Function Multiverses for Dynamic Variability

Valentin Rothberg, Christian Dietrich, Daniel Lohmann
{rothberg, dietrich, lohmann}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany
Alexander Graf
agraf@suse.de
SUSE Linux GmbH

***Abstract*—Run-time variability is a necessary mean to adapt to a configuration or state of a system, which can only be determined during execution. However, implementing such dynamic variability oftentimes results in a conglomerate of a highly branched control flow, with negative impacts on performance. In this paper, we present an approach to dynamically adapt a running system to a specific configuration by means of binary patching. Instead of adding yet another architecture-dependent binary patching technique, we implement the functionality directly in the compiler. With specially annotated config variables, the compiler can generate multiple versions of a function and dynamically binary patch the running system to use the version of the current configuration. Our approach is work-in-progress with developers at SUSE.**

1. Introduction

Static configurability of software product lines (SPLs) such as the Linux kernel is commonly used to support a wide range of hardware architectures and platforms, as well as software features such as scheduling [1, 2]. Although static configurability has proved to work well to implement variability and portability for software projects of various domains [3], its limitations are becoming a serious threat to the maintainability and evolution of software product lines. Considering the ARM hardware architecture, developers face a heterogeneous fruit salad, where devices can show fundamental differences in their instruction sets or only minor differences in their hardware configurations (e.g., addresses of I/O ports). Supporting *n* of such devices by means of static configurability implies building and shipping *n* different, highly redundant software products – a nightmare for vendors, who target a short time-to-market and low costs.

In recent years, Linux developers have filled the gap between dynamic device configurations and static configuration with the concept of device trees (DTs) [4]. DTs allow to describe generic and concrete hardware devices in tree-like data structures, which can be translated into run-time–loadable and easy to parse binary files. Instead of statically compiling the hardware description into distinct kernel images, the kernel loads a specified binary file at boot time, which contains all necessary information to initialize and further boot the system on the current hardware device. Currently, there are more than 800 supported ARM hardware configurations (i.e., device trees). Device trees allow a *dynamic* configuration of the system and have considerably improved the way the Linux kernel implements variability for the ARM architecture. As shown in Figure 1, since 2011 (i.e., the introduction of DTs) the amount of C preprocessor #ifdef blocks is decreasing for ARM related source code by nearly 40 percent while increasing by 25 percent in the rest of the kernel. Notice, that other hardware architectures, such as x86, do not make use of DTs since the variability in terms of platforms and sub-architectures is limited (e.g., x86 32-bit and x86 64-bit).

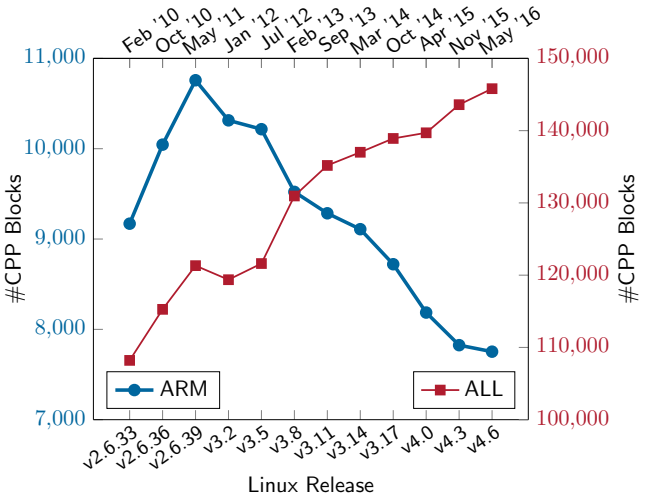

Figure 1. The evolution of CPP blocks in the ARM architecture compared to the evolution of all architectures from Linux v2.6.33 (Feb' 2010) to v4.6 (May 2016).

Problem Statement

Although device trees enable a dynamic configuration of the system at boot time, they are restricted to the description of hardware and, hence, do not help in adapting to other, even more dynamic use cases. Such dynamic use cases range from user-configurable switches (e.g., to choose a scheduling

strategy) to variation points in the control flow of the networking stack, which, depending on the current network load, alter the processing of network packets. Especially variations in the control flow have considerable performance impacts with respect to cache behavior, branch prediction and the amount of executed instructions. As a consequence, developers are reluctant to replace static variability with dynamic variability [3].

