# Cocoon: Custom-Fitted Kernel Compiled on Demand

Bernhard Heinloth, Marco Ammon, Dustin T. Nguyen,
Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
{heinloth,marco.ammon,nguyen,thoenig,sieh,wosch}@cs.fau.de

## Abstract

As computer processors and their hardware designs continuously evolve, operating systems provide many different assembly-level implementations for the same functionality. This enables support for new platforms and ensures backward compatibility for older ones at the same time. However, the source code of operating systems grows more complex and becomes much harder to maintain.

In this paper we explore ways to build made-to-measure system software by relegating work to the compiler which has necessary knowledge about the system at hand. We propose Cocoon, an approach for compiling a system-tailored and -optimized kernel at boot time. For two operating systems (i.e., Linux and FreeBSD) we demonstrate the soundness of the approach by hands of a prototypical implementation. The implementation shows various aspects of Cocoon, such as the ability to remove hard-to-maintain code while preserving and even increasing the system performance.

## 1 Introduction

To efficiently exploit the achievements of new hardware designs (i.e., performance improvements), made-to-measure system software [5, 11] is a *must*. But as complexity grows [16], it becomes increasingly difficult to provide hand-crafted software routines that leverage hardware offerings in the most efficient manner. Instead, system software (i.e., operating-system kernels) must be adaptable [13] and made-to-measure with a broad set of tools—in particular, with support from the compiler [10]. However, it is a chicken-and-egg problem: Compilers require an operating system (OS) to build a custom-fitted OS kernel. Hence, general-purpose OS kernels are commonly prebuilt by the distributor and shipped in binary format, without tailoring to an individual system, as shown in Figure 1. This approach, however, rules out the efficient use of target-system–specific build-time optimizations which, for example, exploit extensions to the instruction set architecture (ISA).

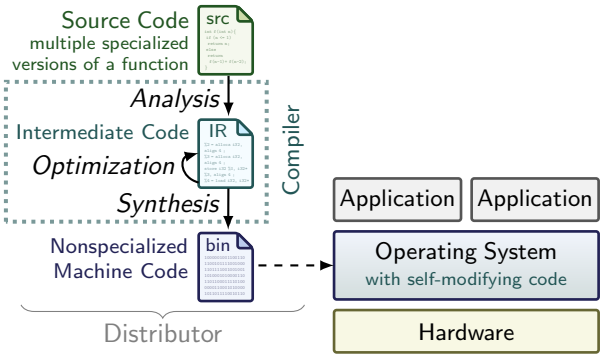


**Figure 1.** The status quo of today's OS distribution is shipping a generic binary kernel built by the distributor, while the system tailoring might be performed by code patching during run time.

OS distributors provide kernels that are generic for a specific instruction set architecture. These kernels run on a broad set of machines and provide support for a wide range of different processors and peripheral devices. The decisive disadvantage is that system-specific adjustments (i.e., the use of available processor extensions and selection of drivers) are relocated to run time. Therefore, distributors have to simultaneously deal with two contrary issues: On the one hand, adding functionality to support all possible peripheral devices (thus bloating the kernel), on the other hand, finding the lowest common denominator of the ISA to achieve machine-code compatibility—relinquishing all improvements potentially offered by processor extensions. Today's operating systems like Linux work around both issues at run time: Support for loadable modules reduces the memory footprint by loading only the required drivers, while self-modification selects the functions most suitable to the available ISA extension(s). For example, Linux provides no less than three different variants of the memory-copy function `memcpy` on the x86_64 architecture [17, `arch/x86/lib/memcpy_64.S`]. At run time, the OS kernel chooses from the available `memcpy` implementations in order to efficiently execute memory-copying operations on different x86_64 CPUs.

