The Case for an Interwoven Parallel Hardware/Software Stack

Kyle C. Hale*, Simone Campanoni†, Nikos Hardavellas†, and Peter A. Dinda†
khale@cs.iit.edu, simonec@eecs.northwestern.edu, nikos@northwestern.edu, pdinda@northwestern.edu
*Illinois Institute of Technology
Chicago, IL, USA
†Northwestern University
Evanston, IL, USA

***Abstract*—The layered structure of the system software stacks we use today allows for separation of concerns and increases portability. However, the confluence of widely available virtualization and hardware partitioning technology, new OS techniques, rapidly changing hardware, and significant advances in compiler technology together present a ripe opportunity for restructuring the stack, particularly to support effective parallel execution. We argue that there are cases where layers, particularly the compiler, run-time, kernel, and hardware, should be interwoven, enabling new optimizations and abstractions. We present four examples where we have successfully applied this interweaving model of system design, and we outline several lines of promising ongoing work.**

***Index Terms*—interweaving, layering, operating systems, compilers**

I. Introduction

Advances in virtualization and hardware partitioning now make it possible for a single system to securely host multiple software stacks simultaneously [9], [59], [38], [39]. Similarly, in some high performance computing and cloud environments, systems with diverse software stacks can securely coexist side-by-side through network traffic segregation. Containerization simplifies deployment, and fast boot technologies allow us to think of an entire software stack in much the same way we have thought about processes in the past. The result of these advances has been an explosion of innovation across the software stack, abetted by concomitant hardware advances. We are no longer bound by the commodity software/hardware stack. Simply put, we can put whatever stack we want *next to* the commodity stack.

We have been working to leverage these advances to improve the state of software/hardware stacks *specifically for parallel programs*. As we know from the 50+ year history of parallelism, parallel programs are quite different from sequential and ordinary concurrent programs in terms of languages, patterns, collective behavior, and execution models. Despite these profound differences, today it is often assumed that only small variations in the commodity software/hardware stack are necessary to support parallelism well. At the same time the commodity software/hardware stack has grown quite rigid, which restricts the imagination of the designers of parallel systems including languages, compilers and run-times, and limits them to a constrained design space. These restrictions are already limiting today and will become hard limiters as exploiting parallelism becomes ubiquitous, the scale of the necessary parallelism expands, and energy efficiency becomes increasingly important.

a) Example Limitations: Current hardware/software stacks for parallelism require virtual memory in the form of paging, which then demands the existence of TLBs and other hardware structures. These in turn have substantial overheads in time and energy. This issue is not limited to parallel systems. Indeed, there is a cottage industry of work on addressing the limitations of paging in the general systems and architecture communities as well. Another limitation of current hardware/software stacks for parallelism is that all memory is kept cache-coherent by hardware means. When a parallel language implementation does not require this, there is a substantial toll on performance and energy. Yet another limitation is that events in current hardware/software stacks are based ultimately on hardware timer interrupts, but these exist at a considerable remove from the parallel runtime because of the need to cross a kernel/user boundary multiple times. The extra latency and overhead this introduces artificially limits the granularity of action in a parallel runtime system.[1]

b) Big Picture: Our focus is on layering, in particular the layers of language, compiler, run-time, operating system, and hardware. While not impermeable, the existence of these layers inherently warps the design of a parallel system. Another critical aspect is the notion of privileged (e.g., kernel) versus unprivileged (e.g., user) execution, which in turn is a major enforcer of layering. This notion also means that a substantial portion of the hardware functionality is unavailable to all but the kernel, and that higher layers are forced to use the kernel's own abstractions.

c) Interweaving Model: We have been investigating how these layers of a parallel system can and should change, particularly when the privileged/unprivileged distinction is

This project is made possible by support from the United States National Science Foundation through awards CNS-1718252, CNS-1763612, CNS-1763743, CNS-1730689, CCF-1757964, CCF-2029014, CCF-2028851, CCF-2028958 and CCF-1453853.

[1]Sections IV and V give examples of our interweaving work that address the example limitations we note here.

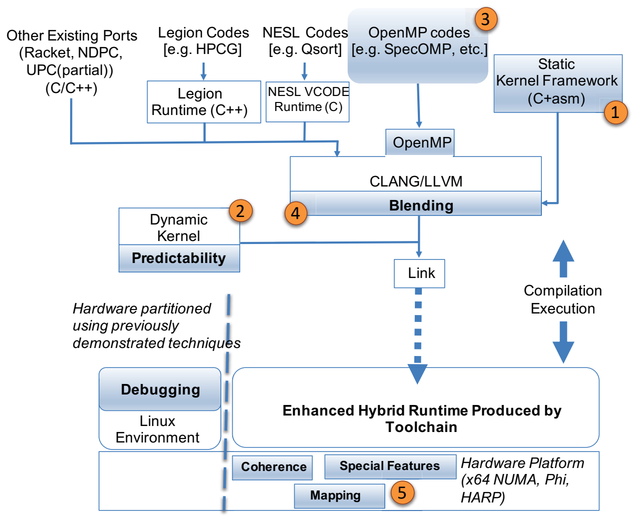
Other Existing Ports (Racket, NDPC, UPC(partial)) (C/C++)
Legion Codes [e.g. HPCG]
NESL Codes [e.g. Qsort]
OpenMP codes [e.g. SpecOMP, etc.]
3
Static Kernel Framework (C+asm)
1
Legion Runtime (C++)
NESL VCODE Runtime (C)
OpenMP
CLANG/LLVM
Blending
4
Dynamic Kernel
2
Predictability
Link
Compilation
Execution
Hardware partitioned using previously demonstrated techniques
Debugging
Linux Environment
Enhanced Hybrid Runtime Produced by Toolchain
Coherence
Special Features
Mapping
5
Hardware Platform (x64 NUMA, Phi, HARP)