In this paper, we present a work-in-progress approach to dynamically adapt to specific run-time configurations by extending the C compiler to generate multiple versions of a function, each tailored towards specific configurations of the control flow. By means of binary patching, we can exchange the currently used instance of a function and hence implement dynamic reconfiguration during the system's execution.

The remainder of the paper is structured as follows. In Section 2, we present our *multiverse* approach and explain how multiple versions of a function can be tailored towards specific configurations, and how the system can be patched at run-time with as little impact as possible. In Section 3, we discuss the use-cases of our approach and explain the advantages and limitations, for instance compared to just-in-time compilation. We further discuss how we seek to improve our approach in future work.

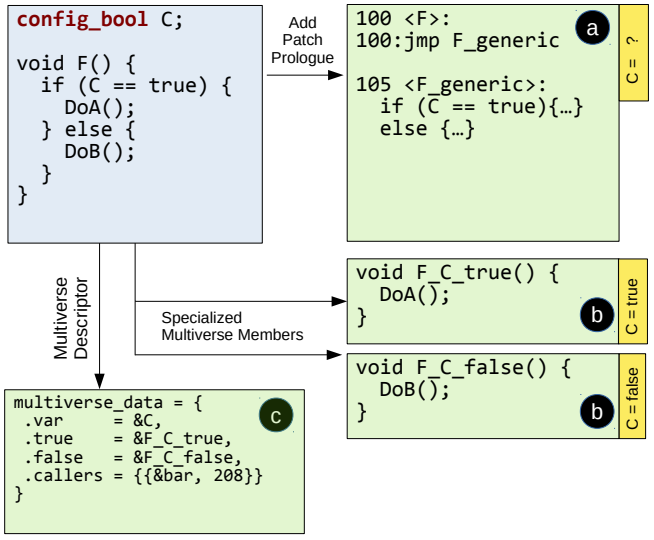

Figure 2. Overview of the compiler multiverse approach. Variables that are marked by the developer (`config_bool` instruct the compiler to a) prepend a patchable prologue to the original function body b) generate members of the function multiverse c) include patch information into the binary for the run-time library.

2. Approach

In a nutshell, function multiverses are a hybrid technique between static configuration (e.g., using the C preprocessor and #ifdef blocks) and fully dynamic configuration switches at run-time. The compiler will emit several specialized instances of a function that is influenced by the configuration, while the run-time system will dynamically patch the binary to reflect the specialized implementation. In our approach, we will extend the C compiler and the run-time system with function-multiverse support. We will discuss how the compiler detects functions suitable for multiverse generation, explain which static code fragments have to be emitted, and how the run-time system can switch between members of the multiverse.

The foundation of the multiverse concept are global, boolean *config variables*, which are annotated by the developer. Together, the values of all config variables constitute the dynamic *configuration vector* of the program. In the code, config variables can be used to distinguish between different critical code paths. Since they will introduce nearly no run-time overhead, they can even be used in tight inner loops. In Figure 2, `C` is a config variable that controls whether the function `F` calls `DoA()` or `DoB()`. Without multiverse support, this decision would be done at run-time every time `F` is activated, even if `C` is rarely or never modified.

The granularity of our multiverse concept will be at the function level. For each function, the compiler collects all references to config variables; if at least one configuration dependency is detected, a function multiverse is emitted into the binary. A function multiverse contains three artifacts that are produced during the compilation process: a generic multiverse-member function that dynamically evaluates the configuration vector, several multiverse-member functions specialized for exactly one assignment of the referenced config variables, and a multiverse descriptor used by the run-time system to switch between specializations.

The generic function is suitable for all dynamic configuration assignments. Its body is the original unmodified function code, which references the configuration at run-time, extended by a *patch prologue*. This patch prologue is large enough to hold a single absolute-jump instruction and is modified by the run-time system to point to other multiverse members. The generic function is only a conservative fall back for all places in the program that are inaccessible to dynamic patching by the run-time system (e.g., function pointers buried in user-defined variables). In Figure 2, the generic function label `<F>` (a, address 100) points to the patch space, which initially jumps to the generic implementation.

