Asynchronous Abstract Machines: Anti-noise System Software for Many-core Processors

Sebastian Maier, Timo Hönig, Peter Wägemann, Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

ABSTRACT

Today's systems offer an increasing number of processor cores, however, the chance to operate them efficiently by dedicating cores to specific tasks is often missed. Instead, mixed workloads are processed by each core which leads to system noise (i.e., interferences, scheduling overheads) and yields subpar performance, only.

We therefore propose a system design based on *Asynchronous Abstract Machines* (*AAM*s). An *AAM* features a light-weight scheduler and is dedicated to a specific group of tasks with common characteristics (i.e., shared code and data). It offers an asynchronous, task-based interface for efficient interaction between *AAM*s. Just like applications are built from *AAM*s, even the OS itself consists of *AAM*s that are interfaced by applications via asynchronous messages instead of synchronous system calls. A dedicated OS component, which is aware of all *AAM*s in the system, is responsible for dynamic and exclusive allocation of cores to *AAM*s depending on their current workload. Thus, cores rarely switch between heterogeneous workloads of different *AAM*s. And, at the same time, frequent switches between homogeneous tasks become fast, local operations of an *AAM*, which do not involve the OS kernel.

In this paper, we describe shortcomings of existing operating systems, our new system design concept, and present evaluation results of our prototype implementation.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Software and its engineering → Operating systems**; **Scheduling**; **Multithreading**; **Multiprocessing / multiprogramming / multitasking**; **Concurrency control**; **Message passing**.

KEYWORDS

system noise, interference, locality, dynamic resource allocation, partitioning, message-based system call, thread pool, scheduling

ACM Reference Format:
Sebastian Maier, Timo Hönig, Peter Wägemann, Wolfgang Schröder-Preikschat. 2019. Asynchronous Abstract Machines: Anti-noise System Software for Many-core Processors. In *9th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'19), June 25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3322789.3328744

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ROSS'19, June 25, 2019, Phoenix, AZ, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6755-4/19/06...$15.00
https://doi.org/10.1145/3322789.3328744

1 INTRODUCTION

Historically, computer systems featured only a few processor cores and operating systems had no choice but to schedule heterogeneous workloads on these cores. However, even today, in the presence of many-core systems, common operating systems (e.g., Linux) continue to share processor cores between unrelated threads that often originate from different processes. In addition, the processing time assigned to a thread is fragmented into the execution of a variety of different application or library modules, on the one hand, and the OS, on the other hand. Even a single system call can have a significant impact on the perceived user-space performance of an application for several thousand CPU cycles [24]. While a system call introduces direct costs in the form of the mode switch and the time spent inside the kernel, there are also indirect costs that are caused by executing OS code and accessing OS data in memory. This interference causes a decreased instructions-per-cycle (IPC) performance of the CPU directly after a system call occurred and stems from increased cache/TLB misses and branch mispredictions. With the introduction of kernel page-table isolation (KPTI) [14] to mitigate a widespread hardware security bug in Intel CPUs (i.e., Meltdown [12]), costs of system calls, and interrupts have increased even further [21, 22].

But indirect costs are not limited to system calls, they also appear when switching between different workloads within user space (e.g., different application or library modules). Therefore, existing systems often operate processor cores inefficiently, while at the same time more cores become available that could as well specialize in a certain group of tasks and thus avoid expensive transitions between tasks with heterogeneous workloads.

To solve these problems, we propose *Asynchronous Abstract Machines* (or short *AAM*s) that group tasks with shared resources (i.e., data, code) together and execute them on a dedicated set of processor cores, to increase cache locality and decrease interference. The number of cores assigned to an *AAM* may change dynamically over time depending on the current workload. To enable efficient interaction between *AAM*s, they feature an asynchronous, task-based interface that even works for communication across isolation boundaries. Exclusive resource allocation and reduced interference make the *AAM* approach and its anti-noise system design especially interesting for domains like staged server applications and high-performance computing (HPC).

The paper is structured as follows: Section 2 explains shortcomings of existing operating systems regarding the utilization of many-core systems. Section 3 describes our approach of building systems based on *AAM*s and Section 4 gives an overview of the prototype implementation and first evaluation results. Section 5 explains identified challenges and states future work in the context of *AAM*s. Section 6 discusses related work, and Section 7 concludes.

2 PROBLEM STATEMENT

In the following section, we first outline our system model. Subsequently, we discuss the three problems, addressed by *AAM*: The problems contain (1) the missing native support for threads executing homogeneous work in current operating systems, (2) the performance degradation due to switching between threads in kernel space, and (3) the static pre-allocation of resources (i.e., cores, threads) leading to poor utilization.

2.1 System Model

Domains where many-core systems are already prevalent are server applications and high-performance computing (HPC). Server applications typically handle a massive number of independent requests per second that mostly consist of short-lived tasks with intense OS interaction. HPC applications, in contrast, utilize these cores for computation-intense workload while the OS only provides infrastructure for I/O or communication and should apart from that not generate any overhead or interference with the computation-intense workload at all. In fact, isolating the OS noise from the HPC application is of such a big concern, that often a full-weight kernel (e.g., Linux for I/O) is combined with a light-weight kernel (LWK) running on different cores of the same machine [9, 17, 23, 30].