Fig. 1. Example system.

removed. The *interweaving model*, a particular approach to layering, is our approach. Here, the specific functionality needed by a specific parallel system (or even application) is implemented by integrating ("interweaving") in a custom manner functionality formerly kept distinct at each layer. The custom software/hardware stacks possible in the interweaving model have the potential to lead to higher performance, more scalability, and more energy efficiency.

Figure 1 illustrates the compile-time and run-time structure of the system we are building as we pursue our research agenda. We build upon Nautilus, a static (1) and dynamic kernel framework (2), which we describe in the next section. Nautilus provides predictable behavior through a variety of means, including hard real-time scheduling. We use the Clang/LLVM compiler toolchain to build both Nautilus and the parallel application, with OpenMP applications (3) being an important current target. Using custom code transformation passes, we blend the code of the application and the code of Nautilus at a low-level (4), including below the level of individual functions. Other compiler transformations add code that allows us to solve traditional problems, such as protection and mapping of memory, and timing, using alternative means. Finally, the generated code can take advantage of specialized hardware features such as FPGA-based operators, or relaxed coherence, that emerging hardware (5) can provide and that higher-level parallel languages can exploit.

Given this setting and the overall interweaving model, we have developed alternative approaches to a range of systems problems, which we elaborate on. These provide some evidence that the interweaving approach has legs.

II. Related Work

While the HPC community has been reconsidering operating system design for tightly-coupled parallel computing for decades now [45], [30], [48], [8], the strict separation between layers of the stack has remained largely stagnant, especially at the user/kernel boundary.

Multi-kernels [63], [28], [61], [75], [7], [37], [27] attempt to strike a middle ground between general-purpose system software and specialized OSes by space-sharing OSes across a system, but leave opportunities for co-design across layers on the table.

In the cloud landscape, Unikernels, aided by ubiquitous virtualization, allow for high performance for a specific target set of workloads [46], [54], [69], [74], [60], [14]. Their success has bled into other areas, including serverless computing [47] and high-performance computing [50]; a Unikernel target for Linux is now in the works [66].

Some Unikernels are constructed from application code using a high-level language [55], a natural progression from classic library OSes [24]. As more sophisticated systems languages like Rust come to prominence, decade-old ideas on using language features to provide or enhance kernel mechanisms like protection or isolation [10], [41], [64] are resurfacing in the form of OSes and Unikernels like Theseus [13], RedLeaf [58], and RustyHermit [49]. However, the compiler is left out of the loop here; we argue that there is significant opportunity for bringing compiler technology and co-design across layers to bear for efficient parallelism.

III. Background: Nautilus

We build our exploration of Interweaving on Nautilus [35], a publicly available, open-source OS kernel[2] that currently runs directly on x64 NUMA hardware, including Xeon Phi. Nautilus comprises over 331K lines of code and Nautilus was designed with the goal of supporting hybrid run-times (HRTs). An HRT is a mash-up of an lightweight OS kernel framework, such as Nautilus, and a parallel run-time system [34], [33]. Nautilus can help a parallel run-time ported to an HRT achieve very high performance by providing streamlined kernel primitives such as synchronization and threading facilities. It provides the minimal set of features needed to support a *tailored* parallel run-time environment, avoiding features of general purpose kernels that inhibit scalability.

Nautilus has a range of features that help make the execution of an HRT faster and more predictable. Identity-mapped paging with the largest possible page size is used. All addresses are mapped at boot, and there is no swapping or page movement of any kind. As a consequence, TLB misses are extremely rare, and, indeed, if the TLB entries can cover the physical address space of the machine, do not occur at all after startup. There are no page faults. All memory management, including for NUMA, is explicit and allocations are done with buddy system allocators that are selected based on the target zone. For threads that are bound to specific CPUs, essential thread (e.g., context, stack) and scheduler state is guaranteed to always be in the most desirable zone. The core set of I/O drivers developed for Nautilus have interrupt handler logic with deterministic path lengths. Finally, interrupts are fully

[2]https://github.com/hexsa-lab/nautilus

steerable, and thus can largely be avoided on most hardware threads. Application benchmark speedups from 20–40% over user-level execution on Linux have been demonstrated, while benchmarks show that primitives such as thread management and event signaling are orders of magnitude faster [35], [36]. This background description is reproduced from our prior work, and more details can be found there [35], [29].

IV. Interweaving Examples

Below we describe several completed and ongoing efforts within the larger Interweaving umbrella that we believe demonstrate the promise of our approach.

