Multiverse: Compiler-Assisted Management of Dynamic Variability in Low-Level System Software

Florian Rommel, Christian Dietrich, Michael Rodin, Daniel Lohmann
{rommel, dietrich, lohmann}@sra.uni-hannover.de, michael@rodin.online
Leibniz Universität Hannover, Germany

Abstract

System software, such as the Linux kernel, typically provides a high degree of versatility by means of static and dynamic variability. While static variability can be completely resolved at compile time, dynamic variation points come at a cost arising from extra tests and branches in the control flow. Kernel developers use it (a) only sparingly and (b) try to mitigate its overhead by run-time binary code patching, for which several problem/architecture-specific mechanisms have become part of the kernel.

We think that means for highly efficient dynamic variability should be provided by the language and compiler instead and present *multiverse*, an extension to the C programming language and the GNU C compiler for this purpose. Multiverse is easy to apply and targets program-global configuration switches in the form of (de-)activatable features, integer-valued configurations, and rarely-changing program modes. At run time, multiverse removes the overhead of evaluating them on every invocation. Microbenchmark results from applying multiverse to performance-critical features of the Linux kernel, cPython, the musl C-library and GNU grep show that multiverse can not only replace and unify the existing mechanisms for run-time code patching, but may in some cases even improve their performance by adding new dynamic variability options.

Keywords operating systems, compilers, Linux, dynamic variability, binary patching

ACM Reference Format:
Florian Rommel, Christian Dietrich, Michael Rodin, and Daniel Lohmann. 2019. Multiverse: Compiler-Assisted Management of Dynamic Variability in Low-Level System Software. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3302424.3303959

1 Introduction

The Linux kernel is known for its versatility with support for 32 architectures (v4.16.7) and application to a broad range of domains from small embedded systems up to supercomputers. Most of this flexibility is provided by means of *static variability*: More than 17 000 Kconfig options influence the code generation in the Kbuild build system and, on finer granularity, via the C preprocessor.

```
inline
void spin_irq_lock(raw_spinlock_t *lock) {
#ifdef CONFIG_SMP
    irq_disable();
    spin_acquire(&lock);
#else
    irq_disable();
#endif
}
```

Ⓐ Static Binding

Ⓑ Dynamic Binding

```
inline
{+ __attribute__((multiverse)) +}
void spin_irq_lock(raw_spinlock_t *lock) {
  if (config_smp) {
    irq_disable();
    spin_acquire(&lock);
  } else {
    irq_disable();
  }
}
```

Ⓒ Multiverse

[avg. cycles]	Ⓐ	Ⓑ	Ⓒ
SMP=false	6.64	9.75	7.48
SMP=true	28.82	28.91	28.86

Figure 1. Impact of static, dynamic, and *multiversed* variability in a (slightly simplified) Linux spinlock.

Take `CONFIG_SMP` in Figure 1.A as an example, a feature flag for the kernel's *symmetric multiprocessor* (SMP) system support. Here, it controls the conditional acquisition of a spinlock in `spin_irq_lock()`, a function used inside low-level kernel code for interrupt synchronization. By omitting the spinlock acquisition if not needed (i.e., in a *uniprocessor* (UP) system), the run time of this performance-sensitive operation could be reduced from 28.82 to 6.64 cycles on average. However, to yield these benefits, it has to be decided at compile time that the kernel is supposed to run on UP systems only, which in practice prevents all major Linux distributions from exploiting the option. The SMP configuration is more general, and multicore hardware appears to be "standard anyway" – a perception that is not necessarily true: Even on multicore hardware, the system may utilize only a single CPU most of the time to save energy. Low-cost virtual machines offered by cloud providers typically provide only a single CPU (but of course, more CPUs could be added later at run time for extra money). Such systems might even switch from UP to SMP and back to UP at run time.

Figure 1.B shows an alternative implementation using *dynamic variability* that would make it possible to support UP and SMP systems with the same implementation: In this setting, `config_smp` has become a global integer variable that is set at early boot time and expected to change only rarely at run time. The `spin_irq_lock()` implementation, however, has to check it on each invocation to decide if taking the extra spinlock is required or not. While this seems to reduce the run time in the UP case to 9.75 cycles at no extra cost for the SMP case, the numbers from Figure 1 reflect only the microbenchmark situation with a warm branch target buffer (BTB). In real kernel execution paths, the induced branch has a high chance to be mispredicted, which causes a penalty of 15–20 cycles[1] that would effectively kill the possible benefit. Kernel developers are aware of the costs of extra branches and avoid them as far as possible in critical paths.

With *multiverse*, it becomes possible to provide such dynamic variability without the extra branching costs by means of partial specialization and run-time binary patching. This is sketched in Figure 1.C: In a nutshell, the programmer just has to mark the `spin_irq_lock()` function and the `config_smp` variable (not shown) with an additional attribute; the GNU C compiler (GCC) then generates additional specialized versions of the function for `config_smp = {0, 1}` that omit the extra test and, thus, any potential penalty from branch misprediction. At run time, these special versions (multiverses) can be patched or even inlined into all call sites of `spin_irq_lock()` when the value of `config_smp` changes. Thereby, multiverse, in this case, effectively combines the performance of static variability (case A) with the flexibility of dynamic variability (case B). Compared to the `#ifdef` approach, the multiverse solution has the additional benefit that every code path gets compiled and statically checked.

1.1 Problem Statement

For the system-software developer, dynamic variability is a double-edged sword. On the one hand, it enables the run-time adaptation of the system to the actual environment (such as availability of extra processor features), while on the other hand, additional tests and branches affect the performance on every invocation – even though configuration decisions are rare events and in many cases occur only once at boot time. In general, the issue is dealt with in three ways:

Avoidance – the induced overhead outweighs the potential benefits. Example: The discussed spinlock implementation.

Acceptance – the overhead is considered marginal in comparison to the functional benefits. Example: Support for non-ascii search patterns in GNU grep.

Mitigation – the overhead is too high, but the flexibility is highly in need. Reduce the overhead by patching the decision into the binary code. Example: `alternative` macros in Linux to patch in processor-specific instructions.

In performance-critical parts of the code, Linux developers either employ the *avoidance* or, if unavoidable, the *mitigation* strategy. In fact, for the latter the Linux kernel contains several special-purpose mechanisms that all aim to mitigate the costs of dynamic variability by patching the binary code: The `alternative` and `alternative_smp` families of macros are used to mark and align single instructions in the code so that they can be overwritten by alternative instructions later at boot time. For instance, the *Supervisor Mode Access Protection* (SMAP) feature is deactivated at boot time by overwriting it with `nop` instructions if the boot processor does not support it. Xen paravirtualization [4] is supported by invoking critical instructions (such as `cli` and `sti`) not directly, but as *PV-Ops* – small functions – via a function pointer. However, the kernel patches the code at boot time and replaces the indirect calls by direct calls or even the respective target instructions (i.e., performs some sort of function inlining). Besides this, also Ftrace, Ksplice [3], and kpatch [29] bring their own code patching facilities to mitigate the costs of dynamic branches at run time.

Overall, it is apparent that the existing mitigation mechanisms based on binary code patching provide little reusability. They are highly architecture-specific and tend to be tricky to use, which increases long-term maintenance costs. Hence, they are rarely used, and dynamic variability is often avoided – or its overhead is simply accepted. We think that instead of establishing dozens of home-grown binary code patching mechanisms, the necessary means to patch the code for highly efficient dynamic variability should be provided directly by the instance that knows it best – the compiler.

