Paths to OpenMP in the Kernel

Jiacheng Ma
Northwestern University
United States

Wenyi Wang
Northwestern University
United States

Aaron Nelson
Northwestern University
United States

Michael Cuevas
Northwestern University
United States

Brian Homerding
Northwestern University
Argonne National Laboratory
United States

Conghao Liu
Illinois Institute of Technology
United States

Zhen Huang
Northwestern University
United States

Simone Campanoni
Northwestern University
United States

Kyle Hale
Illinois Institute of Technology
United States

Peter Dinda
Northwestern University
United States

Abstract

OpenMP implementations make increasing demands on the kernel. We take the next step and consider bringing OpenMP *into* the kernel. Our vision is that the entire OpenMP application, run-time system, and a kernel framework is interwoven to *become* the kernel, allowing the OpenMP implementation to take full advantage of the hardware in a custom manner. We compare and contrast three approaches to achieving this goal. The first, *runtime in kernel* (RTK), ports the OpenMP runtime to the kernel, allowing any kernel code to use OpenMP pragmas. The second, *process in kernel* (PIK) adds a specialized process abstraction for running user-level OpenMP code within the kernel. The third, *custom compilation for kernel* (CCK), compiles OpenMP into a form that leverages the kernel framework without any intermediaries. We describe the design and implementation of these approaches, and evaluate them using NAS and other benchmarks.

CCS Concepts

• **Software and its engineering** → **Operating systems, compilers**; **Runtime environments**; • **Computing methodologies** → **Parallel computing methodologies**; • **Blended systems**;

Keywords

parallelism, OpenMP, operating systems

ACM Reference Format:
Jiacheng Ma, Wenyi Wang, Aaron Nelson, Michael Cuevas, Brian Homerding, Conghao Liu, Zhen Huang, Simone Campanoni, Kyle Hale, and Peter Dinda. 2021. Paths to OpenMP in the Kernel. In *The International Conference for High Performance Computing,Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis,MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476183

This project was supported by the United States National Science Foundation via grants 1763743, 1718252, 1763612, 1730689, 1908488, 2028851, and 2028958.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476183

1 Introduction

OpenMP [2, 16, 62] is arguably the most widely-employed approach for the linguistic expression and realization of shared memory parallelism, in part because it extends existing sequential languages like C, C++, and Fortran with parallel features. As a consequence, it can be incrementally adopted. While OpenMP's origins are in compact expression of loop-level data parallelism on SMPs, it has grown to include support for heterogeneous parallelism (including memory and devices), and task parallelism (including fine-grained and recursive tasks).

An OpenMP implementation is split between the compiler, which understands its language directives (`#pragma omp ...`) in the context of the sequential host language and lowers them to sequential code, and a run-time system that the lowered code invokes to dynamically create and manage parallelism. Underneath both lies the kernel, which implements primitives for memory, thread, task, and synchronization management that the run-time system uses, and the hardware itself, which the compiler, run-time system, and kernel ultimately try to leverage in the most performant way possible.

In a typical implementation, the OpenMP compiler and run-time system target the user-mode process model of a general-purpose kernel. This means that neither the generated code nor the run-time system have access to the full feature set of the hardware, which is only visible in kernel mode. Additionally, both are limited to the features and execution model of the user-level process abstraction the kernel exposes via system calls and other mechanisms. In today's implementations, the OpenMP application becomes a multithreaded Linux process.

There is reason to believe that by removing these limitations, performance and efficiency gains are possible [28, 32, 49]. Consider a parallel program that, instead of being a process, is itself a special-purpose kernel. Such an implementation of the program can directly leverage all hardware capabilities, including those that

match well to parallel language features but are typically unavailable in user mode [31]. Furthermore, the kernel abstractions used by the program can be accelerated [29], or even specialized [25, 75].

While such a "parallel application is a kernel" approach has demonstrated promise with other parallelism models, there is currently limited support of it for OpenMP. The goal of this paper is to show how to change this—to bring OpenMP into the kernel.

The design space for achieving this goal is large, and we report on three distinct points within it that represent particularly interesting trade-offs. The first of these, *runtime in kernel* (RTK) involves no changes to the compiler. In RTK, the OpenMP runtime and its immediate dependencies are ported to (or reimplemented within) the kernel. The application is then compiled as normal and linked directly with the kernel codebase, to create a custom kernel. This represents the tightest kernel/application coupling possible without changes to the compiler, but it requires considerable effort, particularly if the application has other dependencies.

In *process in kernel* (PIK), the kernel codebase is modified to create a special process abstraction that *behaves* like the user-level process abstraction, but, in fact, all code runs in kernel mode. In our implementation, the usual user-level compilation and linking steps are slightly modified, and the unmodified OpenMP run-time system is simply linked in. This allows a normally user-level program to be compiled and linked into a form that can be dynamically loaded into a running kernel, somewhat similar to a Linux kernel module. The environment it sees, however, emulates the user-level process environment of Linux. This allows kernel mode features to be leveraged incrementally. The PIK approach requires minimal effort of the user and can seamlessly handle additional dependencies. However, it is also the loosest coupling of the kernel and application.

The *custom compilation for kernel* (CCK) point of the design space allows the modification of the compiler itself. In our implementation, specialized LLVM analysis and compilation passes handle OpenMP directives (and add automatic parallelization where possible), lowering them down to a form that uses a tiny task-based run-time instead of the OpenMP run-time system. This run-time system is then directly implemented within the kernel. The CCK approach promises the tightest possible coupling of the OpenMP application, the kernel, and the hardware.

Our contributions are as follows.

- We make a case for kernel-level OpenMP support.
- We describe the design and implementation of the *runtime in kernel* (RTK) approach.
- We describe the design and implementation of the *process in kernel* (PIK) approach.
- We describe the design and implementation of the *custom compilation for kernel* (CCK) approach.
- We provide a performance evaluation of the approaches using NAS and other benchmarks.
- We compare and contrast these approaches in detail.

While it is not our goal here, we also note that enabling OpenMP within the kernel, specifically the RTK design point, also presents the opportunity to write traditional kernel-level code using OpenMP. This may become useful as general purpose kernels need to deal with increasingly larger scale machines. Our code can be found via *http://interweaving.org*.

2 Software, testbed, and benchmarks

Our work is built on the LLVM implementation of OpenMP, and the Nautilus kernel framework. We compare with the same OpenMP implementation on Linux using two well-known benchmark suites on node hardware with up to 192 cores and 8 sockets.

2.1 Software

Clang/LLVM: LLVM [50] is a widely-used compilation framework in academia and industry that enables sophisticated code analyses and transformations. In this work, we use the framework in two respects. First, we use the Clang/LLVM 9 implementation of the OpenMP directives in C/C++. Clang/LLVM lowers OpenMP code to the sequential LLVM intermediate representation (LLVM-IR), within the "middle-end" of LLVM.[1] For RTK and PIK our goal is to use Clang/LLVM without modification, meaning that identical object code is created for a user-level and kernel-level program.

`libomp`*:* `libomp` is the OpenMP run-time system that the code generated by Clang/LLVM invokes. `libomp` comprises about 75K lines of C++ and C, and 2K lines of assembly (all measured by `sloccount`). It targets the user-level process model of Linux and has several dependencies beyond this. For RTK we port `libomp` and its dependencies into the Nautilus kernel with minimum possible changes. For PIK we employ the unchanged user-level binary `libomp` directly.

NOELLE: For the CCK approach an alternative compilation path, implemented within the Clang/LLVM framework, is used both to handle OpenMP directives and to do automatic parallelization from sequential code. This builds on a powerful new analysis framework, NOELLE [55], 46,750 lines of C++. The same lowered sequential code is produced for the user-level and kernel-level target.

VIRGIL: The sequential code generated by CCK uses a custom, task-based run-time system, named VIRGIL, instead of `libomp`. Two versions of VIRGIL exist: a user-level version that uses C++ 17 abstractions to build on top of C++ threads (e.g. `clone()`) and C++ synchronization (including `futex()`) on Linux, and a kernel-level version that directly uses the kernel's internal task system, which operates similarly to the SoftIRQ mechanism in the Linux kernel.

Nautilus kernel framework: Nautilus [29] is a publicly available open-source OS kernel that currently runs directly on x64 NUMA hardware, including Xeon Phi. It is independent of the Linux codebase. Nautilus comprises over 331K lines of code as measured by `sloccount`. Nautilus was designed with the goal of supporting hybrid run-times (HRTs). An HRT is a mash-up of an extremely lightweight OS kernel framework, such as Nautilus, and a parallel run-time system [27, 28]. 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

[1] Fortran OpenMP programs could also be supported using the Flang front-end to LLVM. The middle-end transformations and the run-time system are the same.

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 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 [29, 30].

