MELF: Multivariant Executables for a Heterogeneous World

Dominik Töllner
Leibniz Universität Hannover

Christian Dietrich
Hamburg University of Technology

Illia Ostapyshyn
Leibniz Universität Hannover

Florian Rommel
Leibniz Universität Hannover

Daniel Lohmann
Leibniz Universität Hannover

Abstract

Compilers today provide a plethora of options to optimize and instrument the code for specific processor extensions, safety features and compatibility settings. Application programmers often provide further instrumented variants of their code for similar purposes, controlled again at compile-time by means of preprocessor macros and dead-code elimination. However, the global once-for-all character of compile-time decisions regarding performance-, debugging-, and safety/security-critical features limits their usefulness in heterogeneous execution settings, where available processor features or security requirements may evolve over time or even differ on a per-client level.

Our *Multivariant ELF (MELF)* approach makes it possible to provide multiple per-function compile-time variants within the same binary and flexibly switch between them at run-time, optionally on a per-thread granularity. As MELFs are implemented on binary level (linker, loader), they do not depend on specific language features or compilers and can be easily applied to existing projects. In our case studies with SQLite, memcached, MariaDB and a benchmark for heterogeneous architectures with overlapping ISAs, we show how MELFs can be employed to provide per-client performance isolation of expensive compile-time security or debugging features and adapt to extended instruction sets, when they are actually available.

1 Introduction

Modern compilers provide a plethora of options to statically optimize and instrument the code. Natural examples for such at-compile-time tailoring include support for hardware-specific processor extensions, but also compiler-specific debugging, program instrumentation, and sanitizing aid. These options commonly do not alter the semantics of the code, but influence its *nonfunctional* properties with respect to performance, safety, security, and compatibility. They are put under the control of the developer, because they reflect important tradeoffs: Exploiting special instruction-set extensions can greatly improve performance [1], but at the cost of losing compatibility to smaller or older processors. Letting the compiler instrument the code with extra sanity checks increases safety and security [2]–[6], but comes at a significant performance cost. This also holds for many higher-level instrumentations that are inserted manually by the developers, often by means of the preprocessor: Typical examples are executable asserts [7], [8], tracing, and logging support, which have been shown to actively increase safety [9] and security [10], but are commonly disabled at compile time in production builds due to their performance overhead.

However, in a world of increasing dynamic hardware and use-case heterogeneity, the global once-for-all character of compile-time decisions regarding performance-, debugging-, and safety/security-critical features limits their usefulness: In heterogeneous cloud environments or on machines with heterogeneous ISAs, the availability of processor features may change over time, even for individual threads. The performance costs of the extra sanity checks might be acceptable while processing input from external users, but not in general. In a DevOps setting, it would be useful to temporarily enable tracing and logging, but only for that specific client who is having troubles.

In short: It would be useful to decide at run time, depending on the dynamic context, on features that are technically bound at the compile time of the code. We call this flexibility *semi-dynamic variability*, which conceptually lays between static and dynamic variability in that the code of all variants is still generated at the compile-time of the project (thus, facilitating whole-program optimization), but the actually used variants can be decided on at run time.

While there are special-purpose solutions for semi-dynamic variability in some domains (e.g, math libraries [11] or the Linux kernel [12]) that adapt at load or initialization time to the actually available hardware features, these solutions are limited in scope, require manual intervention, and typically involve costly extra indirections via proxies [13] or inherently fragile means of fine-grained run-time code patching [12]; often they have to prevent inlining.

