Message passing and scheduling in Osprey

Jan Sacha, Henning Schild, Jeff Napper, Noah Evans, Sape Mullender
Alcatel-Lucent Bell Labs
Copernicuslaan 50
2018 Antwerp, Belgium
Jan.Sacha@alcatel-lucent.com

ABSTRACT

Future systems for multi-core architectures will have to be highly parallelizable to utilize hardware and will have to avoid sharing to achieve good performance. Consequently, novel operating system designs gradually move towards asynchronous message-based communication and decentralized scheduling. In this paper we describe a message passing and scheduling architecture which provides main communication and synchronization instruments for the Osprey operating system. The architecture supports blocking and non-blocking communication, allows user-level threads, processes, kernel tasks, and cores to be suspended, supports address space transitions, and provides real-time scheduling.

1. INTRODUCTION

It is generally predicted that processor clock speeds will not increase significantly in the mid-term future, and machines will scale up by adding more processors and cores, potentially specialized in performing particular functions. Shared memory will become increasingly expensive because cache-coherency protocols will introduce high overhead, and interconnects will likely become performance bottlenecks.

Future software will have to be highly parallelizable to be able to utilize abundant processors and cores. Sharing will have to be limited to reduce synchronization costs, e.g., stalls due to locking, and to reduce the stress on the memory subsystem. Localized processing, cache affinity, and loosely-coupled asynchronous communication will be key to achieving maximum performance on multi-core, many-core, and potentially heterogeneous hardware. Consequently, many novel operating systems (OS) designs are based on asynchronous message-based communication and decentralized scheduling [1, 2, 6, 9, 10]. Initial experiments show that messaging implementation and inter-core synchronization have a strong impact on OS performance [2–4].

However, designing an efficient and scalable messaging architecture for multi-core hardware poses a number of challenges. The designer needs to consider performance trade-offs such as whether to poll or use inter-core interrupts, whether to use a single queue with a lock or multiple queues with no locks, or whether to block the producer when the queue is full, grow the queue, or drop messages. The optimum configuration often depends on the underlying hardware and application scenario considered. For instance, locking might be appropriate on 16 cores but inefficient on 64 cores. Hence we argue that the operating system abstractions should be flexible, adaptable to hardware, and should allow applications to choose appropriate communication mechanisms to achieve good performance.

In this paper we describe the message passing and scheduling architecture developed for the Osprey [7, 8] operating system that we believe addresses many requirements of future multi-core machines. We consider both message passing and scheduling because, by definition, blocking communication requires scheduling. Our architecture aims to decouple computations performed by different cores to enable parallelism. It reduces sharing between cores, limits the use of locks, and enables asynchrony between system components. At the same time, it is flexible and supports multiple communication and synchronization styles. For example, it allows threads, processes, and entire cores to suspend to save energy, or busy wait to minimize latency. It also allows messages to be processed lazily in batches to maximize throughput, or to be handled immediately to meet real-time constraints. We believe our design provides a consistent and powerful model for building concurrent systems.

2. SYSTEM OVERVIEW

Our architecture is based on a hierarchy of schedulers that both route messages and schedule the receiving entities. An example of such a hierarchy is shown in Figure 1, where every core schedules its own tasks and processes, and a multi-threaded process schedules its own threads. We use the term task for a kernel-level schedulable entity, process for a user-level process, and thread for a cooperative thread (coroutine) running inside a process.

A process is represented in the kernel as a task. A process is executed by first scheduling its task and then switching to user mode. Similarly, when a process is preempted, i.e., receives an interrupt, causes an exception, or traps, it runs as a kernel task and is subject to task scheduling. Every process has one kernel stack belonging to its task and one

