Design Challenges of Scalable Operating Systems for Many-Core Architectures

Andreas Schärtl

Friedrich-Alexander University Erlangen-Nuremberg (FAU)

06/12/2016
Introduction
We are moving from multi-core to many-core [1]. This introduces some changes:

- 1,000 and more cores
- Differences in computer architecture [5]
We are moving from multi-core to many-core [1]. This introduces some changes:

- 1,000 and more cores
- Differences in computer architecture [5]

It is important that operating systems scale on this hardware!

- OS is the base for all applications
- It is helpful to differentiate between load scalability and structural scalability [4]
We are moving from multi-core to many-core [1]. This introduces some changes:

- 1,000 and more cores
- Differences in computer architecture [5]

It is important that operating systems scale on this hardware!

- OS is the base for all applications
- It is helpful to differentiate between load scalability and structural scalability [4]

What are the design challenges for an OS on many-core hardware?
Introduction

Locks

Caches and Locality

Reliance on Cache Coherent Shared Memory

Conclusion
Locks
Locks

- One job of any OS is managing system resources [14]
  - Network
  - Memory
  - ...
- Sometimes exclusive access to resources is required
- Often locks enforce critical sections [15]
Locks may not Scale

Figure: Taken from *Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores* [15]
Locks hurt OS scalability. What can be done about it?

Use more fine-grained locks?

Done today in OS development [15]

But: Parallelizing code is prone to errors and already parallelized code is hard to parallelize even more.

Use better locks?

Boyd-Wickizer et al. replaced spin locks in the Linux kernel with more modern MCS locks [10]. This resulted in considerable performance improvements [7].

But: Lock contention remains.

Both approaches offer poor structural scalability because they still rely on locks.
Locks hurt OS scalability. What can be done about it?

- Use more fine-grained locks?
  - Done today in OS development [15]
  - But: Parallelizing code is prone to errors and already parallelized code is hard to parallelize even more
Locks hurt OS scalability. What can be done about it?

**Use more fine-grained locks?**
- Done today in OS development [15]
- *But*: Parallelizing code is prone to errors and already parallelized code is hard to parallelize even more

**Use better locks?**
- Boyd-Wickizer et al. replaced spin locks in the Linux kernel with more modern MCS locks [10]. This resulted in considerable performance improvements [7].
- *But*: Lock contention remains
Locks hurt OS scalability. What can be done about it?

Use more fine-grained locks?
- Done today in OS development [15]
- *But:* Parallelizing code is prone to errors and already parallelized code is hard to parallelize even more

Use better locks?
- Boyd-Wickizer et al. replaced spin locks in the Linux kernel with more modern MCS locks [10]. This resulted in considerable performance improvements [7].
- *But:* Lock contention remains

Both approaches offer poor structural scalability because they still rely on locks.
How does fos [15] avoid locks?

- fos is optimized for systems with hundreds to thousands of cores
- Operating system is *factored* into small parts
- *Servers* offer OS functionality (e.g. networking, paging)
  - Each server runs on a dedicated core
  - Servers are organized in *fleets* that provide the same functionality
  - Servers process requests in a sequential manner
Locks

Sequentializing OS services

\[ S \]

\[ A_0 \]

\[ A_1 \]
Locks

Sequentializing OS services

S

malloc

A₀

A₁
Locks

Sequentializing OS services

\[ S \]

\[ \text{JOB } 0 \]

\[ A_0 \]

\[ A_1 \]
Locks

Sequentializing OS services

\[ S \]  

\[ JOB 0 \]  

\[ malloc \]  

\[ A_0 \]  

\[ A_1 \]
Locks

Sequentializing OS services

\[
\begin{array}{c}
S \\
\hline \\
\text{JOB 0} \quad \bigcirc \\
\text{JOB 1} \quad \bigodot \\
A_0 \\
A_1
\end{array}
\]
Locks

Sequentializing OS services

\[ S \rightarrow \text{Reply} \rightarrow A_0 \]

\[ \text{JOB 1} \]

\[ A_1 \]
Locks

Sequentializing OS services

\[ S \]

\[ \text{JOB 1} \]

\[ A_0 \]

\[ A_1 \]
Dedicated server cores make it possible to sequentialize OS services [15]

- No locks are required on a core
- Takes advantage of optimistic synchronization
Dedicated server cores make it possible to sequentialize OS services [15]
- No locks are required on a core
- Takes advantage of optimistic synchronization

New challenge: How to synchronize servers in a fleet with each other?
- *Notional locks*
- Transaction servers
- Distributed algorithms
Dedicated server cores make it possible to sequentialize OS services [15]

- No locks are required on a core
- Takes advantage of optimistic synchronization

New challenge: How to synchronize servers in a fleet with each other?

