Draft: Have you checked your IPC performance lately?

Steven Smith[†], Anil Madhavapeddy[†], Christopher Smowton[†], Malte Schwarzkopf[†]
Richard Mortier[‡], Steven Hand[†], Robert M. Watson[†]
[†]*University of Cambridge* [‡]*University of Nottingham*

Abstract

IPC mechanisms in UNIX systems were developed for a range of different purposes, but their performance is increasingly unpredictable due to diversification in multicore hardware layouts and the popularity of OS virtualisation. Perhaps surprisingly, no standard benchmarking suite exists to compare the performance of different IPC mechanisms on a machine. To address this need, we developed `ipc-bench`, and discovered significant and sometimes counter-intuitive differences when running on many-core and virtualised hardware.

In this paper, we characterise a subset of these results on hardware available to us. However, we also make a much larger data set available, and provide means to enable a wider community to gather further data on a larger scale. `ipc-bench` integrates with revision control to make it easy to record and publish performance results, aggregate them and track their provenance using a popular code sharing website. We hope that making this tool available will result in a corpus of useful open data to guide the development of hypervisors, kernels and programming frameworks.

1 Introduction

Inter-process communication (IPC) facilities have existed in operating systems for many decades. Ever since the first parallel applications ran on time-sharing machines, operating system kernels and processes have had a need to efficiently communicate. The first networked applications introduced the concept of sending a message to a remote process with the outcome of triggering a remote action. Today, these primitives are more relevant than ever before: the advent of multi-core and many-core hardware means that explicit message-passing between concurrent processes running on different cores is a common operation. Future operating systems may be heavily based on IPC [1, 8], and as data centres are increasingly composed of many-core machines, high-performance data processing is increasingly challenged to bridge the gap between traditional multithreaded SMP programming and distributed parallel processing.

One would expect that, with this prominent a rôle in contemporary and future software, tools to evaluate and compare the performance of different IPC mechanisms would be abundant. To our surprise, we found no standard IPC benchmarking suite for comparing mechanisms on the same or different hardware and OS platforms.

In this short paper, we describe such a benchmark that we developed: `ipc-bench` (§2). We also present a selection of results which highlight dramatic differences in IPC performance across communication mechanisms (depending on locality and machine architecture), and observe that the interactions of communication primitives are often complex and sometimes counter-intuitive (§3). Furthermore, we show that virtualisation can cause unexpected effects due to OS ignorance of the underlying, hypervisor-level hardware setup.

Faced with this evidence and assuming future software is to rely on IPC as much as is currently expected, tuning software to the particular characteristics of the platform and application at hand becomes a necessity. To enable such tuning, we will need access to a comprehensive corpus of benchmark data, such as that produced by `ipc-bench`. We hence make our set of experimental results freely available in an online database, along with the benchmark suite (§4). We also discuss how this database can be extended by the community via running `ipc-bench` and making the results available in a persistent, traceable and easily aggregatable way.

2 The `ipc-bench` suite

UNIX provides a number of mechanisms for inter-process communications, each with their own benefits and tradeoffs. We have included a set of those which are suitable for low-level, high-performance communication

Type	*Benchmark*	*# copies safe*	*# copies unsafe*
shared memory	`mempipe-spin`	0	1
	`mempipe-futex`	0	1
	`shmem-pipe`	0	1
vmsplice	`vmsplice-coop`	–	1
POSIX	`pipe`	2	–
	`unix-sock`	2	–
	`tcp-sock`	2	–
	`tcp-sock-nd`	2	–

Table 1: Micro-benchmarks in `ipc-bench` with the number of data copies performed.

in `ipc-bench`. Our focus is on local, same-chassis, communication mechanisms (such as those based on shared memory), but we also include some, such as TCP, which are also capable of communicating with remote processes and are often used locally too.

We now describe the low-level transport mechanisms included as micro-benchmarks (§2.1), explain the higher-level macro-benchmarks (§2.2), and then discuss our findings from running the suite on a set of machines—a 4–15× difference in throughput between the best and worst performing mechanisms on a standard multicore machine, as well as unexpected performance artefacts.

2.1 Micro-benchmarks

There is a wide design space of communication mechanisms, and Table 1 summarises those we include in `ipc-bench`. As simple, high performance and potentially zero-copy transports, we include a set of custom shared-memory communication implementations (§2.1.2) and the Linux `vmsplice` mechanism (§2.1.3). Finally, we include benchmarks based on UNIX pipes and the conventional socket interface (§2.1.1).

