FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Florian Rommel

Multiverse: Compiler Assisted Dynamic Variability Management in the Linux kernel

Masterarbeit im Fach Informatik

2. November 2017

Please cite as:
Florian Rommel, “Multiverse: Compiler Assisted Dynamic Variability Management in the Linux kernel” Master’s Thesis, University of Erlangen, Dept. of Computer Science, November 2017.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Multiverse: Compiler Assisted Dynamic Variability Management in the Linux kernel

Masterarbeit im Fach Informatik

vorgelegt von

Florian Rommel

geb. am 31. Dezember 1990
in Kaiserslautern

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
	Prof. Dr.-Ing. habil. Daniel Lohmann
	Christian Dietrich, M.Sc.
	Andreas Ziegler, M.Sc.
Beginn der Arbeit:	**1. Mai 2017**
Abgabe der Arbeit:	**2. November 2017**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Florian Rommel)
Erlangen, 2. November 2017

ABSTRACT

Linux can be considered as a prime example of variability and configurability in software systems, regarding its various areas of operation from small embedded devices to supercomputers. In most cases, variability requirements are resolved statically, during the compilation of the kernel, by selecting and inserting the configured features into the final compilation product. However, configuration can not always be determined statically. Sometimes, it is necessary to defer the adaption of a software system to the run-time, making the configuration dynamic.

Dynamic variability is, due to its application at run-time, often linked to considerable performance penalties, arising from branches in the control flow. In the Linux kernel, run-time binary patching mechanisms are applied in order to eliminate these performance issues by removing branches in the control flow. Unfortunately, such live patching techniques are often difficult to apply and usually incorporate complex implementations which increase maintainability costs.

Function Multiverse is a new approach to handle dynamic variability requirements. In this work its applicability as an alternative method to the current solutions in the Linux kernel is investigated. Due to its compiler-based approach, Multiverse promises to make binary patching easier and more efficient to realize.

As part of this work, Multiverse was successfully applied in the Linux kernel. Results from microbenchmarks show an increased performance in many cases. Besides this, a possible complexity reduction in the source code is suggested in case of a more extensive use of Multiverse throughout the kernel.

KURZFASSUNG

Linux ist mit seinen vielseitigen Einsatzgebieten, vom kleinsten eingebetteten Gerät bis hin zum Supercomputer, ein Paradebeispiel für Variabilität und Konfigurierbarkeit in Softwaresystemen. In den meisten Fällen werden die Variabilitätsanforderungen statisch, also zum Zeitpunkt der Kompilierung des Betriebssystemkerns, aufgelöst, indem die konfigurierten Leistungsmerkmale ausgewählt und in das Kompilat integriert werden. Konfiguration kann jedoch nicht immer statisch festgelegt werden. Um die Flexibilität eines Systems zu gewährleisten, kann es nötig sein, bestimmte Anpassungen erst dynamisch, während der Ausführung des Programms vorzunehmen.

Dynamische Variabilität ist, bedingt durch die Anwendung zur Laufzeit, oft mit erheblichen Leistungseinbußen verbunden, welche durch Verzweigungen im Kontrollfluss entstehen. Um diese Einbußen möglichst auszuräumen, werden im Linux-Betriebssystemkern Mechanismen zur Code-Ersetzung (*Binary-Patching*) während der Laufzeit angewendet, mit dem Ziel die Verzweigungen zu eliminieren. Solche Binary-Patching-Techniken sind jedoch oft schwierig anzuwenden und bringen in der Regel komplexe Implementierungen mit, was den Wartungsaufwand des Quelltextes erhöht.

Mit dem Einsatz von *Function Multiverse*, einem neuen Ansatz zur Behandlung dynamischer Variabilitätsanforderungen, wird in dieser Arbeit eine alternative Methode zu den bisherigen Lösungen im Linux-Kern untersucht. Durch seinen compiler-basierten Ansatz verspricht Multiverse Binary-Patching zur Laufzeit einfacher und effizienter umsetzen zu können.

Multiverse konnte im Rahmen der Arbeit erfolgreich im Linux-Kern eingesetzt werden. Messergebnisse von Microbenchmark-Tests zeigen in vielen Fällen eine Verbesserung der Leistung. Es wird außerdem für eine mögliche Reduzierung der Komplexität des Quellcodes argumentiert, ausgehend von der Annahme einer umfassenderen Anwendung von Multiverse im Linux-Kern.

CONTENTS

1 INTRODUCTION

The Linux kernel is well-known for its versatility. Despite not initially designed with variability in mind, Linux now supports a variety of different hardware architectures, peripheral devices and different features, making it one of the most adaptable software systems. One and the same operating system kernel is used on embedded devices, desktop computers, servers, smartphones and supercomputers.

Of course, although running Linux, the different target systems do not run the same binary code. Linux's extensive configurability allows to build highly customized binaries, tightly tailored to the specified requirements during compilation. This type of build-time variation is referred to as static variability.

Although, static variability being the backbone of the kernel's flexibility, it is not sufficient for all configuration determinations. Sometimes, it is required or simply more convenient to defer a variation decision to the run-time. The need of this dynamic form of variability is heavily dependent on the target system the kernel runs on. For instance, an embedded system with fixed hardware components that will never change, may not need the kernel to employ dynamic variability mechanisms. In contrast, a Linux-based desktop operating system, which is meant to run on a variety of different processors and to communicate with diverse peripheral hardware, has for sure dynamic variability requirements.

Dynamic variability is, due to its application at run-time, often linked to considerable performance penalties. This is caused by branches in the control flow which are needed to establish the desired adaption to the run-time-determined configuration. To eliminate the performance issues, binary patching mechanisms are used in some places in the Linux kernel. Binary patching uses code modification to remove branched code, replacing it with non-branching variants according to the current configuration. Unfortunately, such patching techniques are often difficult to apply and usually incorporate complex implementations. This makes them difficult to maintain and limits their profitable usage in the kernel.

Function Multiverse is a dynamic variability approach, proposed and developed by Rothberg et al. [29]. It is designed to combine the flexibility and simplicity of control flow branches with the performance of binary patching. Therefore, Multiverse provides a compiler-assisted hybrid approach between dynamic and static variability. Configuration

possibilities are detected during compilation and used as a basis to compile specialized code manifestations which are put in place at run-time.

In this work the applicability of Multiverse in the Linux kernel is investigated and assessed. The main objectives consist of finding appropriate use cases in the kernel and adapting Multiverse to be usable for these application areas. Furthermore an evaluation of possible benefits and drawbacks, compared to other dynamic variability solutions, is also part of this work.

The thesis is structured as follows. In Chapter 2, a detailed definition of dynamic variability is given, before examining common solutions to dynamic variability requirements. After that an overview of run-time variability in the *Synthesis* project is given. Chapter 3 elaborates on current solutions to dynamic variability in the Linux kernel. This includes loadable kernel modules, machine instruction patching and operations for paravirtualized kernels (PV-Ops). A detailed overview of *Function Multiverse* is given in Chapter 4. Its concepts, components and functionality are explained. As part of this work, Multiverse was applied in two places in the Linux kernel. In Chapter 5, these application targets are presented and expected benefits resulting from the use of Multiverse are discussed. Furthermore, problems and challenges, that were caused by the application of Multiverse in the kernel, are described. In Chapter 6, the results of microbenchmarks, which were performed to compare Multiverse to current dynamic variability solutions, are presented. Beside this, it is elaborated on possible code complexity reductions. The chapter closes with a brief outlook to possible future work.

2 FUNDAMENTALS

This chapter tries to give an overview of fundamental concepts and common techniques related to variability in software systems. First, detailed definitions of variability in general and dynamic variability are given. After that, the two common concepts of dynamic dispatch and dynamic binding are revisited in the light of dynamic variability. The chapter concludes with a short disgression to the variability approach applied by the *Synthesis* kernel.

2.1 Variability in software systems

Variability describes the ability of a software architecture to be modifiable in a preplanned way [8, p. 31] in order to produce derivatives, differing from the original architecture in certain well-defined aspects. Thus, it enables the adaption of a software artifact by configuration, so that different user requirements can be met. Variability can be realized at different stages during a software artifact's lifecycle. This includes variations during architecture design (architecture derivation), compilation, linking or run-time [36].

The basis of variability are *variation points*, which represent a single point in the code where a choice between distinct *variants* can be made based on some type of *configuration*. There is a lot of terminology in use that is linked to software variability. The following list gives an overview of the terms that are used in this thesis. They are based on the terminology in other works [36, 1, 37].

- **Variant.** This can be described as a manifestation of code, realizing a defined piece of the modifiable behaviour in the software system.
- **Variation point.** This denotes a point in the software system where a choice between different variants can be made. Oftentimes a collection of specific variants is tied to it.
- **Configuration.** It represent a certain choice that can be made, concerning variability. It is used to determine the active variant in a variation point. Therefore, it must have a suitable form (e.g. a run-time variable or a preprocessor symbol).

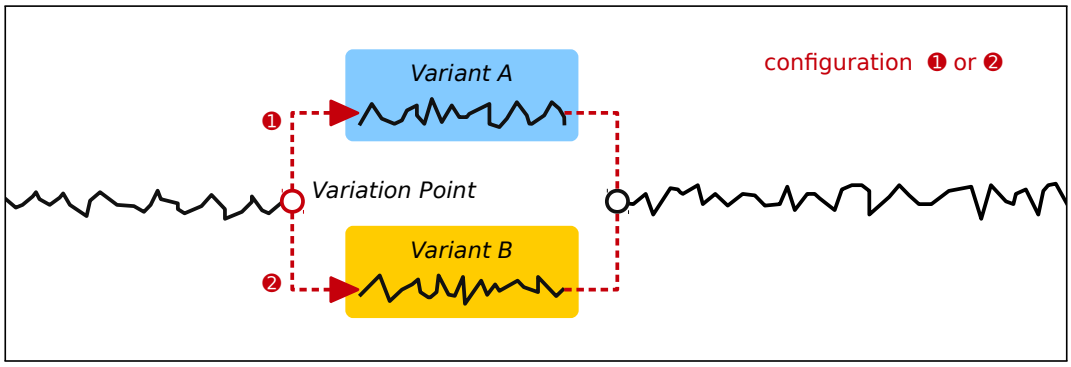

Figure 2.1 – Variation point with two variants *A* and *B*. They are selected according to the configuration state *1* and *2* respectively.

Figure 2.1 illustrates an example variation point with two associated variants that are selected according to a configuration state.

Variation points can be seen as the basic building blocks of variability management. They are often aggregated to features, which model coherent, high-level configuration entities [32]. Features are an abstraction of requirements [36]. Especially in complex systems, features may be arranged in a tree structure with constraints and dependencies between them.

Not only extraordinary large and complex programs but most of non-trivial software artifacts may need to realize variability at some point. Most variability mechanism kick in during compile-time or link-time[1], and thus are applied before the system is deployed. This is referred to as static variability.

As an example for a simple static variability implementation consider Listing 2.1: a C preprocessor (CPP) conditional statement produces different code depending on a configuration switch. The two branches are the variants in this case. They are tied to the variation point which is denoted by the `#ifdef` preprocessor directive. The preprocessor symbol `CONFIG_X` represents the configuration.

```
1 #ifdef CONFIG_X
2   // Implementation of the desired behaviour when X is "switched on"
3 #else
4   // Implementation of the alternative behaviour
5 #endif
```

Listing 2.1 – A variation point realized by the use of the C preprocessor (CPP)

[1]Note that link-time refers to static linking in this case. See Section 2.3.2 for a detailed look at dynamic linking.

2.2 Definition of dynamic variability

Besides the described static variability, there is also a dynamic form of variability. The difference between the two forms is in the time the configuration gets evaluated and applied. Dynamic variability is about resolving the configuration at run-time, rather than during the design or build process of a program [15]. This means that variants at variation points are selected and bound during the execution of a program – opposed to the build-time or design-time binding in static variability.

Of course this implicates that configuration variables, variants and variation points must be somehow contained in the executable form of the software artifact. This is different to static variability, where all the appliance connected to this form of variability is removed after building the program.

There is a possibility in between run-time and build-time binding: *load-time binding*. This means that the variant selection and binding occurs when the program is loaded by the operating system. Loading happens before the actually execution of the program but after building the executable. I regard this type as dynamic variability. The reason for this is that it can be seen as conceptually more similar to run-time binding due to its occurrence after the deployment of the software system, leading to a regular re-evaluation of the configuration on every start of the program.

Unlike variant binding, the generation of possible variants must not necessarily happen during run-time or load-time in dynamic variability. Often, the software artifact is equipped with all the necessary variants generated before its deployment[2] [15]. However, there exist dynamic variability solutions were variant generation is performed during run-time as well. A prominent example is the Synthesis kernel that is described in Section 2.4.

In the previous section the C preprocessor (CPP) was employed to express static variability (see Listing 2.1). CPP macros will expand during compilation; therefore, the configuration variable, the variation point and its variants will have totally disappeared in the executable. To migrate the example to dynamic variability it is sufficient to replace the `#ifef` with a C conditional statement (an "ordinary" `if`) and to change the `CONFIG_X` switch to be a C variable. Listing 2.2 shows the migrated example.

```
1 if (config_x) {
2   // Implementation of the desired behaviour when X is "switched on"
3 } else {
4   // Implementation of the alternative behaviour
5 }
```

Listing 2.2 – A variation point realized by a conditional statement

[2] Rothberg et al. [29] refer to this approach as "hybrid technique" due to its static aspect, regarding the precompiled variants.