For each function multiverse, the compiler extracts a list of referenced config variables and selects a set of assignments for which specialized members are emitted. In a basic multiverse implementation, the compiler will generate a specialized member for each possible combination of values. For each assignment, the compiler copies the body of the original function and replaces all references to a config variable by its currently assigned, constant value. To make use of this additional static information within the specialized function bodies, the compiler does another round of function optimization (e.g., inlining, constant propagation, dead code elimination).

In Figure 2, two specialized functions are generated (b); one for each assignment of the dynamic configuration vector $\langle C \rangle = \{\langle false \rangle, \langle true \rangle\}$. After the read access to `C` is replaced by $true$, respectively $false$, the guarding if

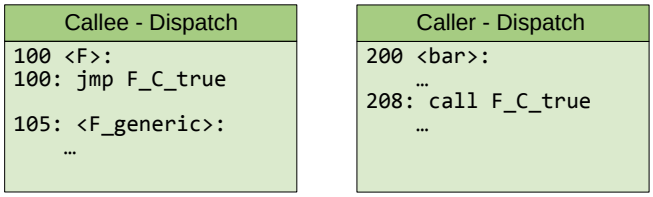

Figure 3. Result of a muliverse switch to `C := true`. The patching proglogue in `F` is redirected to `F_C_true`; all known caller sites of `F` are binary patched to call the specific multiverse member directly.

condition becomes a tautology and the dead code elimination removes the unreachable branch within the specialized body.

As a third component, the compiler emits a descriptor for each multiverse. The run-time library uses the descriptors to determine what code has to be patched at run-time when the dynamic configuration changes. Besides a pointer to the config variable and pointers to the specialized functions, the descriptor contains a list of all known call sites that activate the original function. Since all call sites to a given function are known at link-time, the list of call sites for a given function could, for instance, be placed in a dedicated section by the linker. In Figure 2, a simplified multiverse descriptor for the function F is given (ⓒ).

Function multiverses require support from the run-time system to switch between their different specialized members. The run-time system, which is included in the compiler's support library, exposes functions to modify the current dynamic configuration. When any configuration variable changes, all multiverses that reference the value are updated to reflect the changed configuration. First, the run-time system selects among the specialized functions a compatible one; if no suitable member is found, we fall back to using the generic implementation. The resulting function pointer (e.g., `&F_C_true`) is then used to modify the machine code of the patch prologue and all known call sites.

In Figure 3, the situation when `C` is set to $true$ is shown. The call site in the `bar()` function is replaced by a direct call to `F_C_true`; it will not introduce any run-time overhead for the specialized activation. Besides these zero-cost modifications, the patch prologue of the generic function redirects all function pointers and undetected call sites with an absolute jump to the specialized implementation. Here, a single absolute, unconditional jump is introduced into the code.

3. Discussion

As already mentioned, our approach is a hybrid between a fully static approach of configurability, where no decision is made at run-time, and a fully dynamic approach, where all decisions are made at run-time. Nevertheless, other techniques, like just-in-time (JIT) compilers (e.g., HotSpot [5]) can be found on this spectrum. Compared to JITs, function multiverses are less flexible. Nevertheless, since the run-time library is less complex than a full JIT compiler, function multiverses have the advantage to reach application scenarios, like deeply embedded devices, where JITs are unsuitable.

We see different benefits stemming from function multiverse support in publicly available compilers. The most obvious one is the run-time benefit from removing dynamic decisions that read config variables. The specialized function body runs as efficiently as if the config variable would have just been inserted as a constant. Even more, dead branches are removed, the register pressure decreases, in short the compiler can generate more efficient code. At run-time, the data cache is not polluted with configuration values that evict a useful cache line, even if read only once. Furthermore, since we patch directly at the call site, no overhead is introduced to activate the specialized implementation.

Although the extensive specializing of functions will clearly increase the final size of the binary, multiverses can, in some scenarios decrease the memory consumption at run-time. Since the code is partitioned along the config variables into the specialized functions, the linker can co-locate code for the same dynamic configuration. This code partitioning along configurations allows the unmapping of entire memory regions of dead functionality, if the dynamic configuration settles and is known to remain static (e.g., after some program initialization).

But function multiverses would not only impact the properties of the actual binary and the running program, but also the overall code quality. When function multiverses became a widely deployed feature in off-the-shelf compilers, many performance-enabling work-arounds could be eliminated; configurability can be expressed directly where it is intended to act, even in an inner loop.