1.2 About This Paper

We present *multiverse*, an extension to the C programming language and GCC, for exactly this purpose. Multiverse is easy to apply to existing software and provides dynamic variability via the well-known interface of conditional statements and function pointers. At run time, multiverse removes the overhead of evaluating them on every invocation. We claim the following contributions:

1. The multiverse language as a minimally intrusive extension to the C programming language to express dynamic variability (Section 2).
2. A compiler-assisted approach to ahead-of-time variant generation (Section 3).
3. Run-time support for highly efficient function-level binary-patching in both, user and kernel mode (Section 4).

[1] e.g., Intel Skylake: 16.5/19–20 cycles, depending μOp cache hit/miss. https://www.7-cpu.com/cpu/Skylake.html

We describe our implementation in Section 5. Our evaluation results (Section 6) from applying multiverse to performance-critical features of the Linux kernel, cPython, the musl C-library, and GNU grep show that multiverse replaces and unifies existing mitigation techniques for run-time code patching without any compromise on performance and makes it easy to improve existing system software by new or more efficient means for dynamic configuration. We discuss the implications of multiverse in Section 7, present related work in Section 8, and finally conclude our findings in Section 9.

2 The Multiverse Language Extension

With multiverse, we provide an extension to the C programming language that enables the developer to express dynamic variability in performance-critical paths. We target program-global configuration switches in the form of (de-)activatable features, integer-valued configurations, and rarely-changing program modes. These switches can be used to change the control- and data flow while multiverse removes the run-time overhead of evaluating them on every invocation.

In order to decrease the initial hurdle for the developer to use multiverse, we follow the principle of the least surprise and keep close to the regular C semantic: (1) configuration switches are annotated integer-like global variables that can be read, used, and written like regular variables. (2) performance-critical parts are explicitly annotated functions and all introduced optimizations work on the granularity of whole functions. (3) the developer has full control over the multiverse run-time mechanism and the employed binary patching must be invoked explicitly. All in all, multiverse keeps the program behavior (nearly) unchanged and integrates seamlessly into the C language.

As the only extension to the C syntax, we added a declaration attribute `multiverse` that can be added to global variables and function definitions. Annotated global variables are treated as configuration switches, while annotated functions are considered variation points where the switches take effect. For our implementation, we restricted the possible variable types to signed and unsigned integer types, as well as enumeration types. However, it should be possible to extend multiverse to globally-defined records that act as namespaces for configuration switches.

```
#define multiverse __attribute__((multiverse))
multiverse int config_smp;
multiverse void spin_irq_lock() {...};
```

If the multiverse API is not invoked, the multiversed program defaults to the usual execution, and all configuration switches are evaluated dynamically. Only with an explicit *commit* library call, we inspect the current assignment of all configuration switches and choose pre-generated, optimized variants for each annotated function. The selected variants are then installed in the running process image via binary patching such that these optimized variants are called from now on. In the following listing, we select and install the SMP variant for `spin_irq_lock()` and all other variation points that depend on `config_smp`:

`int multiverse_commit(void)` Inspect all multiversed variables, select optimized function variants, and install them in the process.
`int multiverse_revert(void)` Revert all modifications of the process image and use generic function variant in all places.

`int multiverse_commit_refs(void* variable)`
`int multiverse_revert_refs(void* variable)`
Commit and revert functions that reference a variable.

`int multiverse_commit_func(void* function)`
`int multiverse_revert_func(void* function)`
Commit and revert a single function.

Table 1. The Multiverse API

```
void hotplug_add_cpu() {
   nrcpu++;
   config_smp = true;
   multiverse_commit();
}
```

With the commit, the semantic of the multiverse function changes as they no longer dynamically evaluate the configuration switch, but assume it to be constant. Hence, a change in the configuration switch has no direct effect until the user explicitly re-commits the change or requests that the binary is *reverted* to the original, unmodified state. Multiverse guarantees that the currently committed function variant is called under all circumstances – even through function pointers or external calls (see Section 4).

Phrased more abstractly, we *bind* the functionality of the variation point at the commit time, instead of binding it at compile (`#ifdef`) or execution time (`if()`). Thereby, we come close to the run-time benefits of static variability while keeping the flexibility of dynamic variability. By default, we bind all referenced configuration switches within a function at commit time. However, we also provide support for binding a subset of the referenced variables for individual functions.

Table 1 gives an overview of the provided multiverse API. In addition to the universal commit and revert functions, which affect all multiversed functions in the program, there are more constrained alternatives. The `multiverse_{commit|revert}_func` class of functions only applies to a single function. The functions with the `ref` suffix make it possible to update all functions that reference a particular configuration variable. In contrast to the

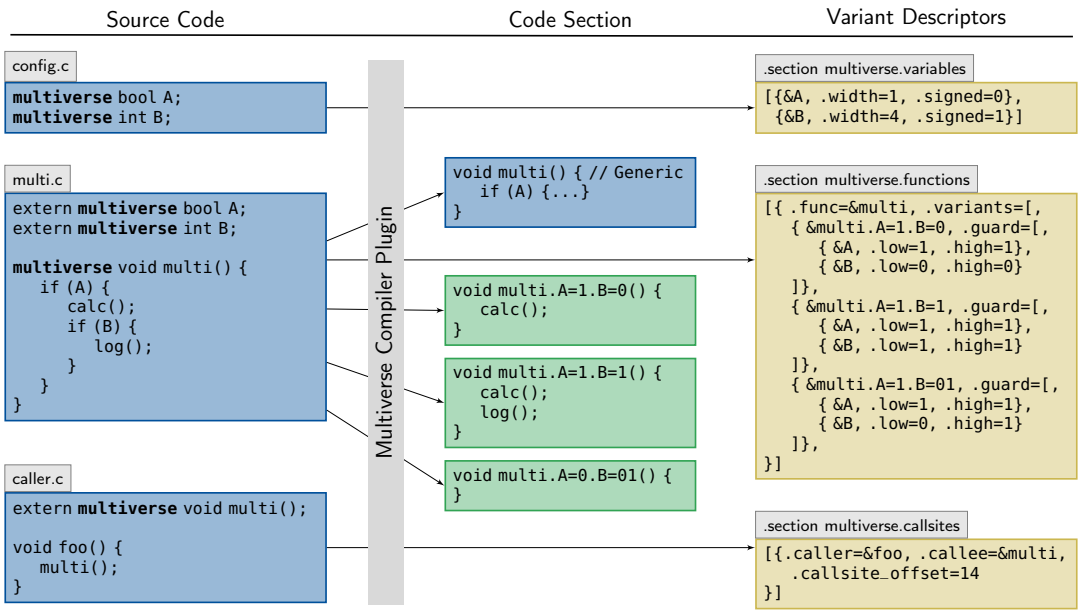

Figure 2. Ahead-of Time Variant and Descriptor Generation with the Multiverse Compiler Plugin. Besides the generic version of `multi()`, four variants are initially generated. Since both function bodies for A=0 are equal after optimization, we merge them and their descriptors. The generated descriptors reference the multiverse variables, functions, and relevant call sites and are located in separate sections.

universal API calls, these constrained alternatives give the developer fine-grained control over the patching state.