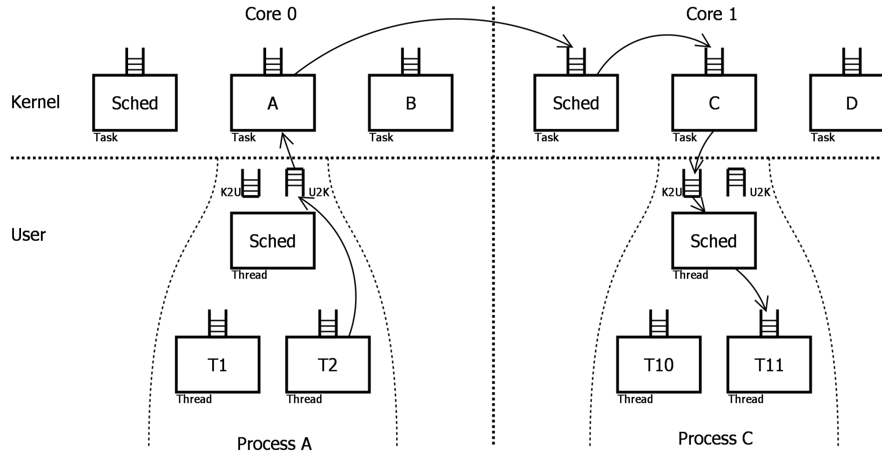

Figure 1: Sample system with two cores, each running two tasks and a process consisting of three threads. The arrows show the message route from thread T2 to thread T11.

or more user stacks belonging to its threads.

A regular process has its own address space. The top part of this address space is shared with all other processes and is used by the kernel. This global kernel space is consistent across all cores and is used by all tasks. However, we partition most kernel state at data structure level, as in the multikernel model [2], to avoid cross-core synchronization.

Processes may share parts or entire address spaces. Memory sharing processes can run in parallel and may be considered as heavy-weight threads. However, from the message passing and scheduling viewpoint they are treated as separate processes. We reserve the term thread for coroutines that run inside a process, as in Plan 9 [5]. Our threads can never run in parallel, but allow concurrency and can asynchronously handle messages. Threads have their own stacks and schedule each other cooperatively.

Every scheduling entity, i.e., thread or task, has exactly one message queue on which it can block waiting for incoming messages. Additionally, a process has a special pair of queues, one for sending and one for receiving, that it uses to communicate with the kernel. Apart from the blocking receive, message queues also support non-blocking receive and non-blocking send.

Every message has a destination address consisting of a task id and thread id. Since a task is always assigned to a core, message address determines the destination core. Messages are routed hierarchically by the schedulers. A message is first delivered to the destination core, then the destination task, and finally the destination thread, if applicable. As the schedulers relay messages, they make the receiving entities runnable.

Processes and tasks can migrate between cores, for example to balance load. If a message is routed to a process during migration, it may be forwarded by the scheduler on the old core to the scheduler on the new core. Once migration completes, messages are routed directly.

Figure 1 shows a sample scenario where thread T2 in process A on core 0 sends a message to thread T11 in process C on core 1. The message is first appended to the user-to-kernel (U2K) queue so that it can be transferred to the kernel space. The process may continue executing, but eventually it traps and runs in the kernel mode. The kernel then picks up the message from the U2K queue and forwards it to the scheduler's queue on core 1. The scheduler, potentially triggered by an inter-core interrupt, receives the message, appends it to task C's queue, and readies task C if needed. Task C eventually executes and checks for incoming messages. It inserts the message into the K2U queue and switches to user mode. Finally, the thread scheduler of process C receives the message from the K2U queue, relays it to thread T11's queue, and makes thread T11 runnable.

In the described scenario, the message is relayed through three message queues before it reaches its destination queue. Inevitably, this procedure adds overhead. However, two important points have to be made here. First of all, the described scenario is a worst case where the message latency is the highest. In many cases the message path is much shorter. For example, threads running inside the same process, or tasks running on the same core, can message each other directly. Secondly, as we elaborate later, our architecture allows fast-path queues between any entities, where messages are exchanged directly through shared memory, and wakeups, which by definition require scheduling, are routed through the schedulers using the algorithm described above.

2.1 Message queues

We have implemented message queues using shared memory buffers containing cache-aligned message slots, currently cache line sized, where messages are copied in both send and receive operations. A message buffer consists of a list of pages that can be allocated and added on demand, for example using *malloc*, and thus a queue can never be full.

All message queues have only one consumer and in most cases multiple producers. Producer and consumer synchronize by checking the current message slot in the queue. Mul-

tiple producers synchronize either by disabling interrupts or using a lock. The former method is cheaper, as it merely requires setting a core-local register, but can only be used if the producers run on the same core.