2.1.1 POSIX-based (`pipe`, `unix`, `tcp`)

The POSIX sockets API is widely implemented in most multi-process operating systems, and specifies at least TCP streams, domain sockets and pipes.

TCP sockets are used by applications that potentially communicate outside the chassis; but can of course also be used to communicate with local processes too. This makes it transparent to the application if the endpoint is a local or remote process, simplifying the deployment model. It is, however, somewhat inefficient: the network stack continues to run its packetisation, congestion control and retransmission algorithms, even though these are redundant on a loopback connection.

Domain sockets are far simpler than TCP sockets, with the absence of packetisation, retransmission, or congestion control, while still preserving a similar socket interface. Nonetheless, they cannot take full advantage of the potential for sharing memory between local processes, as the socket interface mandates at least one data copy.

Pipes present a similar interface to domain sockets, but are uni-directional[1] and stream-based only, and are internally implemented in a completely different manner.

2.1.2 Shared memory (`mempipe`, `shmem-pipe`)

Shared memory offers a high-bandwidth, low-latency alternative to sockets when both end-points are processes on the same kernel. Although POSIX specifies how to establish shared memory segments, there is no standard for shared memory *communications*, and so bespoke implementations are usually used. We now describe two variants we developed for `ipc-bench`.

The `mempipe` transport uses a ring buffer split up into variable-length messages, each with a header and a message body. Pointers to messages are held in private memory at the producer and consumer.

Each message has a cache-line-sized header indicating its freshness and payload length. To receive a message, the receiver waits for the freshness flag to be set in the next header. When the flag is set, it reads the message size header and consumes the payload directly from the ring. The message is then consumed, the freshness flag is cleared and the consumer pointer advances. Once finished with the message, it clears the flag (acknowledging the message) and advances the consumer pointer.

The free space is precisely that between the producer and next acknowledgement pointers, and so the transmitter might need to advance the next acknowledgement pointer to ensure that enough space is available. The transmitter waits for the unconsumed-message flag to be cleared in the relevant message header, and then advances the pointer by the size of the acknowledged message. Once the pointer is advanced far enough, the transmitter can send its message.

To transmit data, the application populates the payload area with data. The transmitter then finds the header immediately *after* the payload and clears the unconsumed-message flag there, before moving back to the header *before* the payload and setting the size and unconsumed-message flag there. This two-step process ensures the receiver does not advance ahead of the transmitter.

Constructing a good shared memory transport requires making a number of decisions about the end-points. For instance, while there is a logical transfer of ownership from the producer to the consumer in `mempipe`, there is no attempt to enforce it. We can thus measure the cost

[1] Some operating systems do provide bi-directional pipes, but POSIX.1-2001 only requires them to be uni-directional.

of enforcement (copying data into private buffers) separately as a "safe" variant—unlike with the socket API, which operates on the basis that both sides are untrusted.

We consider two implementations of the `mempipe` transport, differentiated by the way they wait for flags in the message headers to change (indicating whether a message is valid and ready for consumption): in `mempipe-spin`, the process spins and tests the flag repeatedly, while in `mempipe-futex`, the Linux `futex` system call is used to set up a kernel notification when the flag memory has potentially changed.

The `mempipe-spin` approach requires neither expensive system calls nor atomic operations; it does, however, keep the CPU spinning and potentially hurts performance of co-located processes. The `mempipe-futex` approach does not spin, but is less portable and requires more expensive atomic operations. The choice of approaches is further complicated by virtualisation, where spinning may have external effects on another co-scheduled VM, despite the process believing it has a CPU to itself. Similarly, the system call cost is higher in the virtualized case due to extra privilege checks.

The `mempipe` mechanism has two important weaknesses: it cannot be integrated with existing `poll`- or `select`-based event loops, and messages must be processed entirely in order. These restrictions are lifted with `shmem-pipe`, which, like `mempipe`, is zero-copy, and communicates payload data via a shared memory area, but communicates message metadata out-of-band using a pair of ordinary POSIX pipes. Rather than managing the shared memory area as a contiguous ring, it is managed using a heap allocator operating in the producer.