Using standard language techniques, like such a simple conditional statement, is a straightforward way to realize dynamic variability. In some cases, however, run-time performance may considerably suffer from the constant re-evaluation of the configuration variable.

Therefore, there exist more advanced dynamic variability mechanisms that involve installing a variant permanently without re-evaluating the configuration variable in every pass. The usual way to achieve this is to modify the executable code at run-time. This is called run-time (or live) binary patching. In contrast to ordinary control flow modification techniques, binary patching implementations usually do not have programming language support and require specific knowledge of the processor architecture and the run-time environment.

2.3 Common approaches to manage dynamic variability

Dynamic variability requirements have been around for a long time during the history of software engineering. This section introduces *dynamic dispatch* and *dynamic linking* – two common approaches of dynamic variability management. Furthermore the relationship between control flow modification by standard language features and dynamic variability is elaborated.

2.3.1 Dynamic dispatch in object-oriented languages

Polymorphism via *dynamic dispatch* in object-oriented programming (OOP) can be considered as a form of run-time variability. Polymorphism is an important principle in object-oriented languages, meaning that a variable can be of more than one possible type [7]. This is in contrast to monomorphic languages where each variable has fixed static type.

Listing 2.3 shows a simple example written in C++. Template metaprogramming is used to realize polymorphism. Function `call_foo` takes a parameter of any type that provides a fitting method named `foo`. Variability results from varying implementations of `foo` in different type definitions. These implementations can be considered to be the variants, whereas the method call is the variation point.

```
1 template<typename T>
2 void call_foo(T poly_param) {
3     poly_param.foo();
4 }
```

Listing 2.3 – Static polymorphism via template metaprogramming in C++. Function `call_foo` takes a parameter (`poly_param`) of any type that provides a fitting method named `foo`.

However, the decision which parameter type is used for a specific invocation of `call_foo` is resolved during compile-time. Thus, the polymorphic behaviour in Listing 2.3 is static. Dynamic variability can only be realized by using the dynamic form of polymorphism. It allows the developer to defer the decision which type is used to the run-time.

Sticking with the previous example, using dynamic polymorphism involves creating an abstract class and declaring the method `foo` as virtual (see Listing 2.4). In this case polymorphic behaviour is realized via dynamically dispatching the correct implementation for `foo` depending on the specific type of the argument (in this case a sub-class of the abstract class `Fooable`). This allows the type implementation to be selected at run-time.

```
1 class Fooable {
2     public:
3     virtual void foo() = 0;
4 };
5
6 void call_foo(Fooable *poly_param) {
7     poly_param->foo();
8 }
```

Listing 2.4 – Dynamic polymorphism via dynamic dispatch in C++. An abstract class is used. Potential types for the `call_foo` parameter are required to inherit from this class.

In statically typed languages, such as C++, dynamic dispatch is usually implemented involving lookup tables for each type [26]. The correct member function is called indirectly using the appropriate pointer in the table. Despite the employment of various optimized mechanisms, dynamic dispatch causes a performance loss on modern processor architectures mainly due to the indirect invocations [25]. This is even worse in dynamically typed languages where all the method calls are dynamically dispatched [26, 39].

2.3.2 Dynamic linking

Dynamic linking is the prevailing method to link libraries to the program during load-time or run-time. It is realized by modifying and extending the binary data of a process in the main memory. Traditionally, the main goal is to reduce the memory footprint of programs, but it is also used to satisfy patching and variation requirements. Thus, dynamic linking can be seen as variability mechanism in the broader sense.

In contrast to *static linking* which is carried out after compilation, dynamic linking allows global symbols in the code (variables and procedures) to stay undefined until the program's execution [16]. This is accomplished by deferring the binding of these symbols to the launch of the program or even to a later point. This comes with many benefits,

such as memory savings (due to sharing code between processes) and the possibility to patch libraries without modifying the referencing programs [2].

The basis of linking are modules. Each module encapsulates types, variables and procedures. Some of these symbols are exported to be used (imported) by other modules, which on their part may also export symbols. This results in an acyclic dependency hierarchy [14]. Some modules constitute a program, whereas others are combined to form a library, depending on their position in the dependency hierarchy. Figure 2.2 shows an example hierarchy consisting of a program and its dependencies.

The job of a dynamic linker is to find the libraries which a program depends on, load them and establish the links to the imported symbols. This is a recursive process because the libraries can have dependencies themselves [14]. To make this possible, every compiled module (also referred to as *object file*) contains two directories: the *entry table* which lists the exported symbols together with their position in the code, and the *link table* which contains all the module's imports [14].

There are various strategies to realize dynamic linking. They differ in the point in time the linking is done (load-time vs. run-time) and the way how the actual links to external symbols are established.

The simpler methods of dynamic linking install all modifications to the process' memory at load-time, that is, before the program is actually running. This is what many

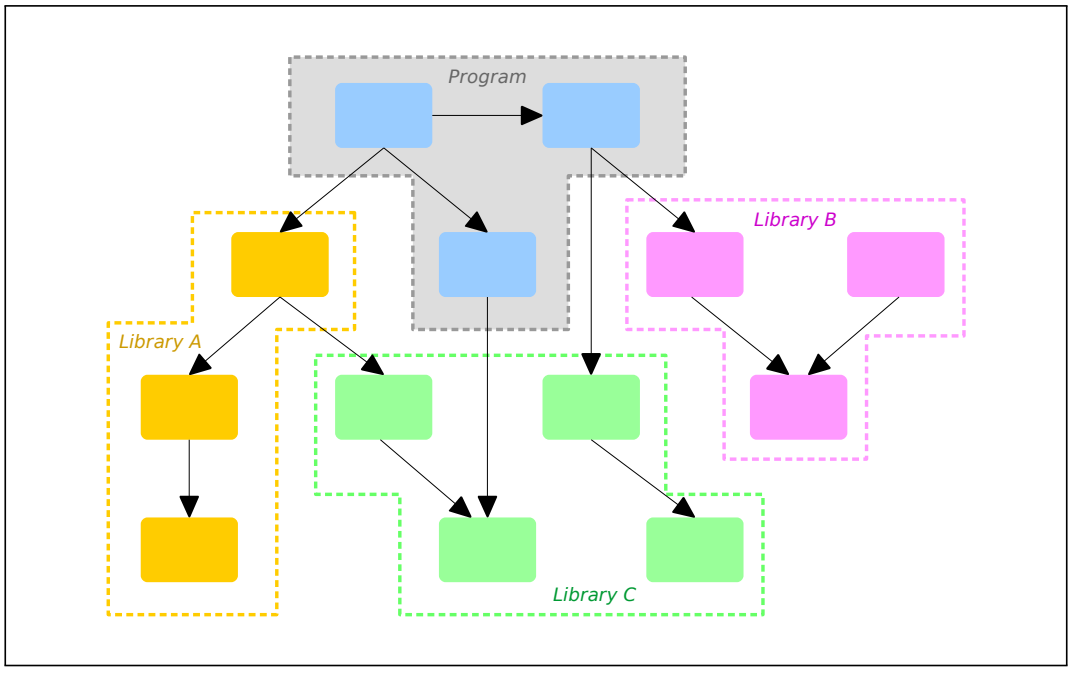

Figure 2.2 – Program and library modules and their dependencies. An arrow denotes a dependency relationship.

Unix-like operating systems implement [16]. Regarding link-establishment, either a table lookup approach or a code modification mechanism is used.

In the *table lookup* approach [14] the linker loads all libraries that the program depends on and replaces the symbolic references in the link tables with actual pointers to the respective memory locations. Each reference in the code can then be resolved *indirectly* through the module's link table.

Due to the indirect access, a table lookup on each reference is problematic when it comes to performance on modern processors. The *code modification* [14] method improves this by replacing references to external symbols directly in the code, making the indirection obsolete. Due to the modification of actual executable code, this method can be considered as a typical binary patching approach. This implicates that every reference to an imported symbol must be somehow recorded to be available to the dynamic linker. As a drawback, the method may reduce the ability to reuse code pages in the main memory due to modifications tailored to the specific memory layout of the process.

Besides these widespread load-time mechanisms, there are different approaches to defer the dynamic linking process to the run-time. The historic *Multics* system, which was the first system to implement dynamic linking [9], uses a table lookup approach but delays the establishment of links to the run-time. In Multics, dependent modules are loaded on demand [18]. The link table, which lists the external reference data, is called linkage segment [10]; it contains a pointer pair for each referenced symbol, allowing to indirectly access the symbol. Initially these pointer pairs are set to an invalid value because the referenced modules have not been loaded yet. When the pointer pair is accessed for the first time, the invalid value causes the processor to trap [18, 10], transferring control to the dynamic linker. The system is then able to load the needed dependencies into memory, replace the invalid link section pointer with the respective address and finally continue with the program's execution beginning with a restart of the interrupted access operation. This approach makes dynamic linking in Multics very flexible. The technique uses tables and indirect access, and thus makes it very similar to the mechanisms used to realize dynamic dispatch in object-oriented languages (see Section 2.3.1).

As stated earlier, many Unix-like systems implement load-time dynamic linking, which does not offer the same flexibility that Multics had. However, some of them provide another technique [38, 16], next to the common load-time approach, which allows to realize dynamic linking at run-time. It takes the dynamic linking concept one step further by making the access to external modules entirely programmable. This is accomplished by a library-based dynamic linker that can be used by the program during run-time. The library provides functions that allow the loading of arbitrary libraries, searching them and retrieving pointers to their symbols.

The mechanism is also known as *dynamic loading* and is available in POSIX-compliant operating systems and in Microsoft Windows [21, pp. 107ff.]. Dynamic loading is different from other dynamic linking approaches, as it makes references to external symbols programmable – opposed to the compile-time determined placement of links in other approaches. This allows to have references which must not be known at compile-time or

link-time, making this technique a powerful variability tool. For example, it can be used in a program to load user-provided plugins – an use-case not possible to realize with any of the other dynamic linking methods.

In summary, the term dynamic linking is used to refer to various different approaches which defer the linking process past build-time. All mechanisms have in common that they extend the executable code by loading additional modules (usually aggregated in libraries) to the process' memory. Some mechanisms run during load-time, before the program is executed, others are applied during run-time. Another difference is the used link establishment method. Either indirect referencing or binary-patching-enabled direct access can be used to resolve links to external modules.

2.3.3 Dynamic control flow modification

Complex approaches of dynamic variability management were introduced in the previous sections. However, the most straightforward realization of dynamic variability is to use standard language facilities to modify the control flow. This is called *dynamic control flow modification* in this work. It refers to *conditional statements* (branches) and *function pointers*.

At this point, the reader might ask if plain conditional statements can really be considered as a realization of dynamic variability. The answer that is suggested here is: it depends. Variability is determined by intention. That means identifying variability is not a question of the applied method but of the developer's objective. A control structure can be considered as a variability mechanisms when it is not part of the program's application logic. A variability mechanism embodies a variation point connected to a configuration variable. Usually the variation point can be removed (by hard-coding one specific variant) without breaking the application logic.

Take a conditional statement as an example. It could be used in the validation of user input in which case it is absolutely vital for the program's application logic. Thus, it is not a variability mechanism; removing it would break the program. In contrast, a conditional statement realizing a choice between two possible input methods can be seen as a variability implementation. It could be omitted in favor of always using one specific method. Of course this would somehow restrict the program but it will not turn it incorrect.

Binding in dynamic control flow modifications happens during run-time. Actually, for conditional statements, binding occurs immediately before executing the variant because the configuration is re-evaluated every time the control flow reaches the variation point. With function pointers this may be different, in cases where the pointer is not simultaneously regarded as the configuration variable – then binding occurs when the function pointer variable is assigned a new value.

Variants are defined manually during development. They are fixed at compile-time and statically embedded in the binary code. In case of function pointer usage, the specific

variant count is not bounded. Theoretically every function with a matching signature can be used. However these functions are still statically defined.

2.4 Dynamic variability in Synthesis

The *Synthesis* kernel goes one step further in realizing dynamic variability than the previously discussed approaches. Synthesis is an operating system kernel that aims to offer higher performance than traditional kernels by employing a variety of new techniques, including *run-time code synthesis* [23, pp. 1f.]. This is a technique that allows to create executable machine code at run-time to adapt frequently used kernel routines to their precise requirements at a specific system state.

The idea of run-time code synthesis arises from the observation that traditional operating system kernels, when handling kernel calls, spend considerable time in traversing data structures (e.g. linked lists), before performing the actual functionality [24, p. 8]. This is because the system state is maintained in these data structures. The kernel has to first reach the starting state that is needed for the current operation, before executing it. Later the previous state has to be restored. The idea behind code synthesis is to generate new code at run-time that captures the system state at a specific point in time. This is called data-dependent optimization [23, p. 24].