2.2 User-level cooperative threading

Every multi-threaded process has a thread scheduler, which is a thread itself. It receives messages from the kernel, dispatches the messages to other threads, and schedules threads. Threads yield to the scheduler explicitly using the *yield* function, which is normally invoked when a thread attempts to receive from an empty queue. A thread is made runnable again when it receives a message.

Cooperative threads do not allow parallelism but are a convenient programming instrument to handle non-determinism in input-output. They allow a process to issue multiple concurrent requests to the kernel and at the same time use traditional synchronous (blocking) semantics in each thread. Since a cooperative thread can never be preempted by another thread running in the same process, critical code sections can be executed without locking. Finally, threads are cheaper than processes because they can be created and scheduled entirely in user space.

2.3 User-kernel transition

Processes communicate with the kernel mainly using messages. Similar to FlexSC [9], our architecture requires only two exception-based system calls: *sleep* to wait for incoming messages and *flush* to process outgoing messages. The *flush* system call is mainly intended for real-time processes.

Every process has two memory segments containing special message queues: user-to-kernel (U2K) and kernel-to-user (K2U). A thread sends a message to the kernel by simply writing it to the U2K queue. It may generate a *flush* system call to process the message immediately or continue executing. The thread scheduler receives a message from the kernel by reading it from the K2U queue. If this queue is empty and there are no runnable threads, the scheduler executes a *sleep* system call.

Once the process switches to the kernel mode (on a system call, interrupt, or exception) and becomes a kernel task, it processes the U2K messages, potentially forwarding them to other kernel tasks or generating response messages and putting them on the K2U queue. It also receives and processes messages from its task queue delivered by other kernel tasks. Finally, if the process is waiting for a message (signaled by the *sleep* system call) and the K2U queue is empty, the task performs a blocking receive on its task queue to yield to the kernel scheduler and suspend the task (and therefore the process).

Pages are added to the U2K and K2U queues on demand (i.e., on page fault). If a process uses all pages in the U2K segment, it invokes a *flush* system call. If the kernel uses all pages in the K2U segment, it assumes the corresponding process is unresponsive and terminates it.

Asynchronous message-based communication, in contrast to exception based system calls (traps), allows batching kernel requests and replies, saving context switches and thus overall performance. Such an asynchronous communication model is particularly convenient in combination with lightweight user-level threading which transforms event-driven communication into a traditional synchronous execution model.

2.4 Kernel message routing

Kernel tasks can deliver messages directly into each other's receive queues if they run on the same core. Producers can synchronize cheaply by temporarily disabling interrupts.

If a message needs to be delivered to a task running on a remote core, the message is first sent to the scheduler on that core. The task scheduler, similar to the thread scheduler, is a task itself. It receives all messages coming from other cores and dispatches them to local tasks making them ready if necessary. If the scheduler has no runnable tasks and its receive queue is empty, it puts the core into a low-power state, setting an *idle* flag in a globally visible data structure. When a task delivers a message to another core and its *idle* flag is set, it uses an inter-core interrupt to wake up the remote scheduler. We show further a synchronization algorithm that guarantees that wakeups are never missed.

On some hardware architectures, a core can also be idled by executing a memory monitor instruction, such as *monitor* and *mwait* on x86. The core executing such an instruction is woken up when a chosen memory region is written. This mechanism can be used as an alternative to interrupts, since an idling core can monitor its own receive queue, waking up each time a message arrives.

Scheduler message queues are accessed by multiple producers that potentially run on different cores and hence require synchronization, for example using locks. An alternative to locks is to replace N scheduler queues, where N is the total number of cores, with a matrix of $N \times N$ message queues, so that each core has a private message queue to any other core in the system. Each such queue is accessed by producers running on the same core and thus synchronization can be done cheaply by disabling interrupts. Further, if N is large, the $N \times N$ matrix can be avoided by adding an extra level to the core hierarchy and routing messages over a spanning tree.

3. DIRECT QUEUES

The architecture described so far introduces potentially high overhead because messages are relayed by multiple schedulers before they reach their destination. Our model allows threads, processes, tasks, and cores to be suspended. However, many applications perform non-blocking communication or do not block in the common case and thus do not need the overhead introduced by the schedulers.

