SCALABILITY AND HETEROGENEITY

MICHAEL ROITZSCH
Application
Runtime
OS Kernel
ISA
Core
Physical RAM
Non-Uniform Memory Access

- core-to-RAM distance differs
- various interconnect topologies: bus, star, ring, hypercube
- more general: different access latencies to data
- consider cache latency, shared resource contention
<table>
<thead>
<tr>
<th>Operation</th>
<th>Access</th>
<th>Time</th>
<th>Std Dev</th>
<th>NUMA Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>local</td>
<td>37.420 s</td>
<td>$6.60 \cdot 10^{-2}$ s</td>
<td>1.000</td>
</tr>
<tr>
<td>read</td>
<td>interleaved</td>
<td>48.405 s</td>
<td>$3.20 \cdot 10^{-2}$ s</td>
<td>1.294</td>
</tr>
<tr>
<td>read</td>
<td>remote</td>
<td>53.223 s</td>
<td>$3.53 \cdot 10^{-2}$ s</td>
<td>1.422</td>
</tr>
<tr>
<td>write</td>
<td>local</td>
<td>23.555 s</td>
<td>$7.17 \cdot 10^{-3}$ s</td>
<td>1.000</td>
</tr>
<tr>
<td>write</td>
<td>interleaved</td>
<td>23.976 s</td>
<td>$1.18 \cdot 10^{-3}$ s</td>
<td>1.018</td>
</tr>
<tr>
<td>write</td>
<td>remote</td>
<td>23.976 s</td>
<td>$1.48 \cdot 10^{-3}$ s</td>
<td>1.018</td>
</tr>
</tbody>
</table>
NUMA MECHANISM


TU Dresden

MOS: Scalability and Heterogeneity

6
NUMA POLICIES

- fundamental options: migrate thread vs. migrate data
- use performance counters to monitor
- dynamic management shows >10% performance benefit compared to best static placement
TU Dresden MOS: Scalability and Heterogeneity

**Layer Cake**

- Application
- Runtime
- OS Kernel
- ISA
  - Core
  - RAM
  - Coherency
  - Interconnect
  - Core
  - RAM
The Multikernel: A new OS architecture for scalable multicore systems


TU Dresden

MOS: Scalability and Heterogeneity
Barrelfish concept: multikernel, implementation: Barrelfish
- treat the machine as cores with a network
- no inter-core sharing at the lower levels
- “CPU driver” plus exokernel-ish structure
- traditional OS functionality as a tailored distributed system with state replication
REALITY SO FAR

App
RT
OS
ISA
Core
RAM

App
RT
OS
ISA
Core
RAM

Messages
Interconnect
driven by scalability issues of shared kernel designs and cache coherence

this may not be a pressing issue today

The architecture of future computers is far from clear but we expect tools like these to support a message-passing abstraction will become widespread.

Increasing system and interconnect diversity, as well as core heterogeneity, will prevent developers from optimizing shared memory structures at a source-code level. It seems that the OS design itself does not rely on it. In contrast, we propose an OS architecture for heterogeneous multicore machines, which we call the BARRELFISH model.

Within a multikernel OS, all inter-core communication is performed using explicit messages. A corollary is that no memory is shared between the code running on each core, except for that used for messaging channels. As we have seen, using messages to access or update state is more efficient than shared-memory access or update, as the number of cache-lines involved increases. We rapidly becomes more expensive to exploit it.

This upheaval in hardware has important consequences and latency to access main memory, and the complex optimizations, and future system software will have to adapt its communication patterns and mechanisms at runtime to the collection of hardware at hand. It seems fit for such a design.

Since the architectural changes are driven by hardware developments that exposed scalability and performance bottlenecks, particularly the adoption of multiple processors in commodity PCs. Mainstream OSes are currently moving towards the center, where several “hot” research systems take this even further with mechanisms for distributed systems approach to gain improved performance, natural support for hardware heterogeneity, distributed systems approach to gain improved performance, natural support for hardware heterogeneity, and between systems with varying interconnect topologies.