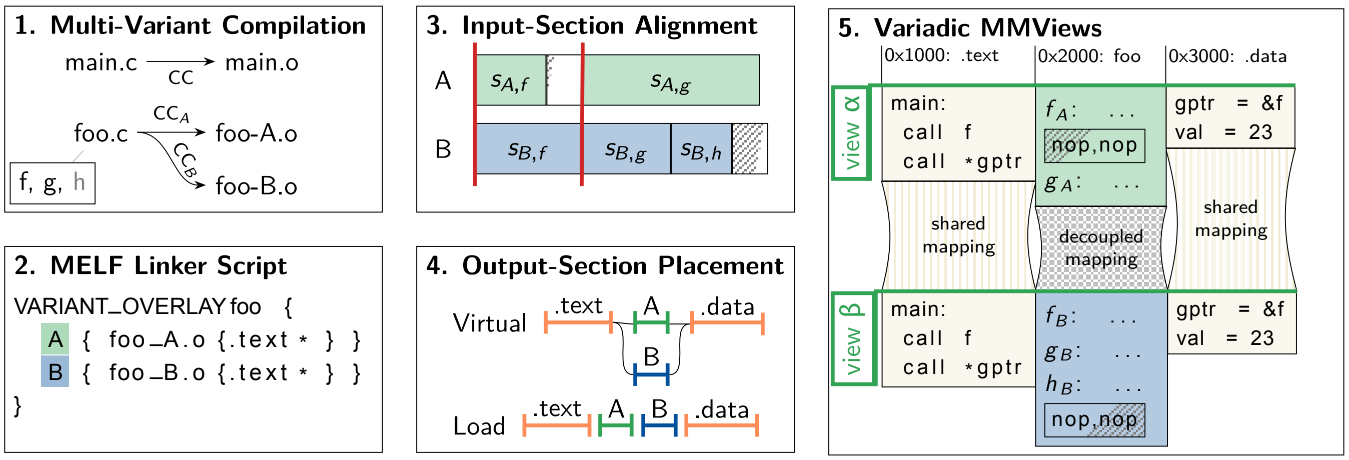

Figure 1: Overview about the MELF approach. At compile time (1), parts of the program are compiled into multiple variants (A and B), which are captured and organized in the linker script in the variant overlay `foo` (2). In the linker, the matched input sections are aligned (3) and the resulting output sections (4) are placed in the virtual- and load-address space. At run time (5), the variants are loaded into different MMViews, which share everything but the decoupled regions; common symbols and pointers are stable across views.

About this Paper

We present *multivariant ELF (MELF)* as an easy-to-apply approach for the inclusion of multiple compile-time variants within the same binary and flexible switching between them at run time on function/section granularity. MELFs are implemented solely on binary level, hence mostly independent of the employed languages and compilers (as long as they produce ELF-compatible objects), which also makes them easy to apply to existing software projects. Function variants are aligned by the MELF linker to the same virtual address, so that existing pointers or relocations remain valid even in case of a variant switch at run time. They can optionally be loaded by the MELF loader into additional in-process address spaces (with the MMViews kernel extension, taken from [14]), where multiple variants can coexist at the same time to be applied on a per-thread level.

In particular, we claim the following contributions:

- We provide the MELF concept, as an end-to-end solution for semi-dynamic variability.
- We describe our MELF linker (an extension to LLVM's LLD linker) and the MELF loader.
- We demonstrate the MELF benefits and costs in four case studies from different domains.

2 The MELF Approach

The MELF approach (see Fig. 1) provides semi-dynamic variability on function granularity for compiled functions by aligning functions (and data) with a modified LLD at link time. At run time, either one of variants is loaded into the process's address space or, with the help of the MMViews, multiple variants can coexist simultaneously. We describe our approach for *executable and linking file format (ELF)* and Linux processes, but are confident that our approach is generalizable to other binary formats (e.g., COFF, Mach-O) and process models.

2.1 System Model

We target programs written in compiled languages (e.g., C, C++, Rust, ...) that organize their binary code in regular, hierarchically-called functions (e.g., unlike Haskell, Forth). For a subset of all functions (i.e., not `main()`), it is intended to include multiple function variants in the final binary. For these functions, the signature (including the mangled symbol name) must be equal and their side effects on the program's object space must be compatible. The compiler must be able to put functions into their individual binary section.

2.2 Compile-Time: Static Variant Generation

First (Fig. 1, step 1), we have to statically generate multiple variants of our functions at compile time. These static variants can, for example, originate from translating the same translation unit multiple times with different compiler flags or CPP configurations. But also, manual variant encoding (e.g., directly in assembler) or programmatically in the compiler via guided-function specialization [15] is possible.

In Lst. 1, we show that this variant generation is simple with modern build systems, like CMake [16]. In the example, which is taken from our SQLite3 case study (Sec. 3.1), we compile the SQLite source files twice into two static libraries.[1] Both libraries are compiled with different predefined CPP macros and linked into the `main` executable.

```
# Collect SQLite 3 Source files
file(GLOB SRCS sqlite3/*.c)

# The Non-Debug Version
add_library(sql-ndebug STATIC ${SRC})
target_compile_options(sql-ndebug -DNDEBUG=1)

# The Debug Version
add_library(sql-debug STATIC ${SRC})
target_compile_options(sql-debug -DSQLITE_DEBUG=1)

# Case-Study Executable: main
add_executable(main main.cc melf_loader.c)
target_link_libraries(main sql-ndebug sql-debug ..)
```

Listing 1: Multi-Variant compilation with CMake

Besides the multi-variant compilation, the ELF [17] standard forces us to put each function and each data object into their own section. In order to understand this requirement, we now take a quick detour into the ELF standard, which is also necessary to understand the MELF linker.