As multiverse hands over the explicit control over the commit mechanism, it is the users' responsibility to ensure the program is in a patchable run-time state. In order to be integratable into complex run-time environments, like the Linux kernel, it is crucial to make the consistency model independent of the dynamic-variability mechanism. Therefore, multiverse deliberately avoids synchronization. A transaction pattern with an additional object-layout change might look like this:

```
void subsystem_set_config(bool _A, bool _B) {
   wait_sync_and_lock(&subsystem);
   A = _A; multiverse_commit_refs(&A);
   B = _B; multiverse_commit_refs(&B);
   translate_objects(&subsystem);
   unlock(&subsystem);
}
```

Summarized, multiverse collects annotated configuration switches and generates specialized function variants with a compiler plugin. We remove the run-time penalty for evaluating the switches by binary patching (or even inlining) the variants into the call sites.

3 Ahead-of-time Variant Generation

Multiverse is split into two parts: the generation of variants with bound configuration switching and installing the variants according to the current configuration. For multiverse, we decided to do the variant generation as a compiler plugin, since there we already have a rich code-analysis and optimization machinery at hand. Thereby, we also avoid the requirement for a run-time code-generation framework, which should ease the acceptance within conservative system-level software projects, like the Linux kernel. Only a small run-time library (see Section 4) is required for the binary patching. Due to the compiler integration, we can also provide more reliable information for binary patching than other ad-hoc mechanisms that utilize inline assembler (see Section 1.1).

The compiler plugin works in four phases at different points in the code analysis and generation: (1) collect configuration switches and their value domains; (2) clone and specialize annotated function bodies for the cross product of all referenced switches; (3) merge function bodies that become equal after optimization; (4) generate descriptors used by the run-time library for configuration switches, variants, and call sites. In Figure 2, we give an overview of the variant-generation results.

First, the plugin identifies all variables that are annotated with the multiverse attribute as configuration switches, and we decide on a domain of values for which we want to generate variants. For integer-typed variables, we default to 0 and 1 as they act as the different boolean values in C (see

stdbool.h). For enumeration types, we choose all declared enumeration items as specialization values. Besides these default policies, we also provide an extended syntax for the multiverse attribute, where the user can explicitly set the domain of a configuration switch. In Figure 2, we can identify two configuration switches A and B, which have both the domain $\{0, 1\}$.

After the immediate-code generation, but before the optimization passes, we generate the different variants for every defined and annotated function. For that, we inspect all references to global variables in the function body and calculate a set of the used configuration switches. We build the cross product of the domains of the switches and clone the generic function body once for every possible assignment. For each clone, we replace each read of a switch with the constant value from the assignment and emit a warning if a switch is written. Furthermore, we rename all clones and mark the generic variant as non-inlineable to avoid the uncontrolled spread of configuration-switch dependent code. We will discuss the topic of variant explosion, which stems from the value cross product, in Section 7.

Since we did the value replacement before the compiler's optimization passes, the function variants are optimized perfectly for their configuration assignment. Of special effectiveness are the constant propagation, constant folding, and dead-code elimination as they directly benefit from the introduced constants. Due to the optimization, some cloned function bodies become equal, and we can discard all but one variant to save space. In Figure 2, the assignment $A = 0, B = 0$ and $A = 0, B = 1$ result in the same (empty) function body of `multi()`, and we have to keep only one variant.

Besides the code for generic functions and the optimized variants, we also generate descriptors for variables, functions, and call sites that hold important information for the run-time library (see Figure 2). For each configuration switch, we record the variable address, its width in bytes, and its signedness such that the run-time library can inspect the current value. For each annotated function, we record the address of the generic function and all variants. We also record a guarding expression over the configuration switches that indicates for which assignment a variant is usable. In the guarding expression we use value ranges ([`.low`, `.high`]) instead of single values to be able to cover multiple merged variants (see `multi.A=1.B=01`).

With the third class of descriptors, we collect information about every call site in the program that references a multiversed function. For that, we hook into the code-generating compiler backend to place a label exactly at the emitted call instruction and reference it from the call-site descriptor. Multiverse is limited to the call sites that the compiler is able to see. This precludes indirect calls via function pointers and calls from assembly or other foreign languages without multiverse support.

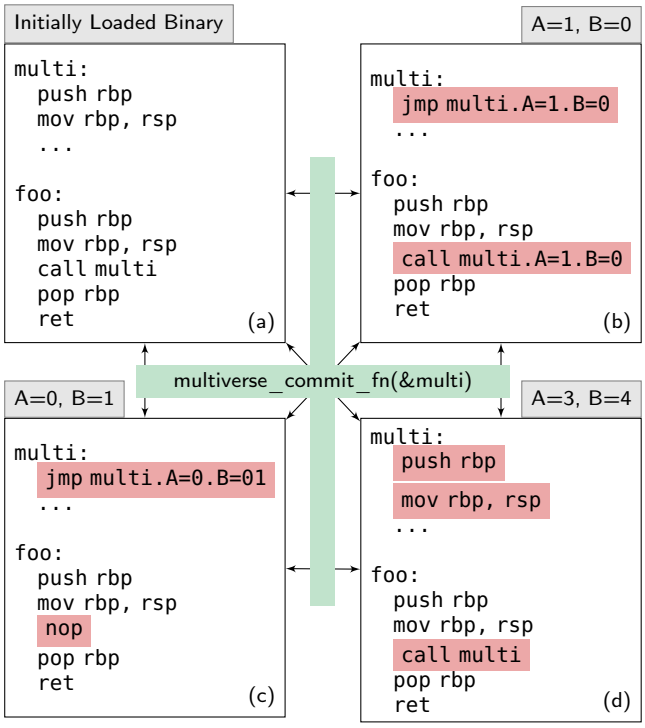

Figure 3. Binary Patching Mechanism. After writing the multiverse variables, the user must explicitly trigger the reconfiguration of the text segment via the multiverse API. If the run-time library does not find a valid variant, the text segment is reverted to the initial state (d).

4 Late Feature Binding via Binary Patching

The multiverse run-time library is a light-weight binary-patching mechanism that interprets the provided descriptors, selects variants, and installs them into the running process image. For this, it patches a concrete variant into the call sites and the address of the generic function. Figure 3 shows different text-segment configurations for the example in Figure 2.

For the commit, the run time inspects the configuration switches for their values and searches a viable function variant where all guard expressions are fulfilled. If no suitable function variant is found, we use the generic function, which still exhibits the correct behavior for the current value (see Figure 3 d). Since the generic variant lacks the being-bound semantic, we signal this situation to the user.

After a function is selected for installment, we inspect its call sites, check if they point to a expected call target, make the code location writable, and insert the new call target (see Figure 3 b). In order to force all function pointers and calls from assembly or foreign languages to execute the selected variant, we save the first bytes of the generic variant and overwrite it with an unconditional jump. If the user wants to revert the changes, we restore this overwritten prologue and

install the generic function in the call sites. After all changes to the text segment are done, we undo our write-permission changes and flush the instruction cache for the respective locations.

While the described mechanism already fulfills the multiverse semantic, we added a few optimizations and extensions that increase the utility of multiverse. First of all, the library detects if the function body of a variant is smaller than a call instruction, and we inline the body directly into the call site.[2] Thereby, we remove all function-call overheads and are able to eradicate empty function bodies completely by inserting a suitably large nop instruction (see Figure 3 c).