The *AAM* concept is orthogonal to any OS architecture, be it monolithic or based on nano-, micro-, macro-, or exokernels. We rely on a monolithic architecture targeting shared-memory systems.

2.2 Missing OS-level Support for Teams

Especially server applications can often be partitioned into distinct computation stages [10, 27] (e.g., receiving, handling, and answering requests). In order to parallelize these stages and utilize available cores, one common approach is to employ thread pools [25, 28]. They are used together with work queues (for the individual jobs) to reduce the frequency of costly thread creations and context switches. However, we argue that thread pools often lack native OS support (just like many operating systems lack support for user-level scheduling in general) and that an adaption of the concept is required to operate many-core systems efficiently. Since an OS like Linux has no notion of thread pools and their work queues, it is unaware of the fact that these threads form a team [5, 6] of equivalent workers and their current workload (i.e., number of jobs). Threads that could process homogeneous workload without interference from other threads are therefore often intermixed with unrelated threads on the same core.

Our Approach: We form groups of homogeneous tasks that are handled by *AAM*s. As *AAM*s and their current workload are known to the OS that can optimize core assignments to these *AAM*s and therefore avoid frequent, expensive transitions between machines and their heterogeneous workload.

2.3 Heavy-weight Threads and System Calls

Switching between kernel-level threads (e.g., because of a blocking system call or synchronization primitives) is expensive, since it is a kernel-space operation [1, 15]. Preemptive scheduling increases the scheduling overhead even further. Furthermore, in a process-based system, those threads may consume a lot of memory for the kernel and user stacks even when they are inactive (e.g., waiting for their first execution). User-level scheduling can eliminate most of these disadvantages. However, without native support from the OS, user-level scheduling is prone to the blocking anomaly [20]. That anomaly occurs when a system call has to block execution and its thread therefore becomes temporarily unavailable for the user-level scheduler and its tasks.

Our Approach: We use light-weight tasks with run-to-completion semantics. To schedule these tasks, we leverage user-level scheduling and lazy context allocation. Therefore, frequent task switches between the short-lived tasks of an *AAM* are performed without involvement of the OS kernel. Lazy context allocation guarantees that a task occupies a stack only when it is actually running or currently blocked. System calls are based on asynchronous requests and waiting for their responses can easily be done in user space without suffering from a blocking anomaly.

2.4 Static Allocation of Resources

A common practice to reduce interference from the OS is to offload system functionality to a number of dedicated OS cores [3, 29]. However, we argue that static partitioning of resources is inherently susceptible to sub-optimal utilization when the distribution of the workload varies over time. Static allocation of cores therefore either leads to a bottleneck for I/O intensive tasks or under-utilized cores.

Similar problems occur when determining the size of a thread pool [25]. Decisions are typically based on assumptions regarding the expected workload of the thread pool and the load in the rest of the system. Often, more threads are allocated than cores are utilized, since threads may become (at least temporarily) unavailable when they are blocked because of a system call or lock. While switching between threads is already an expensive operation, the fixed number of threads represents another problem: It limits the ability to adapt to changing workloads. Since each thread comes with a kernel and user stack, changing workloads may lead to wasted memory, an increased scheduling overhead, or idling cores.

In order to improve locality, applications and runtime libraries tend to pin their threads to specific cores [16]. While this may actually increase cache locality in certain scenarios, an application is only aware of its own threads. Other applications following the same strategy can still interfere and therefore pinning can ultimately even increase contention for resources of a specific CPU (e.g., CPU time, cache, branch predictor).

Our Approach: A dedicated OS component—the *Machine Manager*—is responsible for dynamic core allocation to machines. It is aware of all machines that compete for resources and their current workload. Even the OS implements its services in the form of *AAM*s that offer their functionality via an asynchronous interface.

3 THE *AAM* APPROACH

A system designed according to the *AAM* approach is shown in Figure 1. Applications are composed of one or more *AAM*s, whereas the OS kernel consists of at least one *AAM* and the *Machine Manager*. *AAM*s perform the actual work by scheduling their respective tasks and providing their functionality to other *AAM*s in the form of a task-based interface (cf. Section 3.2). This interface allows *AAM*s to trigger predefined tasks on other *AAM*s. Since *AAM*s are encapsulated components that offer a well-defined interface, *AAM*s can

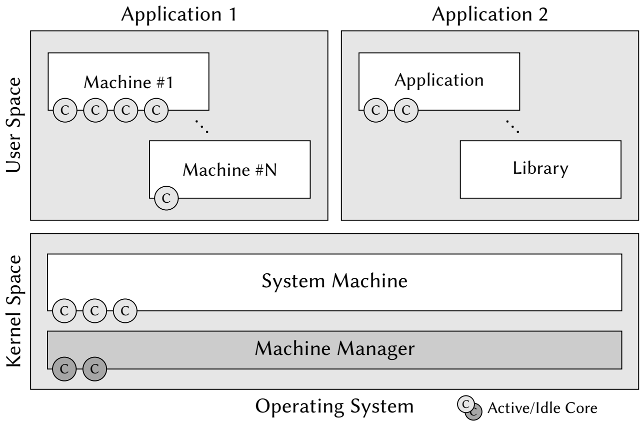

Figure 1: Asynchronous Abstract Machines and their allocated cores

be reused just like libraries. Typically, all *AAM*s of an application reside within the same address space. However, for low-overhead communication via the task-based interface it is sufficient when the server machine can access the address space of the client machine. This property enables one-sided isolation between *AAM*s in the form of nested address spaces and the protection of OS services.