Of course, this synthesized code is only valid for a specific time until a relevant system state change happens. Therefore it has to be recreated frequently at run-time. This is done by the *kernel code generator* [23, pp. 23ff.]. It does essentially what a compiler does but during run-time. In contrast to the compiler which runs as an offline process, run-time code generation is a more performance critical task. Therefore, it has to be more careful with performing expensive code optimization operations. The code generator in Synthesis mainly uses three types of efficiently applicable optimizations: constant folding, constant propagation and procedure inlining [23, p. 25]. This is sufficient to reach significantly better performance in data-dependent code than by using the respective generic implementations.

Run-time code synthesis is used in various places throughout the Synthesis kernel. [23, pp. 30ff.]. This includes buffers and queues, context switches, interrupt handling and system calls.

2.5 Summary

Dynamic variability is a form of variability that is realized during the run-time or load-time of a program. Like static variability, it is based on variation points associated with a set of implementations (variants) realizing the configuration possibilities. Implementing dynamic variability is usually complex and often comes with some sort of run-time penalty. However, many programs require variability during run-time. Three common approaches of dynamic variability management were introduced in this chapter:

- **Run-time polymorphism by means of dynamic dispatch** in object-oriented programming (OOP)
- **Dynamic linking**
- **Dynamic control flow modification** via conditional statements and function pointers

Another rather exotic method is employed by the Synthesis operating system. It uses a run-time code generation approach that is very powerful but adds a lot of complexity to the software.

3 DYNAMIC VARIABILITY IN THE LINUX KERNEL

As many other pieces of system software, the Linux kernel has various variability requirements which result from the need to support different architectures, peripheral hardware, and different features. Most of the variability requirements are solved statically. This means that choices are specified before compilation, and then realized by the C preprocessor or conditional compilation. In general, this is a good solution, as it provides full flexibility during compile-time with zero run-time overhead. For some features, however, it is required to be not compile-time determined. Most commonly, these are features that are dependent on volatile properties of the target system, such as the specific hardware configuration. In these cases, using static solutions would result in the undesirable situation that the binary is very tightly tailored to the specific setting. In contrast to static variability, there is no generic solution for dynamic variability requirements in the Linux kernel. Instead, there exist various application-specific mechanisms.

3.1 Coarse-granular variability via loadable kernel modules

Loadable kernel modules provide a generic dynamic variability mechanism in the Linux kernel. They allow to add and remove functionality in the kernel at run-time [6, p. 11]. Loadable kernel modules usually consist of multiple functions, global variables and type definitions. This makes them operating on a coarse-granular level, compared to the other variability methods in the Linux kernel that work with much smaller code units (on the level of single functions or instructions).

The use of loadable modules helps to keep the kernel small by making higher-level components not part of the main binary. Modules are separate ELF object files [6, p. 844], allowing the user to load them as required. They run in the privileged kernel mode. Many device drivers, file systems and network layers can be compiled as kernel modules [6, p. 842]. This lets Linux support a variety of different high-level features without increasing the kernel binary size or having to build highly specialized versions. Loading kernel modules is done either explicitly from user-space via a command line program or, in some cases, automatically when the functionality is requested [6, pp. 850f.].

A loadable kernel module can interact with the static kernel code and other modules. Many global symbols in the kernel can be referenced from module code. The kernel uses special *kernel symbol tables* to store their addresses in order to be able to resolve these references when loading a module [6, p. 846]. This means that the kernel replaces all the references to kernel symbols with the corresponding symbols' real addresses at the load-time of a module. C preprocessor (CPP) macros (`EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL`) are used to make a specific symbol in the static kernel code accessible. Modules can also export symbols themselves by using the same macros. When a module is loaded, these symbols are then available to other modules, just like the exported kernel symbols [6, p. 846].

Module-to-module interaction via referencing symbols results in dependency relationships between modules. For example, a module *A* using a function from module *B* requires *B* to be already in memory before *A* can be loaded. To keep track of this, every module knows its dependencies and the kernel maintains a list of depending modules for each loaded module [6, p. 847]. Loading a module only succeeds when all the dependencies are met. Unloading can only happen if there are currently no dependent modules loaded.

Loadable kernel modules differ from other variability mechanisms in that the variation points are less tangible. In general, variability is realized on a higher level than in other approaches. Consider module-based file system implementations as an example; different file system drivers are implemented by kernel modules which can be considered as the variants in this case. The common interface code, using the different file system implementations to make them available as a file hierarchy, provides the variation points.

Variant generation, which is the compilation of the modules, happens usually in the kernel build process. However, modules can also be built separately. A module can, for example, be programmed, compiled and then immediately loaded into the currently running kernel. So, variant generation may happen during run-time but it is usually done with the kernel build process. The binding of variants, which is the loading of modules, happens at run-time.

There are similarities between loadable kernel modules and dynamic link libraries (see Section 2.3.2). They are both separate files and get loaded into memory at some point. Loading kernel modules is particularly similar to the dynamic linking approach that is known as dynamic loading. Both mechanisms allow arbitrary units of code to be loaded at any time during execution.

3.2 Application-specific solutions

Loadable kernel modules are very generic in that they are not tailored to a single use case. The mechanism is not specific to a special kind of problem. But kernel modules cannot be used for every kind of dynamic variability requirement due to the generic and coarse-granular design. In the following section there are introduced application-specific variability approaches. They have built-in knowledge about the variability requirement that they are used for.

3.2.1 Run-time patching of instructions based on processor capabilities

The *alternatives* patching mechanism is a low-level variability mechanism in the Linux kernel. It allows to switch between two or more machine instruction sequences at run-time, depending on the availability of certain CPU features. Currently, this mechanism is implemented for the x86 [20] and the arm64 [28] processor architecture. The *alternatives* mechanism allows the use of new processor capabilities in generic kernel binaries without breaking the support for older CPUs of the same processor family.

The mechanism is based on C preprocessor (CPP) macros that can be used to denote a variation point in the architecture-specific code. The variants of this variation point are specified as assembler instruction sequences. Variant selection is based on the availability of specific features in the computer's main (bootstrap) CPU. Listing 3.1 shows the anatomy of such an *alternatives* application site.

```
1 alternative(<original instructions>,
2             <alternative instructions>,
3             <CPU feature>)
```

Listing 3.1 – Anatomy of an *alternatives* application site. The `alternative` macro takes three arguments: the original instruction sequence that will be placed into the executable statically, the alternative instruction sequence and the processor feature that is used as configuration variable to determine if the original instructions should be replaced by the alternative sequence.

The instruction sequence, that is specified by the developer to be the original one, is placed into the application site during compilation to be the default variant. The alternative sequence gets written to a special ELF section of the kernel binary. Furthermore, an entry in another section is created which holds metadata about this *alternatives* application site. This includes its position in the binary, a pointer to the alternative instruction sequence and the CPU feature the variation depends on. Figure 3.1 gives an overview of the section layout.

The actual patching is executed during kernel startup, before other CPUs are initialized, to avoid problems with additional processors executing code that is currently being patched [20]. The mechanism has to iterate over the generated metadata section to perform the patching. The section has all the necessary information to determine if the original instructions should be replaced or not.

The *alternatives* approach was initially introduced to be able to replace legacy memory barrier instructions with better ones that were newly introduced with the Pentium 4 processor [20]. Listing 3.2 shows the definition of the C preprocessor (CPP) macro that is used to place a memory barrier in the kernel.

```
1 #define mb() alternative("lock; addl $0,0(%%esp)", "mfence", ↘
      X86_FEATURE_XMM2)
```

Listing 3.2 – Memory barrier macro definition using *alternatives*. The original instruction sequence is the legacy memory barrier operation. The `mfence` instruction is faster and can be used on modern processors. `X86_FEATURE_XMM2` is the feature flag that is used to identify the availability of the `mfence` instruction.

In recent kernel versions, there can be used instruction sequences of different sizes for the original and the alternative variants [5]. In case the original sequence is shorter than the alternative, the missing space is padded with NOPs by the `alternative` macro. This makes the replacement with the alternative sequence possible without overwriting the following instructions. If the original sequence is longer than the alternative then the run-time patching code will add padding NOPs when the replacement happens.

An example for differently sized instruction sequences is the kernel's usage of Intel's *Supervisor Mode Access Prevention (SMAP)* technology [3]. This security feature allows to prevent the kernel from accessing user pages when a specific processor flag is set [17, p. 4/4].

Listing 3.3 shows the definition of the *STAC* operation which is used to set the SMAP-enable processor flag. Due to the fact that SMAP is only supported on recent Intel processors, the operation is realized as an alternative sequence. The original instruction sequence is empty because there is no corresponding operation on processors that do not support the SMAP technology. Thus, in this case the size of the original sequence is zero. The *alternative* macro will make sure that the space is padded with appropriate NOP instructions to allow the STAC operation to be patched in correctly.

```
1 static __always_inline void stac(void)
2 {
3     alternative("", __stringify(__ASM_STAC), X86_FEATURE_SMAP);
4 }
```

Listing 3.3 – STAC operation implementation using alternatives. The STAC instruction is only available when the processor supports the SMAP technology (`X86_FEATURE_SMAP`). There is no equivalent on other processor models. Thus, the original instruction sequence is empty.

There are also versions of the *alternatives* macro that can use more than one alternative instruction sequence [4]. In this case the developer has to provide one CPU feature flag for each alternative because the availability of a processor feature is a binary choice. Patching the instructions is done in sequence [4]. This causes the last patch to overwrite all previous ones. Thus, the last specified feature flag takes precedence over the others.

The *alternatives* patching is a straightforward dynamic variability mechanism. The variation points and the variants are clearly defined. Its narrow use case, only with

processor features, and the variant specification on assembler level are restrictions of the mechanism. On the other hand, it gives the developer detailed control over the used instructions. This matches the main use cases of the mechanism: Increase performance through the use of the best available instructions and work around bugs in specific processor models [28].

Variant generation in the *alternatives* variability management method happens at compile-time, when the *alternative* macros are expanded and the specified instruction sequences get embedded into the code. Variant binding occurs at run-time. However, it is only done once, early in the kernel's bootstrap process [20]; the ELF sections holding the variant information are discarded after the kernel startup is finished. In this regard the *alternatives* mechanism is more similar to load-time binding approaches than to other run-time patching mechanisms that allow to change the configuration at any time during execution.

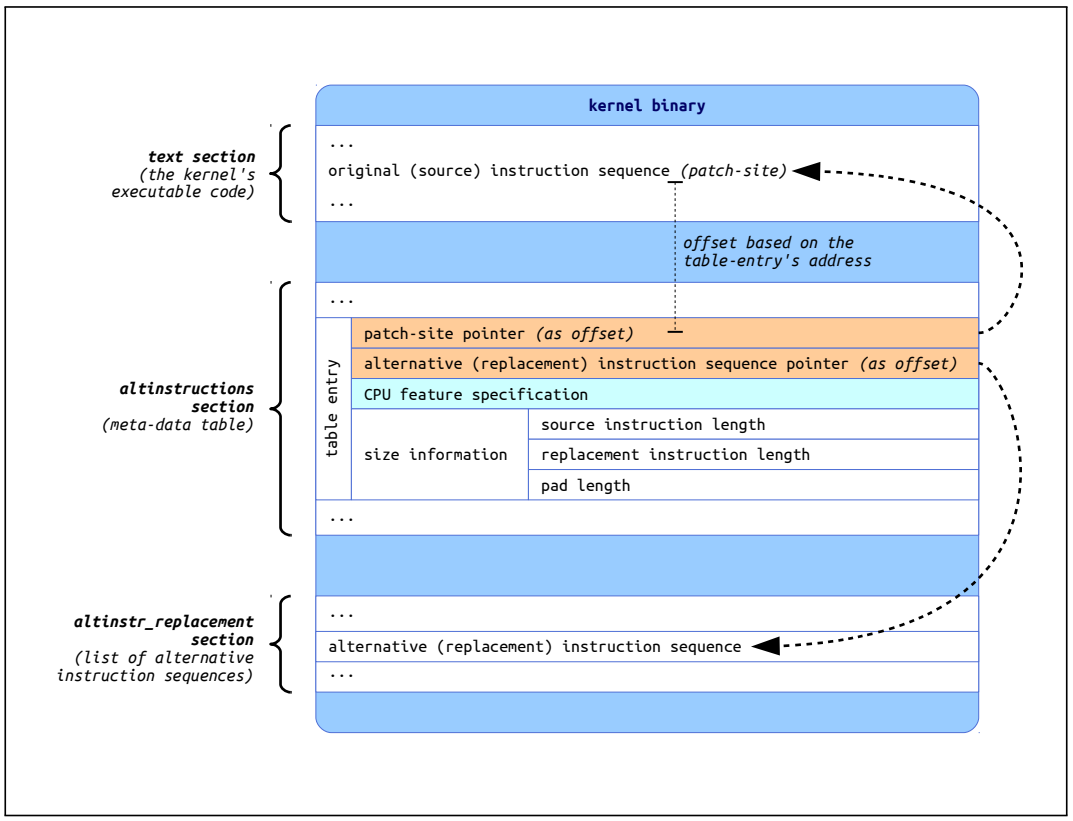

Figure 3.1 – Conceptual overview of sections involved in the *alternatives* patching mechanism. The figure shows a patch-site, its associated metadata entry and alternative instruction sequence.