In this paper, Nautilus is used only as a stand-alone OS kernel that runs directly on bare metal with no virtualization. No Linux is used in any way when running Nautilus. While we don't use it here it is also possible to run Nautilus on top of commodity virtualization platforms. Of note for security and deployment concerns, Nautilus *can* run side-by-side with Linux in a multi-kernel configuration either using a hybrid virtual machine (HVM) [33, 34] or using the Pisces co-kernel framework [64] for a multi-kernel setup on bare metal. In a multi-kernel configuration, Nautilus and Linux can be compartmentalized (mutually protected) by HVM or Pisces, and rebooting the Nautilus part of the configuration can be done at timescales similar to a process creation in Linux.

2.2 Testbed and benchmarks

Testing and performance measurement is done on PHI, a Colfax Ninja Xeon Phi server, which is based on a Supermicro K1SPE motherboard that includes a 1.3 GHz Intel Xeon Phi 7210 (64 cores, 256 hardware threads) mated to 16 GB of MCDRAM and 96 GB of DRAM. We use this machine because it allows us to consider relatively large scales on a machine where we also have the full bare-metal access necessary for kernel testing.

PHI is used with hyperthreading off, and with the flat memory model. In this model, the MCDRAM is given a distinct NUMA zone with high distance to every CPU. As a consequence, the DRAM is preferred by any NUMA-aware OS. The DRAM consists of 6 16 GB DIMMs and is configured as 6-way interleaved. Both Nautilus and Linux are booted directly on this platform. In both cases, for the problem sizes used in our evaluation, only the DRAM is used. The Linux kernel involved is version 5.8.0. It is a tickless kernel driven by the LAPIC one-shot timer, as is Nautilus. The Linux distribution is CentOS 7 and it was configured according to Intel requirements by ColFax on delivery. Huge pages are enabled, transparent huge pages is set to `madvise`, and compaction is set to `always`.

Additional performance measurement is done on 8XEON, a SuperMicro 7089P-TR4T server with eight 2.1 GHz Intel Xeon Platinum 8160s (192 cores, 384 hardware threads total) mated to 768 GB of DRAM spread evenly across eight NUMA zones. Hyperthreading is off. Both Nautilus and Linux are booted directly on this platform. The Linux kernel is 5.4.0 and is tickless, as is Nautilus. The distribution is Ubuntu 20.04.2 LTS. Huge pages are enabled, transparent huge pages is set to `madvise`, and compaction is set to `madvise`.

To evaluate our work, we use the Edinburgh OpenMP Microbenchmark Suite [9–11] (EPCC) and the NAS 3.0 Application Benchmark Suite [3, 38] as ported to C+OpenMP [61]. EPCC measures the overhead of OpenMP directives. NAS is a well-known suite of benchmarks geared towards aerospace applications.

3 Runtime in kernel (RTK)

The runtime in kernel (RTK) model adds the application code, runtime system, and other dependencies directly into the kernel, making building of these part of the kernel compilation process. Any part of the kernel can then use OpenMP, not just the application.

3.1 Compilation

OpenMP provides the programmer with the ability to annotate statements in the base language with directives (pragmas) that control how the statement is to be parallelized. In the Clang/LLVM implementation of OpenMP, the compilation process produces object code that invokes the `libomp` run-time system. The application code may have other dependencies as well, for example on `libc`, `libstdc++`, `libm`, and so on. Our compilation process assumes that the necessary dependencies have been ported to the kernel. The need to port arbitrary dependencies to the kernel is a key limitation of the RTK approach. If an application has many such dependencies, other approaches may be preferable.

In RTK, no source code changes relating to OpenMP are required. However, the compilation and linking process needs to be adjusted for incorporation into the kernel. For the most part, this requires changes in compilation flags. However, since `main()` is now the kernel, an alternative means of starting the application needs to be added, which we do by converting the application's `main()` into a Nautilus shell command.

The x64 ABI provides for several features that do not exist within kernel code. As a consequence, the compilation process must be adjusted by changing or adding compilation flags. Two critical elements are the *memory model* and *red zone use*. Because the application is now a part of the kernel, the kernel's memory model must be specified. Red zone is a bit more challenging to understand. The red zone part of the x64 ABI allows the compiler to use a limited amount of stack space without allocating it. This can make leaf functions faster. Unlike user-level code, however, kernel code must be correct in the presence of interrupts. For performance reasons, Nautilus handles interrupts on the current thread's stack. Consequently, an interrupt would clobber such unallocated stack state. Therefore, the application and its dependencies must be compiled without red zone support.

As a practical matter, there are essentially two ways to incorporate these changes into the application and its build process: (1) porting the application's build process to the kernel's build process as a subdirectory/submodule of the kernel, or (2) separately building the application, respecting the necessary compilation flags, into a static library that is linked into the kernel. In both cases, it is the kernel's link process that is ultimately used, and this targets bootstrap in a physically addressed environment.

3.2 Runtime system

The Clang/LLVM OpenMP runtime system, `libomp`, must be linked into the kernel in order for OpenMP-compiled code to work. The

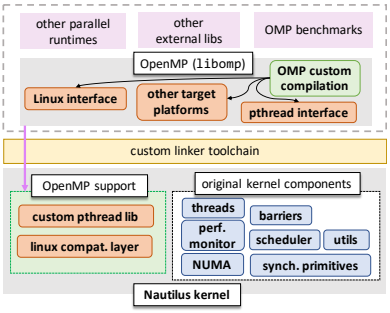
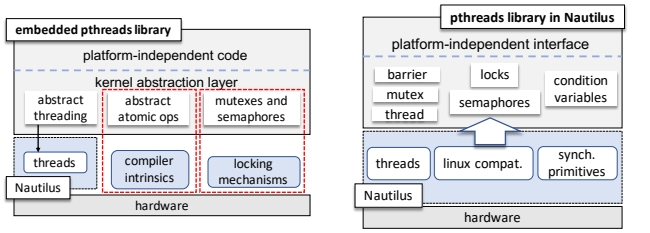


Figure 1: Integration of `libomp` runtime system into Nautilus.



(a) Port of embedded pthreads library to Nautilus.

(b) Customized embedded pthreads in Nautilus.

Figure 2: Pthreads in Nautilus: (a) Simple port of embedded pthreads library; (b) customized embedded pthreads.

problem is that `libomp` is not a standalone library, but rather a user-level library with many dependencies on the target platform (most commonly Linux and Windows). We explored two approaches to this problem: (1) elimination of these dependencies through in-depth porting of `libomp` directly to Nautilus internal interfaces, and (2) implementation of the required dependencies of the Linux target within Nautilus. Recall from §2 that `libomp` is a large, complex codebase (77K lines of C/C++/assembly). Approach (1) requires substantial effort, and more importantly, a deep understanding of `libomp`. It also makes it difficult to track changes to the mainline of `libomp`. However, it can provide the maximum flexibility in adapting `libomp` to take advantage of being in the kernel. In contrast, approach (2) has a lower effort, makes tracking of the mainline much easier, and still leaves room for incrementally taking advantage of the kernel context. It may seem surprising that (2) is lower effort, but note that while `libomp` has many dependencies, it uses these dependencies in very specific ways, and only these require emulation. We describe approach (2) in this paper.

In our design, we retain `libomp`'s default `autoconf`/`cmake`-based configuration and compilation process for its Linux target, but adjust the configuration so that it produces a static library that is suitable for incorporation into Nautilus. This means adding the special compilation flags of §3.1, selecting the appropriate compiler, and choosing an appropriate featureset. Essentially, we provide a wrapper script on top of the existing compilation process.

Figure 1 illustrates how `libomp` is then integrated into Nautilus. `libomp` assumes it is targeting Linux (albeit the dependencies are minor), and is using POSIX threads (pthreads) for its own implementation. Nautilus has been extended with a compatibility layer that includes a pthreads interface, a Linux compatibility layer just sufficient for the needs of `libomp`, and support for hardware-enabled thread-local storage (hwtls), which `libomp` (and the compiler via `__thread`) assumes.

The Nautilus linking process and custom linker scripts have been adjusted to add the `libomp` library, and other dependencies, as well as to appropriately handle variables marked as being thread-local. To avoid circular dependencies, the compatibility layer uses no OpenMP features.

3.3 Pthreads in Nautilus

Unlike its other dependencies, `libomp` makes extensive and elaborate use of the POSIX threads interface (pthreads). This interface is absent in Nautilus because Nautilus's thread, fiber, task, synchronization, and interrupt models aim to grant a parallel runtime more subtle control of concurrency. Compared to the Nautilus threads interface, pthreads is much more complex. The complexity of pthreads comes from various attributes associated with primitive objects and functionality significantly diverges for higher-level objects that build on them. We built a compatible pthread interface that, as expected, bases the primitive objects of pthreads on the primitives available in Nautilus. However, our implementation's design decisions are specifically made for the `libomp` use-cases. In other words, our pthread implementation is aware of the OpenMP runtime and geared to it. Within the kernel, a pthread thread is a variant of a kernel thread.