Since the run-time library already includes a binary patching mechanism, we extended the approach to capture another commonly-used form of dynamic variability where the variant generation is done manually by the developer: function pointers. For this, we also allow function pointers to be attributed as configuration switches such that the compiler plugin records all call sites. When such a function pointer is committed, we reuse the patching mechanism to insert it into the call sites.

5 Implementation Caveats

In this section, we report on technical details of the compiler plugin and the run-time library that do not influence the general operation of multiverse but are important for its actual application.

As C projects normally use separated compilation, we have to handle multiple translation units where variable and function definitions, and the call sites can be located in different source files. To keep the multiverse semantic over all object files, we demand that the attribute is added to the declaration (in the header file), such that the compiler knows for every occurrence of a function or variable that it is multiversed.

In order to collect the descriptors from different translation units, we use a separate binary section per descriptor type. Since the linker concatenates all sections with the same name from different objects, we can address the descriptors as a regular array. Furthermore, since we use the address-of operator in the descriptors, the compiler emits relocation entries and the linker (or loader) injects the actual numerical addresses. Thereby, we get support for relocatable and position-independent code for free. However, our current implementation does not support dynamic linking since we only inspect the descriptors of the binary itself. Nevertheless, there is no general problem in inspecting also the descriptors of all dynamically loaded modules.

Since we want to provide a unified binary-patching mechanism for function-level dynamic variability, the portability of multiverse is an important aspect. Hence, we separated the architecture-dependent functions (e.g., call-site patching, short function-body detection) and the platform-dependent functions (e.g., memory allocation, changing code protection) from the rest of the library. All in all, the whole run-time library consists of less than 850 lines of code. We currently support binary patching for IA-32/AMD64 (130 lines) and work on support for ARM. As platforms, we currently support the Linux user space (41 lines), the Linux kernel (63 lines) including the early boot, and the OctopOS [26] research operating system (49 lines).

The run-time library compiles down to 6.5 KiB code for AMD64/Linux user. For the descriptors, we add 32 bytes for every configuration switch, 16 bytes for every call site, and (48 + #variants · (32 + #guards · 16)) bytes per multiversed function to the binary.

The compiler part is implemented as a GCC plugin[3] and is tested for version 6.3 to 8.2. It works with the shipped Ubuntu and Debian Unstable GCC packages. While we tested the plugin only for IA-32/AMD64 and ARM code, we are confident in the portability as there is no platform specific code in the plugin since we operate on the architecture-independent intermediate representations of GCC (GIMPLE, RTL). In total, the GCC plugin consists of 2439 lines of code, whereby 1289 lines are located in GCC compatibility headers that we inherited from the Linux kernel.

6 Evaluation

In order to evaluate our approach in the kernel-space and user-space, we applied multiverse in different places in the Linux kernel and in cPython, the musl C library, and GNU grep. We used microbenchmarks to measure the performance of multiversed variation points. In this section we report on the integration of multiverse in these codebases and present our benchmark results.

6.1 Lock Elision and Paravirtual Operations

In the Linux kernel, we applied multiverse in two places: spinlocks and paravirtual operations. Both are already targeted by variability management (static or dynamic). Also, both mechanisms are known to have a considerable performance impact and thus are heavily optimized. This makes them perfectly suitable to assess the applicability of multiverse and its possible performance benefits.

We chose microbenchmarks to measure the performance changes caused by our modifications. This decision is based on the assumption that possible performance benefits will be rather small and thus difficult to observe in the ambient noise of an application-level benchmark. Besides this, our changes are applied to an already strongly optimized codebase. The measurements were performed on a recent desktop computer equipped with an Intel® i5-7400 CPU. On the software side we worked with the stable Linux kernel version 4.16.7

[2] On IA-32, a far-call site is 5 bytes large.

[3] Source code is available as free software at: https://github.com/luhsra/multiverse

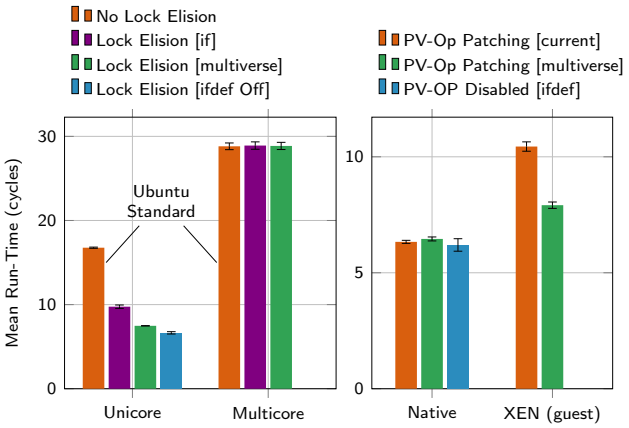

Figure 4. Average Run time (n=100 million) for spinlocks (lock+unlock) and paravirtual operations (sti+cli) with different elision and patching mechanisms.

and with GCC 7.3.0. We used the processor-local Time-stamp Counter (TSC) to capture the execution time with a high accuracy and minimal disturbance. The TSC provides a resolution similar to the processor's maximum frequency [19, p. 17-42]. Its value can be retrieved with a single machine instruction (`rdtsc`). We used the Linux `rdtsc_ordered` function, which combines the instruction with an appropriate memory fence (as suggested in Intel's Developer's Manual [19, p. 4-541]) to prevent inaccurate time-stamps caused by out-of-order execution. For each measurement we recorded 1 million samples, each consisting of 100 calls to the respective functions. In all result sets a small amount (not exceeding 0.04%) of clearly distinguishable outliers could be observed, presumably attributable to the occurrence of processor interrupts during measurement. These outliers were excluded in the evaluation.

In the case of lock elision, we transformed a build-time configuration switch to a run-time variation point (see Figure 1). We measured the spinlock functions in four different kernel variants: (1) the unmodified mainline kernel with an SMP-capable configuration without lock elision as used in all major Linux distributions, (2) a modified kernel with lock elision through control flow branching, (3) a modified kernel with multiverse-enabled lock elision, (4) the mainline kernel configured without SMP capability (which results in static lock elision). Each kernel variant was benchmarked in unicore (UP) mode and once again in multicore (SMP) mode – except kernel no. 4, which is statically determined to UP. The measurements are performed after the startup stage of the kernel, shortly before the invocation of the first user-space process. This point was chosen to minimize the interference with other system activities.

Figure 4 shows the average CPU cycle count that it takes to acquire and release a spinlock in each of the variants. In the multicore case the measurements reveal no significant difference between the three SMP-capable kernels. Regarding the mainline kernel and the multiversed kernel, this is consistent with our expectations as these two candidates should virtually run the same code. In this scenario, the presence of an additional control flow branch does not make the spinlock code in the modified non-multiverse kernel run measurably slower. As already stated (see Section 1), the reason for this lies in our microbenchmarking setup, which runs repeated invocations in a tight loop, resulting in a favorable situation for branch prediction techniques. When looking at the unicore results, we observe considerable differences between the candidates. The statically tailored UP kernel yields the best benchmark results, followed by the multiversed kernel, which is roughly twice as fast (in regard to spinlock functions) as the mainline SMP kernel. The modified non-multiverse kernel scores better than the mainline kernel but is significantly slower than its multiversed counterpart.