The *Machine Manager* is part of the OS and responsible for resource allocation to *AAM*s and signaling between *AAM*s. Being aware of all machines and applications, the *Machine Manager* is well-prepared to assign processor cores and memory in an efficient manner. While *AAM*s and their interfaces are instantiated dynamically during runtime, their definition occurs at design time. In the current implementation, it is the responsibility of the developer to partition the application into *AAM*s by identifying suitable components. The selection is based on the following considerations:

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

Although the current implementation requires manual partitioning of the system into reusable *AAM*s, we see potential to automatically partition the system into application-specific *AAM*s [11], which is part of future work.

3.1 Components of an *AAM*

Figure 2 shows all important components of an *AAM*. In addition to the machine-specific task implementations, an *AAM* features an independent light-weight task scheduler, a task-based interface, and a dedicated memory allocator to reduce interference between machines and improve locality.

The confined interface between *AAM* and *Machine Manager* allows each machine to implement its own tailored task scheduler and memory allocator, but typically *AAM*s use the optimized default implementation and infrastructure provided by the *AAM* framework. Migrating existing modules or libraries into an *AAM* can therefore be performed with minimal effort and often boils down to specifying the interface.

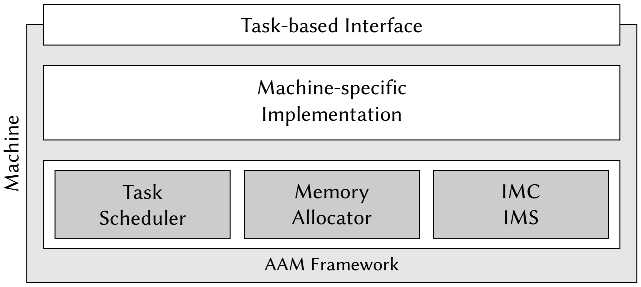

Figure 2: Software components of an Asynchronous Abstract Machine

3.2 Inter-machine Communication (IMC) and Machine Calls

The machine interface allows other machines to perform machine calls and thereby schedule a predefined list of tasks on a remote machine (e.g., the SQLite *AAM* offers tasks for database operations). As efficient communication between machines is essential for the overall system performance, we use task-based interfaces that avoid costly system calls in the common case.

Figure 3 visualizes that regular *inter-machine communication* (*IMC*) is performed directly between two *AAM*s via queues that reside in the address space of the client machine. The server machine may reside in the same or another address space, as it suffices if the server machine has access to the address space of the client machine. This characteristic, for example, allows efficient *machine calls* to OS machines while still preserving isolation between the OS and applications.

The queues of the interface allow to communicate tasks from the client to the server machine and finished tasks, including their results, back from the server to the client machine. An indicator flag per queue allows the sender of a message to identify if the receiver is actively monitoring the queue. If that is not the case, *inter-machine signaling* (*IMS*) is necessary to make sure that the request or response is actually handled. Performing *IMS* involves interaction with the *Machine Manager* in the form of a synchronous system call.

To issue a machine call, the following steps are required: creating a task; enqueuing it into the interface; waiting for the result. However, the task-based interface is actually hidden from the programmer and instead exposed in the form of three interface variants:

- **asynchronous:** returns immediately with a *future* [8] and therefore allows for latency hiding and batching
- **synchronous:** calling task waits for completion and another task is scheduled in the meantime
- **event-based:** schedules a specified task on completion and delivers the result to this task

It is important to note that these different interface variants do not require an enormous redundant implementation effort within the machines (like the individual software subsystems for synchronous IO and AIO in Linux) but instead essentially come for free.

Regarding the ordering of tasks issued to the interface, it is up to the client machine to specify whether the task order should be

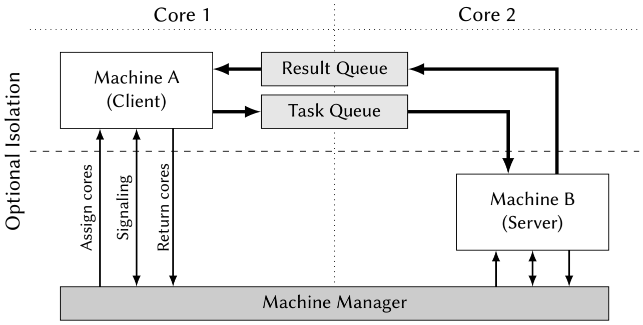

Figure 3: Inter-Machine Communication (thick arrows) and Signaling Mechanisms (thin arrows)

preserved implicitly (e.g., for multiple writes to a file descriptor) or whether it ensures the correct order explicitly on its own. In addition, the server machine may limit the number of tasks that are processed concurrently for a certain client machine.