3.2.2 Run-time modification for uniprocessor systems

Run-time modification for uniprocessor systems is a specialized version of the *alternatives* mechanism. It is also referred to as *SMP alternatives* and allows to switch at run time between customized code versions for uniprocessor (UP) and symmetric multiprocessor (SMP) systems [22]. The mechanism is only available for the x86 architecture.

The originally proposed approach of *SMP alternatives* included a general instruction sequence patching facility, just like the processor feature based *alternatives* method [22]. In current kernels, *SMP alternatives* patching handles only *lock* prefixes. The ability to patch arbitrary instructions based on the running system's UP/SMP property has been removed [19].

The *lock* instruction prefix in x86 CPUs causes an atomic read-modify-write operation when the prefixed instruction is accessing memory [17, p. 2/25]. It is used for reliable interprocessor communication. Thus it is not required in UP systems.

A C preprocessor (CPP) macro is provided that is used whenever the lock prefix is needed in the kernel code in inline assembly statements or separate assembler files. Like in the feature based *alternatives* mechanism, pointers to the lock prefix occurrences are placed into a special ELF section, but in this case without extra metadata. The pointers can then be used to remove the lock prefixes in cases where the kernel is not running on a SMP system. The prefixes are not replaced with NOP instructions but with meaningless *DS segment override prefixes*. This saves an extra CPU cycle for each patch-site.

In contrast to the processor feature based *alternatives* mechanism, patching in *SMP alternatives* is reversible and can be initiated at any time – not only during startup [22]. This is especially useful for CPU hotplugging.

The code shown in Listing 3.4 is taken from the `atomic.h` file in the x86 architecture specific code in the Linux kernel. It provides a set of atomic operations for general use inside the kernel. The CPP macro `LOCK_PREFIX` places the lock prefix into the inline assembly statement and creates the patch-site.

```
1 static __always_inline void atomic_add(int i, atomic_t *v)
2 {
3     asm volatile(LOCK_PREFIX "addl %1,%0"
4                  : "+m" (v->counter)
5                  : "ir" (i));
6 }
```

Listing 3.4 – Atomic add operation with lock prefix using *SMP alternatives*. The macro `LOCK_PREFIX` inserts an lock prefix and creates a patch-site.

As in the CPU feature based approach, variant generation in the *SMP alternatives* variability management method happens at compile-time. Variant binding occurs at run-time. In contrast to the feature based approach the binding can be re-evaluated at any time during execution – not only during the kernel startup.

3.2.3 Operations for paravirtualized kernels (PV-Ops)

Paravirtualization is a very performance-critical field, where dynamic variability mechanisms are applied in the Linux kernel. The term paravirtualization refers to a virtualization technique in which the emulated platform is not exactly reproduced by the virtualization layer. Instead, features that are difficult or expensive to reproduce are replaced by a simpler interface [33, pp. 159f.]. This change of the virtualized architecture requires the software, that runs inside the virtualization, to be aware and to adapt of the modified interface.

The Linux kernel can run as a guest system inside a hypervisor that uses paravirtualization. A general problem with paravirtualized operating systems is that the kernel cannot run in the most privileged processor mode anymore, because the hypervisor takes this place [33, p. 160]. The kernel code is executed in a less privileged mode in this case. Usually (e.g. in the x86 architecture) this implicates that certain operations, such as enabling or disabling interrupts, are not directly available anymore. To execute one of these operations, the kernel has to delegate it to the hypervisor. The mechanism, that is usually employed to achieve this is called *hypercall* [33, p. 161]. It is similar to a system call in user-space programs.

Thus, to make the Linux kernel work as a paravirtualized guest, all the privileged operations have to be replaced with appropriate hypercalls matching the used hypervisor's interface. For the x86 architecture, Linux uses a generalized approach, encapsulating all sensitive operations in a common interface that is implemented by the different hypervisor adaptions [30]. 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 functions are assigned to these function pointers. They either implement the native operation or perform a call to a specific hypervisor [31, 40]. The method can be interpreted as a run-time variability management approach; a function pointer usage defines a variation point, whereas the PV-Ops implementations constitute the variants. The approach allows to use the same kernel binary for native hardware and all paravirtualized environments.

Using function pointers involves indirect calls which are much slower than direct calls on modern processor architectures. Due to performance issues with heavily used PV-Ops like interrupt manipulation operations, the variability approach has been equipped with a binary patching facility to get rid of the indirect calls [41].

The binary patching mechanism replaces the slow indirect function pointer invocations with fast direct calls. Additionally, the binary patching function itself is implemented as a paravirtualized operation. This allows different paravirtualization implementations to modify the patching behaviour according to their needs. The native PV-Ops implementation uses this to replace some indirect calls with inlined function bodies. This is possible in cases where the function body is shorter than the indirect call instruction that

it overrides. It is done for a few manually selected, performance-critical operations, like enabling or disabling interrupts.

Similar to the *alternatives* patching method (see Section 3.2.1), this mechanism uses a special ELF section to store information about the PV-Ops patch-sites. This information is used at run-time to enable the patching. Figure 3.2 gives an overview of the section layout.

Currently, the patching mechanism is executed once on startup, shortly after the paravirtualization detection and the PV-Ops variant selection. Thus, like in the *alternatives* approach, this variability solution is more comparable to load-time mechanisms than to other run-time solutions. The functions that can be assigned to the function pointer variables can be seen as the variants. They are defined statically; therefore, variant generation happens at compile-time.

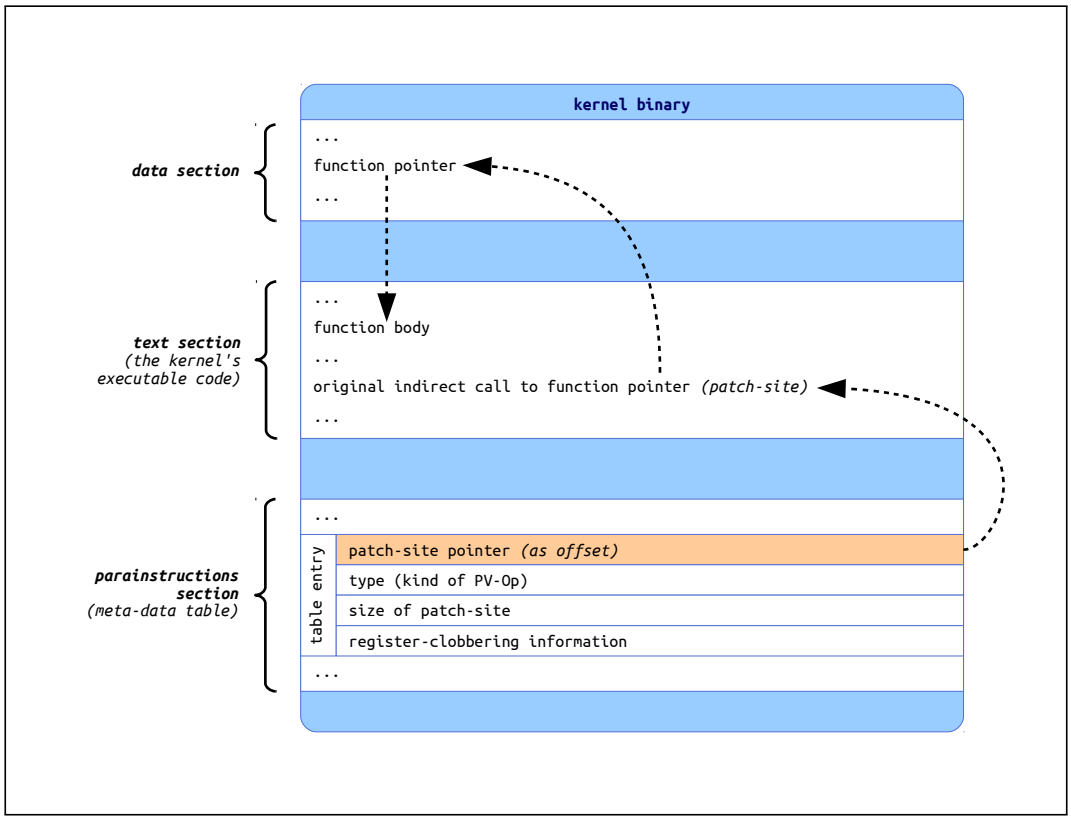

Figure 3.2 – Conceptual overview of sections involved in the *PV-Ops* patching mechanism. The figure shows a patch-site and its associated metadata entry in the kernel. The metadata table is used by the binary patching code to find and classify the patch-sites.

3.3 Summary

The Linux kernel comes with different dynamic variability management solutions. *Loadable kernel modules* can be seen as a coarse-granular mechanism. They allow to load units of code into a running kernel and thus, are comparable to dynamic linking in user-space programs. A more low-level dynamic variability solution is provided by the *alternatives* patching approach based on CPU capabilities. It allows to patch machine instruction sequences to adapt the kernel to the underlying hardware at run-time. Very similar to this is the *SMP alternatives* mechanism which is able to optimize the kernel based on the availability of symmetric multiprocessing (SMP).

The solutions have different properties concerning the time of variant generation and variant binding. Table 3.1 gives an overview of these properties. Note that variant binding occurs at run-time in all approaches. However, it can be restricted to a specific stage during run-time. This is the case in the *alternatives* and the *PV-Ops* patching approaches where variant binding is only done once during kernel startup.

Table 3.1 – Time of variant generation and variant binding of dynamic variability mechanisms in the Linux kernel

Variability mechanism	Time of variant generation	Time of variant binding
Loadable kernel modules	run-time	run-time
Alternatives patching	compile-time	kernel startup
SMP alternatives patching	compile-time	run-time
PV-Ops	compile-time	kernel startup

4 FUNCTION MULTIVERSE

Function Multiverse (or *Multiverse*) is a generic, compiler-based approach to address dynamic variability requirements. The main objective is to provide a tool to realize run-time variability efficiently by means of binary patching, without the need to develop application-specific solutions. Furthermore Multiverse tries to keep the run-time overhead small by doing as much work as possible during compilation. This chapter tries to give an overview of the components and functionality of Multiverse, and explains its objectives and use-cases.

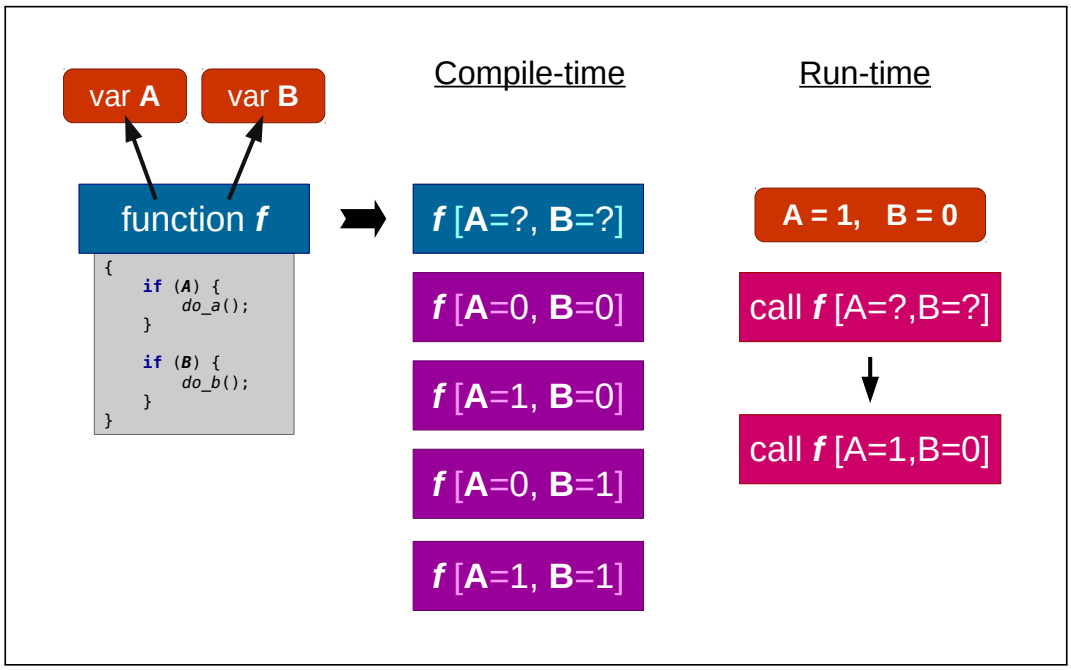

Figure 4.1 – Conceptual overview of Multiverse's functionality. Function *f* references two config variables *A* and *B* (first column). Besides the generic version, the compiler generates additional variants of function *f*, determined to each particular value combination of *A* and *B* (second column). During run-time each callsite of *f* is patched to call the variant corresponding to the current configuration (third column).

4.1 Motivation and Concept

The ability to adapt to different environmental conditions and user needs is a basic requirement for many programs. Responding to such parameters can either happen by implementing static or dynamic variability (see Chapter 2). In many cases, static variability is the preferred form due to its ease of use and the absence of any run-time performance penalty. Sometimes however, it is necessary or simply more convenient to delay variation determination to the run-time of a program.