Multiverse prevents the compiler from inlining the generic variant. This could be a performance drawback. However, Linux kernel spinlocks are usually not inlined, despite in some rare cases where this is explicitly turned on.

Apart from the benchmark results, our kernel with multiversed spinlocks showed no errors or misbehavior during normal use. Multiverse records 1161 call sites of spinlock functions. Patching all these call sites takes approximately 16 milliseconds. Looking at the zipped kernel image, the total increase in size is 40 KiB, compared to the mainline kernel with the same configuration (total size: ~10 MiB).

In the area of paravirtualization, the Linux kernel is already equipped with a binary-patching mechanism. To be able to run as a paravirtualized guest, privileged operations have to be replaced by appropriate hypercalls, matching the hypervisor's interface. In the x86 architecture, Linux encapsulates all sensitive operations in a common interface that is implemented by the different hypervisor adaptions [32]. The set of these operations is referred to as operations for paravirtualized kernels (PV-Ops). PV-Ops are realized as function pointers. Depending on whether the kernel is running on real hardware or in a paravirtualized environment, appropriate implementations are assigned to these function pointers. They either implement the native operation or perform a call to a specific hypervisor.[4] This approach allows the use of the same kernel binary on native hardware and in paravirtualized environments alike. Due to performance issues with heavily used PV-Ops like interrupt enable/disable, the variability approach has been equipped with a binary patching facility to replace the indirect invocations with direct calls during run time. From a technical perspective the binary patching mechanism is similar to the one that multiverse implements. The approach is also capable of inlining function bodies in some special cases, and it also uses an ELF section

[4]LKML: Chris Wright. "[PATCH 0/7] x86 paravirtualization infrastructure" https://lkml.org/lkml/2006/10/28/191

to store information on patch-sites. Unlike multiverse, the PV-Ops patching mechanism does not come with a compiler extension. Therefore, every call to a PV-Op function pointer has to be manually wrapped inside a preprocessor macro, which generates the necessary inline assembler directives to record the call site's metadata.

We multiversed two PV-Ops: *enable interrupts* and *disable interrupts* (`sti` and `cli`). To evaluate the impact of this modification, three benchmarking kernels were built: (1) an unmodified baseline kernel with PV-Ops patching, (2) a modified kernel with multiversed interrupt operations, and (3) an unmodified kernel with statically disabled paravirtualization support (which causes all PV-Ops to be statically determined to use the native operations). We benchmarked the kernels on bare metal and as paravirtualized guests running inside the XEN hypervisor.[5] Figure 4 shows the benchmark results. In the non-virtualized environment, all the three candidates appear to perform similarly. The two dynamic kernels (1) and (2) are not worse than the kernel with statically disabled paravirtualization. The reason is that both patching mechanisms (current and multiverse) are capable of inlining these simple function bodies (consisting of a single `sti` or `cli` instruction) into the call site.

In the paravirtualized environment, the microbenchmark results show a difference between the baseline and the multiversed kernel. The reason for this is the usage of a custom calling convention in the current PV-Ops implementation, which has no volatile (or scratch) registers, i.e. all registers have to be saved and restored by the callee. Thus, if the register pressure on the caller side is low, a lot of unnecessary store and load operations are performed. Apparently, this is the case in our microbenchmarking code and the multiversed candidate, which uses the standard calling convention, is faster. In general, it is beneficial to integrate the variant generation into the compiler and let it handle the low-level details, such as calling convention, instead of implementing them manually. We assume that multiverse could fully replace the current PV-Ops mechanism without performance loss.

6.2 User-Level Case Studies

In order to demonstrate its broad applicability, we report on the integration of multiverse into three important user-space source-code bases: cPython (version 3.6.6), the musl C library (version 1.1.20), and GNU grep (version 3.1). For each project, we manually chose existing configuration switches and variation points that we multiversed, and measured the run-time impact with microbenchmarks. We used GCC version 8.2.0 to compile the assessed software projects. Each program was built with its default compile flags and configuration.

[5] http://www.xenproject.org

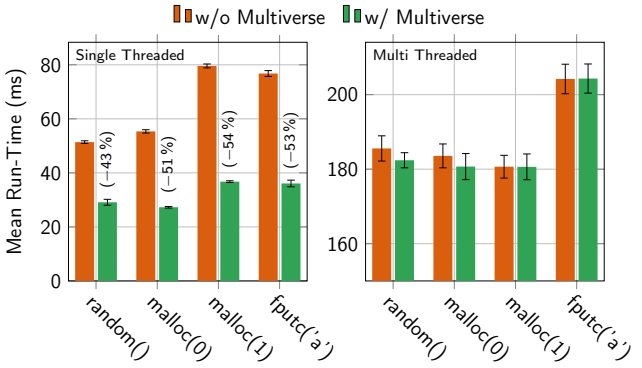

Figure 5. Improvements for the musl C library. Accumulated run time for 10 million invocations if musl is in single-threaded (left) or in multi-threaded (right) mode.

6.2.1 CPython

For cPython, we identified the boolean `enable` flag in the garbage collector as a good potential multiverse candidate, as it is only modified through API calls (`gc.enable()`, `gc.disable()`) and influences the object-allocation path (`_PyObject_GC_Alloc()`). For the multiverse application, we modified 12 source-code lines in a single file. Although we intensely tried to minimize any jitter (compile benchmark into interpreter, use Linux single user mode, core pinning, real-time priority), we could not produce stable results (not even for the unmodified baseline). Therefore, we cannot report on a significant influence of multiverse on the object-allocation time for cPython.

6.2.2 musl C Library

In the musl C library, an alternative implementation of the C standard library, we multiversed the locking mechanisms that ensure POSIX semantics in multi-threaded processes. For that, we extend the normal (owner-less) spinlock (`__lock()`) and the (owned) stdio file-object locking (`__lockfile()`) such that we skip the lock if only one thread is running. To detect the single-threaded situation, we use the existing `threads_minus_1` integer variable that musl keeps up-to-date with every `pthread_{create,exit}()`. We mark it as a configuration switch, annotate the lock and unlock functions as variation points, and call `multiverse_commit()` before/after the second thread is spawned/has exited. Thereby, the multi-threaded but multiversed scenario remains equal to the unmodified musl standard codebase. Overall, we had to change 67 source-code lines and 10 files.

We quantify the run-time impact on three C library functions with microbenchmarks that run the function 10 million times in a tight loop. We measure the whole benchmark program, including the overhead introduced by multiverse, with the `perf stat` tool, which is provided by Linux. In Figure 5, we show the average and the standard deviation

for 10 000 runs. The benchmarks were performed on an Intel® i5-6400 processor in the Linux single user mode on an Ubuntu 18.04. In order to minimize interference with other processes, we pinned the benchmark process to a single CPU and executed it with the highest real-time priority. We ran the benchmark in single-threaded or in multi-threaded mode (`threads_minus_1` $\in \{0, 1\}$) and without/with a multiverse commit. As `malloc(0)` is a special case in the specification (it can return `NULL`), we show results for the arguments 0 and 1.