A. Compiler- and Runtime-based Address Translation (CARAT)

Nautilus has no protection mechanisms or kernel-user distinction, by design, and, in fact, has a single-address space with identity mapping between physical and virtual addresses using the largest possible page size. The result is that a parallel program on top of it faces no compromises or surprises in terms of TLB misses. Can we add the benefits of virtual memory back into the equation without losing these benefits? Can the benefits of virtual memory be achieved without paging and the concomitant entanglement of cache design and TLBs?

We have developed a technique in which compiler-based analyses and transformations of existing code at the LLVM IR level make it possible to achieve both protection and mobility of data without any hardware support—all code runs using physical addresses. This result frees hardware architects from constraints that might limit them in the search for highly-efficient platforms.

Simply put, our analyses identify the subset of memory accesses and allocations that need to be checked at run-time while our transformations add tracking and protection checks to them. We do so while scaling to the entire codebase, including the kernel. Conceptually, protection check code is introduced at each read or write, and data movements operate similarly to a garbage collector in a managed language. However, our techniques apply, with few limitations, to any code that can be compiled to LLVM. An important result is that we can demonstrate that it is possible to massively reduce the potentially high costs of the compiler-introduced protection and tracking code in most cases. This is because modern code analysis techniques can provide the information necessary to aggregate and hoist protection and tracking code, thus taking it out of the critical path in most instances. This is particularly true for parallel codes from a range of benchmarks from NAS [43], Mantevo [6], and PARSEC [11], where the overheads are $<6\%$ (geometric mean). An additional benefit of this approach to virtual memory is that memory can be managed at arbitrary granularity, instead of being restricted to page sizes. Details of our approach and our evaluation can be found elsewhere [72].

We have also built an enhanced version of this technique within Nautilus with support for separate compilation. Based on the PIK model (Section V-A), a Linux user-level program can be compiled, transformed, linked, and cryptographically attested such that it can run as a part of Nautilus, at kernel-level, using physical addresses, in a simulacrum of a process. The "process's" tracking code and protection code directly interacts with Nautilus, and Nautilus can perform per-"process" and whole system memory defragmentation.

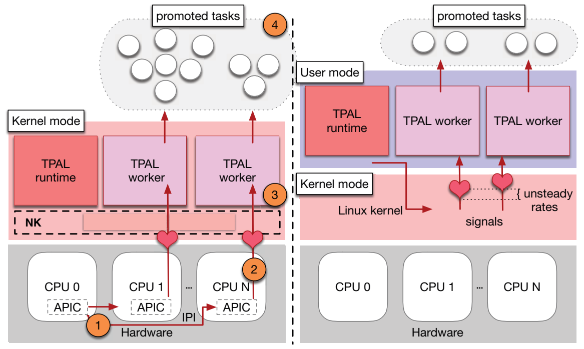

Fig. 2. Heartbeat signaling mechanisms in Nautilus (left) and Linux (right). This figure is reproduced from our prior work [65].

B. Low-Overhead Event Notifications for Heartbeat Scheduling

Heartbeat scheduling [2] is a recently proposed technique for scheduling recursively parallel task-based programs within a work-stealing model, for example, Cilk [12], [26]. The essential idea is that the programmer exposes all available parallelism in the program, and then compiler-based techniques are used to generate both parallel and sequential variants at all levels. The runtime system dynamically promotes sequential code to the parallel variants as needed to "right-size" the extant parallelism, and it can do so in a sound manner that provides provable bounds on performance of the algorithm. The runtime is periodically triggered by a "heartbeat" event that is ultimately caused by a hardware timer interrupt.

The event mechanisms available in Linux were not designed for the purpose of driving heartbeat interrupts at fine granularity (typically $\heartsuit = 20\mu s$—$100\mu s$ or smaller) at the scale of even a tiny modern machine (e.g., 16 CPUs). As others have shown [36], existing software mechanisms in Linux are unable to achieve predictably low latencies for out-of-band event signaling—these are nowhere near the latencies, rates, and jitter that the underlying hardware is capable of.

In collaboration with the heartbeat scheduling team, we developed a Nautilus-based HRT that uses the same compilation process as the user-level implementation, but does signaling directly using the x64 hardware, and thus can achieve the lower limit on architected, out-of-band event signaling the hardware is capable of. Figure 2 illustrates the different heartbeat mechanisms in Linux and Nautilus. TPAL is the name of the heartbeat compilation and run-time system. In the Nautilus implementation, a LAPIC timer interrupt on CPU 0 (1) is broadcast via IPI (2) to the TPAL workers on other CPUs (3), which in turn promote latent parallelism (4).

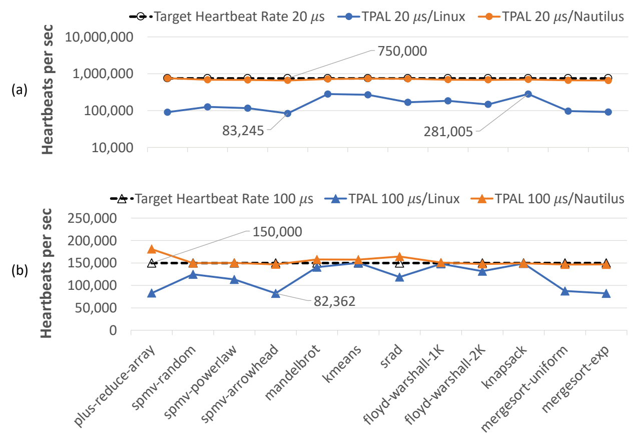

Fig. 3. Achieved and target heartbeat rate in Nautilus and Linux. This figure is reproduced from our prior work [65].