This usually results in implementing dynamic variability based on control flow modification using standard language facilities (see Section 2.3.3). This way, variability is easy to realize but it can lead to serious performance problems caused by variation points in the control flow depending on rarely changing configuration variables. In contrast to their counterparts in static variability, these variation points increase the amount of executed instructions, cause register clobbering and cache pollution.

Run-time binary patching is a common way to overcome this performance drawback caused by dynamic variability (see Chapter 3). However, implementations are often tightly tailored to fit the specific use case, and are difficult to express in high-level languages, which usually results in non-canonical code.

Multiverse aims to solve this problem by allowing to use the same notation for binary patching enabled variation points as in ordinary control flow modification. Concretely, this allows the use of conditional statements to enable binary patching in a program. In Multiverse, the variability configuration constitutes of developer-selected global variables, called *config variables* [29]. The mechanism works on a function level granularity. During compilation, Multiverse will generate specialized variants for each function referencing one or more config variables (e.g. using them in a conditional statement). The specialized variants can then be used during run-time by a dedicated binary patching facility, which is able to patch all callsites to point to the function variant matching the current configuration (see Figure 4.1).

Thus, Multiverse employs an approach in between static configuration and fully dynamic variability [29]. Possible configuration values are detected during compilation and then used as a basis to compile code specialized to the respective configuration manifestations.

4.2 Components

As a result of its hybrid approach, Multiverse consists of two components, a plugin for the GNU C Compiler (GCC) and a run-time support library. In this section a brief overview of the components and their basic functionality is given. A detailed description of the inner workings follows in the next sections.

4.2.1 GNU C Compiler plugin

The compiler is responsible to generate all the function variants that will be necessary at run-time. For this purpose the GCC's ability to be extended via plugins is used. Such extensions can add new functionality on all levels during the compilation process. Plugins are added to the compiler as dynamic libraries. They can use a subset of the GCC's API, e.g. connecting to different events during the compilation process or even adding new passes [34, pp. 637ff.].

To cause the Multiverse plugin to become active, the developer has to annotate the declaration of every designated config variable with a special attribute. This determines the set of variables considered as configuration base for variability management.

Config variables must be global and of enumeral or integer type. The variables will be used to generate different specialized instances of the functions that are influenced by the configuration. Each generated instance is a variant of the original (generic) function, with the difference that every occurrence of a config variable is set to a specific constant value. The entirety of variants for a function is referred to as the function's multiverse. The plugin also collects and records all callsites of these functions for later use by the run-time library.

Listing 4.1 shows a simple example with a single config variable and a function that uses this variable. Alongside the original generic version, there will be generated two additional variants of this function for each value of `A` (`true` or `false`).

```
1 bool __attribute__((multiverse)) A;
2
3 void __attribute__((multiverse)) foo() {
4     if (A)
5         do_a();
6     else
7         do_b();
8 }
```

Listing 4.1 – Declaration of a config variable and a function annotated with the Multiverse attribute.

```
1 void foo_A_true() {
2   do_a();
3 }
4 void foo_A_false() {
5   do_b();
6 }
```

Listing 4.2 – Specialized variants (equivalent C code) of function `foo` depending on the boolean config variable `A`.

Listing 4.2 shows the equivalent C code for the variants that will be generated. Note that not only the config variable but also the function has to be annotated with the Multiverse attribute in order to be considered for variant generation.

4.2.2 Run-time library

After compiling a program using Multiverse, all the specialized function variants will be part of the generated binary. However, there are still the original generic instances in place. The run-time support library is capable of performing the actual switching between different variants. This is accomplished by binary patching the appropriate callsites previously recorded by the compiler plugin. The library exposes a set of methods to switch between the variants of multiverse functions to the current state of configuration (see Table 4.1).

Presuming the function and config variable from Listing 4.1, the developer would have to call one of the *commit functions* after changing the value of `A` and before calling `foo` the next time (see Listing 4.3).

```
1 A = true;
2 multiverse_commit_refs(&A);
3 ...
4 foo();
```

Listing 4.3 – Update all functions that reference a variable. To avoid undefined behaviour, the commit function must be called after changing the value of `A` and before calling `foo` the next time.

Table 4.1 – Library functions for updating multiverse variants (excerpt)

Function	Description
`multiverse_commit`	Update all function variants to match the current configuration.
`multiverse_commit_fn`	Update only a single function to use the instance that matches the current configuration.
`multiverse_commit_refs`	Update all functions referencing a specific config variable to use the instance that matches the current configuration.

4.3 Compile-time functionality

As previously described, the GCC plugin is chronologically the first part that springs into action when using Multiverse. The plugin itself consists of two main parts, the variant generation and the descriptor construction.

4.3.1 Variant generation

Variant generation consists of three steps added to the compilation process: examination of config variables, variant generation itself and a subsequent pass to eliminate duplicate variants.

After registering the *multiverse* attribute in the compiler, the plugin iterates through all variables annotated in such way. The goal is to determine possible values for every config variable in the compilation unit. This finally results in a *configuration vector* for each one of the variables [29]. For a boolean variable this configuration vector consists of the two possible values, *true* and *false*. For variables of enumeral type it contains the respective *enum variants*. Integer types are more difficult because they have too many values to cover them all. In this case, Multiverse tries to infer relevant values from context, or otherwise, falls back to an configuration vector consisting of 0 and 1.

In the variant generation process the configuration vector is then used to generate specialized instances for the multiversed functions. This is accomplished by finding all references to config variables within each annotated function and then generating variants for combinations of possible assignments based on the configuration vectors. This is achieved by simply replacing each config variable reference with the constant value determined by the respective combination. Unreachable branches will be cut off later in compilation process by the compiler's standard optimization passes [29].

Besides the specialized instances, the unmodified generic body of a multiversed function is still preserved. In case a specialized variant gets installed for this multiversed function, an appropriate jump instruction to this variant gets written to the start of the function body. This is necessary for cases that will not be accessible for the dynamic patching facility, e.g. the use of pointers to multiversed functions.

It may occur that two or more generated variants are equivalent. This can result from simply using only one of many enum variants in a conditional expression or from using logical operators to combine different config variables. Therefore a dedicated pass is executed after variant generation. It detects and merges duplicate variants.

4.3.2 Descriptor construction

After variant generation, the compiler has to gather and store all the information that is needed during run-time. Therefore, the plugin generates descriptors for function multiverses, variants, config variables and callsites and embeds them in the compilation output.

Figure 4.2 shows a simplified overview of the descriptors and illustrates their relationships. Each function that possesses a multiverse gets a descriptor containing the function's name, a pointer to the body (the actual code) and a lists of its associated variants' descriptors. A variant descriptor contains, aside from the body pointer, a list of all involved config variables in the form of special assignment descriptors. Config variable descriptors hold a pointer to the actual variable. To be able to access all places where a multiversed function is used, the compiler has to also collect and record all callsites of these functions. The corresponding descriptors include the address to the respective callsite (the call label) and a pointer to the function it refers to.

The plugin cannot overlook the whole program. Due to its operation in the compiler, it works on the level of compilation units. Therefore, there are separate sets of descriptors embedded in every unit (object file) that contains multiversed functions. It is the run-time library's task to merge the separate descriptors to a global list that is valid for the whole program.

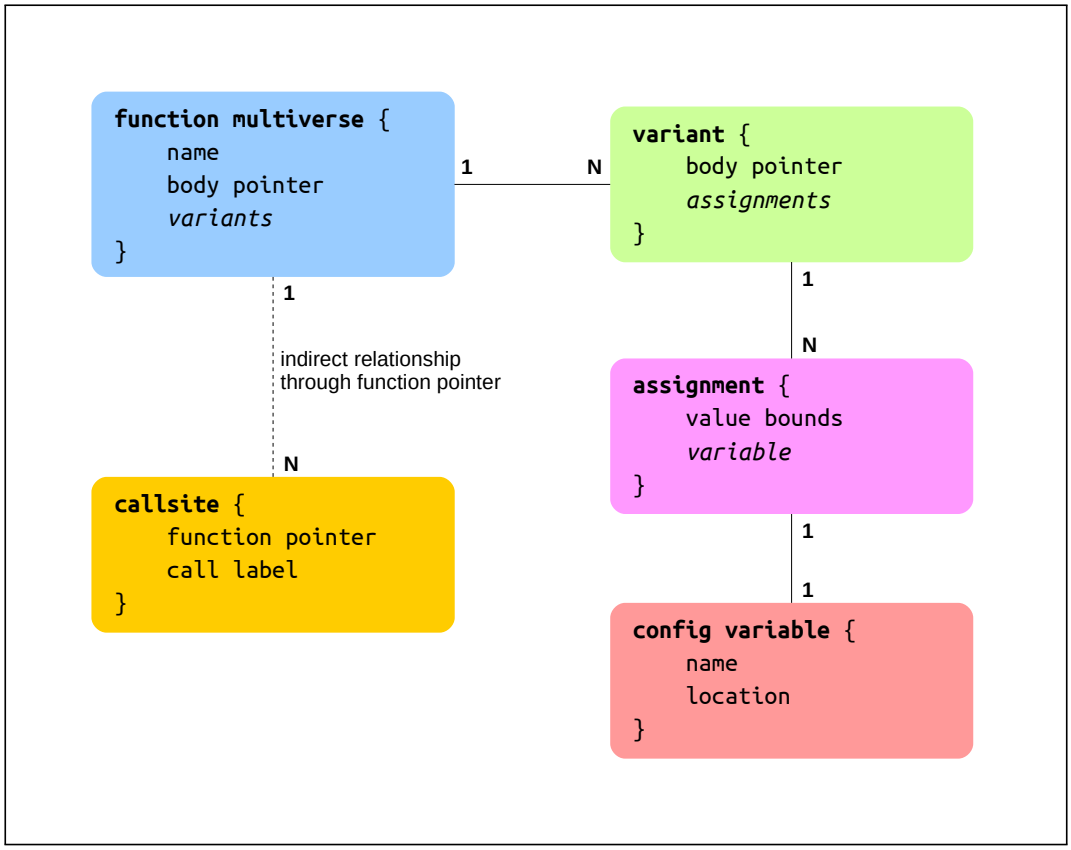

Figure 4.2 – Overview of descriptors and their relationships (simplified)

4.4 Run-time functionality

To actually use the generated function multiverses and switch between different variants, the run-time support library is required. It exposes an interface that allows the programmer to update callsites according to the current state of the config variables.

4.4.1 Initialization of run-time data structures

Multiverses uses *constructor functions*, a special GCC feature. It allows the automated execution of code prior to the program's actual entry point [35, pp. 432f.]. The compiler plugin emits such a constructor in each compilation unit. It will add the unit's descriptors to a global list which lives in the run-time library. This is necessary due to the compiler plugins inability to generate a joint list for all the compilation units. The global list's head pointer is hard-coded in each constructor function and defined in the library.

Before actually using the library, the developer has to manually call the run-time library's initialization function. It will add further information to the descriptors and connect them to allow efficient variant switching.

Most importantly, the initialization code has to connect each config variable assignment to the config variable it refers to. This must be done at run-time because a config variable is not necessarily defined inside the same compilation unit in which it is referenced by a multiversed function. The code will not only connect the assignment descriptors to the variable descriptors but also add a list of all referencing functions to each variable descriptor. The result is that it is now possible to efficiently retrieve referencing functions for each config variable and the other way round. This is required for the use of the commit functions (see Table 4.1).

On some architectures, the library detects special variants for later optimization purposes. The objective is to enable direct inlining of code during the variant switching process, instead of placing function calls. Inlining is possible if the variant body's instructions length is shorter or equal to the length of the call instruction that it aims to replace. This is achieved by analyzing the variants' function bodies. The architecture dependent code will detect some hard-coded body types like empty bodies (NOPs) or simple value returns. It will add an additional data structure to the variant descriptor that identifies the special body type and, in some cases, holds additional information, for instance the value in a simple value return.

4.4.2 Variant switching via binary patching

The switching between different function variants is the core part of the run-time functionality. The developer has to manually trigger the binary patching process by calling one of the commit functions (see Table 4.1). This has to be done every time the configuration is changed.

During the process, the descriptors are used to identify the relevant callsites that need to be updated to match the current state of the config variables. The selected callsites are

then replaced by calls to the new function variant. This involves operations specific to the platform and architecture.

As a first step, Multiverse must be able to write to the memory regions in which the machine code resides. Usually the platform (the underlying operating system in most cases) prevents this type of access due to security considerations. The run-time library will call the platform to repeal this protection before the patching code runs. After the replacement is done, it will request the platform to restore the write protection and to trigger a processor instruction cache cancellation for the affected memory region.

Of course, the actual patching is architecture-dependent. The current call instructions to the generic function or to a function variant will be replaced with call instructions to the newly identified variant. This requires the library to assemble a suitable instruction that works as intended on the processor architecture. In some cases, special function bodies allow inlining code into the callsite itself. This is currently implemented for the Intel x86 processor architecture. It relies on the previously classified special function bodies.

4.5 Supporting different architectures and platforms

Multiverse aims to not depend at all on the used hardware architecture and software platform from the perspective of an user of the tool. Internally, it is designed to be easily extended to support to different architectures and platforms.