Furthermore, the proposed method is a controlled, language-semantic preserving mean to express dynamic binary patching. The Linux kernel is a good example that this need is actually present, since it contains several live patching mechanisms: there are at least three proposals for live updating (ksplice, kGraft, kpatch); paravirtual operations are patched at run-time depending on the presence of a hypervisor; `ftrace` patches arbitrary call sites for tracing, `static_cpu_has` removes CPU feature checks with dynamic patching; spinlocks are removed from a multicore-enabled kernel if run on a unicore machine. In our opinion, the compiler is the exactly right spot to place such functionality and to replace many ad-hoc solutions; not only in Linux, but also in many userspace programs.

Currently, we aim only for boolean-typed config variables, since they provide a base layer of variability; features can be enabled and disabled. In an extension to the presented approach, we also want to include specialized implementations for enumeration-typed, and user-annotated–discrete integer-typed config variables. One problem, we already face with booleans, but which will become even more pressing with other data types, is the exponential explosion of variants, if several config variables are used within one function. We plan to tackle this problem by restricting the multiverse size and by selecting the most promising configuration vectors (e.g., config variables in the inner loop are prioritized) for specialization.

Another issue we identified for the run-time library is the concurrency model we aim for. On the one hand, we believe that only the application knows when it entered a safe state where the current dynamic configuration can be applied. On the other hand, it should be possible to use *compare-and-swap* operations to concurrently patch the running program in a thread (and interrupt-safe) manner.

4. Related Work

Dynamic reconfiguration of operating systems has been a long-standing [6, 7, 8] and still very active [9, 10, 11, 12, 13] field of research, even though the motivating "trends" have changed over time: Most recent work aims at a (more or less) disruption-free dynamic patching/updating of the operating system, especially with respect to security fixes. The related work addresses research kernels, such as K42 [9, 14] or PROTEOS [13], but also aims to facilitate dynamic patching in commodity Linux installations, such as DynaMOS [10] and KSplice [11]. Another approach that has been suggested for this purpose is dynamic aspect weaving [15, 16, 17].

Earlier work, in contrast, was mostly motivated by application-/scenario-specific extension and specialisation of the kernel at run-time, which is more related to the multiverse goals: Synthesis [8], for instance, employs a built-in kernel-code synthesiser (basically an early idea of a JIT compiler) to generate optimized thread-specific variants of kernel features; Synthetix extends this further by means for optimistic and incremental specialisation, also featuring a dynamic function replacer called "replugger" [18]. Choices [19] features dynamic reconfiguration for the concretion and conversion of kernel objects at run-time via its object structure; earlier systems, such as DAS [7], supported this by segment-based dynamic modification. SPIN supports application-specific extensions (written in a type-safe languages) to be loaded into the kernel address space [20].

Compiler-based approaches are, however, mostly centered around optimizing kernel performance by staged, partial specialization. A good overview is provided in [21], which employs Tempo [22], a partial evaluator for C programs, and a set of specialization predicates to identify functions automatically for specialization within the kernel.

However, with the exception of the KSplice system [11] (used mostly for applying security fixes), none of the above approaches has actually established as a major technique to implement dynamic variability in system software – probably, because they mix mechanisms and strategies and are considered by kernel developers as too implicit, heavy-weight, architecture-dependent, and performance-critical to be carried into the kernel. In this realm, we propose multiverses as a light-weigt, but general mechanism to solve the most pressing issues of the domain.

5. Conclusion

In this paper, we presented an approach to dynamically reconfigure a running system to adapt to a defined set of configurations. The *multiverse* approach extends the compiler to generate multiple versions of a function, each tailored and optimized towards a specific configuration. Configurations are exposed by developer-annotated boolean-typed config variables in the source code. The run-time system materializes the current configuration, by dynamically binary patching function-call sites. The presented approach is a generic solution to express a common pattern of dynamic variability within the semantics of the C language. Implementing function multiverse in the compiler provides a widely applicable and portable mechanism for binary patching, and can be used in the context of dynamic reconfiguration beyond the Linux kernel.

Acknowledgements. This work has been supported by the German Research Council (DFG) under grant no. LO1719/3-1 and the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89, Project C1).

References

[1] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. 2007.