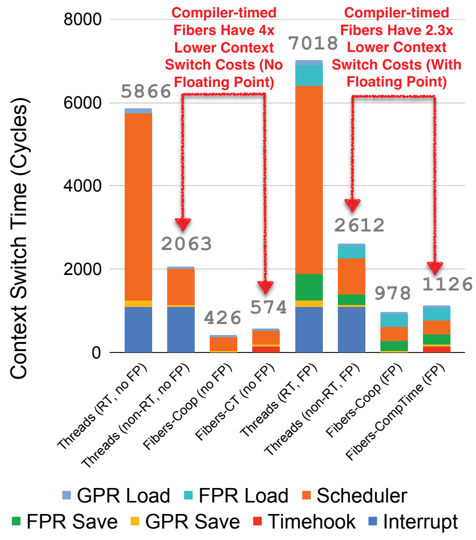

Fig. 4. Cost of context switches for real-time and non-real-time threads, fibers, and compiler-timed fibers on specialized kernel on Phi KNL. Linux non-real-time thread context switches with FP state take about 5000 cycles on this platform. Compiler-timed fibers can achieve a >4x lower granularity. Reproduced from our prior work [29].

Across a range of benchmarks, the scheduling overheads are 13–22% on Linux, and reduce to at most 4.9% in Nautilus. Furthermore, while the best Linux mechanism cannot sustain heartbeat signals at a consistent rate for all benchmarks, even at $\heartsuit = 100$ μs and a scale of 16 CPUs, Nautilus not only hits the target, but it also delivers a consistent, stable rate at both 100 μs and 20 μs (Figure 3). More details can be found elsewhere [65].

C. Compiler-based Timing for Fine-grain Preemptive Parallelism

Fully exploiting a modern machine of any kind depends on extracting and leveraging parallelism across a wide range of granularities ranging from below the level of individual instructions to beyond the level of independent, long-running jobs [56], [23], [1]. Future machines are likely to further expand this requirement [52], [20], [25]. Fine-granularity parallelism is of significant interest within the HPC community (e.g., OpenMP tasking [5]), and is an expectation of theoretically well-grounded parallelism models for higher-level parallel languages [12], [26], [40], [2]. Others in this community have pointed out the increasing importance of *granular computing* [51].

Preemptive threads are a natural abstraction for many of these uses, but, unfortunately, due to their high overheads, are not generally used. The high cost of preemptive threads is due in large part to the high costs of handling hardware timer interrupts. Even when there is no kernel/user boundary to cross, these are consequential. Thus, using preemptive threads puts too high a bound on the parallel granularity that can be achieved. Instead, these systems are based on callbacks, with all the challenges this entails.

The fundamental issue here is that timing is based on a hardware/software co-design—a hardware timing device drives a hardware interrupt dispatch mechanism, which leads into a software-based timing framework, which leads to a software-based context switch. What if we replace this with a software/software co-design involving the compiler toolchain and the kernel? Compiler-based timing does exactly this, and our design, implementation, and evaluation is described in more detail elsewhere [29].

In compiler-based timing, the entire codebase of the system, including the kernel itself, is processed using modern compiler analyses and transformations to introduce `call`s into the timer framework that replace hardware timer interrupts. The timer framework can in turn induce thread context switches. Because the timer framework is now invoked with the overhead of a `call` instruction instead of the overhead of an interrupt, it can be invoked more often within the bounds of some limit on overhead. Furthermore, because no interrupt context is involved, the design of threads can be considerably simplified and sped up. In fact, they become fibers, with "preemption" provided by `yield()`s executed by the timer framework. These elements combine to reduce the overhead of threads.

Figure 4 illustrates the benefits on a Intel Phi KNL chip. For comparison, a (non-real-time (non RT)) Linux user-level thread context-switch, including floating point state, takes about 5000 cycles on this platform. Our kernel's (non-RT) thread context switch using hardware timers ("Threads (non-RT, FP)" in the figure) is about half that. Using compiler-based timing, it is slightly more than halved again ("Fibers-CompTime (FP)" in the figure)[3] . As a consequence, our system can support preemptive threads with granularities that are over four times smaller than those possible in the commodity Linux environment. The granularity limit on this machine is less than 600 cycles, which is so low that floating point state

[3]The remaining bars of the figure illustrate other options in the parameter space of {RT, non-RT} x {Threads, Fibers } x {Cooperative,Compiler-timed}.

management becomes the bottleneck.

A major challenge here is that the compiler transform needs to introduce timing calls *statically*, so that they occur *dynamically* at some desired rate regardless of the code path taken through the kernel+application ensemble as it runs. Modern compiler analysis makes this possible.