The GCC plugin is completely independent from both, architecture and platform. Since it operates on the part of the compiler's backend that is independent fro the target architecture, it works with an abstract, hardware-independent representation of the program code [34, pp. 117ff.].

As previously described, there are dependencies in the run-time library due to the binary patching functionality but it is designed to be easily ported to new architectures and platforms. This is achieved by encapsulating dependent code in distinct modules that implement a generic interface. This keeps the remaining parts of the library independent.

4.6 Summary

Multiverse is designed as a generic solution to efficiently realize dynamic variability independently from architecture and application. The approach is based on boolean, enumeral or integral configuration variables and works on function-level granularity. It employs a hybrid technique with pre-compiled function variants and run-time binary patching.

The tool consists of two components, a GCC plugin and a run-time support library. The first creates multiple specialized versions of functions, based on the possible values of specially annotated configuration variables. The latter is used during run-time to switch

between the generated variants according to match the current state of configuration. The switching is accomplished by patching the respective function callsites.

5 MULTIVERSE IN THE LINUX KERNEL

As stated in Chapter 3, there are various mechanisms built into the Linux kernel that address dynamic variability requirements. This chapter describes the employment of Multiverse in places where it substitutes current variability mechanisms or in completely new areas. Expected advantages over current mechanisms are discussed. Furthermore, specialties of the application in operating system kernels and the resulting extensions to Multiverse are explained.

5.1 Expected benefits

Applying Multiverse is connected with the expectation of benefits which result from Multiverse's objective to provide a generic and performance-oriented dynamic variability management solution.

5.1.1 Maintainability improvements

Using Multiverse as an unified solution to dynamic variability management is expected to improve source code maintainability in the Linux kernel. All mentioned dynamic variability management mechanisms in the kernel use run-time patching. Such live patching techniques are always specific to the processor architecture and usually incorporate complex implementations. In some cases there must be even made assumptions about compiler behaviour or compiler determinations have to be overridden explicitly.

Multiverse addresses this by placing the implementation of dynamic variability completely into the compiler and the support library. Responsibility is taken away from the kernel source code, and thus from the kernel developer. Various application-specific binary patching mechanisms could be removed if current solutions were replaced completely by Multiverse.

5.1.2 Performance improvements

Performance improvements are primarily expected in areas where special variability mechanisms have not yet been applied. This refers to locations which have dynamic variability requirements that are currently handled by ordinary control flow modification, without applying further performance optimization techniques.

The compiler-based approach makes it easy to apply Multiverse in such cases. In contrast to other dynamic variability solutions, Multiverse does not interfere with the semantic structure of the programming language. Conditional statements can just stay as they are. It is sufficient to enhance the code by annotating the targeted function with the Multiverse attribute and calling one of the commit functions after a configuration variable change.

Thus, Multiverse could be much more easily applied in the kernel than the current mechanisms, while adding only little complexity. Assuming that this facilitation actually leads to a broader use of dynamic variability optimization mechanisms, it could increase overall kernel performance.

5.2 Multiverse in kernel space

Running code in an operating system kernel imposes certain difficulties that are not present in user space programs. This is also the case for the Multiverse run-time library.

From the beginning, the library has been designed to be easily portable to new software platforms (see Section 4.5). An extension to this design was made to also allow the application in unhosted environments where at most a freestanding standard library implementation is available [35, p. 6]. However, the platform has to provide basic dynamic memory management functionality, as this is essential to build the run-time data structures. In hosted platforms this is implemented by standard library functions. In Linux, its kernel equivalents must be used.

Current dynamic variability mechanisms in the Linux kernel often kick in on early stages during the bootstrap process [20], before multiple CPUs are enabled and preemption is initiated. Despite this being an easy way to avoid concurrency problems during binary patching, it is a problem for the application of Multiverse. The reason is, that certain functionality is not available during startup. There are no dynamic memory capabilities available at this point. Multiverse requires GCC constructor functions to be called at startup to initialize its run-time data structures. Constructor functions can be used in the kernel, after enabling them via configuration switch [27], but they will not have been called at this early stage. The problems cannot be easily worked around. Therefore, replacing such variability solutions with Multiverse may implicate redesigns of the respective solutions in the kernel.

5.3 Application of Multiverse in the Linux kernel

In principle, most of the dynamic variability solutions presented in Chapter 3 can be replaced by Multiverse. But there are also new places where Multiverse can be applied which are currently not covered by special variability mechanisms. As part of this work, Multiverse was applied in two places in the Linux kernel. The implementation and its challenges are described in this section.

5.3.1 Spin-lock elimination in uniprocessor systems

Spin-locks are used all over the kernel to synchronize access to shared resources in symmetric multiprocessor (SMP) systems. Threads acquire spin-locks through a busy-waiting process until the current holder releases the lock. Spin-locks are necessary in cases of true parallel execution. In uniprocessor (UP) systems there is no need to use them. Due to the fact that Linux is a SMP-capable kernel, the employment of locking mechanisms is mandatory when accessing shared resources. Thus, spin-locks are used by default, even if the kernel is running on an UP system. To improve performance on such systems, Linux can be configured to disable SMP support (via a build system config switch). This static configuration option causes all spin-lock operations to be removed completely. This saves processor cycles but results in a kernel targeted to run exclusively in an UP configuration[3].

Multiverse is able to combine the performance benefits from statically disabling spin-locks with the flexibility of run-time patching. Listing 5.1 illustrates the application of Multiverse in the kernel's spin-lock implementation. It shows the function used to acquire a spin-lock. The boolean config variable (`mv_uniprocessor_locks`), that is referenced there, is set up during kernel startup according to the hardware's SMP/UP configuration. In this case, the compiler plugin generates two variants of the function, each specialized to implement one arm of the branch. The `__LOCK` macro used by the UP variant is no actual spin-lock implementation. It is used to satisfy compile-time requirements, like parameter usage, and it disables preemption in the kernel.

Applying Multiverse in Linux's spin-lock implementation enables SMP kernels to adapt to the machine's processor count. Effectively, it allows a configuration option, previously only determinable during compile-time, to be set up at run-time.

[3] At least in the x86 architecture, a kernel without SMP support will run on a SMP system but it will only be able to use a single CPU.

```
1 void __lockfunc __attribute__((multiverse)) ↘
      _raw_spin_lock(raw_spinlock_t *lock)
2 {
3     if (mv_uniprocessor_locks) {
4         __LOCK(lock);
5     } else {
6         __raw_spin_lock(lock);
7     }
8 }
```

Listing 5.1 – Multiversed *spin-lock acquire* function. The variable `mv_uniprocessor_locks` is a boolean config variable determining if the kernel is running on an UP system or not. The function `__raw_spin_lock` contains the actual lock implementation. `__LOCK` is a macro, satisfying compile-time requirements (instruction ordering, parameter usage) and disabling preemption.

5.3.2 Operations for paravirtualized kernels (PV-Ops)

Operations for paravirtualized kernels (PV-Ops) in the x86 architecture are already handled by a specialized variability mechanism (see Section 3.2.3). The current mechanism uses function pointers to allow changing the functionality of operations when paravirtualization is enabled. Binary patching is used to replace the indirect function pointer calls to these operations with direct calls. The current approach allows to simply activating new implementations for PV-Ops by assigning new values to the respective function pointer variables and re-running the patching process.

This solution differs from the way Multiverse works. It conflicts with the idea that variants are derived from a generic function. Instead, a special variant generation process is not necessary in this case, as the variants are manually provided and installed by the developer (via function pointer assignment). This discrepancy concerning variant generation prevents replacing the current solution with the original version of Multiverse. Apart from this problem, the Multiverse run-time library is very well suited to replace the PV-Ops replacement mechanism, regarding the very similar binary patching methodology. To allow the application in cases like this, Multiverse was extended to support the described function pointer usage.

Implementing the support for specially recognized function pointers in Multiverse requires changes in the compiler plugin and in the run-time library. First of all, with annotating function pointers, there is now a third way how the Multiverse attribute can be used (next to annotating functions and config variables). Therefore, the compiler plugin was modified to generate a *function multiverse descriptor* for every annotated function pointer. No actual variants are placed into the descriptor, as there are none of them determined during compile-time – they originate at run-time by assigning some value (a function body) to the pointer variable. The library uses the multiverse descriptor's emptiness as a marker to differentiate between function pointers and functions. On

committing a new configuration state, the library creates a temporary variant descriptor, pointing to the function body that is currently assigned to the function pointer variable. This descriptor is then used to determine the active variant for the binary patching process. A call to a commit function will completely remove the old descriptor and replace it with a newly generated one. There are also differences in the callsites. Instead of direct calls, callsites associated with function pointers involve indirect calls. On the x86 architecture this results in differently sized instructions, which has to be taken into account by the patching code.

Listing 5.2 illustrates how multiversed function pointers work from a users perspective. Note that the function pointers are not recognized as config variables by Multiverse (though they conceptually are). Instead, they are associated with a multiverse descriptor at run-time. They can be seen as function multiverses without predefined variants.

```
1  int foo1(void) {
2      return 23;
3  }
4
5  int foo2(void) {
6      return 42;
7  }
8
9  __attribute__((multiverse)) int (*fp)(void);
10
11 int main(void) {
12     multiverse_init();
13
14     fp = foo1;
15     multiverse_commit_fn(&fp);
16     assert(fp() == 23);  // foo1 is called
17
18     fp = foo2;
19     multiverse_commit_fn(&fp);
20     assert(fp() == 42);  // foo2 is called
21 }
```

Listing 5.2 – Using a multiversed function pointer. Different values are assigned to the annotated function pointer `fp`. A call to the commit function patches the callsite to perform a direct call to the function hold by the pointer.

The described extension is a step forward towards the usage of Multiverse for PV-Ops. However, there is another issue that causes incompatibilities. In current kernels, PV-Ops are grouped into structures according to their area of operation. Listing 5.3 shows the definition of two of the groups in the x86 architecture code as an example. Reasons for the grouping might be the improvement of code structure or the minimization of global namespace pollution. Multiverse, however, does only work with global variables and functions. Allowing *C struct members* to be Multiverse attributed would be problematic

since struct variables can also be declared locally. Therefore, the PV-Ops design must be changed in order to apply Multiverse. This means suspending the groups and declaring the function pointers as global variables.

```
struct pv_irq_ops pv_irq_ops = {
    .save_fl = __PV_IS_CALLEE_SAVE(native_save_fl),
    .restore_fl = __PV_IS_CALLEE_SAVE(native_restore_fl),
    .irq_disable = __PV_IS_CALLEE_SAVE(native_irq_disable),
    .irq_enable = __PV_IS_CALLEE_SAVE(native_irq_enable),
    .safe_halt = native_safe_halt,
    .halt = native_halt,
    /* Additional members ommited */
};

struct pv_cpu_ops pv_cpu_ops = {
    .cpuid = native_cpuid,
    .get_debugreg = native_get_debugreg,
    .set_debugreg = native_set_debugreg,
    .read_cr0 = native_read_cr0,
    .write_cr0 = native_write_cr0,
    /* Additional members ommited */
}
```

Listing 5.3 – PV-Ops grouped in structures according to the area of operation (x86 architecture). This snippet shows two groups: interrupt related operations (`pv_irq_ops`) and CPU related operations (`pv_cpu_ops`). In total, there are seven groups in Linux, version 4.13.

Two PV-Ops were migrated to use Multiverse: `irq_enable` (enable processor interrupts) and `irq_disable` (disable processor interrupts). Listing 5.4 shows them, declared as global function pointer variables. Suspending the groups also involves changing calls to the function pointers accordingly. Luckily, the PV-Ops pointers are generally not referenced directly, but instead called through wrapper functions. Thus, the impact of the design change is limited.

```
extern __attribute__((multiverse)) void (*pv_irq_enable)(void);
extern __attribute__((multiverse)) void (*pv_irq_disable)(void);
```

Listing 5.4 – PV-Ops for enabling and disabling interrupts as multiversed function pointers

5.4 Summary

The usage of Multiverse for dynamic spin-lock elimination and PV-Ops implementation prove that the tool is applicable in the Linux kernel. However, a modification to Multiverse was necessary to make it feasible: The tool was extended to support function pointer based dynamic variability. Expected benefits of using Multiverse in the Linux kernel are maintainability improvements due to code reduction and performance improvements mainly in new areas where dynamic variability mechanisms are currently not applied.

6 EVALUATION

Section 5.1 lists expected benefits when applying *Multiverse* in the Linux kernel, replacing current dynamic variability management mechanisms. Besides the unification of different solutions, performance improvements and code complexity reduction were suggested. To verify these claims, the *Multiverse* applications in the kernel (see Section 5.3) are evaluated in this chapter.

6.1 Performance comparison via microbenchmarks

Micro-benchmarks were performed to asses the performance of Multiverse in the Linux kernel and compare it to current dynamic variability solutions. This section presents the results, after a short explanation of the benchmarking methodology and the experimental setup.

6.1.1 Method

According to Ehliar & Liu [11] there are two practical approaches to benchmarking: application level benchmarks and microbenchmarks. Application level benchmarking refers to performance measurements of typical applications of the whole program, whereas microbenchmarks are targeted to single aspects of the software artifact.