In Figure 5 (right), we see that the removal of the early return by multiverse has only a minor impact on the multi-threaded scenario. However, for the single-threaded scenario, we see an immense impact of multiverse on all three functions that ranges from −43 (`random()`) to −54 percent (`malloc(1)`). For `fputc()`, which emits one byte per function invocation, we achieve an increase in output bandwidth from 124 MiB/s to 264 MiB/s. The impact of multiverse stems from call-site inlining and the thereby reduced number of branches (−40 % in the case of `malloc(1)`).

6.2.3 GNU Grep

GNU grep is a broadly used tool to search for regular expressions in text files. We identified a mode variable that affects the inner matching loop as a potential multiverse candidate: At start, grep decides upon the current language settings (i.e., locale) and the search pattern, whether the matching algorithm has to take care of multi-byte characters (i.e., UTF-8) or not. Since this mode is fixed after the setup is done, we can commit the specialized matching algorithm with multiverse. In total, we changed 50 source-code lines in 4 files.

We performed an end-to-end measurement for the run time of grep with the same hardware setup as for the musl C library. We invoked grep with the search pattern "`a.a`" on a 2 GiB large file of hexadecimal-formatted random numbers and measured its run time with perf. The processed file was placed in a virtual RAM drive (ramdisk) to ensure that the workload is CPU-bound. Averaged over 100 measurements, multiversing the mode variable reduces the overall run time by 2.73 percent (7.84±0.01 s → 7.63±0.01 s). Given the relevance and maturity of grep, we can assume its inner loop to be a well crafted and optimized piece of software; therefore, we consider the impact of multiverse as a noteworthy result.

7 Discussion

7.1 Design of the Patching Mechanism

We decided to implement multiverse's variability on function-level granularity. This has several advantages. It makes the patching process during run time very simple and robust because it basically consists of replacing a single call instruction. There is no need to perform complex binary decoding and run-time code generation. It also facilitates multiverse's hybrid approach with pre-compiled variants, which gives us the advantages of an ahead-of-time compiler despite the usage of run-time variability. Another important point is that function-level granularity offers great flexibility: There is no space limit or functionality restriction for variants.

The big threat arising from a function-level approach is the possibility of combinatorial explosion. This can easily happen when multiple multiversed configuration variables are referenced in a function. In many cases, however, it is not necessary to generate all possible variants, but only those with a clear influence on performance. The generic version of a function can still be used in all other cases. So our mitigation to combinatorial explosion is to give the developer fine-grained control over which variants are generated: The domain of a configuration variable can be explicitly specified (see Section 3). In addition, multiverse supports partially specialized function variants in which only some of the referenced configuration variables are fixed to a constant value.

Another design choice of multiverse is the usage of call-site patching to install variants. As an alternative we also considered body patching, that is replacing the function body with the to-be-installed variant instead of modifying the call sites. Body patching would be faster and would make it unnecessary to record the call sites. The main point against it is that it would require the multiverse support library to relocate variant bodies. This would cause a significant complexity increase of the library and thus contradicts our concept of a compiler-based approach with a small and simple run-time system. Also, in our opinion, the speed of patching is not crucial to overall performance, as changes to the configuration variables are rare and are executed relatively fast anyway – even with many call sites (see Section 6.1). Apart from that, the current patching mechanism allows the inlining of simple bodies into the call site, which in some cases leads to a huge performance improvement. Run-time body inlining could even be expanded in future versions by adjusting the sizes of call sites in certain cases to allow the inlining of larger bodies. Of course, there is always a trade-off between sophisticated body-inlining capabilities and the simplicity of the run-time library.

On the other hand, we chose to disallow the compiler to perform inline expansion on multiversed functions; more specifically, on the generic variant (see Section 3). This is a conscious choice as functions that have inlined a multiverse function would have an unclear semantic. Furthermore, prohibiting inlining stops the uncontrolled spread of multiverse-variable references throughout the binary, which makes it easier for the user of multiverse to ensure the system is in a patchable state. Also, function boundaries limit the number of multiverse variables and thus the number of generated variants.

All optimizations other than inline expansion are applied to multiverse functions (generic and specialized variants). Of course, the prohibition of inlining may have a negative impact on performance. In general, inline expansion can

improve execution speed in two ways: It removes the call overhead and allows further optimizations due to now larger function bodies [10]. Especially the more comprehensive optimization scope can be quite significant, depending on the function that is to be inlined. However, during our evaluation, we did not observe any conditions in which the overall performance of multiversed functions deteriorated. In many cases, inline expansion is not possible anyway, because caller and callee are located in different compilation units (e.g., Linux spinlocks). In some other situations, we observed that the compiler does not choose to inline a specific function despite the possibility to do so (e.g., grep).

7.2 Impact of Run-time Code Modification

Run-time binary patching enables multiverse to achieve a performance close to static variability. Yet modifying the text segment also entails some drawbacks and pitfalls. The ability to perform binary patching requires a von Neumann processor architecture. This prevents many small microcontrollers from using multiverse. Another requirement is that a program is allowed to write to its text segment. While this is usually prohibited by default, operating systems provide the means to change the protection of memory regions through system calls (POSIX[6], Microsoft Windows[7]). This is necessary for some widespread techniques, like just-in-time (JIT) compilation or loading dynamic libraries during run time. In principle multiverse should also work with BSD's *W^X (Write xor Execute)* security mechanism[8] where a memory page cannot be executable and writable at the same time. However, this worsens the synchronization problem: The developer would have to ensure that no thread executes code which is located in the affected pages during patching. Apart from that, there are no further special prerequisites concerning the execution environment. Other security mechanisms, like ASLR do not interfere with multiverse. In order to minimize security risks, multiverse makes the required memory locations writable only during the patching process (see Section 4). During normal operation the multiversed program is protected. An attacker would have to tamper the multiverse descriptors and interrupt the patching process in order to be able to execute code. This already requires a certain degree of control over the program. At this stage it would probably be easier to invoke the respective protection system calls directly.

An alternative to the usage of multiverse is the employment of ordinary function pointers to express dynamic variability. This eliminates the mentioned security risks and removes the need for synchronization and the patching cost. However, in comparison to function pointers, multiverse has two main benefits: the compiler-assisted variant generation and the usage of direct calls. Automatic variant generation allows expressing variability in a natural way (via conditional statements) – yet multiverse is also compatible with variability via function pointers. The additional cost of indirect calls is significant in various situations as we showed in our evaluation.

Another impact of run-time code modification concerns the use of debugging tools. We experienced that the minimally intrusive nature of multiverse mitigates the effects on debuggers. In GDB, we observed that patched call sites are not displayed correctly: It always shows the original call. Nevertheless, it is possible to break and step into multiverse variants as expected.

7.3 Potential of Protocol Violation

The patching process in multiverse must be initiated manually by the developer via an explicit commit library call (see Section 2). We are aware that this is a potential source of bugs caused by missing commit calls. As an alternative, we also considered an implicit approach where multiverse would automatically detect a write to a configuration variable and update the code accordingly. Though, this approach is accompanied by other problems and disadvantages. Most importantly, it is not always advisable to patch immediately after every variable change. In some cases, the developer might want to consolidate multiple writes and commit several configuration variables changes at once, especially when the program must apply some kind of synchronization mechanism. Furthermore, in C the compiler is not able to deduce every write to a variable (e.g., certain indirect modifications through pointers). This makes automatic change detection incomplete and thus does not fully solve the problem of potentially missing commits. Apart from that, the developer may prefer to use multiverse's functionality only selectively in certain execution paths in the program. By making the commits explicit, we follow the principle of minimal intrusion: The commit semantics aligns well with the design of the C language, which is unsafe but in return very flexible. The goal is to make multiverse usable in existing codebases without enforcing code refactoring. Despite this, the developer is free to use appropriate abstractions to prevent unintentional behavior, for example by using accessor functions to multiverse variables.