Excursus: ELF Sections vs. Functions The *executable and linking file format (ELF)* is a format that is used for linking *and* for loading programs. An ELF contains multiple byte streams (code, data, debug info, ...) that are arranged in two views: In the link view, those byte streams are called *sections*, while they are called *program headers* (or segments) in the load view. Additionally, the link view makes use of *symbols* as named offsets into a section. Further, *relocations* specify how to edit a byte stream while linking it to a virtual address. In a nutshell, the linker arranges the link-time sections into load-time segments while resolving (most) relocations.

The basic unit of linking is the section and not a (language-level) function or (global) data object. In fact, the linker has no idea about those, and it cannot (due to compile-time resolved relocations) break up a section back into smaller pieces. Therefore, as we want to align the function's start address, we have to instruct the compiler[2] to put each function (and data-object) into its own section, named like the (mangled) function name. For example, the C++ function `void foo(int)` will end up in the section `.text._Z3fooi`.

For data objects (i.e., global variables and read-only data), we require that all variants share the same (data-) object space. For this, the linker has to throw away all but one instance, which requires each variable to be located in its own section[3]. Furthermore, we restrict the program's interpretation of the shared data objects: As objects can be accessed from different variants, we *require* that the interpretation of objects, statically *and* heap allocated, must be compatible across all variants. This means that matching `struct` fields have to be aligned, that language-level types have to be of equal size, and that variants must have a common understanding of the object state. Nevertheless, this restriction holds for many automatically-applied program transformations as they are nonfunctional by design. In Sec. 4, we will discuss this topic further.

After the multi-variant compilation, we end up with a set of ELF object files whose sections s can be categorized as follows: A *variant* $v = \{s_{v,1} \ldots s_{v,n}\}$ is a collection of sections that should be visible together; all sections of one variant have to have the same *section type* (e.g., code, read-only data, ...), which becomes the *variant type*. A *variant overlay* (overlay, o) is a collection of $|o|$ equally-typed *variants* and a *variant-overlay group* (group) is a set of overlays that are semantically connected. For a program, multiple independent overlays and groups can exist. For example, besides a math-code overlay (non-AVX vs. AVX2), there can another overlay group (code and read-only data) for the SQLite library that allows to en/disable executable assertions. We call all sections that are not covered by a variant the *remaining* sections.

2.3 Link-Time: Virtual-Address Alignment

After the compile-time preparation, the linker generates the multi-variant ELF (Fig. 1, steps 2-4), for which it must match and align sections from the different variants within an overlay such that they end up with the same virtual address. For our implementation, we modified LLD [18], the linker of the LLVM project that is a drop-in replacement for the GNU `ld` and `gold`. We will describe the required linker modifications on base of this implementation. However, they should be easily generalizable to other linkers as well.

First (Fig. 1, step 2), the developer must be able to express the relationship between sections, variants, and overlays. For this, we add a `VARIANT_OVERLAY` statement to the linker-script language, which is the command language of `ld` (and `gold`/`lld`). The statement contains multiple variant statements, which define, similar to other commands, patterns that are matched against the *input sections*, which the linker extracts from the object files. The lexically first variant of an overlay is its *primary* variant. In the overview example, we see a linker-script fragment that defines an overlay `foo` with two variants (`A`, `B`), which collect the code (text) sections from the respective `foo_{A,B}.o`.

Thereafter (Fig. 1, step 3), we align the input sections within an overlay: For this, we first match sections from the different variants and identify those sections that occur only in one variant: Starting with all sections captured by the overlay, we group the sections by the key (variant, section name) and select a single section as representative for that key. While there is usually only one candidate section, ELF

[1]Static libraries are fundamentally different from dynamic libraries. They are only collections of object files, (nearly) transparent for the linking process, and induce no run-time overhead.

[2]GCC/Clang: `-ffunction-sections`, MSVC: `/Gy`

[3]GCC/Clang: `-fdata-sections`, MSVC: `/Gw`

$v\backslash n$	f	g	h	X	J
A	$s_{A,f}$	$s_{A,g}$		$s_{A,X}$	$s_{A,J}$
B	$s_{B,f}$	$s_{B,g}$	$s_{B,h}$	$s_{B,X}$	$s_{B,J}$
C	$s_{C,f}$	$s_{C,g}$			$s_{C,J}$

Figure 2: Input-Section Table (extended running example)

section groups[4] and weakly-defined functions[5] can result in multiple candidates. In the former case, we can choose any section as the group representative, in the latter case we apply the usual override semantic for weakly-defined functions, but only within the group. With the representatives, we end up with one section $s_{v,n}$ per (variant, name)–pair and form the overlay's *input-section table*, which tabularizes the results. In the table (Fig. 2) for the (extended) running example, we see that f and g are present in all variants and h is private to variant B. For partially-filled columns (e.g., X) that contain more than one entry, we report an error.