Our implementation is based on the POSIX Threads for embedded systems (PTE) library [39], which is itself based on pthreads-win32 [40], a GPL-licensed pthreads library for Microsoft Windows. PTE trades platform-dependent optimization for portability. To port PTE to Nautilus, we needed to supply only a thin OS abstraction layer. Figure 2(a) illustrates this port. Although redundancies are easy to spot, it is still reasonably efficient and pushes most performance issues down to the platform-dependent layer we supply. Later, we revisited the pthread implementation, focusing on customizing it to the Nautilus environment. This included directly leveraging some higher-level constructs such as condition variables, barriers, and thread management in Nautilus. Figure 2(b) illustrates the structure of the customized pthread interface.

3.4 Other dependencies and issues

Although `libomp` has dependencies on `libc`, etc., the important cases basically boil down to access to environment variables, and use of the Linux `sysconf()` call to get access to hardware/software configuration information. These are not performance critical, but essential for correctness and to manipulate the application (for example to choose the number of threads to use). We implemented a general purpose environment variable mechanism for kernel code, as well as a `sysconf()` that supports a limited number of keys.

`libomp` and the code generated by Clang/LLVM's OpenMP implementation makes extensive use of hardware support for thread-local storage. On x64, hardware TLS is based on the use of the `%fs` and `%gs` segment register overrides, where the corresponding `FSBASE` and `GSBASE` MSRs point to the TLS block. In Nautilus, we require the use of `%gs` to point to the per-CPU state in the kernel, so we restrict the compiler to use `%fs` when it generates TLS code. We added support for context-switching `FSBASE` as part of a thread context switch, as well as support for `arch_prctl()` configuration of `FSBASE`. Linking and loading of the kernel was modified so that

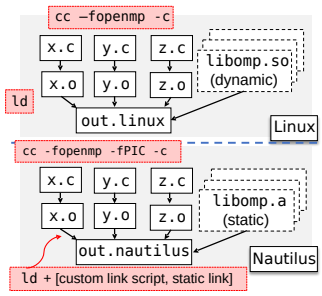


Figure 3: PIK compilation/linking compared to Linux.

TLS data and BSS segments are supported and handled correctly. Thread launch clones TLS data and BSS to complete the support.

During testing, we encountered issues with SSE (and higher) floating point state being corrupted. Because Nautilus integrates kernel and application code, it cannot restrict the use of SSE registers like a general purpose kernel, and instead must manage them as a part of kernel thread/fiber state. We found that Clang/LLVM was aggressively using SSE registers to optimize interrupt handlers, for which SSE state was not managed. To address this, we added a lazy SSE save/restore model for interrupts, with the added feature that it can point out interrupt code that is causing it to be invoked. We then used this feature to give the kernel interrupt code it identified the no-SSE attribute.

4 Process in kernel (PIK)

The process in kernel (PIK) implementation allows for separate compilation and linking of the application and kernel, much like the Linux user-level model. However, the separately compiled application executable is dynamically loaded and run as part of the kernel. Unlike a kernel module, however, it is not linked to the kernel, but rather runs within a specialized kernel-mode process abstraction. PIK completely avoids adding application and run-time dependencies to the kernel itself, instead providing only a system call interface. This greatly simplifies the porting of applications—dependencies are handled exactly as they are at user-level—and allows for incremental use of kernel-mode features. However, in contrast to RTK, the barrier for using kernel-mode features is higher, and OpenMP cannot be used elsewhere in the kernel.

4.1 Compilation

Figure 3 compares and contrasts the Linux user-level build process, and the PIK build process. The PIK build process for an application almost exactly reflects the application's original user-level build process. The implementation supplies a script, `nld`, which wraps the linker for the common case. The same C compiler can be used for both Linux and Nautilus.

Only one additional compiler flag is needed: position-independence (`-fPIE`). Position independence is required because Nautilus's loader is placing the executable into the physical address space, and the ultimate location depends on the state of prior kernel memory allocations. Disabling red zone use is not necessary because, when compiled for PIK, the kernel is configured to use a trampoline stack on interrupt so as to avoid disturbing the application's red zone variables. On a syscall instruction, the syscall handler subtracts from its stack pointer to avoid the redzone in a similar way.

After compilation, all objects, and dependencies (libraries) are linked together using a custom linker script. Note that the entire, unmodified `libomp` run-time system is simply linked in. The linker script preserves the position-independence of the entire linked executable ("static PIE"). The compiler, C, and C++ runtime startup code (e.g., `crt0`) is integrated carefully, and with an assumption that the kernel will be providing a "pre-start" environment for it. The linker script also attaches a custom-designed 64-bit variant of a multiboot2 header at the very beginning of the output file and as the very first section. While multiboot2 headers are usually used to simplify the loading of an ELF kernel by a boot loader, we use one here to simplify the loading of an ELF executable by the kernel. Because of the position independence, static linking, and the multiboot2 header, the Nautilus loader can largely treat the executable as a simple binary blob that can be placed anywhere in physical memory that is convenient.

4.2 Process abstraction

PIK builds upon Nautilus's kernel-level process abstraction. This abstraction combines the notion of a kernel thread group (which can be gang-scheduled) with optional support for an independent address space (implemented using paging or other means [75]), and optional support for a custom allocator that is layered on top of the kernel-level memory management. The abstraction itself has no concept of user-mode, however, nor system calls.

As might be expected, a process creation involves the creation of an initial thread within the process. Otherwise, a newly created kernel thread joins the process of its creator, if it exists. The initial thread runs a wrapper function (the "pre-start" code) that completes the setup of the process before invoking the user's thread function. This is configurable to support different compatibility models. Other threads similarly start in wrappers that complete their setup with respect to their process before running the user's thread function.

In our implementation of PIK, the initial thread's function invokes Nautilus's loader with a file name. The loader, leveraging the multiboot2 information in the file, allocates memory, copies the file content to it, initializes BSS/TBSS, and then jumps to the entry point. This is quite similar to a Windows-style `CreateProcess()`, but done entirely in kernel.

We added several features to Nautilus to facilitate such processes. First, hardware TLS support and lazy floating point save/restore in the presence of interrupts was included, as described in §3.4. We also eased the red zone restriction by using the hardware's interrupt stack table (IST) feature. We do not handle interrupts on a separate stack, but rather have the initial interrupt handler copy the interrupt frame to the thread stack at an offset that avoids the red zone, and then continue the interrupt on the thread stack.

4.3 Linux compatibility

Figure 4 illustrates the PIK run-time model and compares it with that of Linux. The executable was compiled and linked assuming a Linux-compatible process environment, which we emulate. To achieve this, we provide a system call interface through which we emulate a subset of the Linux syscall interface.

Since Nautilus has no concept of syscalls on its own, the Linux compatible system call interface simply uses the same binary interface as Linux (e.g., the `syscall` instruction or the `int 0x80`

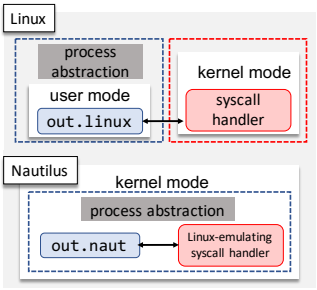


Figure 4: PIK run-time compared to Linux.

instruction). The vDSO is not currently supported, and the "pre-start" code ensures that it is not detected. Unlike in Linux, a system call in Nautilus happens in the same address space, at the same privilege level, and using the same stack as the calling thread (red zone is avoided for both mechanisms).

After implementing these interfaces, we began to implement Linux-compatible system calls. It is important to note that our goal here is not to emulate the entire, gigantic Linux process interface, but just enough to be able to support typical OpenMP programs. Syscall stubs were added for each Linux syscall type so we can see all activity, and respond, by default, with an error.

The most important system calls (i.e. those used by the C runtime and `libomp`) were then implemented iteratively until several test programs were able to execute in an expected manner, consistent with their behavior on Linux. We then continued to expand the implementation until we were able to support all of the benchmarks described in this paper. Other mechanisms processes can use to interact with Linux, especially virtual filesystems such as `/dev`, `/proc` and `/sys`, are not implemented with the exception of `/proc/self`, which is required by `libomp`. In principle, system calls and accessible namespaces can be incrementally added to our implementation as needed.

5 Custom compilation for kernel (CCK)

The OpenMP standard is constantly evolving as the needs of parallel programs and the underlying hardware evolve. This leads to OpenMP runtimes (e.g. `libomp`) that also must constantly evolve. The RTK (§3) and PIK (§4) approaches directly support `libomp`, enabling OpenMP programs to be compiled with any `libomp`-using compiler. The cost is that RTK requires maintaining an additional large and evolving codebase (`libomp` is 77K lines of C/C++/assembly) in the kernel. PIK avoids this, but requires maintaining the kernel-level support needed to be compatible with `libomp` (currently about 2K lines of C and assembly).

