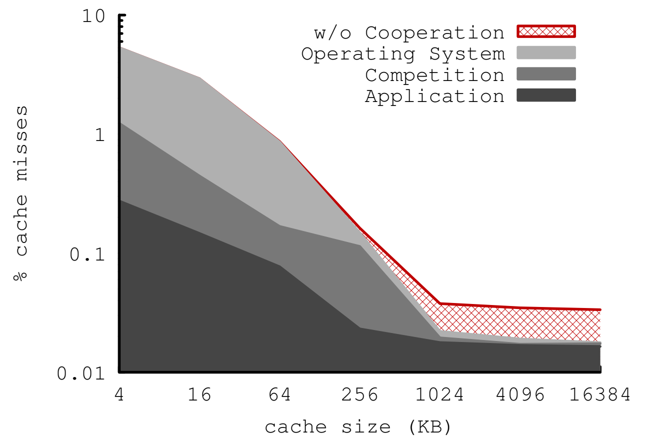



**Figure 2: Cache Miss Rate vs Cache Size for the zip application. Shows shared cache misses attributable to the OS and Application alone as well as the Competition between them. Also shows additional misses that would occur without the Cooperation benefits of a shared cache.**

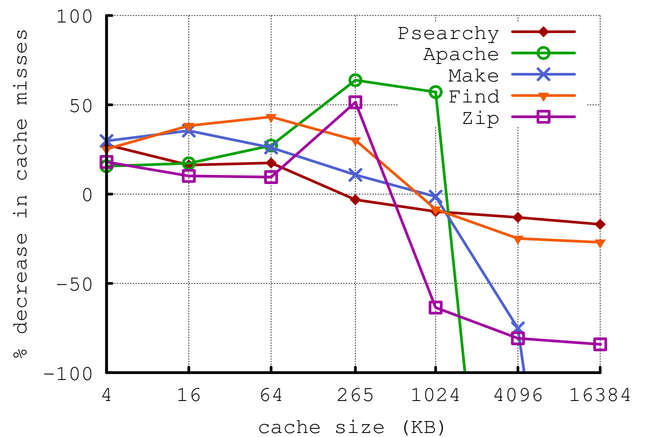



**Figure 3: The percentage decrease in misses caused by splitting the application and OS into separate caches.**

## 2.4 Performance Impact

We studied the performance impact of OS placement on contemporary processors by building a three-level Intel Nehalem cache model. The model includes separate L1 data (8-

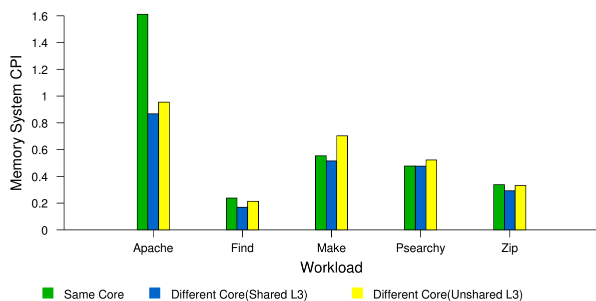



**Figure 4: Memory related Clocks per Instruction (CPI) when executing OS and Application on same core, different cores but same chip, and different cores and different chip. This utilizes a model of the Nehalem cache architecture.**

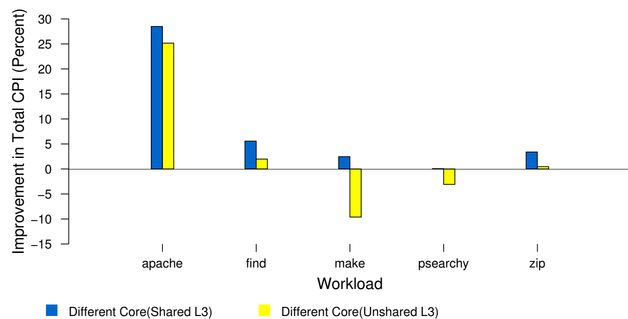



**Figure 5: Overall percentage improvement in the CPI due to memory effects when the OS is run on a different core. The non-memory CPI of each workload is not known, so a conservative value of 1.0 is assumed.**

way associative) and instruction (4-way associative) caches, each with 32 KB capacity. For L2 and L3 we modeled a 256 KB 8-way associative cache and a 8 MB 16-way associative cache respectively. We assume cache latencies of 10 cycles for L2 access, 38 cycles for L3 access, and 191 cycles for local memory access [11]. Using these parameters, we show in Figure 4 the additional cycles-per-instruction (CPI) when compared to a perfect L1 cache hit rate for the following OS placements: all three cache layers shared, separate L1 and L2 but shared L3, and all three cache layers separate. Contention misses caused by shared state were modeled by adding the equivalent latency of the next higher shared cache level. This included using local memory latency when all higher cache levels were unshared. In practice, communication between processors on different dies can be slightly more expensive than local memory accesses [11], but we nonetheless feel that this is a reasonable approximation.