3.3 Inter-machine Signaling (IMS) and Machine Scheduling

The *Machine Manager* is part of the OS and aware of all *AAM*s in the system. It is responsible for resource allocation and *Inter-machine Signaling* (*IMS*). Interaction with the *Machine Manager*, for example to send a signal, is performed with a few synchronous system calls that have a run-to-completion semantic and therefore come with minimal indirect costs.

IMS allows *AAM*s to notify other *AAM*s of an event and potentially activate them. In particular, *IMS* is used to communicate these events: registration of new interfaces and availability of new messages in queues. Machine signals consist of a signal number that encodes the event type and an arbitrary parameter. They are delivered to the destination *AAM* asynchronously and stored in a machine-local buffer for handling (e.g., between the execution of two tasks). Signal handling does therefore not involve traps or synchronous system calls. However, if the destination *AAM* is currently inactive a machine activation is triggered by the *Machine Manager* which may involve waking a processor core.

Machine scheduling is responsible for the allocation of processing cores to *AAM*s and is guided by two goals: maximizing locality and minimizing interference. As switching between two machines comes with direct and indirect costs and may require the installation of a different address space, frequent transitions between machines on a core are avoided. Instead, *AAM*s typically occupy cores for an extended period of time and schedule several tasks before releasing the core to the *Machine Manager* for reassignment. A task switch, machine call or interrupt does typically not require a machine switch. Instead, a machine switch takes place when a core would start to idle, since the task scheduler ran out of tasks, or when rebalancing of the core allocation becomes necessary. As a side effect, exclusive core allocation may hinder side-channel attacks and improve timing behavior while still keeping the overhead for isolation to a minimum.

To optimize utilization, further reduce the number of machine switches, and enable load-based scheduling decisions, *AAM*s and *Machine Manager* share additional information. Data structures mapped into the address space, e.g., allow the *Machine Manager* to communicate the number of idle cores to *AAM*s and enable *AAM*s to communicate their current workload (i.e., number of tasks ready for execution) to the *Machine Manager*. The *Machine Manager* is thus aware of all *AAM*s including their cores and workload. Together with the knowledge about previous core assignments, it has a global view of the whole system alongside the required insight into machines. We currently evaluate different scheduling strategies and implementations that leverage this knowledge to optimize for non-functional properties like locality, performance, and throughput.

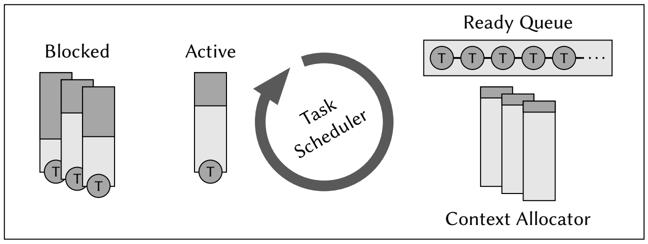

Figure 4: Task scheduling and lazy context allocation within an *AAM*

3.4 Scheduling Tasks inside *AAM*s

Since the *AAM* approach aims for best system performance even in the presence of a large number of short-lived tasks, two requirements become visible: First, the memory footprint of a task should be small because many of these tasks may reside in a queue before actually being executed. Secondly, the scheduling overhead has to be minimized in order to efficiently schedule tasks that run only for a short period of time.

Task scheduling is therefore performed locally inside *AAM*s and without the involvement of the OS kernel. It is based on a lightweight scheduler and lazy context allocation, as shown in Figure 4. A task consists of a handler function, parameters, and a *future* to enable other threads to wait for completion of the task and retrieve a return value. It comes without an execution context (i.e., stack) and thus consumes a minimal amount of memory, when it is awaiting its execution. Instead, a context is assigned to it dynamically when it is first dispatched. Because of the run-to-completion semantic of tasks, only tasks that are in active execution or currently blocked occupy a context. Therefore, it is often possible to reuse the context of the previous task. An additional kernel stack per task is not required at all. The *AAM*-local scheduler is free to accept tasks from the interface queues in alignment with its scheduling strategy.

3.5 User-level vs. Kernel-level Machines

There are only minor differences between user-level and kernel-level *AAM*s. Kernel-level *AAM*s run in privileged mode and therefore have full access to memory and do not require system calls to interact with the *Machine Manager*. To reduce the noise for user-level machines, the *Machine Manager* configures the underlying hardware to direct device interrupts to cores that are either idle or executing a kernel-level machine. This configuration makes sense since interrupts are handled by driver code, which resides in OS machines, and therefore machine switches are avoided.

Like in other operating systems, kernel-level *AAM*s have to validate parameters and pointers that are provided by user-level *AAM*s for security reasons. Additionally, tasks retrieved from an interface have to be copied into kernel space to prevent concurrent modification by user-space *AAM*s. These precautions are omitted when both interacting machines reside in the same isolation domain.

4 IMPLEMENTATION & EVALUATION

The following section describes our prototype implementation of an *AAM*-based system together with tooling support for development and system profiling. Subsequently, we discuss evaluation results in the form of microbenchmarks.

4.1 Prototype System

Our approach has been implemented in a prototype for two architectures: the x86-64 architecture in the form of a native OS and the Linux user space in the form of a 64-bit application binary. The latter uses threads and signals inside a Linux process instead of CPUs and interrupts. While the native variant is suitable for performance measurements, the Linux variant simplifies development and debugging of applications or system components like *AAM*s.