7.4 Soundness and Completeness

As we promise a defined semantic for multiversed functions, soundness and completeness are important aspects: (Soundness) As we generate the function variants by only replacing variable reads with constants and use the compiler infrastructure for the optimization, the resulting functions have, for the respective assignment, the same functionality as the original function. Furthermore, if no specialized variant is present, we fall back to calling the generic, un-specialized function.

[6] https://pubs.opengroup.org/onlinepubs/9699919799/functions/mprotect.html
[7] https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx
[8] https://www.openbsd.org/33.html
SELinux also implements this mechanism.

(Completeness) We inject an unconditional jump at the address of the generic function (which will be revised when multiverse reverts to the generic function, see Section 4). Therefore, we will capture all invocations of the function, also those that were made through function pointers, wild pointer arithmetic, call instructions from the assembler, or run-time generated code. For the aspect of completeness, the collection and the patching of call sites is a mere optimization.

Another general benefit of variability that is implemented with regular control-flow statements instead of preprocessor directives is the increased coverage of static code checkers. As the preprocessor can remove code sections that are enclosed with `#ifdef`, the compiler or another checker can only emit warnings for the code that is included in the configuration unless more complex checking strategies are employed [34, 20]. With control-flow–implemented variability, the static analysis sees all branches at once and can report errors and protocol violations.

7.5 Validity of Benchmark Measurments

In the kernel microbenchmarks we used the Time-stamp Counter (TSC) to capture the execution time. On modern Intel processors this counter is incremented monotonously at a constant rate which is close to the processor core's maximum frequency [19, p. 17-42]. In particular, this rate is independent of the current core clock frequency that may change from time to time. This is a threat to validity as the measurements could be influenced by frequency changes during the benchmarks. We mitigate this effect by performing a high number of repetitions (1 million) and capturing multiple calls per sample (100). Our observations showed that the measurements are quite stable. Furthermore, possible frequency changes would affect all benchmarks equally. An alternative to the use of TSC is the `CPU_CLK_UNHALTED` performance counter, which is incremented in accordance to the processor clock [19, p. 18-9]. However, this counter cannot be retrieved inside a XEN guest due to the lack of privilege in paravirtualization. For the spinlock benchmarks, we repeated the measurements using this performance counter and obtained qualitatively similar results as with the TSC method.

8 Related Work

On module granularity, dynamic code adaptation has a long tradition in operating systems: From dynamic linking in Multics [27] over dynamic module replacement in DAS [17], restartable servers in Mach [1], Spinlets [7], and more, we today find means for run-time loadable kernel modules in basically any general-purpose operating system.

Newer approaches focus more on dynamic update techniques which operate on a fine-granular level, (mostly) for the purpose of run-time security fixes: Examples include K42 [5, 6] and Proteos [16], for which run-time patching has been a design goal from the very beginning, but also more language-oriented approaches based on dynamic aspect weaving [13, 14, 33]. But also for existing systems, especially for Linux, several approaches have been suggested: DynAMOS [21] provides means to update non-quiescent functions, variables, and even datastructures in an unprepared Linux kernel. LUCOS [9] provides similar features if the kernel runs in a XEN VM. Ksplice [3], kGraft [28], and kpatch [29] have been suggested for integration into the kernel itself. Unlike earlier approaches, Ksplice analyzes runtime updates at an object-code level, enabling automated live-patch generation in many cases. A hybrid of kGraft and Ksplice was finally integrated into the mainline kernel. Binary patching is also used in tracing solutions, like Ftrace in Linux. Dtrace [8], which can be used on multiple systems (including BSD, Linux, and Windows), employs dynamic code rewriting to implement zero cost probes and instrumentation without the need of prior static preparation. Several further approaches target not the kernel, but userspace programs [18, 2, 25]. Among them, Ginseng [25] has many similarities to multiverse's hybrid patching approach in that it relies on the to-be-patched functions to be prepared by the compiler with the help of programmer-provided code annotations.

Multiverse, however, has a different scope than the mentioned mechanisms. Live-patching aims to install new code into the system in arbitrary places, whereas multiverse's goal is to facilitate efficient run-time reconfiguration by using predefined variants and patch points. Live-update approaches are typically not capable of call-site patching. Instead, code installation is achieved by using translation tables for patchable functions [5], introducing trampoline calls and redirection handlers [21, 9, 2, 25, 3] or replacing the entire text section [18, 16]. Multiverse is more light-weight and avoids this overhead by call-site patching, which makes it suitable even for very low-level and performance-critical code, such as an adaptable spinlock implementation. Windows dynamic libraries [30] implement late binding by function prologue patching, which is similar to what multiverse uses for redirecting indirect calls to the generic function.

The multiverse approach to install compiler-generated configuration-optimized variants of kernel functions at run time is closely related to in-kernel just-in-time compilation (JIT) techniques. Synthesis [23, 22] adapts frequently used kernel routines to their precise requirements at a specific system state by binding all known parameters and recompiling the code with an in-kernel C compiler. As with multiverse, this results in a partially specialized and optimized version of the kernel function. Recently, also the Linux kernel introduced a JIT compiler with the Extended Berkely Packet Filter (eBPF) [11]. This advanced form of the original Berkely Packet Filter [24] has evolved into a universal in-kernel virtual machine for kernel extensions [12]. eBPF

allows code to be dynamically attached to designated code paths inside the kernel [15]. Due to its event-based nature, eBPF is well suited for packet filtering, debugging, and analysis, but not as a general tool to facilitate dynamic variability in all parts of the kernel, especially the low-level core services. In general, JIT-based techniques are more flexible than multiverse, as they can perform variant generation and variation point determination at run time. The downside is that this requires a respective analysis framework plus an entire compiler to become part of the run-time system. Furthermore, compilation at run time can take considerable time and energy. The built-in compilers tend to produce less optimized code than an external ahead-of-time compiler due to the trade-off between compilation speed and optimization level.

Another compiler-assisted approach to dynamic variability management is the function multi-versioning (FMV) feature of GCC [31]. Like multiverse, it instructs the compiler to generate versions of functions, in this case specialized for the instruction-set extensions of different processor architecture generations. Variant selection and call-site patching is performed by the dynamic loader and cannot be overridden later. In comparison to multiverse, FMV has a very narrow area of application and is not able to handle generic functionality variation.

Finally, there are the existing special-purpose patching mechanisms in the Linux kernel (see Section 1 and 6). PV-Ops and the `alternative`/`alternative_smp` families of macros let the compiler do the variant and metadata generation through an elaborate combination of the C preprocessor, inline assembly statements and assembler directives. Variant determination and patching is done during early boot time. However, these solutions are not generalizable and difficult to maintain. Multiverse, in contrast, delegates the architecture-dependent part to the compiler and comes with an easy to use language interface to define variation points.