The practices mentioned above indicate several issues with the current state of the art. Firstly, OS kernels currently


*PLOS'19, October 27, 2019, Huntsville, ON, Canada*

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365398

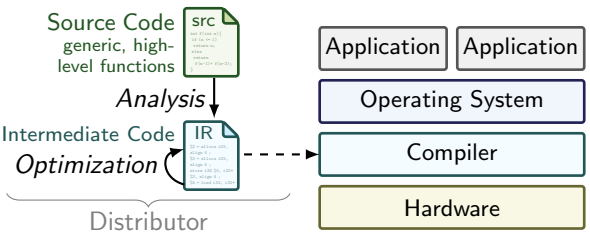


**Figure 2.** When compiling the OS's intermediate-representation code during run time, the kernel can fully utilize the features of the target hardware.

include hand-crafted, manually optimized program code for providing different implementations of various functions. This bloats the source code and requires very high maintenance efforts. Secondly, the segregation of drivers in separate compilation modules as well as the integration of assembly code in high-level programming-language source code act as a barrier for compiler analysis. Thus, optimizations leveraging additional context knowledge cannot be applied. Both maintenance efforts and optimization barriers can be mitigated by providing generic high-level implementations while relocating system-specific adjustments exclusively to the compiler. This leads to a drastic change where generic OS kernels are transformed to highly system-specific ones. By following this path, however, OS distributors would have to provide an OS kernel for each individual system configuration instead of one kernel per supported ISA. As this is infeasible due to the large amount of different ISA-peripheral configurations—and hence kernels—this paper presents a solution to the problem.

Instead of using pre-compiled OS kernels, we propose Cocoon, a *custom-fitted kernel compiled on demand* approach for operating-system kernels. Our approach exploits that compiling an OS kernel on target systems themselves is only a matter of seconds (cf. Section 5.2) with today's technology and that state-of-the-art compilers are capable of building highly optimized kernel binaries (cf. Section 5.3). Compiling OS kernels on the target system yields necessary insight with regards to available hardware and allows the kernel to adapt data structures and synchronization techniques already at compilation time. Cocoon only includes necessary kernel code, since available peripheral devices are known. Thus, loadable-module support can be omitted completely.

Cocoon integrates with existing system designs and extends the bootstrapping process by an additional step. During this step, Cocoon builds and compiles a highly optimized system-specific OS kernel. The resulting OS kernel is used henceforth until there are changes to the kernel, such as a version update or the availability of a security fix. Cocoon either works at the level of source code (e.g., C) or at the level of intermediate representation (IR), a partially translated source code, as shown in Figure 2. The latter is not only sufficient, but also reduces the kernel build time as the compiler-frontend's tasks and most optimizations have already been performed.

The contributions of this paper are threefold: Firstly, we present Cocoon, an approach aiming for target optimization by custom-fitting kernels, compiled on demand. Secondly, we present the implementation of Cocoon as an extensible framework which creates the necessary bootable build environment and utilizes this environment in conjunction with two open-source operating systems (i.e., Linux and FreeBSD). Thirdly, we evaluate the overall approach in two different scenarios: (1) we analyze and compare the start-up time of kernel images built by Cocoon and (2) we analyze and compare the performance of memory-copying functions Cocoon achieves in comparison to stock OS kernels.

To encourage additional research on this topic, we published our tool as open-source software[1].

The paper is structured as follows: Section 2 discusses background information and related work. Section 3 presents a high-level overview of the Cocoon approach, while implementation details are shown accordingly in Section 4. In Section 5, we evaluate the current prototype of Cocoon in real-world scenarios. Section 6 concludes the paper.

## 2 Background

Even though target-specific compilation of software is already quite common in some domains of computer science, most of the research focuses on user space applications. Target optimization in user space is either obtained by manually compiling all required software, or by using software based on the principle of just-in-time (JIT) compilation. However, those concepts can also be adopted and used to enhance the performance of an OS kernel.

Hunt and Larus developed the Singularity operating system written in a variant of the C# language, which is well suited for JIT compilation [7]. Even though they focused on building an operating system in a type- and memory-safe language, their approach can also be leveraged for target optimization. Their research group developed isolated kernel components which are translated from an IR to machine code at the time of installation, when all hardware features are known. The approach explores a wide range of concepts which impact the operating system's performance and security positively. Yet, the Singularity kernel and all software (both user and kernel space) are deeply coupled to C# and other concepts they introduce as part of their application binary interface. Hence, it is not possible to migrate already existing operating systems and applications.