Its main disadvantage is that it requires more system calls, in order to move the descriptors back and forth through the pipes. Fortunately, these operations can often be batched and sometimes exceed the performance of `mempipe-futex`. This scheme is a userspace equivalent to the Xen virtual device model, where guest kernels transfer shared memory pages using a coordination page for the metadata [7].

2.1.3 Other (`vmsplice-coop`)

Linux has a `vmsplice` system call for single-copy IPC, which moves pages into a remote pipe queue (unlike `write`, which *copies* data into freshly allocated pages in the queue). `vmsplice` makes it impossible to determine when the page has been `read` and can be re-used. The `vmsplice-coop` transport augments the data pipe with a consumer metadata pipe to tell the sender when a page is released. Data can thus be copied from the producers's buffers directly into the consumers's buffers. As with `mempipe`, this relies on endpoints trusting each other, and requires a data copy otherwise.

Type	*Benchmark*
memory contention	`mem-contend`
threaded Phoenix++ MapReduce	`thrd-wordcount`
	`thrd-kmeans`
communicating Phoenix++ MapReduce	`comm-wordcount`
	`comm-kmeans`

Table 2: Macro-benchmarks in `ipc-bench`.

2.2 Macro-benchmarks

The earlier IPC mechanisms can be easily evaluated via micro-benchmarks, but are not representative of actual application performance (especially in the presence of memory contention). Hence, we also include a set of macro-benchmarks in `ipc-bench` (see Table 2). The `mem-contend` benchmark creates a contention situation on the memory subsystem by continously sending IPC messages between $N/2$ pairs of cores for a machine with a total of N cores. We consider the cases of NUMA-optimal core pairs, and pessimal pairs which cross NUMA node boundaries.

The remaining macro-benchmarks are based on the Phoenix++ MapReduce system [6], which is a shared-memory implementation of MapReduce. Phoenix++ mappers store intermediate data in a memory grid, and once the map phase has completed in all threads, the reducer code reads directly from this grid without copying. While efficient, this approach only works if an application can be implemented using threads that share an address space, which may not always be possible.

To quantify the cost of using explicit IPC, we compare two approaches: in the `thrd`- cases, we run the unmodified, multi-threaded version of Phoenix, while the `comm`-benchmarks use a modified version which uses sockets for the data shuffle between map and reduce phase.

We primarily tried the common `wordcount` and `kmeans` workloads, the former of which counts the number of occurences of terms in a data set, while the latter performs k-means clustering on a large set of high-dimensional vectors. We found it difficult to isolate significant performance differences in the threaded Phoenix++ macro-benchmarks, since they are affected by many other factors—the OS page cache already holding the data set, or the performance being dominated by the cost of faulting data into memory.

3 Selected benchmark results

We have run `ipc-bench` on a range of different machines and OSes (Linux, FreeBSD, MacOS X), in both native mode and under Xen. We now present a selection of interesting results from these runs.

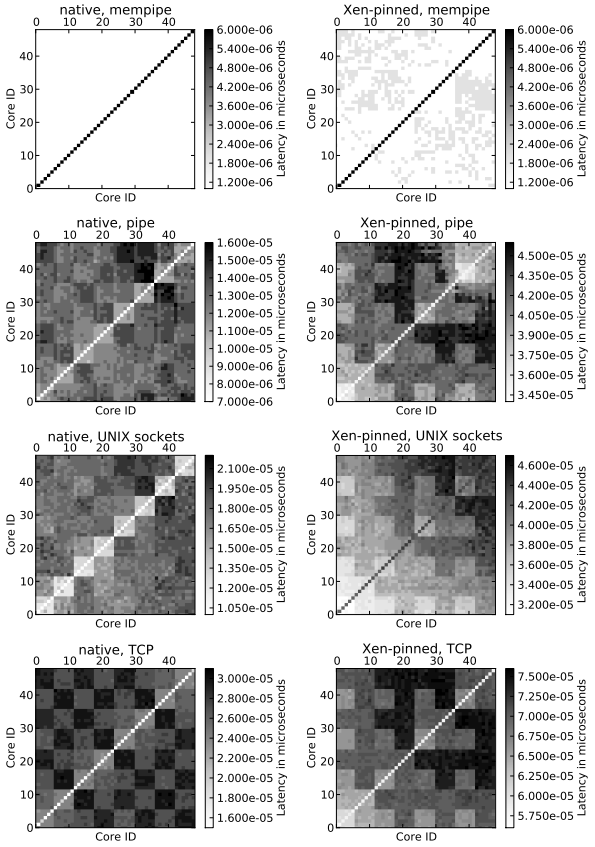

Figure 1: Ping-pong latency in different configurations, using the AMD 48-core system. Xen tests are run in multi-vCPU Linux domains running on Xen, while native tests are run in Linux running natively. In Xen, we manually pinned virtual CPUs to physical CPUs.