- *Notional locks*
- Transaction servers
- Distributed algorithms

In 2011, some OS services were implemented as servers [16]:

- Networking, paging, read-only file system
- Some overhead is introduced
- Compared to Linux, fos offered better scalability
Caches and Locality
Caches and Locality

- Often OS and application share the same core [12]:
  - Caches and TLB lose effectiveness because of poor locality
  - Context switching is expensive in itself

- Impact on load scalability
  - More system calls mean more damage to caches
Caches and Locality

Figure: Taken from *Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores* [15]
How does fos avoid OS/application sharing of cores?
How does fos avoid OS/application sharing of cores?

- On many-core platforms, cores will be plentiful
- It becomes possible to assign one core to every thread
  - If there are less cores than threads, time sharing is used only as a fallback
How does fos avoid OS/application sharing of cores?

- On many-core platforms, cores will be plentiful
- It becomes possible to assign one core to every thread
  - If there are less cores than threads, time sharing is used only as a fallback

Other operating systems also use dedicated cores

- Corey [6] allows applications to dedicate cores to kernel tasks
- Barellfish [2] uses a similar approach to fos
Figure: Improvements of separating OS and application onto different cores on a Linux system. Taken from *Vote the OS off your core* [3].
Reliance on Cache Coherent Shared Memory
Cache Coherent Shared Memory

- Typical PC hardware today offers cache coherent shared memory [6]
- Software running on such systems can make certain assumptions [13]:
  - There exists a single global address space
  - Cache coherence can ensure that caches remain in sync (volatile keyword)
- Cache coherent shared memory can be used for communication between threads and processes [9, 15, 6]
Some researchers believe that many-core architectures will not offer cache coherent shared memory [15, 2, 8].
Some researchers believe that many-core architectures will not offer cache coherent shared memory [15, 2, 8].

Current embedded many-core platforms do not have cache coherent shared memory available [11, 15].
Some researchers believe that many-core architectures will not offer cache coherent shared memory [15, 2, 8].

Current embedded many-core platforms do not have cache coherent shared memory available [11, 15].

Cache Coherence is hard to scale up [8]:

- Power and latency overhead
- Extra space overhead
- Complex implementation prone to errors
Many-Core architectures typically do have on-die networks [5]
- Ring or mesh topologies
- Packet-switching

Performance evaluations are encouraging [2]

Figure: Typical many-core chip layout. Taken from *Thousand core chips: a technology perspective* [5].
Cache Coherent shared memory should still be offered to applications, if supported by hardware [15].

Figure: Application $A_0$ and $A_2$ use application-level cache coherent shared memory. The server cores $S$ and application $A_1$ do not.
Conclusion
Conclusion

- We are moving toward the many-core era. This introduces new challenges for OS development.
We are moving toward the many-core era. This introduces new challenges for OS development.

Locks may not scale
- Lock contention gets expensive
- Using better and more fine-grained locks can be a short term solution
- Structuring the OS to sequentialize requests scales better
Conclusion

- We are moving toward the many-core era. This introduces new challenges for OS development.
- Locks may not scale
  - Lock contention gets expensive
  - Using better and more fine-grained locks can be a short term solution
  - Structuring the OS to sequentialize requests scales better
- OS/application sharing of cores
  - Poor locality impedes performance, especially of the OS
  - Dedicated server cores take full advantage of caches
Conclusion

- We are moving toward the many-core era. This introduces new challenges for OS development.
  - Locks may not scale
    - Lock contention gets expensive
    - Using better and more fine-grained locks can be a short term solution
    - Structuring the OS to sequentialize requests scales better
  - OS/application sharing of cores
    - Poor locality impedes performance, especially of the OS
    - Dedicated server cores take full advantage of caches
  - Reliance on cache coherent shared memory
    - Many-Core architectures may not offer cache coherent shared memory
    - Using message passing instead is a viable alternative
    - Applications can use islands of shared memory
Questions?


Characteristics of scalability and their impact on performance.

Thousand core chips: A technology perspective.
Corey: An operating system for many cores.

Non-scalable locks are dangerous.
Denovo: Rethinking the memory hierarchy for disciplined parallelism.

On micro-kernel construction.


Factored operating systems (fos): The case for a scalable operating system for multicores.  

Fleets: Scalable services in a factored operating system.  
2011.