To address such applications, we allow threads, processes, and tasks to establish direct, fast-path message queues in shared memory. Such queues enable very efficient message exchange in non-blocking mode, e.g., by polling, without any scheduling overhead. Further, they implement blocking operations using wakeup messages routed through the schedulers. Importantly, wakeup messages are generated only if the producer or consumer actually block, i.e., when the consumer attempts to read from an empty queue or the producer writes to a full queue.

```
void msgget(Msgq *q, Msg *msg)
{
  Msg wakeup;

  while(!qget(q, msg)){
    q->wait = mythread();
    if(qget(q, msg)){
      /* race detected */
      q->wait = nil;
      return;
    }
    /* blocking receive */
    msgwait(&wakeup);
    q->wait = nil;
  }
}

void msgput(Msgq *q, Msg *msg)
{
  Msg wakeup;

  /* non-blocking */
  qput(q, msg);
  if (q->wait != nil) {
    wakeup.dst = q->wait;
    q->wait = nil;
    msgsend(&wakeup);
  }
}
```

Figure 2: Send and receive operations on a queue with a single blocking consumer and multiple non-blocking producers.

The (pseudo) C-code for the blocking receive operation is shown in Figure 2. The consumer attempts to read a message from the queue using the *qget* function, and if the queue is empty, it sets the *wait* flag stored in the queue's shared memory to notify the producer that it is going to sleep. The consumer needs to check if the queue is empty twice, before and after setting the flag, to avoid a race condition.

The consumer blocks by calling the *msgwait* function on its main thread queue. It is woken up when a wakeup message arrives. It unsets the *wait* flag and checks the queue again, expecting a message. The consumer has to check the queue and sleep potentially multiple times, hence the *while* loop in pseudo-code, because the producer may generate spurious wakeups when racing with the consumer.

In the presented pseudo-code the producer never blocks. The *qput* function, which inserts a message into a queue, is assumed to add buffer space to the queue when necessary. The producer first appends the message to the queue and then generates a wakeup message if the consumer has set its *wait* flag. The producer unsets the flag to avoid sending redundant wakeups to a sleeping consumer.

The pseudo-code can be easily extended to allow timeouts in *qget*. The consumer simply needs to request a timer message before calling *msgwait*. The design can be further elaborated to allow waiting on multiple queues as in the *select* UNIX system call.

Interestingly, fast-path queues can be used by processes to send requests to the kernel and receive replies. Such a setup would allow a user process to entirely avoid switching to the kernel mode, provided a kernel task can receive and handle

```
void msgsend(Msg *msg)
{
  Task *task = gettask(msg->dst.task);
  if(task->core != mycore){
    /* msg to remote core */
    qput(task->core->sched->q, msg);
    ready(task);
  }else if(msg->dst.task != mytask){
    /* msg to local core, other task */
    qput(task->q, msg);
    ready(task);
  }else if(msg->dst.thread != nil){
    /* msg to a user thread */
    qput(task->k2u, msg);
  }else{
    /* msg to the task itself */
    qput(task->q, msg);
  }
}
```

Figure 3: Message routing function used in kernel.

user requests on a remote core. Multiple policies are possible in such a design. The kernel task could constantly poll (busy-wait) to provide minimum response time. To amortize costs and reduce wasted cycles, the kernel task could poll on multiple queues from different processes. Alternatively, the kernel task could block, for example after a polling timeout, requiring the process to issue a wakeup message. Similarly, the process would be allowed to spin busy-waiting for a kernel response or block.

4. REAL-TIME SCHEDULING

Real-time computing relies on process scheduling with an awareness of the deadlines processes have to meet. Those deadlines might be recurring, in which case a real-time process must be released periodically. A sporadic real-time process is released in response to a non-deterministic event such as a message arrival. In both cases the real-time process must perform its computation within a certain time.

In Osprey, the scheduler on each core maintains its own run queue containing runnable processes sorted by their deadline. For best-effort processes, the deadline is defined as *infinity*, a special value that is always greater than any real deadline. Best effort processes are thus always behind real-time processes in the run queue. Processes are selected for execution according to the Earliest Deadline First (EDF) policy, i.e., from the head of the run queue.