The tests were compiled using `gcc` 4.6.1 with optimisations enabled (`-O3`). All benchmarks were repeated at least 10,000 times, and participating cores were pinned to a particular set of CPUs.

3.1 Impact of NUMA layouts

A key observation from running `ipc-bench`'s micro-benchmarks on various SMP/NUMA machines is that the NUMA layout has a major influence on IPC latency and throughput. Figure 1 summarises some `ipc-bench` results running on Linux 3.1.0 x86_64 on a 48-core AMD Opteron 6168 ("Magny-Cours"). Each point on each grid represents a separate test run, and the axes are the CPU IDs the test was pinned to. The left column represents test run on a native kernel, and the right column are the same tests with a 48-vCPU dom0 under Xen 4.1 (§3.2).

While the `mempipe-spin` transport uniformly achieves the lowest latency (Figure 1a), other transports vary considerably based on the NUMA distance of the

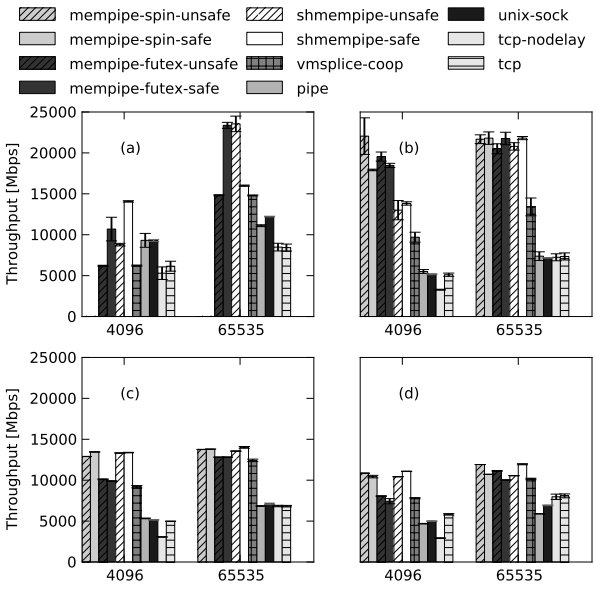

Figure 2: Throughput comparison for different mechanisms on a non-virtualized 48-core Opteron 6168. (a) *same core,* (b) *same socket, same die/MCM,* (c) *same socket, other MCM,* (d) *different socket, other MCM.*

CPUs involved[2]. Communication with different cores on the same socket always achieves best performance, as indicated by the squares along the diagonal. In the `tcp` case, we can also clearly distinguish sockets connected via 2-hop HyperTransport (HT) links and those connected by 1-hop HT links [2]. Overall, the variation in IPC latency introduced by NUMA is up to 2×, while the difference between best (`mempipe`, `pipe`) and worst (`tcp`) transport latency is 5-10×.

IPC latency is important to low-level OS services and some high-performance applications, but in many applications, throughput is the more relevant metric to consider. Figure 2 illustrates throughput differences between IPC mechanisms when on-core, on-socket and on separate multi-chip modules. It becomes evident that IPC throughput is reduced significantly when going off-die (cases *(c)* and *(d)*), dropping from around 20 Gbps to under 15 Gbps with the shared-memory transports. It also becomes clear shared-memory transports outperform POSIX-based ones by 50-300%, depending on the relative locations of the communicating cores, while the `vmsplice` transport is situated in between.

Of course, the results of these micro-benchmarks are only meaningful for a case of only two cores communicating concurrently—which, in a 48-core machine, is not likely to always be the case. The `mem-contend` bench-

[2]Our measurement accuracy was limited to 10^{-6} seconds, so we could not detect any NUMA impact in the native `mempipe` test.