In addition to the implementation of the *Machine Manager* with basic core-assignment strategies and support for *IMS*, the *AAM* framework was developed. Most notably, we implemented a default task scheduler that combines core-local ready queues (i.e., round-robin scheduling) with optional work stealing. The scheduler accepts tasks from interface queues (with a FIFO strategy in accordance with the round-robin scheduling) when no other task is available. The basic idea behind this strategy is locality awareness and reducing noise due to negative cache effects. That is, we focus on finishing short-lived tasks instead of gathering a huge number of tasks that consume resources and interfere with each other.

The introduction of a generic *enqueue handler*, that is always executed after a context switch, allowed us to keep locking in the critical paths of the scheduler to a minimum. This handler is executed in the context of the next task and responsible to either enqueue the previous task into a ready queue or to register it with a synchronization primitive. It therefore makes sure that a task only becomes visible to other cores after it actually ended the execution on its stack and in addition offers implementations of synchronization primitives a generic mechanism to target the lost wake-up problem. Thus, the *enqueue handler* allowed us to implement crucial synchronization primitives like *future* and *counting signal* in a wait-free fashion.

In addition to the implementation of the *AAM* framework and the *Machine Manager*, the following reusable *AAM*s have already been integrated:

- **User-level Machines**
  - SQLite
  - AES Encryption
  - ZLIB/LZO Compression
- **Kernel-level Machines**
  - TCP/IP Stack
  - File System

4.2 Important Tools

During the development of the prototype, multiple tools were created that either simplify the packaging of machines or allow deep insight into the system. With the help of the `igen` tool, the interface of an *AAM* may be specified in a C-compatible `IDL` file. This property allows the reuse of existing interface declarations by including regular C-header files into the `IDL` file. During the build process of a machine, the `igen` tool parses the `IDL` file and generates all the interfaces and data structures required.

The profiling infrastructure of our prototype and tools like `sview` enable us to browse the timeline of events and inspect scheduler behavior. The GUI of `sview` is shown in Figure 5. It visualizes the current load of individual *AAM*s (1), the assignment of cores to machines (2), and how these machines utilize their cores to schedule specific tasks (3). Other profiling tools enable us to collect information from the OS (e.g., the number of machine switches or inter-machine signals) and CPU performance counters (e.g., cache misses or the IPC) to either retrieve per-machine metrics or for deferred processing with Linux tools like `perf`. The insight gained from these tools has proven to be invaluable when designing or debugging scheduling on machine and task level.

4.3 Evaluation Results

Local task scheduling and machine calls are both frequently used operations and key features of our approach. In order to determine the direct costs and the overall latency involved when using these mechanisms, several microbenchmarks have been executed. These microbenchmarks have in common that an application task schedules another `no-op` task and waits for the result of its execution by blocking on a future. The `no-op` task itself does not perform actual work but instead terminates immediately returning a timestamp. It is either executed locally on the same machine (and core) or on a remote machine (and a different core) by issuing a machine call. Each microbenchmark was executed 10000 times and we ensured that no other tasks were active in the system.

The hardware used for our evaluation featured an Intel Xeon CPU (E3-1275 v3 @ 3.50 GHz), 32 GiB RAM, and 4 cores with hyper-threading (8 logical cores), which is more than sufficient for these microbenchmarks. Table 1 shows the direct costs that are perceived by the caller to issue a task for execution (in the form of CPU time) and the overall latency until it is presented with the result of the task execution (using the arithmetic mean and standard deviation). The table distinguishes between the case when both machines are actively monitoring their interface task and result queues throughout the benchmark (no *IMS* required) and the case when both machines are allowed to idle immediately when no tasks are available for execution (*IMS* required). Table 2 serves as orientation and shows the typical costs of frequent system operations on Linux (measured with kernel version 4.4 on the same hardware and all threads/processes pinned to the same core): context switch between two threads of the same process, context switch between different processes, overhead of the system call mechanism, local costs of thread creation.

The following observations are made: (1) Creating a local task is about 3 times faster than the `gettid` system call on Linux. (2) The latency introduced for scheduling a task locally and waiting for its