Every scheduler exports two synchronization variables that can be read by tasks and processes running on other cores. Each of these two variables uses one word in shared memory that can be read and written atomically. Most existing computer architecture guarantee some form of atomic access to memory words. The first variable, *state*, indicates one of five states the scheduler may be in, shown in Table 4. The second synchronization variable, *deadline*, indicates the deadline of the currently running task or process. It is only relevant if the *state* variable is set to *Deadline*, but, for convenience, it is set to *Infinity* if the state is *Slice*.

A task is made runnable when it receives a message or when it is released by a timer interrupt (periodic real-time task). The message delivery function, *msgsend*, is shown in Figure 3. Depending on the destination, the message is either

State	Description	Timer
Scheduling	Making scheduler decisions	Off
Idle	The core is waiting for work	Off
Only	The only runnable best-effort process is running	Off
Slice	One of several runnable best-effort processes is running	On
Deadline	A real-time process is running	On

Table 1: Core states.

```
void ready(Task *task)
{
  if(task->state == Ready)
    return;  /* task is already running */
  if(task->core == mycore)
    enqueue(runqueue, task);
  switch(task->core->state){
  case Scheduling:  /* scheduler is running */
    break;
  case Idle:  /* wake scheduler */
  case Only:  /* wake scheduler */
    wakecore(task->core);
    break;
  case Slice:
  case Deadline:
    /* wake scheduler if task is urgent */
    if(task->deadline < task->core->deadline)
      wakecore(task->core);
    break;
  }
}

void wakecore(Core *core)
{
  if(core == mycore)
    schedule();  /* call scheduler directly */
  else
    intercoreintr(core);  /* use interrupt */
}
```

Figure 4: Ready function is called when task receives a message, potentially from remote core.

forwarded to the scheduler on a remote core, using a private core-to-core queue that does not require locking, or inserted directly to the receiving task's queue on the local core, or forwarded to a user-level thread using the K2U queue. After enqueueing the message, the sender calls the *ready* function that makes the receiving task runnable and decides whether the scheduler must be invoked.

The *ready* function, shown in Figure 4, invokes the scheduler if the core is idle or the scheduler has disabled the clock interrupt and thus is not going to run in the foreseeable future. The scheduler is also invoked if the running task has a greater deadline (lower priority) than the task that becomes runnable. That latter condition is necessary to ensure an EDF scheduling order. The scheduler is invoked either by calling the *schedule* function or by issuing an inter-core interrupt, depending on the location of the task that becomes ready. The inter-core interrupt handler calls *schedule*.

Figure 5 shows the code executed by the scheduler. It is assumed that the scheduler disables interrupts and cannot be preempted. The scheduler first sets its state to *Scheduling* to indicate that it is going to process pending messages and thus no inter-core interrupts are needed. If the current task is still runnable, it is put back on the run queue so that it

```
void schedule(void)
{
  Task *task = currenttask;
  do{
    state = Scheduling;
    if(task != nil && isrunnable(task))
      enqueue(runqueue, task);  /* run later */
    schedmsgs();  /* process messages */
    task = dequeue(runqueue);  /* next task */
    if(task == nil)
      state = Idle;
    else{
      deadline = task->deadline;
      if(deadline != Infinity)
        state = Deadline;  /* real-time task */
      else if(notempty(runqueue))
        state = Slice;  /* time-sharing */
      else
        state = Only;  /* single task */
    }
  /* must check messages again! */
  }while(schedmsgs());
  switch(state){
  case Idle:
    idle();  /* power-saving mode */
    break;
  case Slice:
  case Deadline:
    settimer();  /* program timer interrupt */
  case Only:
    runtask(task);  /* does not return */
  }
}

int schedmsgs(void)
{
  Msg msg;

  if(empty(mycore->sched->q))
    return 0;
  while(qget(mycore->sched->q, &msg))
    if(msg.dst.task == mytask)
      /* process msg */
    else{
      /* forward to task */
      Task *task = findtask(msg.dst.task);
      qput(task->q, &msg);
      enqueue(runqueue, task);
    }
  return 1;
}
```

Figure 5: Algorithm executed by kernel schedulers.

can execute again later. The scheduler then processes pending messages, forwards them to local tasks, and puts the receiving tasks on its run queue. It then makes a tentative scheduling decision, and set the state and deadline variables. Importantly, the scheduler must check the state of its message queue again because messages might have arrived while the scheduler was making its decision. In the unlikely case where the scheduler discovers a pending message, it must remake its decision by running another loop iteration. If there are no runnable tasks, the scheduler switches the core to a power-saving mode and waits for an interrupt. Otherwise, it programs a timer interrupt according to the *state* and *deadline* variables and runs the selected task.