These principles allow the OS to benefit from the rapidly becoming more pronounced in various disciplines. Traditional operating systems, such as Windows and variants of Unix, have evolved from designs where all state is replicated by default and consistency is maintained using agreement protocols. In contrast, we propose an OS architecture for heterogeneous multicore machines, which we call the BARRELFISH model.

In a nutshell, we structure the OS as a distributed system of cores that communicate using messages. A corollary is that no memory is shared between the code running on each core, except for that used for messaging channels. As we have seen, using messages to access or update state is more efficient than shared-memory access or update, as the number of cache-lines involved increases. We rapidly becomes more expensive to exploit it.

Figure 4: Spectrum of sharing and locking disciplines.

- Traditional OSes:
  - Shared state, one-big-lock
  - Finer-grained locking
  - Clustered objects, partitioning

- Multikernel:
  - Distributed state, replica maintenance

View state as replicated instead of shared.

Make OS structure hardware-neutral.

Make all inter-core communication explicit.
INSIDE THE BOX

App
RT
OS
ISA
Core
RAM

Messages

Interconnect

App
RT
OS
ISA
Core
RAM
tu_dresden
MOS: Scalability and Heterogeneity

BEEN THERE

TU Dresden
MOS: Scalability and Heterogeneity
16
spatial allocation of resources over time multiplexing of resources.

client. Figure 4 has a conceptual model of applications and the OS executing on a single core under the fos-microkernel is called an fos applications and OS layer servers executing under it. The code exposes messaging API. The fos-microkernel does not differentiate between under the fos-microkernel and communicate via the fos-microkernel to turn system calls into messages from application to an OS layer exposed to the application writer. A small translation library is used to translate system calls to messages. When an application requires OS services, the underlying communication mechanism is OS layer does not internally utilize shared memory, but rather utilizes a message-based communication or messaging. While coherent shared memory may be quite useful. This is why fos provides the ability for applications to communicate only via the messaging interface provided by the microkernel.

Each server executes solely on a dedicated processor core. Server access, and a fleet which manages process scheduling and layout. Physical memory allocation, a fleet which manages the file system for different functions. For instance, there is a fleet which manages function specific servers. Each operating system function is provided by the OS layer. The OS layer is composed of fleets of application communication primitives. In fos, this functionality is

Applications and OS layer servers act as peers. They all run un-...
Node 0

- VCPU 0
- Machine Pages

Node 1

- VCPU 1
- Machine Pages

Virtual Pages

Physical Pages

As a result, all read-only pages can be shared between virtual machines. Although this reduces the memory footprint, this may adversely affect data locality as most sharers will access the page remotely. However, Disco’s page replication policy selectively replicates the few “hot” pages that suffer the most cache misses. Pages are therefore shared whenever possible and replicated only when necessary to improve performance.

4.3 Running Commodity Operating Systems

The “commodity” operating system we run on Disco is IRIX, a UNIX SVR4-based operating system from Silicon Graphics. Disco is however independent of any specific operating system, and we plan to support others such as Windows NT and Linux.

In their support for portability, modern operating systems present a hardware abstraction level (HAL) that allows the operating system to be effectively “ported” to run on new platforms. Typically the HAL of modern operating systems changes with each new version of a machine while the rest of the system can remain unchanged. Our experience has been that relatively small changes to the HAL can reduce the overhead of virtualization and improve resource usage.

Most of the changes made in IRIX were part of the HAL. Unlike other operating systems, IRIX does not contain a documented HAL interface. In this article, the HAL includes all the platform- and processor-specific procedures of the operating system.