9 Conclusion

Efficient run-time variability is still an insufficiently researched area. Previous work often involves invasive intervention in programming and design paradigms or requires complex run-time environments. In real-world system software, dynamic variability is either avoided, realized with overhead or implemented by special-purpose binary patching mechanisms. With multiverse we provide an extension to the C language to efficiently handle dynamic variability by means of binary patching with compiler assistance. Unlike previous work, we focus on a minimally intrusive mechanism that is easy to integrate into legacy code bases. In our tests, the usage of multiverse was possible without any difficulty in kernel-space (lock elision & PV-Ops) as well as in user-space (cPython, musl C-library & GNU grep). Benchmark results certify a performance benefit in many cases and show that multiverse is able to compete with existing special-purpose binary patching mechanisms. The main objective of multiverse is to narrow the gap between dynamic and static variability by allowing the developer to easily employ run-time configurability at zero or low cost.

Acknowledgments

We thank Valentin Rothberg for his early work on multiverse, as well as our anonymous reviewers and Tim Harris for their detailed and helpful feedback.

This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1 and LO 1719/4-1.

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. “MACH: A New Kernel Foundation for UNIX Development”. In: *Proceedings of the USENIX Summer Conference*. USENIX Association, June 1986, pp. 93–113.

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. “OPUS: Online Patches and Updates for Security”. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM’05. Baltimore, MD: USENIX Association, 2005, pp. 19–19.

[3] Jeff Arnold and M. Frans Kaashoek. “Ksplice: automatic rebootless kernel updates”. In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys ’09)* (Nuremberg, Germany). Ed. by John Wilkes, Rebecca Isaacs, and Wolfgang Schröder-Preikschat. New York, NY, USA: ACM Press, Mar. 2009, pp. 187–198. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519085.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the Art of Virtualization”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*. Vol. 37, 5. ACM SIGOPS Operating Systems Review. New York, NY, USA: ACM Press, Oct. 2003, pp. 164–177. DOI: 10.1145/945445.945462.

[5] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. “Providing Dynamic Update in an Operating System”. In: *Proceedings of the 2005 USENIX Annual Technical Conference*, pp. 279–291.

[6] Andrew Baumann, Jonathan Appavo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. “Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly”. In: *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, USA). Berkeley, CA, USA: USENIX Association, 2007, pp. 337–350. ISBN: 999-8888-77-6.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. “Extensibility safety and performance in the SPIN operating system”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95)* (Copper Mountain, CO, USA). New York, NY, USA: ACM Press, Dec. 1995, pp. 267–283. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224077.

[8] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. “Dynamic instrumentation of production systems”. In: *Proceedings of the 2004 USENIX Annual Technical Conference*. 2004, pp. 15–28.

[9] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. “Live Updating Operating Systems Using Virtualization”. In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE ’06. Ottawa, Ontario, Canada: ACM, 2006, pp. 35–44. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1134767.

[10] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. “The effect of code expanding optimizations on instruction cache design”. In:

IEEE Transactions on Computers 42.9 (1993), pp. 1045–1057. ISSN: 0018-9340. DOI: 10.1109/12.241594.

[11] Jonathan Corbet. *BPF: the universal in-kernel virtual machine*. https://lwn.net/Articles/599755/. May 2014.

[12] Jonathan Corbet. *Extending extended BPF*. https://lwn.net/Articles/603983/. July 2014.

[13] R. Douence, T. Fritz, N. Loriant, J. M. Menaud, M. S. Devillechaise, and M. Suedholt. "An expressive aspect language for system applications with Arachne". In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*. Ed. by Peri Tarr. Chicago, Illinois: ACM Press, Mar. 2005, pp. 27–38.

[14] M. Engel and B. Freisleben. "Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects". In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*. Ed. by Peri Tarr. Chicago, Illinois: ACM Press, Mar. 2005, pp. 51–62.

[15] Matt Fleming. *Extending extended BPF*. https://lwn.net/Articles/740157/. Dec. 2017.

[16] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Safe and automatic live update for operating systems". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)* (Houston, Texas, USA). New York, NY, USA: ACM Press, 2013, pp. 279–292. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451147.

[17] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. "Dynamic Restructuring in an Experimental Operating System". In: *IEEE Transactions on Software Engineering* SE-4.4 (1978), pp. 298–307. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231515.

[18] Deepak Gupta and Pankaj Jalote. "On-line software version change using state transfer between processes". In: *Software: Practice and Experience* 23.9 (1993), 949–964. DOI: 10.1002/spe.4380230903.

[19] *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes 1–4*. Intel Corporation. May 2018. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[20] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)* (Portland). New York, NY, USA: ACM Press, Oct. 2011. DOI: 10.1145/2048066.2048128.

[21] Kristis Makris and Kyung Dong Ryu. "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)* (Lisbon, Portugal). Ed. by Thomas Gross and Paulo Ferreira. New York, NY, USA: ACM Press, Mar. 2007, pp. 327–340. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273031.

[22] Henry Massalin. "Synthesis: An Efficient Implementation of Fundamental Operating System Services". PhD thesis. New York City, NY, USA: Columbia University, 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.4871.

[23] Henry Massalin and Calton Pu. "Threads and Input/Output in the Synthesis Kernel". In: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*. New York, NY, USA: ACM Press, 1989, pp. 191–201. ISBN: 0-89791-338-8. DOI: 10.1145/74850.74869.

[24] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A new architecture for user-level packet capture". In: *Proceedings of the USENIX Winter 1993 Conference*. USENIX'93. San Diego, California: USENIX Association, 1993, pp. 259–269.

[25] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. "Practical Dynamic Software Updating for C". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 72–83. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991.

[26] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "OctoPOS: A Parallel Operating System for Invasive Computing". In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA'11)* (Salzburg). Ed. by Ross McIlroy, Joe Sventek, Tim Harris, and Timothy Roscoe. Vol. USB Proceedings. 2011, pp. 9–14.

[27] Elliot I. Organick. "Intersegment Linking". In: MIT Press, 1972. Chap. 2, pp. 52–97. ISBN: 0-262-15012-3.

[28] Vojtěch Pavlík. *kGraft: Live patching of the Linux kernel*. https://www.suse.com/media/presentation/kGraft.pdf. 2014.

[29] Josh Poimboeuf and Seth Jennings. *Introducing kpatch: Dynamic Kernel Patching*. https://rhelblog.redhat.com/2014/02/26/kpatch. 2014.

[30] Jeffrey Richter. *Advanced Windows*. 3rd. Redmond, WA, USA: Microsoft Press, 1997. ISBN: 1-57231-548-2.

[31] Victor Rodriguez, Abraham Duenas, and Evgeny Stupachenko. *Function multi-versioning in GCC 6*. June 2016. URL: https://lwn.net/Articles/691932/ (visited on 09/23/2018).

[32] Rusty Russell. "lguest: Implementing the little Linux hypervisor". In: *2007 Linux Symposium* 2 (2007), pp. 173–177. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-173-178.pdf.

[33] Wolfgang Schröder-Preikschat, Daniel Lohmann, Wasif Gilani, Fabian Scheler, and Olaf Spinczyk. "Static and Dynamic Weaving in System Software with AspectC++". In: *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Track 9*. Ed. by Yvonne Coady, Jeff Gray, and Raymond Klefstad. IEEE Computer Society Press, 2006. DOI: 10.1109/HICSS.2006.437.

[34] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue". In: *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA, USA). Berkeley, CA, USA: USENIX Association, June 2014, pp. 421–432. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler.