Asynchronous Abstract Machines

Anti-noise System Software for Many-core Processors

June 25, 2019

Sebastian Maier, Timo Hönig, Peter Wägemann, Wolfgang Schröder-Preikschat

System Software Group
Friedrich-Alexander-Universität Erlangen-Nürnberg

Supported by German Research Council (DFG): CRC/TRR 89 (InvasIC, Project C1); SCR 603/8-2 (LAOS)
cores are shared between heterogeneous workload
- different applications and their threads
- application, library and OS code
  ↞ interference, scheduling overhead
  ↞ decreased performance

Is there a better way to operate many-core systems?
AAM

- Asynchronous Abstract Machines (AAMs) as a new system design approach for reduced noise
- address shortcomings of existing systems:
  1. heavy-weight threads and system calls
  2. missing OS-level support for teams
  3. static allocation of resources
1. Heavy-weight Threads and System Calls

Transitions Costs between Workloads

- direct costs
  - time required for actual transition (e.g., mode switch or context switch)

- indirect costs
  - executing other workload causes interference
    - instruction/data caches
    - Translation Lookaside Buffer (TLB)
    - branch prediction units
  - decreased instructions-per-cycle (IPC) performance

Indirect Costs of System Calls

- significant impact on the user-space performance of the CPU for several thousand cycles

---

1 L. Soares, M. Stumm; "Flexible system call scheduling with exception-less system calls"
1. Heavy-weight Threads and System Calls

Kernel-level Scheduling

- requires expensive mode change
- threads have large memory footprint

unsuited for micro-parallelism

User-level Scheduling

- reduced scheduling overhead
- prone to blocking anomaly (w/o native OS support)
  1. user-level task issues a system call
  2. OS blocks the execution context (thread) in the kernel
  3. thread becomes unavailable for user-level scheduler

unsuited for system-intensive workload
2. Missing OS-level Support for Teams

- thread pools: common technique to parallelize tasks and reduce scheduling overhead
- shortcomings
  - OS has no notion of thread pools and work queues
    - is unaware that these threads form a team and execute similar tasks
    - lacks information: amount of tasks (load)
  - subpar scheduling
- optimal number of threads?
  - available resources, future workload, overall system load

---

1 D. R. Cheriton; "The V kernel: A Software Base for Distributed Systems"
3. Static Allocation of Resources

- static allocation of resources
  - offloading system functionality to dedicated cores (e.g., to reduce noise)
  - allocation of a fixed number of threads (e.g., in a thread pool)
- changing workload causes imbalance
  - poor resource utilization
  - performance bottlenecks
Goals of the AAM Approach

- operate cores more efficiently
  - avoid transitions between heterogeneous workloads
    - partition workload into groups of homogeneous tasks
    - dedicate cores to these groups
  - speedup transitions between homogeneous workloads
    - lightweight tasks
    - user-level scheduling

- address problems within user and kernel space
Concept
System Overview

- Asynchronous Abstract Machine (AAM)
  - dedicated to a specific group of tasks (shared code/data)
  - lightweight task scheduler
  - asynchronous task-based interface
- entire system is composed of AAMs (Applications, OS)
- Machine Manager: dynamic allocation of cores to AAMs
AAMs may use their own task scheduler and allocator
AAM Framework offers default implementations

IMC Inter-machine communication
IMS Inter-machine signaling
**Machine Manager**
- machine scheduling
- inter-machine signaling

**Machine Interfaces**
- queues in shared memory
- direct communication between machines
AAMs offer predefined tasks to client machines

\[\rightarrow\] scheduled asynchronously on server machines

- direct IMC does not involve the OS kernel
  (in the common case)
Inter-machine Signaling (IMS)

- Inter-machine signals (delivered by Machine Manager)
  - wake sleeping machines
  - register new interfaces
- involves traditional system calls
  - short and run-to-completion
  ~~~~ minimal indirect costs
Machine Scheduling

- Machine scheduling allocates cores to AAMs
  - maximize utilization
  - minimize interference
- Machine Manager is aware of all machines
  - machine load
  - recent core utilization
  - prior core allocations
Scheduling Tasks within AAMs

- optimized for a huge number of short-lived tasks
  - task identifier, parameters, future
  - run-to-completion
  - lazy context allocation
  ~~~ small memory footprint
- Machine-local scheduler
  ~~~ scheduling does not involve OS kernel
  ~~~ switching between tasks is inexpensive
Implementation and Evaluation
Specifying AAMs at Design-time