The custom compilation for kernel (CCK) approach uses specialized LLVM analysis and compilation passes to lower all OpenMP parallel structures to tasks. The compiler injects trampolines into the code to dispatch independent tasks to a small task-based runtime, VIRGIL, instead of to `libomp`. The implementation of VIRGIL in Nautilus comprises only 550 lines of C and it builds on Nautilus's task system, a component that most other kernels also have (e.g., SoftIRQ in Linux). The user-level version of VIRGIL consists of 620 lines of C++. Not only is this much smaller than the alternatives, it also does not have to evolve; it is the compiler that evolves.

CCK's runtime is significantly simpler than `libomp` (even if we consider just the task components of `libomp`) for several reasons. First, it *only* has to support tasks, rather than the panoply of parallel forms of OpenMP (e.g., section, parallel for, ...). Second, it does not need to support OpenMP attributes (e.g., reduction, nowait, ...). Instead, the compiler uses attributes statically. Finally, the tasks that the CCK runtime sees are *independent* from its perspective. The compiler generates code such that all tasks that are handed to the runtime are immediately ready.

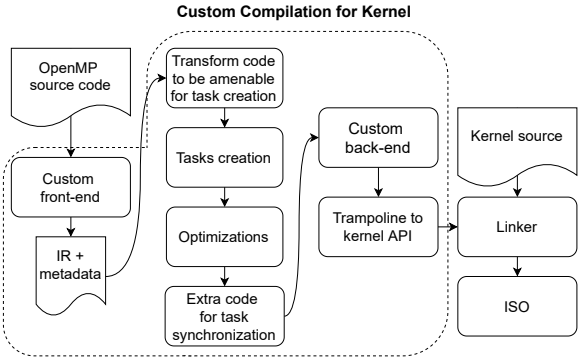


Figure 5: CCK compilation pipeline.

5.1 Compilation pipeline and *AutoMP*

CCK compilation builds upon NOELLE [55], a novel compilation framework that includes state-of-the-art code analyses and transformations. We extended NOELLE's memory analyses to leverage OpenMP semantics, and its code transformations for task generation to implement the parallelism expressed in OpenMP directives. OpenMP directives are translated into metadata that is attached to the LLVM IR to explicitly express the absence of dependences between code regions. The CCK transformations, named *AutoMP*, use this metadata in addition to properties determined via code analysis to automatically parallelize the program.

Figure 5 shows the compilation pipeline. Our front-end lowers the source code to sequential LLVM IR combined with semantic metadata derived from the OpenMP directives. Next, a sequence of custom transformations leverage the semantic metadata to generate tasks suitable for VIRGIL. Further metadata-informed optimizations shave off unnecessary overhead related to task creations and joins. Next, synchronization code is generated together with code that ties to the runtime. Finally, a custom back-end produces an object file that is compatible with the kernel. The object file is then linked as a kernel component to create a bootable kernel image.

5.2 OpenMP to metadata conversion

OpenMP pragmas in the source code specify what code to parallelize, how to parallelize it, and assert that it is *correct* to parallelize it. These pragmas carry rich semantic information into the compiler; for example, that the iterations of a loop are independent or that a section of code is atomic and requires some ordering. Rather than following Clang's conventional OpenMP compilation pipeline, our custom front-end instead embeds this semantic information within the IR without isolating the code specified within pragma.

We modified Clang to simply annotate the Abstract Syntax Tree (AST) of the program being compiled with the OpenMP semantics. This is quite different from Clang's conventional processing, which

wraps OpenMP code regions in new functions (a process called outlining). Outlining partitions the code of a function across multiple functions, which significantly reduces the accuracy of many code analyses in the LLVM middle-end (e.g., memory analysis, data-flow analysis). This accuracy loss is not a problem for a conventional OpenMP implementation because it blindly implements the parallelism specified by the pragmas. We cannot. CCK needs to reshape the parallelism specified by the pragmas to reduce it to tasks. To do so requires high accuracy code analyses and, therefore, outlining code is not an effective option.

Using the annotated AST, for each compound OpenMP statement, CCK's front-end generates unoptimized LLVM IR with the OpenMP semantic information embedded as IR metadata, to wit, a sequential version of the program permeated with OpenMP metadata.

5.3 Task generation

To generate tasks, CCK's middle-end first deconstructs the parallelism forms of the original program into code regions with annotations (e.g., independence declaration between code regions enabled by OpenMP pragmas). Then, it task-parallelizes the code.

The middle-end first computes the program dependence graph (PDG) [20] using state-of-the-art memory analyses [1, 12, 76] that we have enhanced to exploit the code-region annotations mentioned above. These code analysis extensions that exploit the OpenMP semantic metadata is what enables CCK to go beyond the accuracy that a conventional dependence analysis could reach. This extra accuracy allows CCK to find more available parallelism than automatic parallelization techniques can.

The next step is to run a series of code transformations that make the code more amenable for the creation of tasks, including function inlining, loop distribution, and loop fusion. These transformations generate code with single entry and exit points for each code region that could become a task. This is followed by a parallelizer that decides which of these code regions need to become tasks. The selected code regions are then parallelized, generating tasks, using techniques included in NOELLE (HELIX [13, 15] without the OS support [14] and without thread speculation [57], DSWP [63], and DOALL). Each selected code region becomes a function where the region's live-in variables become function parameters. The region's live-out variables are packed into an heterogeneous array and passed as the first parameter to the generated function. These functions are the tasks that the runtime executes.

CCK can often statically determine where in the code tasks will become ready at run-time (e.g., at the beginning of a loop without loop-carried dependences), in which case CCK simply adds a task submission at the identified code point. When readiness of a task cannot be determined statically, the compiler generates the code to check and submit tasks at run-time. In this way, task dependence checking is bespoke to the application instead of being part of the runtime. This significantly simplifies the support that CCK needs from the runtime and the OS.

The last step in CCK's middle-end invokes important optimizations for the parallel execution of applications, such as object privatization and variable reductions. Standard LLVM optimizations are also employed on each task, including loop unrolling and code vectorization. We then add extra code to manage task synchronization as required by the parallelization techniques used.

Aspect	Approach RTK	PIK	CCK
Effort			
Runtime	major	none	minor
Kernel	minor	major	minor
Compiler	none	none	major
Implementation Size (C LOC)			
Runtime	1,600	0	550
Kernel	2,200	13,250	600
Compiler	0	0	6,550 (C++)
Benefits and Opportunities			
Application development	easier	easiest	easy
Leveraging kernel context	easier	difficult	easiest
Decoupled from OpenMP runtime	no	no	yes
Applies to all code in kernel	yes	no	no
Automatic parallelization	no	no	yes

Codes Sizes reflect new code or modifications.

Figure 6: Summary of design and software engineering tradeoffs.

5.4 Binary generation, linking, runtime

Each task is wrapped in a function with live-in and live-out variables made explicit in its signature. We generate a landing task for each set of tasks that are grouped together (e.g., all tasks corresponding to iterations of a loop) to reduce their live-out variables. The landing task is executed when all tasks of the group join (the runtime is unaware of this join). Finally, our back-end generates a single object file that encodes the whole program compiled including the code for the generated tasks. The object file is generated with kernel-compatible options, including avoiding exploiting the x64 red zone. This enables the kernel to arbitrarily intertwine application code and kernel code (e.g., interrupts) with little overhead. Tasks are executed by either the user-level or kernel-level VIRGIL runtime, which was described earlier.

6 Evaluation

It is important to understand that we are exploring the design space of approaches to moving OpenMP into the kernel. We have described the software engineering effort, benefits, and opportunities of the particular points in that design space represented by RTK, PIK, and CCK in their sections. Figure 6 summarizes this discussion.

We evaluated our implementations of RTK, PIK, and CCK for performance, using the Edinburgh (EPCC) and NAS 3.0 benchmarks on the machines described in §2.2. We ran our tests on the default user-level Linux implementation as well. The same compiler and identical compilation flags affecting back-end code-generation were used for Linux, RTK, PIK, and CCK, and are described in more detail in the respective sections. CCK uses custom middle-end analysis and transformations in the context of the compiler. We address the following questions:

- How are OpenMP primitives affected by RTK and PIK?
- How do RTK, PIK, and CCK affect application performance?
- How do the performance gains (and losses) of RTK, PIK, and CCK relate to their tradeoffs in the design space?
- Are these viable paths to bringing OpenMP into the kernel?

We show our evaluation first on PHI, and then repeat it for 8XEON.