[2] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Static analysis of variability in system software: The 90,000# ifdefs issue”. In: *Proc. USENIX Conf* (2014), pp. 421–432.

[3] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. “The love/hate relationship with the C preprocessor: An interview study”. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 37. 2015.

[4] Grant Likely and Josh Boyer. “A symphony of flavours: Using the device tree to describe embedded hardware”. In: *Proceedings of the Linux Symposium*. 2008, pp. 27–37.

[5] Michael Paleczny, Christopher Vick, and Cliff Click. “The java hotspot TM server compiler”. In: *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association. 2001.

[6] Elliot I. Organick. “Intersegment Linking”. In: MIT Press, 1972. Chap. 2, pp. 52–97.

[7] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. “Dynamic Restructuring in an Experimental Operating System”. In: *IEEE TOSE* SE-4.4 (1978), pp. 298–307.

[8] Henry Massalin and Calton Pu. “Threads and Input/Output in the Synthesis Kernel”. In: *12th ACM Symp. on OS Principles (SOSP ’89)*. New York, NY, USA: ACM, 1989, pp. 191–201.

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

[10] Kristis Makris and Kyung Dong Ryu. “Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels”. In: *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2007 (EuroSys ’07)*. Ed. by Thomas Gross and Paulo Ferreira. New York, NY, USA: ACM, Mar. 2007, pp. 327–340.

[11] Jeff Arnold and M. Frans Kaashoek. “Ksplice: automatic rebootless kernel updates”. In: *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2009 (EuroSys ’09)*. Ed. by John Wilkes, Rebecca Isaacs, and Wolfgang Schröder-Preikschat. New York, NY, USA: ACM, Mar. 2009, pp. 187–198.

[12] Chaeseok Im, Minkyu Jeong, Jaedon Lee, and Seungwon Lee. “A dynamically reconfigurable operating system for manycore systems”. In: *28th ACM Symp. on Applied Computing (SAC ’13)*. ACM, 2013, pp. 1622–1627.

[13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. “Safe and automatic live update for operating systems”. In: *18th Int. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS ’13)*. New York, NY, USA: ACM, 2013, pp. 279–292.

[14] Andrew Baumann, Jonathan Appavo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. “Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly”. In: *2007 USENIX ATC*. Berkeley, CA, USA: USENIX, 2007, pp. 337–350.

[15] Yoshisato Yanagisawa, Kenichi Kourai, Shigeru Chiba, and Rei Ishikawa. “A Dynamic Aspect-oriented System for OS Kernels”. In: *6th Int. Conf. on Generative Programming and Component Engineering (GPCE ’06)*. New York, NY, USA: ACM, Oct. 2006, pp. 69–78.

[16] Michael Engel and Bernd Freisleben. “TOSKANA: A Toolkit for Operating System Kernel Aspects”. In: *Transactions on AOSD II*. Ed. by Awais Rashid and Mehmet Aksit. LNCS 4242. Springer, 2006, pp. 182–226.

[17] Wolfgang Schröder-Preikschat, Daniel Lohmann, Wasif Gilani, Fabian Scheler, and Olaf Spinczyk. “Static and Dynamic Weaving in System Software with AspectC++”. In: *39th Hawaii International Conference on System Sciences (HICSS ’06) - Track 9*. Ed. by Yvonne Coady, Jeff Gray, and Raymond Klefstad. IEEE, 2006.

[18] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”. In: *15th ACM Symp. on OS Principles (SOSP ’95)*. New York, NY, USA: ACM, Dec. 1995, pp. 314–324.

[19] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. “Designing and Implementing Choices: An Object-Oriented System in C++”. In: *CACM* 36.9 (Sept. 1993), pp. 117–126.

[20] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. “Extensibility safety and performance in the SPIN operating system”. In: *15th ACM Symp. on OS Principles (SOSP ’95)*. New York, NY, USA: ACM, Dec. 1995, pp. 267–283.

[21] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. “Specialization Tools and Techniques for Systematic Optimization of System Software”. In: *ACM Trans. Comp. Syst.* 19.2 (May 2001), pp. 217–251. URL: http://doi.acm.org/10.1145/377769.377778.

[22] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.– N. Volanschi. “Tempo: Specializing Systems Applications and Beyond”. In: *ACM Comp. Surv.* 30.3es (1998).