In order to overcome the shortcomings of JIT compilation (e.g., high start-up latency) Nuzman et al. describe the concept of fat binaries, which are the combination of compiled programs and their respective IR in one file [12]. This enables

[1] https://gitlab.cs.fau.de/i4/pub/cocoon

starting the program immediately while also being capable of optimizing and recompiling certain parts of the application during execution when using a suited run-time environment. Nuzman et al. build upon the LLVM IR used by the clang compiler, as we do in our approach for the FreeBSD operating system. As their JIT compiler coexists with the running application, it is possible to perform optimizations based on its behavior at run time, especially for hot paths. This approach favors already existing software projects, as it does not require to change the software's source code. Instead, modifications to the build system are sufficient. Nonetheless, due to their sole focus on user-space applications, using their approach for kernel development is difficult.

Another approach for increasing an operating system's performance is described by Pu et al.: They recognized unused potential in the implementation of system calls, which are traditionally built in a generic fashion [14]. Instead, they design a kernel which synthesizes routines, capable of handling specific system calls, on demand. This enables them to create routines incorporating knowledge about the requesting process and environment. Their prime example is the `open` system call and subsequent `read/write` calls: Whenever an application calls `open` on a file, the kernel can create `read/write`-routines associated with the file descriptor. The synthesized routines can omit checks and inline specific code paths, such as the file system, based on the run-time knowledge. Even though Pu et al. achieve promising results, this approach requires a specific kernel design in mind and a great deal of manual labor has to be spent in implementing those optimizations. Cocoon uses modern compiler technology to improve the performance and maintainability of historically grown operating systems, without having to make invasive changes to the underlying source code.

A research group of *Stony Brook University* also tried to improve the FreeBSD operating system by adding JIT-compiler capabilities to compile the kernel from LLVM IR [1]. In addition, it was planned to use the JIT compiler to recompile certain modules of the kernel based on user interaction.

Bellard released the small boot loader TCCBOOT as a proof-of-concept for compiling a Linux kernel at boot time [2]. While the packaged C compiler TinyCC achieves fast compilation and boot times for Linux 2.4.26, neither target-specific tailoring nor optimizations are performed. Due to compiler shortcomings, it requires changes to the build system and source code.

## 3 Approach

Boot loaders often pass through multiple stages in order to load an operating system present as a binary executable. We introduce an additional stage in which we build the actual operating system before seamlessly executing it, as shown in Figure 3. In this stage, we first load a simple and generic kernel together with a minimal environment tailored to the

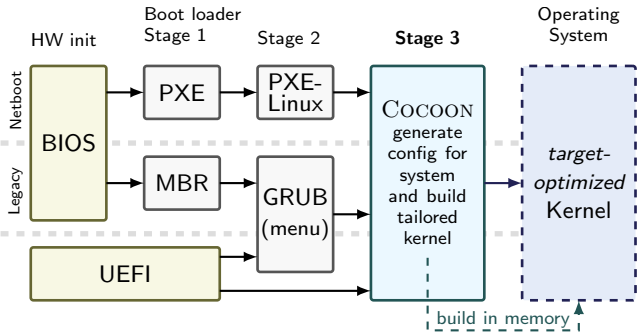


**Figure 3.** Cocoon acts as an additional boot-loader stage and integrates with common ways of booting a system.

requirements of the kernel's build system, with all necessary tools (such as the compiler). Then, a script automatically performs all steps required to build the kernel.

Depending on the preferred way of distribution, the source code of the kernel can be acquired from either a local storage device or downloaded on demand from network locations.

After verifying its integrity, the configuration phase starts: At this point, we can gather further knowledge about the target system to tailor the OS kernel to the actual hardware. The available information not only includes specific instruction sets but also attributes about symmetric multiprocessing (e.g., number of cores) and connected peripherals (for driver selection). Tailoring to the required components leads to a better adjusted system and additionally reduces both compilation time and unnecessary memory usage during run time due to the smaller kernel size.

Now we configure a new kernel for the system according to the information collected earlier. With this knowledge, selecting the correct number and model of CPUs and picking the drivers required for buses and I/O extension boards present in the system is possible. As kernels are targeted towards a wide variety of hardware, configuration options often include "generic" options, ignoring model-specific capabilities. With our approach, specialized options are chosen instead. Because hot-pluggable devices (such as USB hardware) may not be present on system start-up, driver support for them is always included.