If there is only one runnable task, the scheduler disables its timer interrupt and lets the task execute continuously and uninterruptedly until an inter-core interrupt arrives. Such uninterrupted execution is particularly important in high-performance computing applications where parallel processes frequently synchronize and the overall computation time is determined by the slowest process.

5. RELATED WORK

Barrelfish [2] introduced the concept of a multikernel where every core in a machine runs its own kernel. State is partitioned and replicated instead of being shared and communication is based on asynchronous messages. We apply the same principles to Osprey.

Message queues in the current Barrelfish implementation are based on shared-memory buffers. The consumer polls on an empty queue for a fixed maximum duration, after which it blocks waiting for a wakeup. Our messaging architecture supports both polling and sleep-wakeup synchronization, and additionally addresses user-kernel transitions and real-time message delivery.

Recent measurements on a large multi-core machine [4] have revealed that immediate inter-core notifications allow in many cases better performance than polling. This result supports our claim that the OS should not enforce any particular message passing implementation but should rather adapt to underlying hardware and should allow applications to choose models that suit their needs.

FlexSC [9] is a modified Linux system where synchronous, exception-based system calls are replaced with asynchronous, message-like communication. Measurements show that asynchronous communication reduces the number of context switches, significantly improving cache efficiency and hence overall system performance. FlexSC also provides a user-level threading library which exposes a traditional, blocking system call interface to threads on top of the asynchronous kernel API and ABI.

Akaros [6] also uses asynchronous system-calls and cooperative threads. Threads are implemented as coroutines running inside a user process, invisible to the kernel. Threads can block, for example waiting for external events, and be scheduled without trapping to the kernel. Further, Akaros feeds page faults back to the process, which allows the process to continue running while some of its threads are blocked on page faults.

Similarly to FlexSC and Akaros, Osprey is based on asynchronous system calls and lightweight user-level threads. However, unlike Akaros, where threads can preempt each other on page faults, Osprey threads are fully cooperative which allows them to enter critical code sections without locking. Further, in contrast to FlexSC and Akaros, Osprey addresses kernel and inter-core communication and provides a consistent messaging model for threads, processes, tasks, and cores.

NIX [1] introduces a communication model where an application can run exclusively and uninterruptedly on a core sending requests through a shared-memory message queue to a kernel process running on a different core. Such an arrangement allows applications to fully utilize cores and entirely eliminate OS noise, which is desirable in high-performance computing. The same goal can be achieved in Osprey using our message passing and scheduling model.

6. STATUS AND CONCLUSIONS

In this paper we describe the message passing and scheduling architecture for the Osprey operating system. It aims to enable asynchrony between system components to maximize parallelism. It is based on hierarchical scheduling, supports blocking and non-blocking communication, and provides real-time scheduling. We find our model easy to use and we believe it is efficient and scalable. We have implemented a working prototype for x86 32-bit and 64-bit machines which contains approximately 50K lines of code. We are planning to evaluate it in a range of application scenarios.

7. REFERENCES

[1] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano. Nix: An operating system for high performance manycore computing. *Bell Labs Technical Journal*, 2012.

[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*, pages 29–44, 2009.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI*, pages 43–57, 2008.

[4] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe. Early experience with the barrelfish os and the single-chip cloud computer. In *MARC Symposium*, pages 35–39, 2011.

[5] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from bell labs. *Computing Systems*, 8(2):221–254, 1995.

[6] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving Per-Node Efficiency in the Datacenter with new OS Abstractions. In *SCC*, pages 25:1–25:8, 2011.

[7] J. Sacha and S. Mullender. Networking in osprey. In *International Workshop on Plan 9*, pages 14–21, 2012.

[8] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *DSN Workshops*, pages 1–6. IEEE, 2012.

[9] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8, 2010.

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