Microbenchmarks were used to measure the performance changes caused by the modifications that were applied to the Linux kernel. This decision is based on the high specificity of the changes and a general difficulty with application level benchmarking in operating system kernels.

Multiverse was applied in two places in the kernel: dynamic spin-lock elimination and PV-Ops patching (see Section 5.3). Both applications of Multiverse were compared to the respective standard kernel implementations by using microbenchmarks. Linux, version 4.13 was used for all measurements. The benchmarks were run after the startup stage of the kernel, shortly before the invocation of the first user-space process. This point was selected to minimize the interference with other kernel activities. The measurements were not only performed on native hardware but also in a paravirtualized environment,

using the XEN hypervisor[4]. Furthermore, two systems with different processors were used:

- **Intel® Core™ i7-2677M CPU** – a recent multi-core processor (symmetric multi-processor (SMP) system).
- **Intel® Pentium® 4 HT 640** – an older single-core processor. It comes with Intel's hyperthreading technology which provides two virtual CPUs, making it a SMP processor. However, hyperthreading can be turned off in this model. This makes measurements in a real uniprocessor (UP) configuration possible.

Listing 6.1 shows the benchmarking code. The operations to measure have a very short run-time and are thus repeatedly executed (100000 times) to get a measurable result. A monotonic high-resolution clock (`ktime_get`) is used to measure the elapsed time in nanoseconds. The results are written to the standard kernel output buffer (via `printk`).

```
{
    int i;
    ktime_t t1, t2;
    t1 = ktime_get();
    for (i = 0; i < 100000; ++i) {
        /* Measure PV-Ops by disabling and enabling interrupts */
        local_irq_disable();
        local_irq_enable();
    }
    t2 = ktime_get();
    printk("BENCHMARK PV-OPS: elapsed time: %llu\n", t2 - t1);
}
{
    int i;
    ktime_t t1, t2;
    t1 = ktime_get();
    for (i = 0; i < 100000; ++i) {
        /* Measure spin-lock performance by locking and unlocking
           a lock variable */
        spin_lock(&test_lock);
        spin_unlock(&test_lock);
    }
    t2 = ktime_get();
    printk("BENCHMARK SPIN-LOCK: elapsed time: %llu\n", t2 - t1);
}
```

Listing 6.1 – Benchmarking spin-lock and PV-Ops (interrupt operations)

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

6.1.2 Results

On each processor (*Intel Core i7-2677M* and *Intel Pentium 4 HT 640*) the benchmarks, as described above, were run five times for each hardware configuration (SMP/UP). On the Pentium processor, switching to UP mode was done on the hardware[5]. The Core i7 does not allow to disable SMP on a hardware level; therefore, the kernel boot parameter `nosmp` was used to force the kernel into UP mode despite the availability of multiple CPUs.

The described procedure was performed on a kernel that had been extended with Multiverse as described in Section 5.3 (in the following called "multiversed kernel") and for an unmodified kernel (referred to as "standard kernel"). Table 6.1 and Table 6.2 show the averaged results for the respective runs. The pecentage numbers specify the proportional measurement differences between the results produced by the the multiversed kernel in relation to the results produced by the standard kernel. Note that a positive value indicates a performance loss, whereas a negative value signifies a performance gain in the multiversed kernel. A percentage change around zero signifies no considerable differences in performance between the two versions.

Table 6.1 – Averaged benchmark results on a Core i7 processor

	standard kernel		**multiversed kernel**	
	PV-Ops	spin-lock	PV-Ops	spin-lock
SMP	346.476 µs	901.592 µs	277.574 µs	902.088 µs
			-19.9 %	**+0.1 %**
UP	419.764 µs	910.869 µs	277.616 µs	519.430 µs
			-33.9 %	**-43.0 %**

Table 6.2 – Averaged benchmark results on a Pentium 4 processor

	standard kernel		**multiversed kernel**	
	PV-Ops	spin-lock	PV-Ops	spin-lock
SMP	3649.110 µs	3286.647 µs	3641.483 µs	3266.631 µs
			-0.2 %	**-0.6 %**
UP	3650.660 µs	2139.433 µs	3641.763 µs	1940.233 µs
			-0.2 %	**-9.3 %**

Regarding the measurements for spin-lock operations, the results are not surprising. The multiversed kernel comes with a spin-lock elimination mechanism on UP systems (by

[5] by disabling hyperthreading in the BIOS configuration

employing Multiverse), whereas the standard kernel has no such mechanism. Therefore a performance gain is measured in UP configurations (on both processor models) but there are no changes when running in SMP. However, the performance gain is considerably higher on the newer processor (43.0% vs. 9.3%). A reason for this could be that there is at least one other operation than the actual spinning connected to the spin-lock function: disabling preemption (see Section 5.3.1). This operation is also executed in the UP-variant of the spin-lock implementation. It might be that this is slower on older hardware, which would in this case result in narrowing the performance benefit that was gained by eliminating the spinning code.

On the Pentium processor, executing the PV-Ops revealed no performance difference between the current mechanism and the Multiverse solution. This is as expected because the current approach already performs binary patching. On the Core i7 however, the Multiverse solution caused a significant performance increase (19.9% on SMP and 33.9% on UP). This is surprising, as both binary patching approaches have similar capabilities. A reason might be the employment of a non-standard calling convention in the current PV-Ops solution. It expects the caller to preserve more registers than in the standard C convention, resulting in additional instructions before and after the callsites. However, this does not explain the fact that the performance differences are only measurable on the Core i7 processor.

Besides benchmarking the two kernels on bare hardware, it is of interest, especially regarding to the PV-Ops patching, to also measure the performance in a paravirtualized environment. Table 6.3 shows the results of the benchmarks. They were performed using the XEN hypervisor as a paravirtualization host.

Table 6.3 – Averaged benchmark results in paravirtualization

	standard kernel		**multiversed kernel**	
	PV-Ops	spin-lock	PV-Ops	spin-lock
SMP	1046.858 µs	4128.549 µs	1041.850 µs	2090.528 µs
			-0.5 %	**-49.4 %**
UP	493.312 µs	1707.445 µs	1037.521 µs	527.295 µs
			+110.3 %	**-69.1 %**

As in the previous measurements, there are some unexpected results. The multiversed spin-lock implementation is considerably faster compared to the standard kernel implementation. In the case of a SMP system this is surprising. I was not able to determine the the reason for this behaviour but the observed performance increases might be connected to the fact that spin-locks are handled specially in paravirtualization. Ticket lock implementations, like the one used in the kernel, have a negative performance impact when running on a virtual CPU that is scheduled by a hypervisor [13]. Therefore spin-lock

acquiring is partially implemented as a paravirtualized operation. The current spin-lock elimination code could interfere with this.

When looking at PV-Ops, the poor performance of the Multiverse solution is particularly striking. The benchmark run-time is more than doubled compared to the current solution (110.3%). In the SMP configuration, this problem seems to not exist. However, looking at the single measurements instead of the averaged values reveals an erratic behaviour in the performance measurements of the standard kernel running on a SMP configuration (see Table 6.4). The observation can be partially explained by the fact that the benchmarks are running inside a virtualization environment where the CPUs are scheduled by the hypervisor. The hypervisor could in this way, spoil the performance measurements. However, this does not explain why the Multiverse kernel seems not to be affected by this problem.

Table 6.4 – PV-Ops: measured values in paravirtualized kernels (SMP)

kernel	**samples** (µs)				
standard	484.269	1800.810	584.850	562.793	1801.568
multiversed	1057.652	1036.568	1036.574	1036.158	1042.299

6.2 Assessment of code complexity

Current dynamic variability mechanisms in the kernel add a lot of complexity to the kernel source code. In this section, an overview of difficulties with the current solutions is given. Furthermore, it will be analysed how the use of Multiverse could improve this situation. The focus in this section will be on the application-specific variability mechanisms (*alternatives*, *SMP alternatives* and *PV-Ops*; see Section 3.2), as they are most comparable to Multiverse regarding the basic functionality and the use cases.

6.2.1 Current status of code complexity related to dynamic variability mechanisms

When looking at the code of current dynamic variability solutions, one of the first observation is that mechanisms to express a variation point often incorporate complex C preprocessor (CPP) macro invocations and inline assembly statements. This is not surprising, as it is not trivial to express a binary patching site in a high-level language like C. In the PV-Ops approach, for example, the operation callsites are denoted by using a nested preprocessor macro hierarchy, spanning five levels. At the end of the macro invocation chain resides the code shown in Listing 6.2. It uses assembler directives and local labels [12, p.33 & 55] to place the metadata into a special ELF section that is

structured as described in Section 3.2.3. Note that this macro is meant to be used in pure assembly code. There exists a second version for the usage in C code, which implements the same functionality in the form of an inline assembly statement.

```
1  #define _PVSITE(ptype, clobbers, ops, word, algn)   \
2  771:;                                               \
3          ops;                                        \
4  772:;                                               \
5          .pushsection .parainstructions,"a";         \
6           .align algn;                               \
7           word 771b;                                 \
8           .byte ptype;                               \
9           .byte 772b-771b;                           \
10          .short clobbers;                           \
11         .popsection
```

Listing 6.2 – CPP macro used to realize a PV-Ops callsite. The macro places the actual instructions (`ops`) into the code segment. Information concerning register clobbering (`clobbers`) and the type of the operation (`ptype`) are written into the metadata section (`.parainstructions`), together with a pointer to the callsite (`label 771`) and its size (`difference of 772-771`). Numeric local labels are used to realizes pointers. They can be redefined multiple times; note the "b" suffix, stating a backward reference.

Undoubtedly, the macro adds a lot of complexity to the kernel, but this piece of code also reveals another weakness of the PV-Ops patching approach: Every call to a paravirtualized operation must be realized by using the respective macro. Otherwise it will not be recognized by the binary patching code.

Other dynamic variability approaches in the kernel, like *alternatives* and *SMP alternatives* patching, apply very similar techniques and thus, implement virtually the same mechanisms again, with only little changes to match their use case. In addition, the mentioned variability mechanisms (PV-Ops, alternatives, SMP alternatives) are all completely located in the architecture-specific kernel code. There is no shared code or interface between different architecture implementations.

In summary, the current dynamic variability implementations incorporate complex, application-specific code. Defining variation points often does not harmonize well with the language syntax. Therefore, the usage of these mechanisms tends to be difficult and error-prone. Furthermore, the dependency of the current implementations on the use case and the processor architecture results in code duplication.

6.2.2 Improvements by the usage of Multiverse

Multiverse's compiler-based approach keeps the additional code that must be added to apply binary patching relatively small. It is reduced to the addition of the multiverse anno-

tations and calls to the multiverse commit functions after configuration variable changes. This is in contrast to the current solutions in the kernel that come with mechanisms which are complex to apply and maintain.

Besides this, Multiverse offers a solution to binary patching which is generic on different levels. Firstly, it is independent from the use case: It allows variant generation based on standard language control flow modification methods (conditional statements and function pointers). Secondly, its usage is architecture-independent. Of course, internally, the binary patching code is dependent on the target architecture but it is structured in a way, that keeps the majority of the code shared and makes supporting new architectures easy (see Section 4.5).

6.3 Challenges and future work

The application of Multiverse in the Linux kernel is still in an experimental state. To be used productively in the kernel, further work is required.

Executing benchmarks to measure performance showed some improvements compared to current solutions but also revealed some unexpected results and a few open issues, especially concerning the application in kernels running as paravirtualized guest systems (see Section 6.1.2). There are performance issues related to the two measured PV-Ops (enable and disable interrupts). The benchmark results point to a general difficulty of measuring performance in paravirtualization. Solving these issues in a future work will therefore require more extensive benchmarks.

Other challenges relate to specialties of the application of Multiverse in kernel space (see Section 5.2). Current run-time patching solutions become active in very early stages during the kernel startup. This is currently not possible for Multiverse which is relying on GCC constructor functions and dynamic memory allocation. Both features are available in the kernel but initialized relatively late in the startup process. In a future work, alternatives to the usage of these two features could be examined. Solutions, that other patching mechanisms apply, are of particular interest in this regard, e.g. the usage of ELF sections.

Another limitation of Multiverse results from its use of the compiler to generate variants. This makes it currently impossible to use Multiverse in assembler code. Regarding low level patching mechanisms like PV-Ops or instruction patching, this is a serious drawback. It should be investigated if it can be made possible to place Multiverse variation points in assembler code, at least in some limited form.

An important task is the discovery of more potential use cases in the Linux kernel. A complex tool like Multiverse will only be acceptable if it can be profitably employed. This requires the existence of application sites where Multiverse provides real, measurable benefits.

6.4 Summary

Multiverse was not just applied successfully in two places in the Linux kernel, but there are also improvements connected with its application. Performance benefits by applying Multiverse were measurable in many cases. Improvements concerning code complexity and maintainability are only hypothetical at this point because only a small portion of an existing dynamic variability mechanism (PV-Ops patching) was replaced by Multiverse. However, there are indications suggesting a reduction of code complexity and an overall improvement of maintainability.

Despite these benefits, the application of Multiverse in the Linux kernel is currently not ready for productive use. There are unsolved issues concerning the initialization of run-time data structures in the early kernel startup stage and the impossibility to use Multiverse in assembler code. Furthermore there are unresolved performance problems in paravirtualized environments, requiring further research.