Following the configuration step, the actual compilation starts. Here, the compiler optimizes as much as possible, since it has knowledge about the exact CPU type. During linking, we are able to improve the performance even further by applying link-time optimization (LTO), as only a single static kernel binary (without modules) is built.

Because system responsiveness and background services are not important in the building environment, we can fully employ all available system resources for the build process. However, there are several additional improvements that allow speeding it up even further: To deal with minor changes in the source code (such as patches with small bug fixes),

using a persistent compiler cache can have a significant impact on performance. Similar to hidden recovery partitions, we require a reserved partition for the build artifacts such as the kernel binary and debug information. In case no recompilation is required, we are able to immediately proceed with the previously compiled kernel.

Depending on the capabilities of the compiler, the source code can even be distributed in an intermediate representation. This not only avoids time-consuming lexical and syntactical analysis of several files, but also allows applying some optimizations beforehand.

As a final step, we map the new kernel into memory and boot it by jumping to the entry point.

This approach allows to automatically perform even cumbersome optimizations which require multiple runs of the operating system—these have previously been limited to experienced users due to the manual interaction necessary. For example, on initial execution, generation of profiling information can be enabled. The resulting data will be taken into account on subsequent builds to improve either the compilation [18] or selecting kernel features [9].

Any platform capable of self-hosting is suitable for this approach: The only requirement is the ability to compile and load the target operating system.

# 4 Implementation

To be able to support a wide range of existing tools, we implement the approach based upon a small Linux kernel with a temporary memory-based file system—the initial RAM disk (`initramfs`)—providing the build environment. We use the `initramfstools` of the Debian GNU/Linux distribution to create the compressed RAM disk.

Through hooks in the creation process, the compiler and all tools required for the build process are copied into the environment. Currently, we include either the GNU Compiler Collection (GCC) or the LLVM Compiler Infrastructure (in particular `clang`), but additional compilers can be added easily. Commonly required by the build system are basic Unix utilities provided by the GNU Core Utilities or BusyBox as well as build-automation tools like GNU Make. Additionally, we include GNU Wget or Git to retrieve the target operating system's source from remote locations.

A customizable shell script, instructing the retrieval of the source, configuration and build process, constitutes the heart of the approach and is executed automatically during boot. While usually only a small amount of sequential commands is sufficient to build the kernel, our script also performs sophisticated steps to better tailor the target operating system to the hardware. Since user interaction is possible but usually not desired in this step, boot parameters can be used as a flexible way of guiding the script.

After a successful build, we employ `kexec` to start our freshly created kernel: Using this system call, we can load the new binary kernel file into memory and directly boot it by jumping to its entry position, without any additional hardware initialization. The `kexec-tools` [3] provide a command-line interface for this functionality and support different protocols, such as the Multiboot Specification [6].

We demonstrate the universal applicability of our approach with two open-source operating systems on the i386 and x86_64 architectures.

## 4.1 FreeBSD Target

As FreeBSD supports being built with the LLVM-based clang compiler and its related toolchain, it offers us the possibility of using the intermediate representation emitted by the compiler as starting point. Compared to the traditional approach commencing from the original high-level–language code, several analysis and general optimization steps have already been performed ahead of time.

Due to some peculiarities in the build process of FreeBSD, it cannot be booted directly by the `kexec` system call. Therefore, it is necessary to add a boot-loader stage in-between Cocoon and the generated kernel binary. As FreeBSD's build system depends on its specific tooling, it does not run in a Linux-based environment. In order to generate IR files based on the kernel's source code and to extract the commands required for compiling and linking, slight modifications to the build system are needed. We apply these modifications to the 11.2 release for the i386 architecture.

## 4.2 Linux Target

In the case of Linux, we can boot into the latest kernel by updating the source code directly from Linus Torvalds's Git source tree [17] every time. The configuration is adjusted to the system, for example the CPU type and number of cores. While Linux has only rudimentary compile-time support for the ISA extension, we are able to apply patches that allow aggressive machine-specific compile optimizations [4].