6.1 Impact on OpenMP primitive performance

Figures 7 and 8 show the EPCC microbenchmark performance for RTK and PIK, respectively, comparing them to Linux user-level performance on the PHI machine at full scale. There are no microbenchmark numbers for CCK. Recall from §5 that CCK does not

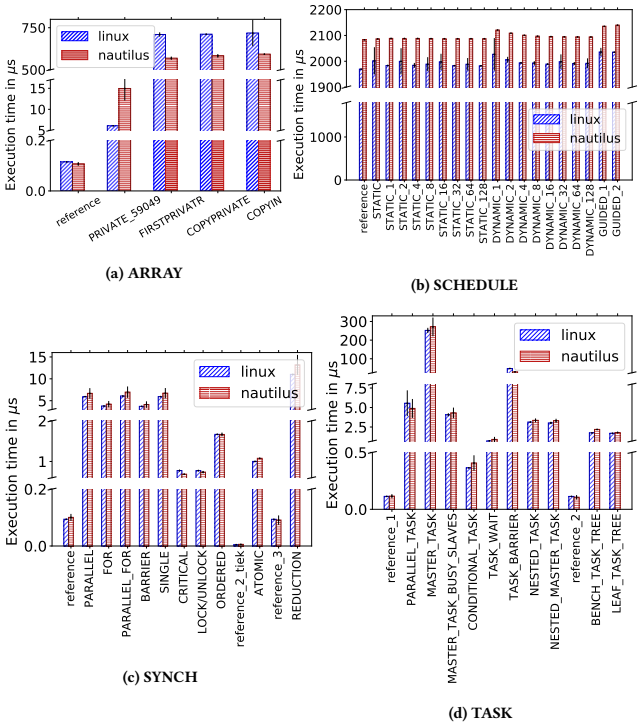


Figure 7: RTK performance compared to Linux: EPCC microbenchmarks on 64 cores of PHI.

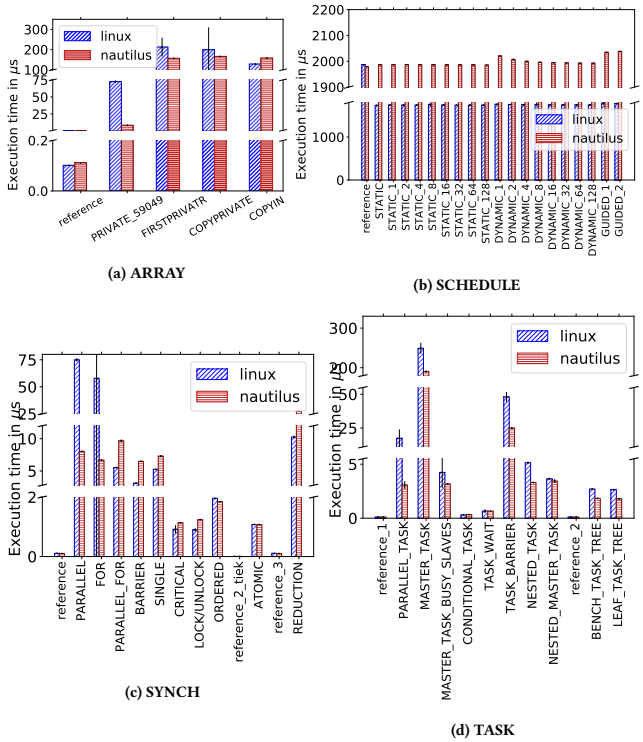


Figure 8: PIK performance compared to Linux: EPCC microbenchmarks on 64 cores of PHI.

directly implement OpenMP directives. Consequently, there are no OpenMP directives that EPCC can measure in CCK.

The numbers represent the overhead of each of the OpenMP directives. Note that we have not yet used any kernel-level features to enhance either RTK or PIK. Hence we are hoping for a rough parity in performance, which is indeed what we see. RTK shows slightly higher overhead than the Linux implementation, while PIK shows slightly lower overhead. In PIK, precisely the same OpenMP runtime, pthread library, and libc/libm are used as with the Linux version. In contrast, RTK uses a port of the runtime, a pthread compatibility layer, and also experiences kernel memory allocation directly. PIK experiences considerably lower variation in overhead than either RTK or Linux—lower jitter is one benefit of bringing code into the kernel, although it also depends on other factors.

6.2 Impact on application performance

To see the impact of RTK, PIK, and CCK on application performance, we ran the NAS benchmarks with C class, with a few exceptions, for each model. The exceptions, where we run B class versions, are due to large static variables (gigabyte-size globals). In RTK and CCK, because these variables are linked into the kernel boot image and are thus loaded into physical memory at boot time, the kernel boot image can end up being large enough that it overlaps an MMIO region. PIK does not have this issue. Where possible, we have modified the benchmarks to use dynamic memory allocation to create these variables when the benchmark is started (instead of at boot time). This avoids the boot overlap problem, but it is not always a fair change because statically allocated multidimensional arrays can potentially be accessed faster. For benchmarks where this might be the case, we do not make this change, and instead use B class. When changes are made, they are used in all four cases.

Figure 9 compares the performance of RTK compared to Linux on PHI and shows the execution time of RTK benchmarks normalized to Linux. At the smallest scale (1 CPU), RTK performs from 4.5% (MG) to 90.5% (BT) better than the Linux user-level code. At the largest scale (64 CPUs), RTK performance varies from slightly worse (-1.2% in FT) to 36% (SP) faster than the Linux user-level code. The average performance gain of RTK across scales and benchmarks is on the order of 22% (geometric mean).

These results may seem surprising given that RTK exhibited slightly higher overheads for the OpenMP primitives in §6.1. Note that unlike the microbenchmarks, the NAS benchmarks do significant computation. This computation benefits from the friendlier kernel environment described earlier. Of note, the Nautilus environment is providing (a) no page faults, (b) extremely rare TLB misses, (c) NUMA-cognizant memory allocations, (d) extremely rare interrupts and otherwise greatly diminished OS noise, and (e) precisely zero competitive threads/processes. When a thread is executing outside of an OpenMP primitive, it does so for long stretches of time, with no competition and with its partner threads running simultaneously on the other CPUs. Gains of 20–40% over user-level execution on Linux have been previously demonstrated for an RTK-like implementation of the Legion run-time system [29], which is in line with what we measure here.

Figure 10 compares the performance of PIK normalized to Linux on PHI. We see that PIK performs generally similarly to RTK, with a few exceptions. The average performance gain of PIK across scales

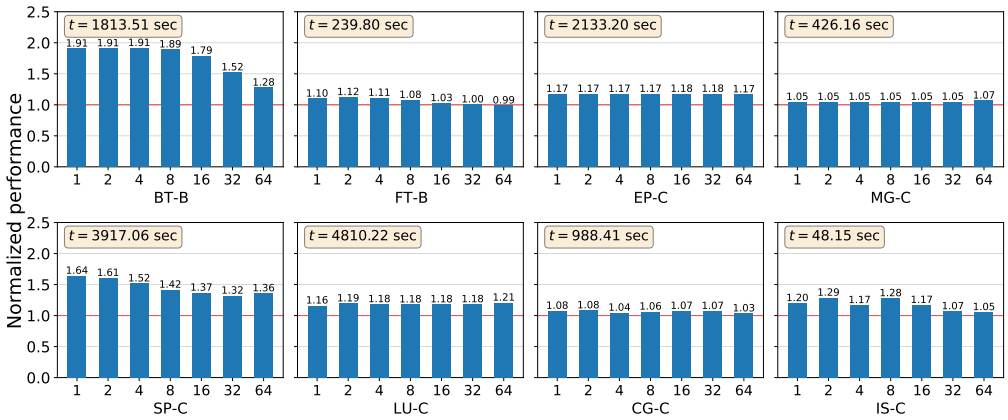


Figure 9: RTK performance relative to Linux as a function of CPUs used: NAS benchmarks on PHI; higher is better. Baseline (Linux OpenMP) is horizontal red bar at 1.0. t is the single threaded Linux absolute performance.

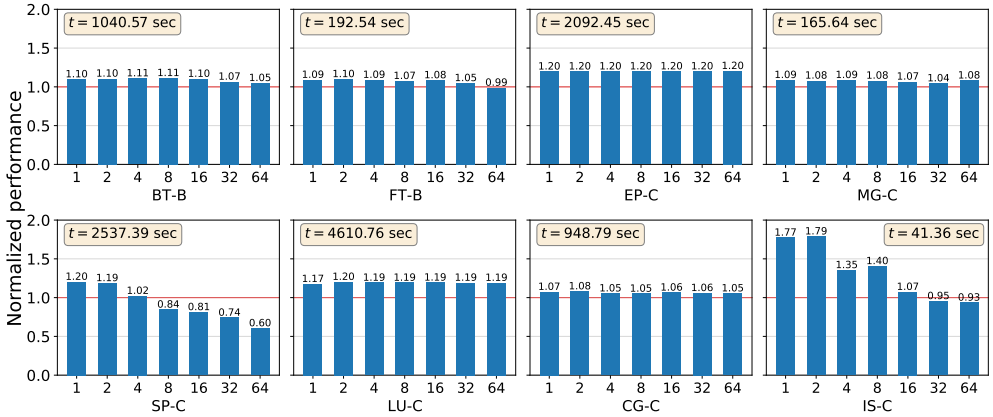


Figure 10: PIK performance relative to Linux as a function of CPUs used: NAS benchmarks on PHI; higher is better. Baseline (Linux OpenMP) is horizontal red bar at 1.0. t is the single threaded Linux absolute performance.

and benchmarks is on the order of 10% (geometric mean). PIK is also a viable and performant path to including OpenMP in the kernel.