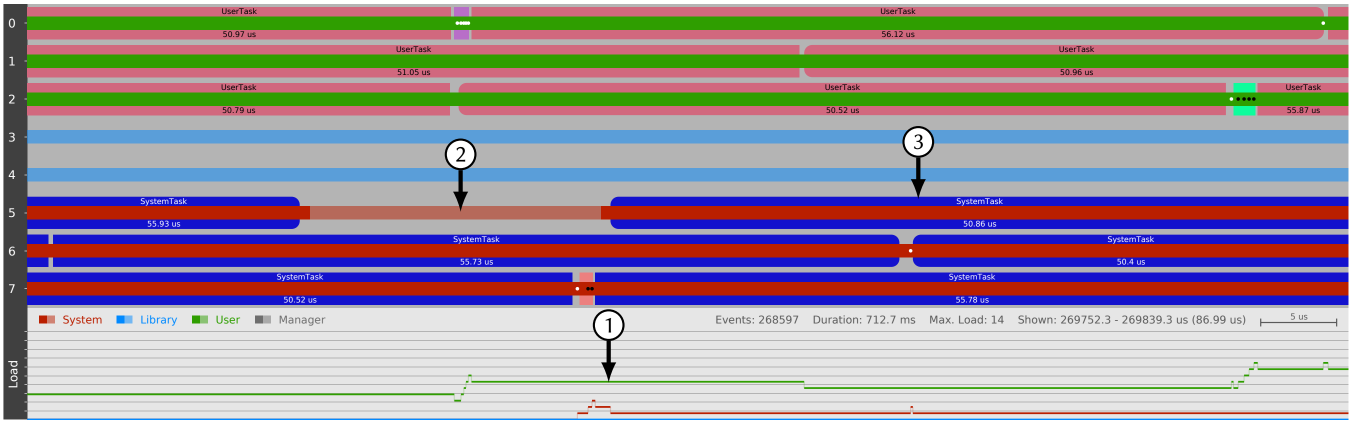

Figure 5: Machine load (1), core assignment (2), and task scheduling (3) over time as visualized by *AAM*'s `sview` tool

Table 1: Direct costs and total latency of frequent operations in *AAM* systems (arithmetic mean and standard deviation)

	Operation	Direct Costs	Total Latency
Active	Local task execution	56 ± 5 ns	277 ± 10 ns
	Machine Call (appl.)	74 ± 23 ns	868 ± 80 ns
	Machine Call (OS)	75 ± 26 ns	882 ± 68 ns
Idle	Local task execution	57 ± 5 ns	278 ± 10 ns
	Machine Call (appl.)	276 ± 10 ns	2808 ± 99 ns
	Machine Call (OS)	294 ± 5 ns	2787 ± 24 ns

result is about 5 times less than the duration of a context switch on Linux. (3) The direct costs for issuing a machine call are only slightly larger than the costs for creating a local task when the machine is active. (4) Issuing a machine call to a machine that is currently not active comes with additional costs in the form of *IMS*. (5) The latency introduced by the machine-call mechanism itself is smaller than the duration of a context switch on Linux but significantly larger than the overhead of a regular synchronous system call. (6) The latency introduced by the machine-call mechanism increases roughly by a factor of 3 when *IMS*[1] is required to wake inactive machines. (7) Since application machines and OS machines share the same mechanisms and have the same overhead for *IMS* right now[1], their performance is almost identical.

We therefore conclude that avoiding indirect costs (by offloading work to a machine on another core) does not come for free and that the latency overhead of the machine-call mechanism increases even further, when the machines are currently inactive. In addition, the actual latency of a machine call in a system heavily depends on the load of the machines. However, depending on the offloaded task, the overhead of the mechanism may actually be negligible and since the core becomes available to the issuing task (or machine) after a short period of time, it might be possible to perform latency hiding or schedule other tasks in the meantime.

Table 2: Typical costs of system operations on Linux

Operation	Duration
Context switch (thread)	1250 ns
Context switch (process)	1725 ns
System call (`gettid`)	174 ns
Thread creation (`pthread_create`)	6980 ns

5 CHALLENGES & FUTURE WORK

Our evaluation in the form of microbenchmarks (cf. Section 4.3) and other synthetic benchmarking applications yielded promising results. Subject of ongoing work is the analysis of *AAMs* to run macrobenchmarks and applications. For example, we have already implemented a key-value store that is based on SQLite and the TCP/IP stack, which we ported and encapsulated into *AAMs* previously. Comprehensive evaluations against related approaches (e.g., Linux) are part of future work.

To enable a fair comparison, we currently work on isolation support, enhanced scheduling algorithms (cf. Section 5.1) and efficient inter-machine communication (cf. Section 5.2). Together with more extensive evaluation scenarios, we will demonstrate and quantify the positive impact of the *AAM* approach on system noise and IPC performance. In the following, we discuss two core challenges that we have identified and will address in future work.

5.1 Interaction between Machine and Task Scheduling

Experiments and insights gained with the aforementioned tooling infrastructure revealed that an overcommitment of cores to a machine may not only result in subpar resource utilization but may actually decrease the overall performance of the machine (depending on the strategy of the task scheduler). We are therefore currently focusing on *utilization-based allocation*, where the *Machine Manager* assigns cores to machines depending on their recent core utilization and the overall load in the system. The task scheduler is aware of all cores that are currently assigned to its machine. It can freely operate these cores (even across short idle phases) until

[1] The current prototype does not implement privilege separation, so the final costs for *IMS* might be slightly higher due to the required mode switch.

the *Machine Manager* reassigns the core to another machine and notifies the task scheduler of the machine with an *IMS*.

Further, the joint use of user-level scheduling and machine scheduling leads to two different levels of scheduling within the system. The existence of two scheduling levels bears challenges and opportunities (i.e., increased design space) likewise. We consider the following key issues to be most relevant for *AAM*s:

- When to trade noise reduction and task locality for core utilization?
- How to maintain fairness between multiple applications regarding the core assignment to their machines and regarding system tasks?
- Where to position machines within a non-uniform memory access (NUMA) system?
- How to handle shortages of cores?
- Which information is shared between the two schedulers?