To speed up the compilation process, we prevent arbitrary value changes in the source files (e.g., timestamps or version counters) and use a compiler cache [15] to access previously compiled fragments instead of recompiling them.

Performing the whole building process is by no means necessary: It is sufficient for us to build only the kernel binary itself (`vmlinux`) and omit the time-consuming steps of compressing and packaging the kernel with a loader (`bzImage`).

# 5 Evaluation

To show the broad applicability of our approach, we evaluate Cocoon with the popular open-source operating systems FreeBSD and Linux on different processor architectures (i386 and x86_64, respectively), followed by an in-depth analysis of the memory-copy function in Linux.

### 5.1 FreeBSD in Cocoon

We measure potential benefits in boot duration when using an IR as the basis, as opposed to source code. The tests were conducted with the FreeBSD source code. The linker used in this scenario is GNU gold (v1.14). Furthermore, we integrate LLVM 8.0.0 with the corresponding clang compiler and lld linker into Cocoon, allowing the usage of LTO and IR. As source files written in assembly cannot be transformed to IR, we generate traditional object files for those.

When compiling to object files, commencing with the 1875 C files takes 382 seconds while using the derived IR takes about a third of the time (127 seconds). Thus, a substantial decrease in compilation time can be achieved. 1621 source files have been translated into the IR of LLVM.

Linking of the resulting object files takes about 2 seconds in both scenarios. When using LLVM's LTO capabilities for compiling and linking the IR code the overall duration is about 242 seconds. For testing, we employ an Intel Core i5-4570 (4 cores at 3.2 GHz) with 32 GB RAM. However, all actions were performed sequentially, only utilizing one core.

### 5.2 Linux in Cocoon

For all Linux-based tests, an Intel Core i5-8400 CPU (6 cores at 2.8 GHz) with 16 GB DDR4 RAM and a 500 GB SSD was utilized. Note that both dynamic frequency scaling and energy-saving features were disabled and the CPU lacks support for simultaneous multi threading, all of which are common sources of unwanted benchmarking jitter.

Boot times with Cocoon generally fall into two categories: The initial build of the kernel and—if there are no changes to source code, configuration, or optimization settings—the subsequent boots with the previously compiled one. The first category's boot duration is influenced by the degree of optimizations performed, the latter is invariant. As a consequence, we measure full build times for a general and a target-tailored/-optimized kernel, as well as the boot time if the kernel is reused. Since the applied configuration can heavily influence compile times, comparable configuration files are mandatory: The unoptimized kernel uses the `defconfig`, a very basic configuration provided by the kernel developers. The optimized kernel utilizes a further tailored configuration and employs the `-march=native` compiler option for tuning the emitted code to the ISA extensions at hand.

A fresh, complete build takes 190 seconds from selecting Cocoon in the stage 2 boot loader until the login prompt of the booted operating system appears. If optimizations are enabled, this boot period is further extended by 3 seconds. In contrast, subsequent booting only takes 33 seconds. However, it is still slower than a traditional, direct boot into the operating system, lasting about 10 seconds. While this might be impractical for daily use on a mobile computer, for other systems, such as servers, this overhead is negligible.

**Table 1.** Comparison of three Cocoon-compiled kernels in multiple `perf bench` microbenchmarks (where higher values are better for `epoll/wait` and `mem/memcpy`, lower values are better otherwise): A traditional vanilla kernel, a kernel with a `memcpy` implementation in C as well as a target-optimized kernel with C `memcpy`.

| Benchmark | Linux Kernel | | | |
|---|---|---|---|---|
| | Vanilla | C | C-opt. | |
| `epoll/wait` | 287 476 | 283 117 | 292 120 | [Op/s] |
| `futex/requeue` | 8.2 | 9.1 | 8.1 | [µs] |
| `mem/memcpy` | 42.5 | 14.2 | 42.5 | [GB/s] |
| `sched/messaging` | 1132 | 1137 | 1010 | [ms] |