Understanding CCK performance is more complex since two elements are at work, the AutoMP compilation process (versus the OpenMP process) and whether Linux or Nautilus is being targeted. Figure 11 shows the absolute performance for all three combinations (the baseline of Linux+OpenMP, plus Linux+AutoMP, and Nautilus+AutoMP.) Figure 12 then shows the relative performance of both AutoMP versions compared to Linux+OpenMP.

The comparison between Linux+OpenMP and Linux+AutoMP highlights the difference in parallelism exposed by CCK compared to the other approaches. Recall from §5 that AutoMP translates the parallelism expressed in OpenMP into declarations of independence between code regions. This independence is then used to generate independent tasks, which allows the runtime to be quite small, simple, easier to maintain, and more stable over time than the OpenMP runtime. However, the cost of AutoMP is the potential performance lost due its normalization of the original OpenMP parallelism, whatever its form, into independent tasks.

FT and EP show that the parallelism generated by AutoMP reaches the same performance obtained by OpenMP—AutoMP's parallelism normalization did not lose any performance. Unfortunately, LU, BT, SP, and IS show a performance loss. This is due to AutoMP being currently unable to exploit OpenMP directives related to object privatization. Consequently, some loops in these benchmarks are left sequential because of the lack of thread-private objects. IS, which we elide entirely, is an extreme case in which no parallelism is extracted due to this limitation.

MG and CG show the benefits of having a compiler's middle-end being able understand the program's parallelism and therefore being able to leverage parallelism-aware code analyses. Here, AutoMP is able to produce more performance than OpenMP because the AutoMP chunks loop iterations differently. More specifically, it chunks loop iterations depending on the estimated latency of an iteration of the loop being parallelized (computed using a parallelism-aware data-flow analysis). In contrast, OpenMP's compiler just blindly follows the OpenMP directives. This leads to the OpenMP compiler choosing a coarse-grained chunking. This is a poor choice. AutoMP's choice of finer-granularity chunking liberates more parallelism, resulting in the performance gains.

When targeting Nautilus (CCK), the compiler is identical. The version of the lightweight VIRGIL runtime used here is simply a

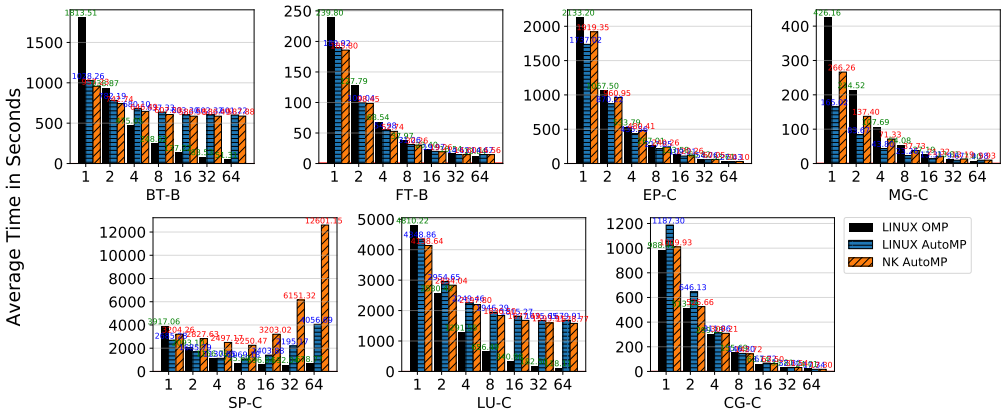


Figure 11: CCK absolute performance on Linux and Nautilus compared to baseline of stock OpenMP on Linux as a function of CPUs: NAS application benchmarks; lower is better.

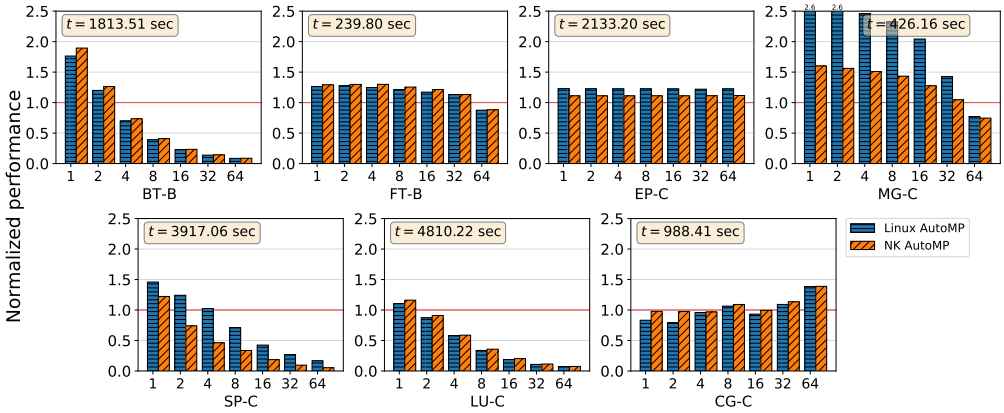


Figure 12: CCK performance relative to Linux as a function of CPUs used: NAS benchmarks on PHI; higher is better. Baseline (Linux OpenMP) is horizontal red bar at 1.0. t is the single threaded Linux absolute performance.

thin veneer over the kernel's task framework. As the figures show, performance of Nautilus+AutoMP (CCK) is broadly similar to that of Linux+AutoMP, with FT, LU, and BT being favorable, EP, SP, and MG being unfavorable, and CG being a wash. CCK shows itself to be a viable path to including OpenMP in the kernel.

6.3 Performance on 8XEON

We repeated all performance tests on the 8XEON, the modern 8 socket server described in §2.2. For 1-24 cores, the same codebase is used as before. For 24+ cores, we have extended Nautilus to use first-touch allocation at 2 MB granularity instead of immediate allocation, similar to Linux. The NAS benchmarks typically use large global arrays. Immediate allocation results in such arrays being assigned to a single NUMA zone, lowering performance when different slices are assigned to CPUs in different zones.

Impact on OpenMP primitive performance: Figure 13 shows the performance of the EPCC microbenchmarks on RTK and PIK, at the largest scale (192 cores, 8 sockets). Except for scheduling, where performance is comparable, RTK and PIK outperform Linux.

Impact on application performance: Figure 14 shows the performance of RTK and PIK relative to Linux for all of the NAS benchmarks. Figure 15 documents CCK and shows the performance of Linux+AutoMP and Nautilus+AutoMP relative to Linux+OpenMP. 1–24 cores is a single socket, 48 cores is 2 sockets, 96 cores is 4 sockets, and 192 cores is 8 sockets. Similar to PHI, on 8XEON, RTK and PIK show ~20% gain (geomeans) compared to Linux.

7 Discussion

Bringing user code into the kernel is not a trivial feat, but the performance benefits can be significant. We now discuss other aspects of bringing OpenMP into the kernel.

Generalizability: While our prototypes are implemented in the context of Nautilus, we expect that in many cases analogous implementations are possible in other kernels, certainly in unikernels and similar models. We think our experience with RTK and CCK extrapolates to Linux as well. Here, we would port to or target the Linux kernel module environment, similar to how early real-time applications have worked in the past (§8). PIK may also be suitable, especially as a PIK executable is already analogous to a kernel module, but there are two issues. First, the footprint of a PIK executable is very large compared to a typical kernel module because it pulls in all necessary user-space libraries statically. Second, Linux might not easily permit a fast kernel-to-kernel system call model.

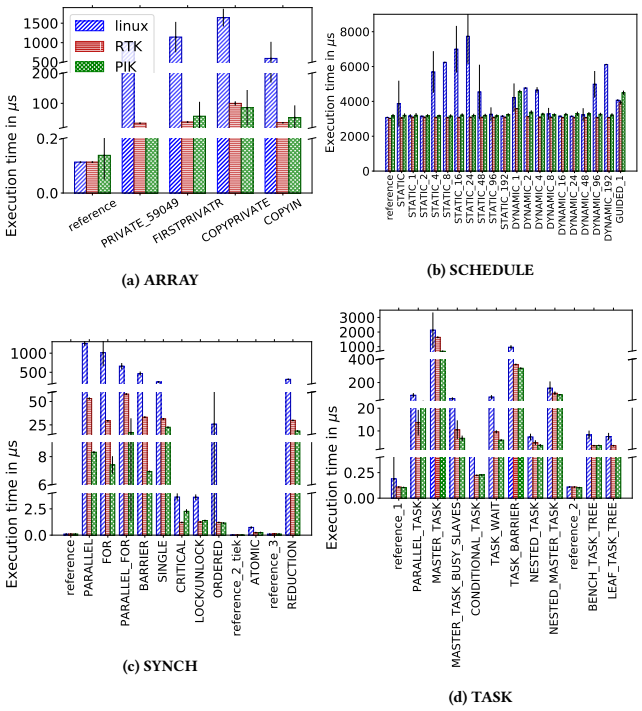


Figure 13: RTK and PIK performance compared to Linux: EPCC microbenchmarks on 192 cores of 8XEON.