Considerations

- duration and cache behavior of operations
- shared data or functionality between operations
- distinct computation stages or system boundaries
- required privileges and isolation requirements

Specification and Reusability

- interface is defined in an IDL file
  - C-compatible format
  - automatic code generation
- self-contained with well-defined interface
  - AAMs are reusable (like libraries)
Asynchronous Interface

\[\sim\] returns immediately with a future; allows for latency hiding and batching

```
1    char buffer[MAX_LEN];
2    auto *rt = System::readAsync(fd, buffer, MAX_LEN);
3
4    // do other stuff ... 
5
6    ssize_t result = rt->force();
```

Synchronous Interface

\[\sim\] calling task waits for completion; another task is scheduled

```
1    char buffer[MAX_LEN];
2    ssize_t result = System::read(fd, buffer, MAX_LEN);
```

Event-based Interface

\[\sim\] schedules a specified task on completion (work in progress)
Current State of the Prototype

Target Architectures

- native OS for x86-64
  - → benchmarking
- Linux 64-bit application
  - → development and debugging

Components

- AAM Framework
  - lightweight task scheduler
  - memory allocator
  - inter-machine communication
- Machine Manager
  - machine scheduler
  - inter-machine signaling
Reusable Machines

- library machines (user level)
  - SQLite
  - AES encryption
  - ZLIB/LZO compression
- system machines (kernel level)
  - TCP/IP stack
  - file system

Tools and Profiling Support

- iGen code generator
- sView scheduling analyzer
- CPU performance counters
- per-machine metrics (IPC, ...)
Evaluation

Costs of Typical System Operations

- local task execution
  1. create task and add it to the scheduler  \(\sim\) direct costs
  2. block active task (waiting for task completion)
  3. execute no-op task
  4. continue original task  \(\sim\) total latency

- Machine Call
  \(\sim\) task execution on different machine and core via IMC

Evaluation Setup

- Intel Xeon CPU E3-1275 v3 @ 3.50 GHz, 32 GiB RAM
- arithmetic mean and standard deviation from 10000 runs
- Linux (used for comparison): kernel version 4.4
**Evaluation Results**

**Direct Costs**

- Task Execution (core-local)
- Machine Call (Library)
- Machine Call (System)

**Total Latency**

- Task Execution (core-local)
- Machine Call (Library)
- Machine Call (System)

- **active** AAMs actively monitor their interfaces
- **idle** AAMs allowed to idle immediately (↝ IMS)
Evaluation Results

active  AAMs actively monitor their interfaces

idle  AAMs allowed to idle immediately (\(\sim\) IMS)

costs of typical Linux operations in gray
Local Task Execution

- task creation is fast
- overhead for task scheduling is low
**Machine Calls**

- **CPU becomes available to the caller after a short time**
  - → latency hiding; schedule other task

- **avoiding indirect costs comes with a latency overhead**
  - → increased latency if IMS is required
  - → still low compared to most system calls or thread creation
Future Work and Conclusion
Future Work

- more micro and macro benchmarks
  - e.g., HPC and server applications
- enhanced machine scheduling strategies
- isolation support
- hardware support for improved IMC performance
  - Software-defined Hardware-managed Queues (SHARQ) \(^1\)
    for communication across isolation domains

\(^1\)S. Rheindt, S. Maier, F. Schmaus, T. Wild, W. Schröder-Preikschat, A. Herkersdorf; “SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures”
Conclusion

- **goals**
  - avoid costly transitions between heterogeneous workload
  - speedup transitions between homogeneous workload
- **AAM concept**
  - partition system into machines with task schedulers
  - assign cores to machines exclusively during runtime
- **addressed problems**
  1. heavy-weight threads and system calls
     - machine-local task scheduling; task-based interface
  2. missing OS-level support for teams
     - Machine Manager is aware of all AAMs
  3. static allocation of resources
     - Machine Manager allocates cores dynamically