Besides our efforts to achieve swift multi-level scheduling by addressing the above challenges, we also work on improvements for *IMC* and *IMS* between *AAM*s that run on individual CPU cores.

5.2 Efficient *IMC* and *IMS* between Cores

As *IMC* performance is crucial for an *AAM*-based system, its scalability and the latency introduced by the communication mechanism itself is important. Scalability is obtained by adding more interface task and result queues as this reduces contention on the respective queues. However, communication and signaling latencies are also limited by the hardware. Whenever *IMS* is required (e.g., as an interface queue is not monitored), a synchronous system call and possibly an inter-processor interrupt is required to wake the receiving machine. Currently, we minimize the chances that *IMS* is required by keeping task and result queues of the interface in the pool of monitored queues for a limited time even when no messages are retrieved from the interface.

Performance improvements are often achieved by utilizing hardware mechanisms that accelerate necessary system-level operations. For example, synchronous system calls benefit from hardware support in the form of specialized CPU instructions for the required mode switch (e.g., `sysenter`). In the context of *AAM*s, a hardware-accelerated queue with advanced signaling capabilities is beneficial. On the one hand a hardware-accelerated queue can speedup *IMC*, on the other hand it can avoid *IMS* under most circumstances. Here, we benefit from our prior work on hardware-accelerated queues [19]. Such a specialized queue is characterized by the following features: the queue is software-defined in the form of a queue descriptor that resides in main memory (i.e., the address space of the destination machine). The enqueue and dequeue operations on the hardware-accelerated queue are allowed even across isolation domains, that is, enqueueing to *AAM*'s interface queues that reside in different address spaces. Furthermore, the queue's automatic signaling mechanism ensures that the destination machine is activated for processing tasks if required. To summarize, by reducing the cost of *IMC* and avoiding *IMS* with hardware-accelerated queues, *AAM*'s costly operations can be mitigated.

6 RELATED WORK

Native OS support for user-level scheduling and mitigations for blocking anomalies are addressed by several approaches [7, 15], most notably by *scheduler activations* [1]. *AAM* builds upon these insights and adds native OS support for task groups within user and kernel space to reduce interferences between heterogeneous workloads (e.g., the OS noise perceived by application tasks).

FOS [29] and Corey [3] are operating systems based on space sharing that focus on scalability of system services on future many-core systems. Their static or explicit allocation of cores to OS services and applications allows the reduction of interference between workloads. In contrast to *AAM*s, these approaches do not exploit the potential for further core specialization *within* applications and the static core allocation can negatively impact the utilization of available computing resources, too.

Solutions like exception-less system calls for Linux, as proposed in FlexSC [24], and the NIX [2] approach, applied to the Plan 9 [18] OS, allow dynamic specialization of cores in OS or application tasks. Both approaches use message-based system calls that leverage shared-memory architectures and therefore implement a similar interface as *AAM*. But unlike *AAM* and limited by the existing system architecture neither of these approaches introduces a generic concept for the management of heterogeneous workloads that offers further specialization of cores within applications or the OS.

Computation spreading [4] introduces a broader vision of specialization. It is based on thread migration and allows for additional core specialization within user and kernel space. However, the implementation of computation spreading requires hardware support and is limited to separation of OS and application tasks.

In contrast to the previous approaches, *cohort scheduling* [10] and *staged event-driven architectures* [27] target core specialization in user space only, by either scheduling tasks of distinct computation stages in cohorts or on dedicated cores. *AAM*s and stages share the focus on executing homogeneous workload by performing local scheduling and queue-based interactions. However, stages are limited to the user space, they use a thread pool, and they lack native OS support.

Tessellation [13] focuses on space-time partitioning by subdividing the system into partitions that either execute applications or system services and by enforcing message-based communication between partitions. The partition manager—a privileged component—is responsible to assign cores to individual partitions for extended periods of time. While partitions have similarities with *AAM*s as they possess their own scheduler, our approach actually decouples resource management (i.e., core assignment) from the address space. This allows multiple *AAM*s to exist within a single address space. Thus, further specialization of application cores without introducing additional overhead for communication can be achieved. Cores within Tessellation partitions are guaranteed to be scheduled simultaneously, a property that can be enforced for *AAM*s by the *Machine Manager*. However, this is not inherently guaranteed or required. Furthermore, an *AAM*-based system typically features one or more privileged OS machines whereas Tessellation offers system services in the form of unprivileged user-space partitions.

HPC systems commonly isolate OS noise from application workloads by dividing up available cores of the machine. A subset of cores is assigned to a full-weight kernel for I/O (e.g., Linux) while the remaining cores are utilized by a light-weight kernel that is dedicated to the HPC application code [9, 17, 23, 30]. Such approaches facilitate the use of specialized kernels that are strictly optimized for HPC workloads while keeping full I/O support and minimizing the effort to support the latest hardware platforms. *AAM*s are a compelling alternative to the coexistence of two operating systems side-by-side, as our approach inherently provides an isolation of the system noise and offers the strong flexibility of custom user-level scheduling within individual machines.