Security: Running untrusted user code in a privileged environment has obvious drawbacks from a security perspective. The main issue arises from the removal of hardware isolation boundaries, such as those enforced by paging. While a detailed threat model is outside the scope of this paper, we do point out two techniques that can minimize damage done by untrusted code. The first involves space partitioning the machine between two OSes, dubbed the multi-kernel approach [22]. In this model, the specialized system (Nautilus in our case) runs on a subset of hardware resources, either using space-partitioned virtual machines [33], or space-partitioned hardware, as is done with co-Kernels [64]. IHK/McKernel [24] is an example of such a multi-kernel system that currently runs on the world's top supercomputer. Isolation can also be enforced by the language and the compiler. There is a rich history of shifting the isolation burden from the OS to the programming language, for example using domain-specific languages for device drivers in Exokernel [19] and enforcing protection with type-safe languages in Singularity [37] and Mirage [53]. In HPC, such managed languages can come with unacceptable performance overheads, and Unikernel approaches that assume a virtualization layer may also be untenable. In this case, the compiler can perform heavy lifting (as in CCK) to enforce isolation [75].

Deployment: The feasibility of deployment of OpenMP in the kernel, in any form, depends on site-specific factors. As previously described, multi-kernels have been deployed in HPC environments (and unikernels have traction in data centers), suggesting a path. Even using a single kernel, for a space-shared environment, a critical issue is boot time. Boot times of a specialized kernel like Nautilus in a multi-kernel environment are on the order of milliseconds. There is no fundamental reason why the same boot times could not be achieved in a single kernel environment, facilitated by better firmware such as Coreboot [56] and specialized subsystems [69].

Multi-node (MPI): Although multi-node execution is not our focus, we note that a "pure" in-kernel MPI implementation would proceed along the lines of RTK or PIK. MPI implementations already have layered designs in which NIC-specific code lies below a HAL. An in-kernel implementation or port would implement the HAL directly on top of kernel drivers. Nautilus already includes drivers for common Ethernet and Mellanox Infiniband NICs. Alternatively, in a multi-kernel model, the "control plane" aspects of MPI and the drivers can be left in the Linux kernel, and only the performance-critical "data plane" elements are in the specialized kernel. Most of the multi-kernels mentioned earlier already provide communication and storage in this split manner.

Programmer effort: The RTK, PIK, and CCK approaches present different levels of a challenge to the application developer. A key benefit of PIK is that the developer does not need to be aware of the fact that their code is in the kernel. In contrast, while RTK and CCK hide the different OpenMP implementation details from the developer, the developer *does* need to port other aspects of the application to the kernel environment. However, RTK and CCK present many more opportunities for optimization than PIK. A compiler could conceivably split the difference by helping with porting, although that is not our focus with CCK.

Implications: As machines scale and become more heterogeneous, an increasing diversity of approaches to performance and efficiency is necessary. Scale also has a track record of making small performance differences compound, as was observed with OS noise [21]. Fortunately, scale itself allows for different approaches to co-exist—the hardware partitioning and multi-kernel techniques described above make increasing sense with increasing scale, for example. This is the case even within a single node.

There is a rapid expansion of the need for parallelism beyond traditional HPC circles, as well as the drive to exascale within those circles. The architecture renaissance is well underway. There is a need and an opportunity to rethink the hardware/software stack of parallel computing, in particular the layering that has now existed for decades, and was never motivated by parallel computing in the first place. Machines have been and can be different. Our exploration of OpenMP in the kernel is in that vein.

8 Related work

Considerable effort has gone into improving the abstractions and performance of parallel primitives user-level. Examples include QThreads [79], MassiveThreads [58], Tiny Threads[17], Lithe [65], Intel's Thread Building Blocks [70], the Converse run-time underlying Charm++ [42]), MPC [67], the Realm event run-time system underlying Legion [78], Light-Weight Contexts [51], and ARGObots [73]. We focus on kernel-level mechanisms.

While the high-performance computing community has been reconsidering operating system design for tightly-coupled parallel computing for decades now [5, 26, 43, 47], the strict separation between layers of the stack has remained largely stagnant, especially at the user/kernel boundary.

Multi-kernels [4, 23, 24, 33, 64, 66, 81] attempt to strike a middle ground between general-purpose system software and specialized

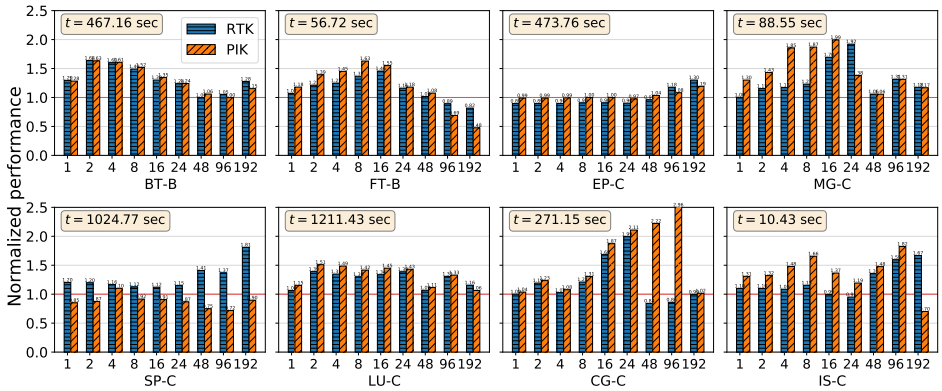


Figure 14: RTK and PIK performance relative to Linux as a function of CPUs used: NAS benchmarks on 8XEON; higher is better. Baseline (Linux OpenMP) is horizontal red bar at 1.0. t is the single threaded Linux absolute performance.

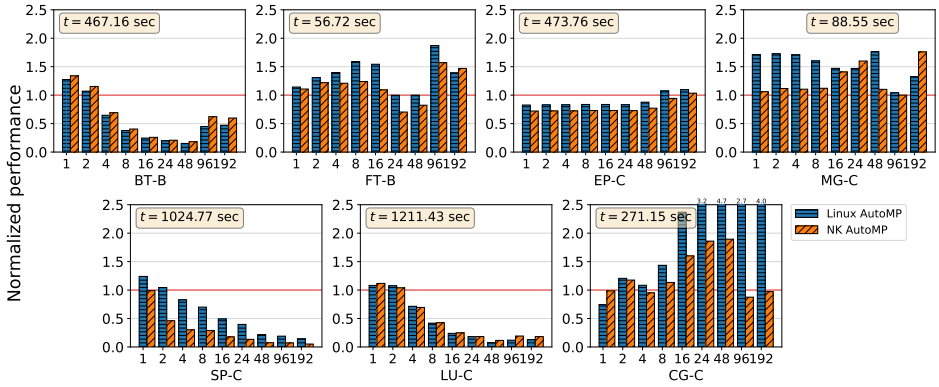


Figure 15: CCK performance relative to Linux as a function of CPUs used: NAS benchmarks on 8XEON; higher is better. Baseline (Linux OpenMP) is horizontal red bar at 1.0. t is the single threaded Linux absolute performance.

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 [8, 45, 52, 60, 72, 80].

Some Unikernels are constructed from application code using a high-level language [53], a natural progression from classic library OSes [19]. This allows unnecessary kernel functionality to be elided from the kernel image (as a library operating system, or libOS). 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 [6, 37, 68] are resurfacing in the form of OSes and Unikernels like Theseus [7] and RedLeaf [59]. However, the compiler uninvolved here; we argue that there is significant opportunity for bringing compiler technology and co-design across layers to bear for efficient parallelism.

Running user-level code within the Linux kernel has been most commonly seen in early extensions to Linux such as RTLinux [82], KURT [35], and RTAI [18] in which hard real-time application components were ported as kernel modules, analogously to RTK. Kernel Mode Linux is notable for providing a way of bringing general user code into the kernel (as we do with Nautilus in PIK) and then working to provide protection via type safety [54]. Software-based protection for managed languages was implemented in Singularity [36], and recent results show the promise of extending this idea to unmanaged languages [75].

HermitCore is a notable related project where OpenMP is run in an HPC-oriented, libOS kernel-context [48, 49]. In contrast, we presented *three* paths to running OpenMP code in the kernel, including via compiler support.