D. Function-Granularity Virtualization

The need for systems support for fine-grained tasks is increasing [51], yet cloud systems that support on-demand scheduling of such tasks (e.g. using the Function-as-a-Service model (FaaS) with serverless computing [68], [42], [19], [71], [32]) often rely on legacy software stacks. To support low-latency startup for such tasks, aggressive snapshotting is applied [17], [22].

We have investigated low-latency, isolated execution of individual tasks and functions by applying the Interweaving model to build such execution contexts from the ground up, culminating in an abstraction that we call *virtines*. Virtines execute in isolated, virtualized environments using a custom software stack (e.g. a Unikernel or minimal runtime shim layer). They are enabled by a microhypervisor (Wasp) and using custom LLVM compiler support. Programmers write code as shown in Figure 5, and the compiler and runtime cooperate to run that function in its own, isolated virtual machine with start-up overheads as low as 100μs.

```
virtine int fib(int n) {
  if (n < 2) return n;
  return fib(n - 1) + fib(n - 2);
}
```

Fig. 5. Virtine programming in C with compiler support.

Our virtine microhypervisor runs as a user-space process on either Linux or Windows using KVM or Hyper-V, respectively, to leverage hardware-specific virtualization features. Other applications (including dynamic compilers and runtime systems) can link with the runtime library to leverage virtines. Our virtine framework can be used with existing code with minimal changes, and with acceptable overheads.

V. Interweaving Next Steps

We now describe several ongoing and future efforts that push the Interweaving model further.

A. OpenMP

We are in the process of completing our implementation of a kernel-level OpenMP as shown in Figure 1. OpenMP involves increasingly complex language, compiler, and runtime support to make it possible to express and exploit node-level parallelism. The result is that the OpenMP run-time system is increasingly looking *like* a kernel, and we are interweaving it with the Nautilus kernel framework so that it *becomes* the kernel.

Three different approaches have been designed and implemented, all of which leverage OpenMP code generation (e.g., `-fopenmp`) in Clang/LLVM. The first, *runtime in kernel*

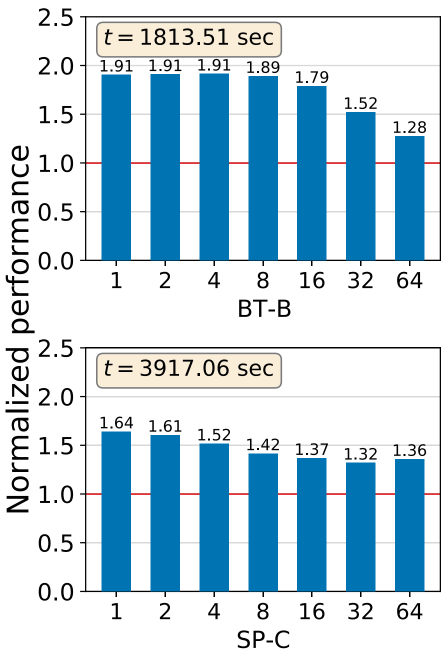

Fig. 6. RTK performance relative to Linux as a function of CPUs used: NAS BT and SP on Phi KNL; higher is better. Baseline (Linux OpenMP) is horizontal bar at 1.0. t is the single threaded Linux absolute performance. This figure is reproduced from our prior work [53].

(RTK), involves a port of the libomp OpenMP runtime to the kernel, allowing any kernel code to use OpenMP pragmas. To use this approach, OpenMP applications must also be ported to the kernel. The second, *process in kernel* (PIK) involves a specialized process abstraction that allows running Linux user-level OpenMP code (and all of its various libraries and runtimes), within the kernel. The code believes it is executing in a traditional process environment in user-mode, but it is actually a part of the kernel, running in kernel-mode. To use PIK, applications need to be recompiled and linked in a specialized manner, but otherwise no porting is needed. The third approach, *custom compilation for kernel* (CCK), involves compiling OpenMP pragmas (and doing automatic parallelization) into a form that leverages the kernel framework without any intermediary. Compilation is porting. Unlike RTK and PIK, CCK always targets a purely task-based execution model, which we map directly to the task framework within Nautilus, which can be viewed as a Linux-like SoftIRQ framework. Unlike SoftIRQs, however, if the compiler can estimate task size, its tasks can be run in the scheduler itself, even in interrupt context.

All three implementations can run the full Edinburgh OpenMP microbenchmarks [15], [16] and the NAS parallel benchmarks [43]. Figure 6 gives example results for the NAS BT and SP benchmarks. The average performance gain of RTK over Linux OpenMP on Phi KNL across all scales and benchmarks is 22% (geometric mean). PIK performs similarly. A repetition of the study on an 8 socket, 192 core

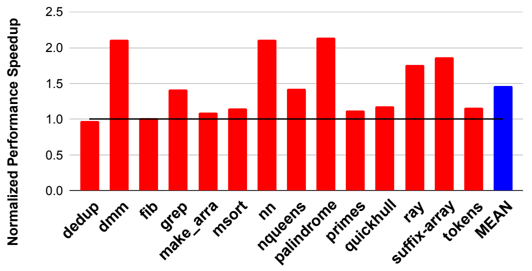

Fig. 7. Speedup for a dual-socket x64 server with our selective coherence deactivation feature driven by a high-level parallel language run-time. (2× 3.3 GHz 12-core CPUs with 32K/256K/2.5M L1/L2/L3 cache structures.)

machine found similar results (∼20% for RTK and PIK). CCK performance is not easily summarized.