4.3.1 Necessary Changes for MIPS Architecture.

Virtual processors running in supervisor mode cannot efficiently access the KSEG0 segment of the MIPS virtual address space that always bypasses the TLB. Unfortu
- motivated by increasing core counts but badly scaling operating systems
- increase system utilization by co-locating multiple badly scaling systems
- allow for future better scaling systems
- employ the opportunities of cache-coherent shared memory instead of dogmatizing the share-nothing kernel
what we want: some unifying runtime / OS service on top of non-cache-coherent hardware

OS service: distributed shared memory?

runtimes from HPC (Charm++, X10) or cloud computing (MapReduce, Dryad)?

distributed-systems-on-chip?

still ongoing research
THE HOLY GRAIL

![Diagram showing the relationship between App, RT, OS, ISA, Core, RAM, and Interconnect.](image)
GPUS TODAY

Application
Runtime
OS Kernel
ISA
Core
Physical RAM

Compute Kernel

HLSL
OpenCL
GPU Driver
ISA
Core
GPU Today

Application

Runtime: C++ AMP

OS Kernel: GPU Driver

ISA

Core

Physical RAM
- idea: heterogeneous ISA systems need some kind of compiler support
- ISA-specific kernels: “satellite kernels”
- provide uniform OS abstractions
- memory management, scheduling
- bootstrap: first kernel becomes coordinator, boots other cores
- share-nothing, even on ccNUMA
- processes cannot span across kernels
- implementation based on Singularity
- applications compiled to intermediate code
- 2nd stage compilation to native code of all available ISAs at install time
- placement based on affinity hints
Placement Example

Source: [NHM 09]

general solution
- transform memory state
- transition control flow between versions
- modify compiler to keep memory state architecture-independent
- runtime stack transformation
- binary translation up to next function call
The expected time to the next call, under three situations: no dummy calls have been added, dummy calls to outermost loops have been added, and dummy calls to second-innermost loops have been added. Figure 4 (first two bars) shows the ETTT for each benchmark. The distribution of function calls is highly irregular. During some phases of execution, function calls are frequent; in other phases, calls are much less frequent. Consequently, the average call frequency when migrating back and forth between an ARM core and a MIPS core.

Figure 3. Performance - Dummy calls in outermost loops

Figure 4. Compilation for migratability (w/o migrations)

The expected time to transformation (ETTT) is the time between calls and the median time between calls are poor predictors of performance. The expected time to transformation (ETTT) is much higher when migrations are infrequent, as is lower at every migration frequency. If migration never occurs, performance remains below 95%. The line below that shows performance when migrations occur. When migrations happen every 10 milliseconds or less the effect on performance is significant. But above 10 milliseconds, performance is too severe. The line in the graph represents the performance if no migration occurs, and accounts only for the overhead of compilation for migratability (w/o migrations). The straight line below that shows performance when migrations occur. The two lowest bars in Figure 3 show the performance of code compiled with dummy calls inserted. Migration overhead is the performance impact is too severe. The presence of dummy procedures calls. This preserves some non-performance is significant. But above 10 milliseconds, performance remains below 95%.

The two lowest bars in Figure 4 show the ETTT for each benchmark. Varying migration frequency is important because it determines how many instructions must be executed by our binary translator, which incur a performance penalty. The other two lines will be discussed in the next section.

The additional transformation opportunities that these changes bring comes at the cost of performance. Injecting dummy calls decreases in ETTT. One way to increase the frequency of migration opportunities is to add more function calls. The long gaps between calls result from long-running loops that do not contain any function calls (or have been inlined). We modify GCC to inject dummy calls into loops. We experiment with two loop selection policies: (1) inserting function calls in outermost loops (parents of in-ternal functions) and (2) inserting function calls in second-innermost loops (parents of innermost loops). We never select innermost loops because the performance impact is too severe. The additional transformation opportunities that these changes bring comes at the cost of performance. Injecting dummy calls decreases in ETTT.
- scalability approaches tend to move the problem upwards into runtimes and apps
- various microkernel-like approaches
- solutions from distributed systems
- today’s challenge: heterogeneity
- compilation as an OS primitive
- future challenge: reconfigurable hardware