Extending LLVM for HPC has spawned an entire workshop/BoF series at the SC conference. CCK is in this vein. Efforts to integrate parallelism into compilation include Tapir [71], OpenMPIR [74], Vector Offload [77], PGAS via OpenSHMEM [44], INSPIRE [41], and HPVM [46]. None of these target kernel-level execution, however.

9 Conclusions

We demonstrated three different, effective techniques in the design space for bringing OpenMP into the kernel. The techniques allow OpenMP programs to benefit from direct interaction with the fully privileged machine, unimpeded by a traditional general purpose kernel. Our techniques have demonstrated performance gains relative to Linux for the NAS benchmarks that average about 22% and can be much larger. The OpenMP gains are similar to those previously observed for other run-times.

References

[1] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. 2020. SCAF: a speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 638–654. https://doi.org/10.1145/3385412.3386028

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

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinksi, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. *The NAS Parallel Benchmarks (NAS 1)*. Technical Report RNR-94-007. NASA.

[4] Andrew Baumann, Paul Barham, Pierre Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the* 22^{nd} *ACM Symposium on Operating Systems Principles (SOSP '09)*. 29–44.

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

[6] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP '95)*. 267–283.

[7] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. 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)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[8] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. 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)*. 250–257. https://doi.org/10.1109/CloudCom.2015.89

[9] J. M. Bull. 1999. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of the First European Workshop on OpenMP*.

[10] J. M. Bull and D. O'Neill. 2001. A Microbenchmark Suite for OpenMP 2.0. *SIGARCH Computer Architecture News* 29, 5 (2001), 41–48.

[11] J. M. Bull, F. Reid, and N. McDonnell. 2012. A Microbenchmark Suite for OpenMP Tasks. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP 2012)*.

[12] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

[13] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing *(CGO '12)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/2259016.2259028

[14] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu. Y. Wei, and David Brooks. 2012. The HELIX project: Overview and directions. In *DAC Design Automation Conference 2012*. 277–282. https://doi.org/10.1145/2228360.2228412

[15] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. 2012. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. *IEEE Micro* 32, 4 (July 2012), 8–18. https://doi.org/10.1109/MM.2012.50

[16] Barbara Chapman, Gabriel Jost, Ruud van der Pass, and David Kuck. 2007. *Using OpenMP: Portable Shared Memroy Parallel Programming*. MIT Prerss.

[17] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. 2005. TiNy threads: a thread virtual machine for the Cyclops64 cellular architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[18] L. Dozio and P. Mantegazza. 2003. Real-time Distributed Control Systems Using RTAI. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*.

[19] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP '95)*. 251–266.

[20] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[21] Kurt Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 ACM/IEEE conference on Supercomputing (SC)*. 1–12.

[22] Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, Robert W. Wisniewski, Yoonho Park, and Bryan Rosenburg. 2016. A Multi-Kernel Survey for High-Performance Computing. In *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*.

[23] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Kengo Nakajima, Yutaka Ishikawa, and Robert W. Wisniewski. 2018. 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)*. 116–125.

[24] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. 2016. 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)*. 1041–1050.

[25] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020. Compiler-based Timing for Extremely Fine-grain Preemptive Parallelism. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2020)*.

[26] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. 2010. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proceedings of Supercomputing (SC '10)*.

[27] Kyle Hale. 2016. *Hybrid Runtime Systems*. Ph.D. Dissertation. Northwestern University. Available as Technical Report NWU-EECS-16-12, Department of Electrical Engineering and Computer Science, Northwestern University.

[28] Kyle Hale and Peter Dinda. 2015. A Case for Transforming Parallel Runtimes into Operating System Kernels. In *Proceedings of the 24th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)*.

[29] Kyle Hale and Peter Dinda. 2016. 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)*.

[30] Kyle Hale and Peter Dinda. 2018. 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)*.

[31] Kyle C. Hale and Peter Dinda. 2018. An Evaluation of Asynchronous Events on Modern Hardware. In *Proceedings of the* 26^{th} *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*.

[32] Kyle C. Hale and Peter A. Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Virtual Execution Environments (VEE)*.

[33] Kyle C. Hale and Peter A. Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the* 12^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'16)*. 161–175.

[34] Kyle C. Hale, Conor Hetland, and Peter A. Dinda. 2016. Automatic Hybridization of Runtime Systems. In *Proceedings of the* 25^{th} *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 137–140.

[35] Sean House and Douglas Niehaus. 2000. KURT-Linux Support for Synchronous Fine-Grain Distributed Computations. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*.

[36] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. 341–354.

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

[38] H. Jin, M. Frumkin, and J. Yan. 1999. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance (NAS 3)*. Technical Report NAS-99-011. NASA.

[39] Ross Johnson. 2008. POSIX Threads for Embedded Systems (PTE). http://pthreads-emb.sourceforge.net/.

[40] Ross Johnson. 2012. Pthreads Win32: Open Source POSIX Threads for Win32. https://sourceware.org/pthreads-win32/.

[41] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. 2013. INSPIRE: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 7–17.

[42] Laxmikant V Kale, Josh Yelon, and Timothy Knauff. 1996. Threads for interoperable parallel programming. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC '97)*. 534–552.

[43] Suzanne M. Kelly and Ron Brightwell. 2005. Software Architecture of the Light Weight Kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting (CUG'05)*.

[44] Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt, and Barbara Chapman. 2015. LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*.

[45] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX*

ATC '14).

[46] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2018)*.

[47] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. 2010. 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 2010)*.

[48] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development *(PLOS '19)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3365137.3365395

[49] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6^{th} International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'16)*.

[50] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[51] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)*.

[52] 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. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *Proceedings of the 12^{th} USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. 559–573.

[53] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 461–472.

[54] Toshiyuki Maeda and Akinori Yonezawa. 2003. Kernel Mode Linux: Toward an Operating System Protected by a Type Theory. In *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Mumbai, India, December 10-14, 2003, Proceedings (Lecture Notes in Computer Science)*, Vijay A. Saraswat (Ed.), Vol. 2896. Springer, 3–17.

[55] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. 2021. NOELLE Offers Empowering LLVM Extensions. arXiv:cs.PL/2102.05081

[56] R. Minnich, J. Hendricks, and D. Webster. 2000. The Linux BIOS. In *Annual Linux Showcase and Conference*.

[57] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. 2016. Performance Implications of Transient Loop-carried Data Dependences in Automatically Parallelized Loops *(CC 2016)*. ACM, New York, NY, USA, 23–33. https://doi.org/10.1145/2892208.2892214

[58] Jun Nakashima and Kenjiro Taura. 2014. *MassiveThreads: A Thread Library for High Productivity Languages.* Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238.

[59] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[60] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15^{th} ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. 59–73.

[61] Omni OpenMP Compiler Group, University of Versailles Saint Quentin en Yvlines. 2014. NAS Parallel Benchmarks 3.0—Unofficial OpenMP C Version. https://github.com/benchmark-subsetting/NPB3.0-omp-C.

[62] OpenMP Architecture Review Board. 2008. *OpenMP Application Program Interface 3.0.* Technical Report. OpenMP Architecture Review Board.

[63] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I August. 2005. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 12–pp.

[64] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. 2015. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24^{th} International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. 149–160.

[65] Heidi Pan, Benjamin Hindman, and Krste Asanović. 2010. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM Conference on Programming Language Design and Implementation (PLDI)*.

[66] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W. Wisniewski. 2012. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Proceedings of the 24^{th} IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '12)*. 211–218. https://doi.org/10.1109/SBAC-PAD.2012.14

[67] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A unified parallel runtime for clusters of NUMA machines. In *Proceedings of the 2008 European Conference on Parallel Processing (EuroPar)*. 78–88.

[68] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the 16^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 291–304.

[69] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the 17^{th} Workshop on Hot Topics in Operating Systems (HotOS XVII)*. 7–13.

[70] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly.

[71] Tao Schardl, William Moses, and Charles Lieserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2017)*.

[72] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Proceedings of the 12^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 671–688.

[73] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 512–526.

[74] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing OpenMP Tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*.

[75] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. 2020. 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)*. 329–345.

[76] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[77] Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, Ayal Zaks, Gil Rapaport, Abhinav Gaba, Vasileios Porpodas, and Eric Garcia. 2017. LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*.

[78] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT 2014)*. 263–276.

[79] K. B. Wheeler, R. C. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 2nd Workshop on Multithreaded Architectures and Applications (MTAAP 2008, colocated with IPDPS 2008)*.

[80] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as Processes. In *Proceedings of the 9^{th} ACM Symposium on Cloud Computing (SoCC '18)*. 199–211.

[81] Robert W. Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. 2014. mOS: An Architecture for Extreme-scale Operating Systems. In *Proceedings of the 4^{th} International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*.

[82] V. Yodaiken and M. Barabanov. [n.d.]. A Real-Time Linux. Presented at USENIX 97; online at http://rtlinux.cs.nmt.edu/rtlinx/u.pdf.