More information about this work can be found in a separate paper [53] that will appear at the SC '21 main conference (hopefully contemporaneously with this ROSS paper.)

B. Coherence and Consistency

The one-size-fits-all approach in today's memory consistency and cache coherence models creates unnecessary constraints that hinder performance. Ordering constraints in consistency models serialize *all* accesses of a particular type, without selectivity. A fence orders writes that produce data before setting the `done` flag, but it also orders all other writes the thread issued, even if they are unrelated to the intended use of the fence. Individual writes within a producer's data production subroutine could semantically proceed in any order, yet x86-TSO [62] unnecessarily enforces a total order. Thread-private data are tracked in the coherence protocol, even though there are no other sharers for the data [21]. Producers and consumers keep stealing each other's cache lines and transfer them across the interconnect, only for them to be stolen back, blindly following the rules of today's reactive coherence protocols, while involving even a third node (the directory) that is often located far away from the producer/consumer cores.

We envision a system that is free from these inefficiencies, where information on parallelism, data sharing, and memory ordering requirements flows from the higher levels of the stack (e.g., high-level programming languages) to the lower levels. Armed with knowledge of the programmer's intent, the compiler, runtime, OS and architecture can decouple data with different requirements from the rest of the data space, and steer their behavior proactively by instructing the hardware to apply specialized memory ordering rules, data sharing mechanisms, and mapping primitives for on-chip data placement. We are currently working toward realizing this goal in a system as shown in Figure 1.

We have developed and are in the process of testing a hardware cache coherence protocol that extends the currently used MESI protocol with support for selective coherence deactivation. This support has been designed with high-level parallel languages in mind, though it is not restricted to them. Via an implementation of the protocol in Sniper [18], we are able to simulate current and future x64 machines of different structures that include the protocol and compare them with machines that do not. Figure 7 gives an example of the preliminary results. Here, the PBBS benchmarks [70] are used, as compiled with a variant of the MPL Parallel ML language implementation [57], [73] that uses the semantics available in this language and in how the implementation manages memory to automatically drive our protocol. In the specific scenario given in the figure, the average speedup is ∼46%, while the interconnect energy (not shown) is reduced by ∼53%. The benefits grow with scale and disaggregation.

C. Blending

We are in the process of reconsidering the Application Binary Interface (ABI) between processes and between a process and a kernel. In particular, we are considering the customization of such ABIs (generated by our compiler) to enable software to blend together at run-time even when developed using a very different execution model (e.g., kernel versus application). Blending continues into the kernel itself.

One candidate application we foresee for blending is *sub-page granularity transparent far memory*. Current far memory systems either operate at page granularity for transparent swapping to remote nodes [31], [3] or require programmer annotations tagging data structures as remotable [67]. Compiler blending can automatically make these decisions and evacuate objects to remote memory transparently.

A second concept we are exploring is *blended device drivers*. Here, the idea is to merge driver code with code throughout the kernel and application. This blurring of the boundary between the driver and everything else may reduce latency through the use of polling and allow more efficient execution by executing driver code during even short periods of waiting elsewhere in the kernel or application. As a proof of concept, we have already extended the compiler-based timing work of Section IV-C to support *distributed device polling* and applied it to simple drivers. The normally interrupt-driven logic of the drivers is straightforwardly replaced with a constant-time poll check, and the compiler injects this polling check throughout the kernel using compiler-based timing. As a result, these devices appear to behave as if they were interrupt-driven, but no interrupts ever occur for them.

D. Pipeline Interrupts

One issue with current hardware (particularly x64 systems, though not limited to them) that we have run into again and again is the unbelievably high cost of interrupt (or exception) dispatch—the time from when an interrupt occurs to the first instruction of the interrupt handler. We have measured this to be on the order of 1000 cycles [29], [36]. In a system with kernel/user separation (e.g., Linux) this cost is generally not the first-order concern since other, higher costs are involved (e.g., context switch due to Spectre/Meltdown mitigation, signal injection cost for delivery to the application) or, for some HPC hardware, the interrupt can be directly mapped to a doorbell for the user-level code. In an interwoven system, however, the interrupt/exception dispatch cost is a major concern. The compiler-based timing work of Section IV-C is all about mitigating it for a specific device (a timer), and distributed device polling (Section V-C) is attempting to extend that mitigation for other devices.

We are also considering how to tackle the problem from the hardware perspective. We have developed a realizable extension of branch prediction logic that would allow a simple interrupt (no privilege level change, etc) in an interwoven system to be delivered as if it were a kind of branch instruction injected into the instruction fetch logic. MSR manipulation, similar to the existing `syscall` instruction, provides the mechanism to return to the interrupted code. The latency would be similar to that of a correctly predicted branch instruction, 100-1000$\times$ better. Because the hardware timer in the LAPIC is already on-chip and next to the core, it is the first interrupt for consideration. Another interest is an instruction exception, for example for #MF/#XF, which would facilitate efficient virtualization of the floating point ISA, and #GP, which would facilitate handling transparent far memory (Section V-C) and protection faults/swapping in CARAT (Section IV-A).