We then used the CPI calculations to estimate overall application speed up. Since actual non-memory CPI is workload dependent, and cannot be estimated by our simulator, we conservatively assume it to be 1.0, the median total CPI for the Pentium Pro [4]. Figure 5 shows projected performance improvements for each of the five workloads. Running the OS on a different core with a shared L3 cache was always better than running the OS on the same core, except for the Psearchy workload where it was equivalent.

## 3. RECOMMENDATIONS

Through utilizing CachEMU to study the effects of OS and application cache interference and cooperation, we have come up with some recommendations for future OS and hardware designs as follows.

RUN YOUR OS ON A DIFFERENT CORE THAN THE APPLICATION

Across all of our benchmarks, we found that when the OS and application utilize the same cache for sizes as would be found in typical L1 caches, the working sets of the OS and application fight and lead to lower performance than if they were executed on separate cores. We also found that for most applications, it was higher performance to not have the OS and application share an L2 cache. In the cases where it was beneficial to share an L2 cache, it was still a performance win to run the OS and application on differing cores because the loss of performance due to sharing the L1 cache outweighed the performance gain of sharing the L2. Therefore, we recommend that future OSes execute the OS on a different core than the application.

One hardware modification which we feel would be beneficial to OSes which execute the OS on a different core than the application is to have faster hardware communication and messaging primitives. We modeled the communication costs between separate cores as the cost of a cache miss as seen though the cache coherence system. We feel that by adding communication hardware which minimizes communication cost between different caches or hardware messaging primitives, system calls can be accelerated and even larger performance gains can be achieved when moving the OS to a different core than where the application is executing.

This result assumes that there are idle (spare) cores to move the OS onto. If there are not spare cores, then the opportunity cost of losing a core to the OS, may outweigh the benefits found in our study. We feel that there is high probability that there will be spare cores on future high-core-count chips. Also, the OS can be aggregated onto a small number of such that the total OS footprint is less than 1:1, OS to application. Last, as Corey [5], Barrelfish [2] and fos [14] have found, through the use of end-to-end tests, dedicating cores to the OS can have both parallelism and working set wins even when the opportunity cost of dedicating cores is taken into account.

RUN YOUR OS ON SAME CHIP AS THE APPLICATION

For all of our benchmarks, we found that it was beneficial to share an L3 cache between the OS and application. There-

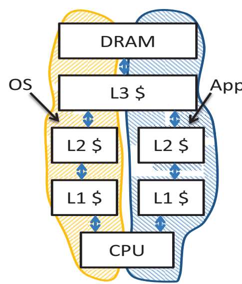



**Figure 6: Processor where the OS and Application share a core, but have different low level caches.**

fore, we recommend that the OS and application execute on the same chip and not across multiple chips which severely hurts communication costs. It makes sense to share the L3 cache because for the applications which we have tested, standard L3 cache sizes for modern microprocessors are able to hold the working set of both the OS and application. Communication between the OS and application is able to happen quicker when on-chip instead of off-chip. If the working sets were so large to cause large amounts of interference in the L3 it may make sense to run the OS and application on differing chips, but this is not what we found.

BUILD DEDICATED LOW-LEVEL CACHES

One idea which our results motivate is modifying chip architecture to have a single core with two private L1 and L2 caches, one for the OS and one for the application. A hardware design such as shown in Figure 6 can save the cost of implementing two complete cores, thereby amortizing the cost of the non-memory portion of a core while still enabling the benefit of segregating the OS and application's working sets.

MAKE CACHES HETEROGENEOUS

There has been much discussion of heterogeneous cores in the multicore computer architecture community. Our results have found that the OS typically has a larger working set size than the application. This suggests that on a heterogeneous multicore, it would be wise to schedule the OS on the cores with larger caches and application on the core with smaller caches. Also, we believe that the OS should compute a real-time estimate of OS and application working set size and schedule components to the appropriately sized caches.

## 4. CONCLUSION

We presented a study on the cache effects of different OS placements for OS intensive workloads. Results were gathered using a new open source tool called CachEMU. We found that *contention* makes a strong case for running the OS on a different core than its application in order to better accommodate each workload's working set. Contrarily, we found that *cooperation* tends to benefit from a shared cache between the OS and the application, allowing for more efficient exchange of data. Thus, a reasonable compromise is to place the OS and the application on separate cores with dedicated caches while still sharing a higher-level on-die cache. This compromise in placement policy was optimal for the five Linux test workloads that we simulated on an Intel Nehalem model, suggesting that *contention* effects can be a sufficient reason to justify running the OS on a different core than the application.

## 5. REFERENCES

[1] A. Agarwal, R. L. Sites, and M. Horowitz. Atum: a new technique for capturing address traces using microcode. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 119–127, 1986.

[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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] D. Bhandarkar and J. Ding. Performance characterization of the pentium pro processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 288–297, Feb. 1997.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[6] 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 *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*.

[7] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 78–89, New York, NY, USA, 1996. ACM.

[8] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 120–133, 1993.

[9] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella,

P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, feb. 2010.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[11] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261 –270, 2009.

[12] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the simos approach. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):34 –43, 1995.

[13] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*. ACM.

[14] 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, 2009.