7 CONCLUSION

In this thesis, it is shown that Multiverse can be applied in the Linux kernel to efficiently realize dynamic variability in performance critical areas. Furthermore, appropriate use cases are presented and possible benefits and disadvantages compared to other approaches are assessed.

Linux already comes with its own dynamic variability management mechanisms in place. The most generic one is the usage of loadable kernel modules. This allows specially designed compilation units to be loaded to the Linux kernel during run-time. There are other, more application specific approaches in Linux. The *alternatives* patching mechanism is able to replace machine instruction sequences in the code, based on the availability of certain CPU capabilities. A very similar method is the *SMP alternatives* mechanism which allows to install uniprocessor optimizations in the code, in cases where the underlying system has only one CPU. Another mechanism, called operations for paravirtualized kernels (PV-Ops), is concerned with adapting the system when it is running in paravirtualization. All these specialized mechanisms in the Linux kernel use binary patching to meet high run-time performance requirements.

There are certain drawbacks connected to the current dynamic variability mechanisms in the Linux kernel. They are often difficult to apply and usually incorporate complex implementations, making them difficult to maintain and limiting their profitable usage in the kernel. By replacing current solutions with Multiverse, it is tried to eliminate these disadvantages.

Multiverse itself is a new dynamic variability approach. It is designed to combine the flexibility and simplicity of control flow modification with the performance of binary patching. Therefore, Multiverse provides a compiler-assisted approach to realize dynamic variability. Configuration possibilities are detected during compilation and used as a basis to compile specialized code manifestations which are put in place at run-time.

Different modifications were applied to Multiverse to make it usable in the Linux kernel. This includes changes to make it applicable in kernel-space. Furthermore, the ability to recognize function pointers as variability sources was added.

Multiverse was applied in two places in the kernel. Two interrupt-related PV-Ops were migrated to use Multiverse and, as a new use case, spin-lock elimination in uniprocessor systems was realized. Microbenchmarks show an increased performance in many cases

but also reveal that further research is required to make a productive and stable use of Multiverse inside the kernel possible.

The focus in this thesis lies on the research connected to make Multiverse applicable in the Linux kernel. Measuring performance and analyzing code complexity is not done extensively in this work. The benchmarks and code complexity assessments should be seen as a basic evaluation to give an insight into what may be expected from Multiverse in the Linux kernel. This opens up a variety of possibilities for future work in this field.

LIST OF ACRONYMS

CPP	C preprocessor
CPU	Central pocessing unit
ELF	Executable and Linkable Format
GCC	GNU C Compiler (also: GNU Compiler Collection)
NOP	No operation (command or instruction that does nothing)
OOP	Object-oriented programming
PV-Ops	Operations for paravirtualized kernels
SMAP	Supervisor Mode Access Prevention
SMP	Symmetric multiprocessor
UP	Uniprocessor

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[1] Nadeem Abbas, Jesper Andersson, and Welf Löwe. "Autonomic Software Product Lines (ASPL)." In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. ECSA '10. Copenhagen, Denmark: ACM, 2010, pp. 324–331. ISBN: 978-1-4503-0179-4.

[2] Varun Agrawal et al. "Architectural Support for Dynamic Linking." In: *SIGPLAN Not.* 50.4 (Mar. 2015), pp. 691–702. ISSN: 0362-1340.

[3] Peter Anvin. *[PATCH 00/11] x86: Supervisor Mode Access Prevention*. Linux Kernel Mailing List. Sept. 2012. URL: https://lkml.org/lkml/2012/9/21/442 (visited on 10/11/2017).

[4] Luca Barbieri. *[PATCH 01/10] x86: add support for multiple choice alternatives*. Linux Kernel Mailing List. Feb. 2010. URL: https://lkml.org/lkml/2010/2/17/72 (visited on 10/10/2017).

[5] Petkov Borislav. *[PATCH v2 03/15] x86/alternatives: Add instruction padding*. Linux Kernel Mailing List. Feb. 2015. URL: https://lkml.org/lkml/2015/2/24/260 (visited on 10/10/2017).

[6] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Thrid Edition. O'Reilly Media, 2005. ISBN: 9780596005658.

[7] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism." In: *ACM Comput. Surv.* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300.

[8] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI series in software engineering. Addison-Wesley, 2002. ISBN: 9780201704822.

[9] Fernando J. Corbató. *A Letter from Prof. Corbató*. Oct. 2000. URL: http://www.multicians.org/corby-letter.html (visited on 10/04/2017).

[10] Robert C. Daley and Jack B. Dennis. "Virtual Memory, Processes, and Sharing in Multics." In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP '67. New York, NY, USA: ACM, 1967, pp. 12.1–12.8.

[11] Andreas Ehliar and Dake Liu. "Benchmarking network processors." In: (Jan. 2002). URL: https://www.researchgate.net/publication/228979633_Benchmarking_network_processors.

[12] Dean Elsner, Jay Fenlason, et al. *Using as – The GNU Assembler Version 7.2.0*. Free Software Foundation. 51 Franklin Street, Boston, USA, 2002. URL: http://www.zap.org.au/elec2041-cdrom/gnutools/doc/gnu-assembler.pdf.

[13] Jeremy Fitzhardinge. *[PATCH RFC 1/4] x86/paravirt: add hooks for spinlock operations*. Linux Kernel Mailing List. July 2008. URL: https://lkml.org/lkml/2008/7/7/318 (visited on 10/25/2017).

[14] Michael Franz. "Dynamic Linking of Software Components." In: *IEEE Computer* 30 (1997), pp. 74–81.

[15] Matthias Galster et al. "Variability in Software Architecture: Views and Beyond." In: *SIGSOFT Softw. Eng. Notes* 38.1 (Jan. 2013), pp. 46–49. ISSN: 0163-5948.

[16] W. Wilson Ho and Ronald A. Olsson. "An Approach to Genuine Dynamic Linking." In: *Software: Practice and Experience* 21.4 (Apr. 1991), pp. 375–390. ISSN: 0038-0644.

[17] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*. Intel Corporation. Oct. 2017. URL: https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf.

[18] *Introduction to programming on Multics*. Honeywell Information Systems. 200 Smith Street, Waltham, USA, 1981. URL: http://www.bitsavers.org/pdf/honeywell/multics/AG90-03_PgmgIntro_Dec81.pdf.

[19] Andi Kleen. *[PATCH] [22/26] i386: Remove smp_alt_instructions*. Linux Kernel Mailing List. Apr. 2007. URL: https://lkml.org/lkml/2007/4/29/384 (visited on 10/14/2017).

[20] Andi Kleen. *[PATCH] Runtime memory barrier patching*. Linux Kernel Mailing List. Apr. 2003. URL: https://lkml.org/lkml/2003/4/21/168 (visited on 10/10/2017).

[21] Ben Klemens. *21st Century C: C Tips from the New School*. Second Edition. O'Reilly Media, 2014. ISBN: 9781491903896.

[22] Gerd Knorr. *[patch] SMP alternatives for i386*. Linux Kernel Mailing List. Dec. 2005. URL: https://lkml.org/lkml/2005/12/13/204 (visited on 10/14/2017).

[23] 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.

[24] Henry Massalin and Calton Pu. *An Overview of The Synthesis Operating System*. Tech. rep. New York City, NY, USA: Columbia University, 1989. URL: https://academiccommons.columbia.edu/catalog/ac:143176.

[25] Scott Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages*. Tech. rep. CSIRO – Division of Information Technology, 1994.

[26] Martin Müller. "Message Dispatch in Dynamically-Typed Object-Oriented Languages." MA thesis. Albuquerque, NM, USA: University of New Mexico, 1995. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.1782.

[27] Peter Oberparleiter. *[PATCH 1/4] kernel: constructor support*. Linux Kernel Mailing List. Feb. 2009. URL: https://lkml.org/lkml/2009/2/3/170 (visited on 10/07/2017).

[28] Andre Przywara. *[PATCH 0/6] arm64: alternatives runtime patching*. Linux Kernel Mailing List. Nov. 2014. URL: https://lkml.org/lkml/2014/11/14/432 (visited on 10/10/2017).

[29] Valentin Rothberg et al. "Function Multiverses for Dynamic Variability." In: *9th International Workshop on Dynamic Software Product Lines - Variability at Runtime*. Augsburg, Germany, Sept. 2016. URL: https://www4.cs.fau.de/Publications/2016/rothberg_16_dspl.pdf.

[30] 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.

[31] Rusty Russell. *[PATCH 3/4] x86 paravirt_ops: implementation of paravirt_ops*. Linux Kernel Mailing List. Aug. 2006. URL: https://lkml.org/lkml/2006/8/7/6 (visited on 10/20/2017).

[32] Alcemir Rodrigues Santos and Eduardo Santana de Almeida. "Do #Ifdef-based Variation Points Realize Feature Model Constraints?" In: *SIGSOFT Softw. Eng. Notes* 40.6 (Nov. 2015), pp. 1–5. ISSN: 0163-5948.

[33] Diomidis Spinellis and Georgios Gousios. *Beautiful Architecture*. First Edition. O'Reilly Media, 2009. ISBN: 9780596517984.

[34] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals – for GCC version 7.2.0*. Free Software Foundation. 51 Franklin Street, Boston, USA, 2017b. URL: https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gccint.pdf.

[35] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection – for GCC version 7.2.0*. Free Software Foundation. 51 Franklin Street, Boston, USA, 2017a. URL: https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc.pdf.

[36] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. "A Taxonomy of Variability Realization Techniques." In: *Software: Practice and Experience* 35 (2001), pp. 705–754.

[37] Reinhard Tartler et al. "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue." In: *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX 2014)*. Ed. by USENIX Association. Philadelphia, PA, USA, 2014, pp. 421–432. ISBN: 978-1-931971-10-2. URL: http://www4.cs.fau.de/Publications/2014/tartler_14_usenix.pdf.

[38] *The Open Group Base Specifications Issue 7 – dlopen*. The IEEE and The Open Group, 2008. URL: http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlopen.html.

[39] Jan Vitek and R. Nigel Horspool. "Compact dispatch tables for dynamically typed object oriented languages." In: *Compiler Construction: 6th International Conference, CC'96 Linköping, Sweden, April 24–26, 1996 Proceedings*. Ed. by Tibor Gyimóthy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 309–325. ISBN: 9783540499398.

[40] Chris Wright. *[PATCH 0/7] x86 paravirtualization infrastructure*. Linux Kernel Mailing List. Oct. 2006. URL: https://lkml.org/lkml/2006/10/28/191 (visited on 10/21/2017).

[41] Chris Wright. *[PATCH 2/7] Patch inline replacements for common paravirt operations.* Linux Kernel Mailing List. Oct. 2006. URL: https://lkml.org/lkml/2006/10/28/196 (visited on 10/21/2017).

APPENDIX

The following tables list the detailed results of the performed microbenchmarks. The results are presented and interpreted in Section 6.1.2.

A.1) Intel Core i7-2677M – Benchmark results for PV-Ops

	kernel	**samples** (nanoseconds)				
SMP	**standard**	321026	347042	370229	347041	347041
	multiversed	276525	277899	277691	277968	277789
UP	**standard**	446216	402355	443143	403825	403281
	multiversed	276555	277829	277898	277828	277969

A.2) Intel Core i7-2677M – Benchmark results for spin-lock

	kernel	**samples** (nanoseconds)				
SMP	**standard**	898772	900114	908635	900184	900254
	multiversed	900114	898727	900318	906959	904321
UP	**standard**	921486	927143	885168	909813	910737
	multiversed	517976	519829	519829	519759	519759

B.1) Intel Pentium 4 HT 640 – Benchmark results for PV-Ops

	kernel	**samples** (nanoseconds)				
SMP	**standard**	3648579	3646414	3648439	3648929	3653188
	multiversed	3642083	3642084	3640966	3641176	3641105
UP	**standard**	3651023	3650046	3650325	3650535	3651372
	multiversed	3641315	3641665	3641805	3642433	3641595

B.2) Intel Pentium 4 HT 640 – Benchmark results for spin-lock

	kernel	**samples** (nanoseconds)				
SMP	**standard**	3212420	3300210	3304261	3313829	3302515
	multiversed	3296020	3228274	3295601	3216960	3296299
UP	**standard**	2148248	2148039	2146426	2108090	2146362
	multiversed	1937817	1937816	1951225	1943823	1930483

C.1) Paravirtualization – Benchmark results for PV-Ops

	kernel	**samples** (nanoseconds)				
SMP	**standard**	484269	1800810	584850	562793	1801568
	multiversed	1057652	1036568	1036574	1036158	1042299
UP	**standard**	507132	507566	483935	483993	483936
	multiversed	1035863	1034046	1033188	1043253	1041253

C.2) Paravirtualization – Benchmark results for spin-lock

	kernel	**samples** (nanoseconds)				
SMP	**standard**	2133068	6289113	2313216	2318218	7589131
	multiversed	2085702	2082613	2115923	2080263	2088139
UP	**standard**	1693594	1712998	1697075	1714852	1718708
	multiversed	518377	518367	542788	518384	538559