NUMA placement	*Avg. throughput*	*Std. deviation*
close / optimal	4171.8 Mbps	123.4 Mbps
far / pessimal	3212.7 Mbps	208.6 Mbps

Table 3: `mem-contend` using the `pipe` transport on the 48-core Opteron; average and std. dev. over 24 pairs.

mark measures throughput under contention: in the case of the Opteron, 24 pairs of cores are communicating concurrently. In the "optimal" NUMA placement, the cores always communicate with their nearest neighbour (as far as possible), while in the "pessimal" setting, they communicate with a core that is furthest away from them in NUMA terms. As Table 3 shows, the overall throughput degrades to about 4 Gbps when memory controllers are contended, but NUMA-pessimal access leads to another drop of about 900 Mbps.

3.2 Virtualization confuses IPC

We also investigated the IPC performance on the Opteron machine when running Linux 3.1.0 as a PV-dom0 on Xen 4.1. Virtualization makes system calls and some memory accesses much more expensive, and we hence expected overall performance to degrade. The latency results in Figure 1 (*(e)–(h)*) surprised us, as we expected the overall pattern to remain consistent between native and virtual kernels. A few factors were responsible for this.

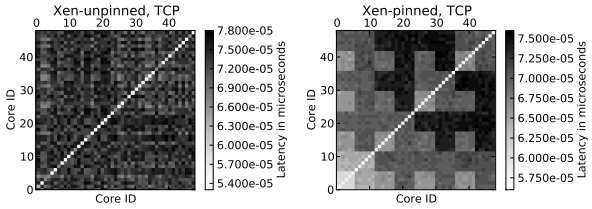

Figure 3: `tcp` on Xen with vCPUs unpinned/pinned.

Firstly, even though the host is running only a single 48-vCPU domain, the Xen scheduler still continuously rebalances the virtual-to-physical CPU mappings within the domain. Thus, the tests where the vCPUs are unpinned are extremely noisy (Figure 3). Manually pinning the vCPU to the corresponding pCPU also does not fully recreate the native Linux experience, as enumeration order and APIC handling varies between the two.

Xen does not currently support communicating the NUMA layout to PV guests, and hence all memory was allocated on NUMA node 0 (cores 0 to 5). This explains[3] the patterns observed: with increasing distance from NUMA node 0, the IPC RTT increases (with the exception of those cores on sockets which have direct HT connections to node 0).

[3]With one exception: we are still investigating why same-core communication in the `unix` benchmark under Xen performs worse than off-core communication, despite `pipe` performng better.

3.3 Examining the microarchitecture

In order to validate our intuitions about the impact of interconnect topologies and NUMA layouts on the micro-benchmark results, we established a lower performance bound by measuring the latency of an inter-processor interrupt (IPI) using a specially modified version of Xen which performs an IPI "self-test" very early in boot (before dom0 has booted).

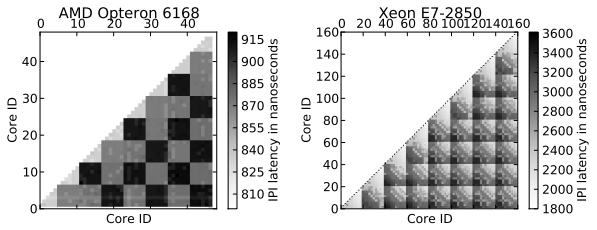

Figure 4: Pairwise IPI latency between cores on the 48-core Opteron and an 80-core Xeon E7-2850 machine.

On the 48-core Opteron (Figure 4 *(left)*), the IPI latencies are between 830 and 910ns, and the pattern exhibited matches intuition and previous results: three different latency categories—same-socket, 1-hop HT (including same-MCM links) and 2-hop HT—exist. This confirms that interconnect topology is an important factor in latency micro-benchmark performance.

We also measured the IPI latency on an 80-core, hyper-threaded Intel Xeon E7-2850 (Figure 4 *(right)*). This machine is more complex than the Opteron, and each cell represents a pair of hardware threads, cores are represented by 2x2 blocks of threads, and sockets represented by 10x10 blocks of cores. The sockets are themselves divided into 2 trays, each of 4 sockets. Again, the best performance is observed when talking to a core on the same socket. However, the latencies observed (2-3.6ms) are more than twice as large as on the Opteron, and vary dramatically when talking to an off-socket core.

This is rooted in the design of the on-chip interconnect in Intel's "Westmere-EX" architecture: on each chip, there is a bi-directional ring connecting the 10 cores with each other and with the QPI socket interconnect [5]. Depending on the relative location of a core to the system interface connection point, between one and four on-chip hops may be required to reach the core. This substantiates itself in the gradient pattern within each block representing a socket, and presents further evidence for the trend towards more complex IPC performance patterns in coming years.