Finally, *thread clustering* [26] schedules threads based on sharing patterns. Such patterns are identified during runtime by leveraging hardware performance counters of the CPU. With *AAM*, the system architect identifies tasks and self-contained subsystems with homogeneous workloads during design time and specifies machines accordingly. Only the instantiation of these machines and the resource allocation to these machines occurs dynamically at runtime.

7 CONCLUSION

In this paper, we presented a system design concept that is based on *Asynchronous Abstract Machines*. *AAM*s offer native support for dynamic core specialization to reduce system noise. It is motivated by the observation that cores are often shared between heterogeneous workload and therefore operated inefficiently because of interference. It targets the following shortcomings of existing systems:

- missing OS-level support for teams
- heavy-weight threads and system calls
- static allocation of resources

To accomplish the goal of dynamic core specialization within user and kernel space, we propose *AAM*s that focus on a limited set of homogeneous tasks that share a lot of code and data. *AAM*s feature local task scheduling and a task-based interface to interact with other *AAM*s efficiently. A dedicated OS component assigns cores to *AAM*s depending on their current workload. I/O and other functionality of the OS is provided by dedicated kernel-level *AAM*s. This approach to anti-noise system design had been implemented in a prototype and an evaluation showed promising results.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Council (DFG) under grant no. CRC/TRR 89 ("InvasIC", Project C1) and grant no. SCHR 603/8-2 ("LAOS").

REFERENCES

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. 1992. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 53–79.

[2] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador. 2012. NIX: A Case for a Manycore System for Cloud Computing. *Bell Labs Technical Journal* 17, 2 (2012), 41–54.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, and Z. Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. 43–57.

[4] K. Chakraborty, P. M. Wells, and G. S. Sohi. 2006. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*. 283–292.

[5] D. R. Cheriton. 1984. The V kernel: A Software Base for Distributed Systems. *IEEE Software* 1, 2 (1984), 19.

[6] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. 1979. Thoth, a Portable Real-Time Operating System. *Commun. ACM* 22, 2 (1979), 105–115.

[7] J. Edler, J. Lipkis, and E. Schonberg. 1988. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*. 1–17.

[8] D. P. Friedman and D. S. Wise. 1976. *The Impact of Applicative Programming on Multiprocessing*. Technical Report. Indiana University, Computer Science Department.

[9] B. Gerofi, Y. Ishikawa, R. Riesen, R. W. Wisniewski, Y. Park, and B. Rosenburg. 2016. A Multi-Kernel Survey for High-Performance Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*. Article 5, 5:1–5:8 pages.

[10] J. R. Larus and M. Parkes. 2002. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC '02)*. 103–114.

[11] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '17)*. 285–298.

[12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. 973–990.

[13] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. 2009. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*. 1–6.

[14] LWN. 2017. The Current State of Kernel Page-Table Isolation. (2017).

[15] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. 1991. First-Class User-Level Threads. *ACM SIGOPS Operating Systems Review* 25, 5 (1991), 110–121.

[16] A. Mazouz, S. Touati, and D. Barthou. 2011. Performance Evaluation and Analysis of Thread Pinning Strategies on Multi-Core Platforms: Case Study of SPEC OMP Applications on Intel Architectures. In *Proceedings of the International Conference on High Performance Computing Simulation (HPCS '11)*. 273–279.

[17] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski. 2012. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing*.

[18] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. 1995. Plan 9 from Bell Labs. *Computing Systems* 8, 2 (1995).

[19] S. Rheindt, S. Maier, F. Schmaus, T. Wild, W. Schröder-Preikschat, and A. Herkersdorf. 2019. SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures. In *Proceedings of the 19th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '19)*.

[20] Y. Seo, J. Park, G. Jeon, and S. Hong. 1999. Supporting Preemptive Multithreading in the ARX Real-Time Operating System. In *Proceedings of the IEEE Region 10 Conference (TENCON '99)*. 443–446.

[21] N. Shenoy. 2018. Firmware Updates and Initial Performance Data for Data Center Systems. https://newsroom.intel.com/news/firmware-updates-and-initial-performance-data-for-data-center-systems/

[22] N. Shenoy. 2018. Intel Security Issue Update: Initial Performance Data Results for Client Systems. https://newsroom.intel.de/chip-shots/intel-security-issue-update-initial-performance-data-results-client-systems/

[23] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa. 2014. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs Targeting High Performance Computing on Manycore Architectures. In *Proceedings of the 21st International Conference on High Performance Computing (HiPC '14)*. 1–10.

[24] L. Soares and M. Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '09)*. 33–46.

[25] M. D. Syer, B. Adams, and A. E. Hassan. 2011. Identifying Performance Deviations in Thread Pools. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*. 83–92.

[26] D. Tam, R. Azimi, and M. Stumm. 2007. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd European Conference on Computer Systems (EuroSys '07)*. 47–58.

[27] M. Welsh, D. Culler, and E. Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 230–243.

[28] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. 2000. *A Design Framework for Highly Concurrent Systems*. Technical Report. University of California, Berkeley.

[29] D. Wentzlaff and A. Agarwal. 2009. Factored Operating Systems (FOS): The Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 76–85.

[30] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. 2014. mOS: An Architecture for Extreme-Scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*. Article 2, 2:1–2:8 pages.