At this point, we have developed an initial proof of concept of this idea within PIN.

E. Bespoke Contexts

We believe new types of virtualized services will be possible using *bespoke execution contexts*, of which the virtines described in Section IV-D are an initial instantiation. These are execution environments tailored to a particular workload's needs. Bespoke contexts eliminate unnecessary overheads and carry little "runtime baggage." For example, if there is no need for device I/O, a runtime environment (or OS) that supports I/O drivers is unnecessary. A piece of code which leverages only integer math need not have the OS layer set up the floating point unit, and so on. Note that bespoke contexts go further than Unikernels [50], [69], [4], [55] and RumpKernels [44], as an application leveraging such services service might not need an OS *at all*; a minimal or *no* runtime environment may suffice. For example, we may even leave the machine in 16-bit mode as it boots up for certain simple services. The key is that these contexts are constructed at compile time, and in that way they can be seen as a type of synthesized runtime environment. Bespoke contexts are one example of an interwoven stack, and the compiler can help to synthesize them.

F. RISC-V / OpenPiton

Our work has generally been very specific to x64. We are currently exploring a port of Nautilus and other components to RISC-V, an open instruction set architecture that includes open implementations, such as in OpenPiton. By working on open hardware, we anticipate being able to more deeply explore hardware changes prompted by the interweaving model. At the present time, Nautilus partially boots on RISC-V.

G. High-level Parallel Languages As Enablers

Based on our experiences with coherence deactivation (Section V-B) and other domains, it has become increasingly clear that high-level parallel languages are enablers for the interweaving model. It may seem very counterintuitive that extremely high-level (indeed, mathematically defined!) abstractions in such languages would have much to do with the guts of a compiler, kernel, or hardware, but they do. The main observation is that due to the rich, well-defined semantics of these languages (including their run-time environment as seen by the programmer), it is much more straightforward to understand what an application is doing. Properties that the lower-level parts of the system could leverage might require immense effort for code analysis to prove about C/C++/Fortran+OpenMP/MPI. In contrast, these same properties are simply available *by construction* in high-level parallel languages. Conversely, properties that fall out of programs written in these languages may prompt new innovation in the lower-level parts of the system.

We expect that this synergy will become ever more important with scale and particularly with heterogeneity of the underlying hardware. Impedance matching heterogeneous hardware (or reconfigurable hardware) to a high-level parallel language is a current focus of ours.

VI. Conclusion

We have been working to improve the state of the hardware/-software stack for parallel programs. We demonstrated several promising examples for which the Interweaving model, where distinctions between traditionally rigid layers are blurred, can produce significant improvements. Parallel task scheduling, event notification, address translation, preemptive scheduling, lightweight virtualization are just a few areas where there is demonstrated potential for Interweaving, and we suspect that there are more.

Availability

The systems described in this work are freely and publicly available, and can be found on our web site at http://interweaving.org.

Acknowledgments

Students leading or involved in the work described here include (Northwestern) Brian Suchy, Mike Wilkins, Vijay Kandiah, Souradip Ghosh, Michael Cuevas, Jiacheng Ma, Aaron Nelson, Wenyi Wang, Siyuan Chai, Zhen Huang, Drew Kersnar, Guarav Chaudhary, Alex Bernat, Kevin McAfee, Kevin Mendoza Tudares, Spencer Colton, (IIT) Brian Tauro, Conghao Liu, Nanda Velugoti, Nicholas Wanninger, and Joshua Bowden. Parts of this work involve collaborations with Michael Rainey, Ryan Newton, Sam Westrick, and Umut Acar.

References

[1] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[2] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, 2018.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the* 15^{th} *European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Thomas E. Anderson. The case for application-specific operating systems. In *Proceedings of the* 3^{rd} *Workshop on Workstation Operating Systems*, April 1992.

[5] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[6] Richard Barrett, Michael Heroux, P. Lin, C. Vaughan, and A. Williams. Mini-applications: Vehicles for co-design. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2011)*, November 2011.

[7] Andrew Baumann, Paul Barham, Pierre Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the* 22^{nd} *ACM Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, October 2009.

[8] Pete Beckman. Argo: An exascale operating system. http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the* 10^{th} *USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 335–348, October 2012.

[10] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, December 1995.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[13] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *Proceedings of the* 14^{th} *USENIX Symposium on Operating Systems Design and Implementation'*, OSDI '20, pages 1–19. USENIX Association, November 2020.

[14] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the* 7^{th} *IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '15, pages 250–257, November 2015.

[15] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of the First European Workshop on OpenMP*, 1999.

[16] J. M. Bull, F. Reid, and N. McDonnell. A microbenchmark suite for openmp tasks. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP 2012)*, 2012.

[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of the* 15^{th} *European Conference on Computer Systems*, EuroSys '20, 2020.

[18] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[19] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, November 2019.

[20] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, November 2005.

[21] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokan Memik, and Alok Choudhary. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the 2012 Design, Automation and Test in Europe Conference*, DATE, pages 479–484, 2012.

[22] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the* 25^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 467–481, April 2020.