As the Cocoon approach ultimately not only strives to enhance performance but also to improve the kernel source code's maintainability by reducing dependencies on hand-written assembly code, we evaluate the performance impact of using a `memcpy` implementation provided in the C language as well as target-specific compiler optimizations. We compare three different kernels, all based on Linux 5.0.0 and compiled with Cocoon: A vanilla kernel with the original assembly implementations compiled with no further optimizations is used as a baseline for comparison to an unoptimized and a target-optimized kernel with a standard implementation in C. Furthermore, the same kernel configurations as in the boot time evaluation are used. We measure the actual results with the kernel-provided `perf bench` tool [17, tools/perf], which comprises several microbenchmarks evaluating the performance of common operations, such as inter-process communication and futex handling. Additionally, it provides an easy way to measure `memcpy` performance from user space.

The insights gained from our results (see Table 1) are threefold: (1) We learn that a compiler-optimized `memcpy` implementation in C can achieve the same throughput as the kernel's hand-written assembly routine. (2) When analyzing all results, the importance of `memcpy` in overall kernel performance is evident: In all microbenchmarks, the kernel with an unoptimized C implementation performs worse than the traditional, assembly-based kernel, which is otherwise completely identical. (3) Furthermore, the overall kernel performance can be increased through target-specific compiler optimizations. We conclude that, despite having the same `memcpy` throughput, the natively optimized kernel achieves better results overall compared to the default one.

### 5.3 Memory-Copy Function

The well-known `memcpy` function, as shown above, is a prime example for optimization at compile time: As copying data in memory is an essential part of almost every application and operating system, both software and hardware manufacturers have a strong interest in improving its performance. Thus,

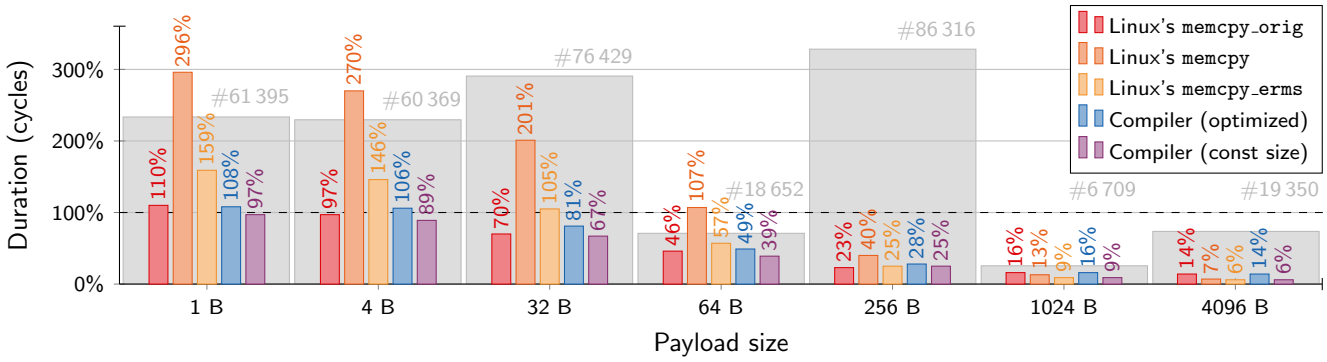


**Figure 4.** OS memory-copy performance of Linux implementations and compiler-generated routines (-O3 optimized with and without context information) relatively to a traditionally optimized (-O2) byte-copy loop at different payload sizes. The bars in the background present the average number of memcpy calls during Linux boot (assigned to the closest payload size as shown).

in recent microarchitectures, both new instructions and improvements to existing ones have been introduced—and have made their way into the Linux kernel, such as the *REP String Enhancement* [8, Section 2.6.6] in memcpy and *Enhanced REP MOVSB Operation* [8, Section 3.7.6] in memcpy_erms. During boot, Linux uses live patching to select the most appropriate assembly memory-copy function. While this mechanism can achieve high performance, it both adds complexity to the codebase and requires the operating-system programmer to know the most efficient implementation. Traditionally, this task lies within the domain of compiler developers.

```
char * memcpy(char * restrict to,
    const char * restrict from, size_t len) {
    for (size_t p = 0; p < len; p++)
        to[p] = from[p];
    return to;
}
```

**Listing 1.** Simple and clean memcpy implementation

Hence, we compare the throughput of all three Linux memcpy implementations with compiler-optimized versions of a simple, high-level memory-copy implementation—a loop copying data byte by byte from source to destination—as presented in Listing 1.