4 Towards longitudinal data capture

IPC performance is highly dependent on the exact deployment environment. We observed a variety of factors influencing the exact IPC performance in a particular combination of hardware, OS and transport mechanisms. At the same time, optimizing IPC performance is a key to building performant software stacks, from the hypervisor up to language runtimes. For instance, we are using the lessons from the shared memory backends to speed up the Xen virtual device system [7], and also integrating them into a new statically-mapped linearly-typed systems language [4].

Many optimization decisions may be informed by information such as that gathered by `ipc-bench`, and as developers of open-source software, it is frustrating to not have a larger, long-term dataset to draw from. For this reason, we are making the complete corpus of results from our runs available online[4]. The number of platforms available to us, however, is tiny compared to the number of possible combinations of hardware, virtualization and OS out "in the wild". Hence, we would like to propose another experiment in open data gathering. We have packaged up portions of `ipc-bench` so that it can operate as an automated means of gathering data, and also a machine discovery script that records as much of the relevant environmental data (without any personally identifiable information).

Instead of submitting this to a closed online database, as many existing bug reporting systems do [3], we wish to keep the code and data open, along with the *provenance* of the data, for anyone to analyse and use. The benchmark code is relatively small and stored in a repository on Github[5], a popular code sharing website, and can be forked by anyone interested. The benchmark driver program also gives the option of pushing results to a fork and sending an automated pull-request to the master repository (and any others specified), allowing new results to be easily pulled in.

Both the code and data results can be stored using distributed version control, and aggregated by us (or anyone else) as demand exists. This approach presents some interesting challenges that we are currently working on:

- **Portability:** while the POSIX tests are easy to run across multiple UNIX variants, others such as the shared memory tests depend on platform extensions such as futexes. It is important to ensure that tests with subtly different behaviours (such as our `mempipe` variants) are recorded appropriately. Even (or especially) minor Linux kernel changes can result in unexpected performance changes!
- **Self-testing:** whilst the user-space libraries are easy to package up, components of `ipc-bench` run directly in the hypervisor (and future tests will run directly in-kernel). This requires a quick self-test to be added to Xen, with results recorded later in the boot cycle.
- **Versioning:** we wish to run this test for years, and will of course evolve the benchmark code. This requires a fine-grained versioning scheme to ensure that all prior data is not invalidated on every code change. Since code and data are recorded immutably, this should be possible to fix up in future analysis.

Our developent of `ipc-bench` was partly motivated by necessity: we are currently working on FABLE, a unified low-level communications and I/O library supported by a distributed cluster naming service. FABLE tries to pick an optimal communication mechanism out of those available using a knowledge base populated by results from `ipc-bench`, and supports reconfiguring I/O channels if dynamic load or security conditions change.

[4] See `http://www.cl.cam.ac.uk/netos/ipc-bench/`

[5] See `http://github.com/avsm/ipc-bench`

References

[1] BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHUPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP 2009* (Big Sky, Montana, 2009), pp. 29–44.

[2] CONWAY, P., KALYANASUNDHARAM, N., DONLEY, G., LEPAK, K., AND HUGHES, B. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro 30*, 2 (Mar. 2010), 16–29.

[3] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 103–116.

[4] PROUST, R., VERLAGUET, J., AND MADHAVAPEDDY, A. Limel: Local computation and linear coordination. In *peer-review at PLACES 2012* (2012).

[5] SAWANT, S., DESAI, U., SHAMANNA, G., SHARMA, L., RANADE, M., AGARWAL, A., DAKSHINAMURTHY, S., AND NARAYANAN, R. A 32nm Westmere-EX Xeon® enterprise processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International* (2011), IEEE, pp. 74–75.

[6] TALBOT, J., YOO, R. M., AND KOZYRAKIS, C. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceddings of MAPREDUCE 2011* (2011).

[7] WARFIELD, A., FRASER, K., HAND, S., AND DEEGAN, T. Facilitating the development of soft devices. In *Proceedings of the 2005 USENIX Annual Technical Conference (General Track)* (April 2005), USENIX, pp. 379–382.

[8] WENTZLAFF, D., GRUENWALD, C., AND BECKMANN, N. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of SOCC 2010* (2010), pp. 3–14.