[23] A. Duran, J. Corbalan, and E. Ayguade. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008)*, 2008.

[24] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, December 1995.

[25] E. Forbes and E. Rotenberg. Fast register consolidation and migration for heterogeneous multi-core processors. In *Proceedings of the 34th IEEE International Conference on Computer Design (ICCD 2016)*, pages 1–8, 2016.

[26] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI 98)*, pages 212–223, 1998.

[27] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Kengo Nakajima, Yutaka Ishikawa, and Robert W. Wisniewski. Performance and scalability of lightweight multi-kernel based operating systems. In *Proceedings of the* 32^{nd} *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 116–125, May 2018.

[28] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In *Proceedings of the* 30^{th} *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 1041–1050, May 2016.

[29] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2020)*, November 2020.

[30] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of Supercomputing*, SC '10, November 2010.

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the* 14^{th} *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 649–667, Boston, MA, March 2017. USENIX Association.

[32] Faisal Hafeez, Pezhman Nasirifard, and Hans-Arno Jacobsen. A serverless approach to publish/subscribe systems. In *Proceedings of the* 19^{th} *International Middleware Conference (Posters)*, Middleware '18, pages 9–10, 2018.

[33] Kyle Hale. *Hybrid Runtime Systems*. PhD thesis, Northwestern University, August 2016. Available as Technical Report NWU-EECS-16-12, Department of Electrical Engineering and Computer Science, Northwestern University.

[34] Kyle Hale and Peter Dinda. A case for transforming parallel runtime systems into operating system kernels. In *Proceedings of the 24th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)*, June 2015.

[35] Kyle Hale and Peter Dinda. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, April 2016.

[36] Kyle Hale and Peter Dinda. An evaluation of asynchronous software events on modern hardware. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulaton of Computer and Telecommunication Systems (MASCOTS 2018)*, September 2018.

[37] Kyle C. Hale and Peter A. Dinda. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'16, pages 161–175, April 2016.

[38] Kyle C. Hale, Conor Hetland, and Peter A. Dinda. Automatic hybridization of runtime systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 137–140, June 2016.

[39] Kyle C. Hale, Conor Hetland, and Peter A. Dinda. Multiverse: Easy conversion of runtime systems into OS kernels via automatic hybridization. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing*, ICAC'17, July 2017.

[40] Troels Henriksen, Niels Serup, Martin Elsman, Fritz Henglein, and Cosmin Oancea. Futhark: Purely functional gpu programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2017.

[41] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[42] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019.

[43] H. Jin, M. Frumkin, and J. Yan. The open mp implementation of nas parallel benchmarks and its performance (nas 3). Technical Report NAS-99-011, NASA, March 1999.

[44] Antti Kantee. *The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, Helsinki, Finland, 2012.

[45] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting*, CUG'05, May 2005.

[46] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, June 2014.

[47] Ricardo Koller and Dan Williams. Will serverless end the dominance of Linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI, pages 169–173, May 2017.

[48] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'10, April 2010.

[49] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS '19, pages 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'16, June 2016.

[51] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS XVII, pages 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[52] M. Lis, Keun Sup Shim, B. Cho, I. Lebedev, and S. Devadas. Hardware-level thread migration in a 110-core shared-memory multiprocessor. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 1–27, 2013.

[53] Jiacheng Ma, Wenyi Wang, Aaron Nelson, Michael Cuevas, Brian Homerding, Conghao Liu, Zhen Huang, Simone Campanoni, Kyle Hale, and Peter Dinda. Paths to OpenMP in the kernel. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21. IEEE, November 2021.

[54] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 559–573, 2015.

[55] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, March 2013.

[56] Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th International Conference on Supercomputing (ICS)*, pages 294–301, 1999.

[57] MPL compiler. https://github.com/mpllang/mpl.

[58] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 21–39. USENIX Association, November 2020.

[59] Jiannan Oayang, Brian Kocoloski, John Lange, and Kevin Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)*, June 2015.

[60] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '19, pages 59–73, April 2019.

[61] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 149–160, June 2015.

[62] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[63] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W. Wisniewski. FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment. In *Proceedings of the 24th IEEE International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 211–218, October 2012.

[64] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 291–304, March 2011.

[65] Mike Rainey, Kyle C. Hale, Ryan R. Newton, Nikos Hardavellas, Simone Campanoni, Peter A. Dinda, and Umut A. Acar. Task parallel assembly language for uncompromising parallelism. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '21. ACM, June 2021.

[66] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. Unikernels: The next stage of Linux's dominance. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS XVII, pages 7–13, May 2019.

[67] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 315–332. USENIX Association, November 2020.

[68] Neil Savage. Going serverless. *Communications of the ACM*, 61(2):15–16, January 2018.

[69] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A framework for building per-application library operating systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 671–688, October 2016.

[70] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, 2012.

[71] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the* 15th *European Conference on Computer Systems*, EuroSys '20, 2020.

[72] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 329–345, June 2020.

[73] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*, 2020.

[74] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the* 9th *ACM Symposium on Cloud Computing*, SoCC '18, pages 199–211, October 2018.

[75] Robert W. Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the* 4th *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, June 2014.