Our testing environment is based on Linux 5.0 with Gcc 8.3.0 on an Intel Core i5-8400 processor. As software in user space can be affected by other processes on the same system, all measurements are performed in a kernel module with both interrupts and task scheduling disabled on the current CPU. Other impacts on performance assessments include hardware factors such as dynamically changing clock speeds and power-saving features, all of which were disabled.

The results presented in Figure 4 demonstrate that recent compilers are able to generate efficient code. Actually, having the right conditions, current compilers emit the same code as provided by Linux. By employing techniques such as constant propagation, it is often possible to determine the size at build time, thus enabling the compiler to choose the best implementation considering the available enhancements.

Furthermore, due to additional factors (e.g., the start-up costs of REP operations), the results clearly indicate that there is no universal superior implementation—it strongly depends on the payload size.

During boot of a Linux kernel, 94 % of the executed memcpy calls copy less than 1 kB, which—on most hardware—results in using a version inferior to the one suggested by the compiler. Profiling might be a sufficient way to gather such knowledge for further compiler improvements.

## 6 Conclusion

Cocoon makes a fundamental change to the development and distribution of made-to-measure operating-system kernels. With the approach presented in this paper, optimization capabilities of modern compilers are exploited to tailor the OS to the underlying hardware. Hand-crafted assembly routines are no longer required in order to utilize a system's potential performance—all this by using high-level language source code, only. To take advantage of improvements of future microarchitectures, updating the OS is no longer necessary: Upgrading to a newer compiler version is sufficient.

For future work, we will take the approach to a further level by letting the compiler reside in memory. This would enable features such as automatic hot-spot optimization and adapting to non-functional requirements (i.e., high performance, low power consumption) during run time.

## Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89 "Invasive Computing" as well as the individual research grant SCHR 603/13-1.

## References


[1] Varun Agrawal, Amit Arya, Michael Ferdman, and Donald E. Porter. 2013. JIT Kernels: An Idea Whose Time Has (Just) Come. Poster presented at the 24th ACM Symposium on Operating Systems Principles (SOSP poster).

[2] Fabrice Bellard. 2004. TCCBOOT: TinyCC Boot Loader. https://bellard.org/tcc/tccboot.html

[3] Eric Biederman, Albert Herranz, and Jesse Barnes. 2019. Kexec Tools. https://git.kernel.org/pub/scm/utils/kernel/kexec/kexec-tools.git/

[4] Alexey Dobriyan. 2019. Linux 5.0-ad1: -march=native support. https://lkml.org/lkml/2019/3/4/698

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. 38–51.

[6] Bryan Ford and Erich Stefan Boleyn. 1995–96. Multiboot Specification version 0.6.96. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html

[7] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[8] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-041.

[9] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS '13)*, The Internet Society (Ed.). 1–18. http://www4.cs.fau.de/Publications/2013/kurmus_13_ndss.pdf

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[11] H. Massalin and C. Pu. 1989. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. 191–201.

[12] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. 2013. JIT Technology with C/C++: Feedback-directed Dynamic Recompilation for Statically Compiled Languages. *ACM Trans. Archit. Code Optim.* 10, 4, Article 59 (Dec. 2013), 25 pages. https://doi.org/10.1145/2541228.2555315

[13] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2, 1 (March 1976), 1–9.

[14] Calton Pu, Henry Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1, 1 (1988), 11–32. http://www.usenix.org/publications/compsystems/1988/win_pu.pdf

[15] Joel Rosdahl and Andrew Tridgell. 2019. ccache — a fast C/C++ compiler cache. https://ccache.dev/

[16] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the Sixth ACM European Conference on Computer Systems 2011 Conference (EuroSys '11)*. ACM, 47–60.

[17] Linus Torvalds et al. 2019. The Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

[18] Pengfei Yuan, Yao Guo, and Xiangqun Chen. 2014. Experiences in Profile-guided Operating System Kernel Optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2637166.2637227