With the table in place, we validate and manipulate the symbol table. With multivariant compilation, the linker will encounter the same symbol, which is a named pointer into a section, multiple times. Instead of reporting an error, we collect duplicate symbols and delete all but the symbol that points to the primary variant. In this process, we verify that each variant's symbol points to the same column and has the same offset.

While we usually report an error if this check fails, some compiler optimizations (e.g., function-body deduplication) can result in two aliased symbols pointing to the same section in one variant but not in the other. However, as we cannot align this section to two different sections in the other variant, we have to solve this rare situation differently: We equip each variant with a jump table (e.g., $s_{A,J}$ in Fig. 2) and insert a `jmp` instruction for one of the aliased symbols. In each variant's jump table, the instruction jumps to the correct section and offset, while we globally redirect the original symbol to the jump-table entry in the primary variant.

With all symbols being aligned within their column, we align the columns by padding each $s_{v,n}$ to $\max_{v_i \in o} s_{v_i,n}$. As this can induce large padding gaps, we use variant-local sections as *gap-filler sections*. Currently, we perform the filling greedy as we did not encounter a situation where a (more) optimal algorithm would be required. In the example (Fig. 1, step 3), the gray areas are padding and $s_{B,h}$ is used for filling the gap in B that $s_{A,g}$ provokes.

After column alignment, we place the variants in the ELF's virtual address space (Fig. 1, step 4). For this, we utilize the fact that the load address (where the loader will copy the section to) and the virtual address (where the section "thinks" it is) can disagree. We combine all sections for a variant (table row) in an *output section*; an established linker-internal concept that acts as an intermediate step between input sections and segments. Each output section is linked (i.e., relocated) to the virtual address of the primary variant, while we load them, by default, sequentially. Thereby, load and virtual address only match for the primary variant. All remaining sections can be linked as usual.

The result of MELF is a regular ELF binary, which is only special with regard for the non-primary output sections, whose load and virtual addresses do not match.

2.4 Run-Time: Multivariant Loading

With the MELF binary constructed, it is time to bring our multi-variant program to execution. As this depends on the indented usage scenario, this section will only provide the necessary primitives from which different use cases can be constructed (see Sec. 3).

First, we have to decide which variant(s) will execute and initialize the program state. As primary variants are loaded to their virtual address, the program automatically starts executing in those variants, and it also loads their data-segment contents. This also requires us to only run the constructors for global variables of the primary variant, which is done by discarding the initialization-array entries of the other variants.

For the usage of MELF's, we provide two operation modes by the MELF run-time loader library: With the *base mode*, only one variant per overlay is active at the same time, which the developer can replace with an explicit call into the run-time library. For this, we use the `mprotect()` system call to make the respective overlay region writable and copy the contents of the desired variant to the primary virtual address. To make overlay and variant regions known to the MELF loader, the linker places symbols with virtual and the load addresses before and after each output section; with these, the program can reference all variants in the program. Usually, the developer will only replace text and other read-only sections, as switching data variants would reinitialize the global variables. Also, for this mode to work, we demand that no thread currently executes or has a call frame for a function from the replaced overlay. This program state is called *global quiescence* [14].

As the base mode is of limited use for multithreaded programs, we also provide the *MMView mode*, based on MMViews [14]. Thereby, multiple variants can be active simultaneously and threads can switch *their* variants for which they only have to be *locally quiescent* (i.e., they do not execute a replaced function). As this mode requires the MMView kernel extension, we give a brief overview of its semantic.

Excursus: MMViews With MMViews, a process can have *multiple*, closely-synchronized, concurrently-active address spaces, which have the same structure: all mappings are equally placed and address-space modifications work si-

[4] e.g., used for deduplicating functions from C++ template expansions.
[5] Weak functions are only used if no non-weak counterpart is defined

multaneously on all MMViews. Also, the contents of most mappings are synchronized by sharing the physical page frames. Only for mappings that the user explicitly marked as *decoupled*, the kernel will establish a copy-on-write mapping, whereby those mappings contain MMView-local memory. Also, threads can switch between MMViews and can create a new MMView by cloning their current view.

The existing [14] MMView Linux extension implements MMViews as separate page-table trees. So, since an MMView is technically a separate address space, they induce higher memory overhead (for the page tables) and increase the TLB pressure if two MMViews are active on the same core. Also, page-table modifications, although the extension synchronizes them lazily, have a higher run-time overhead. However, switching views is rather cheap as the kernel only exchanges a single CPU register.

Coming back to MELF, we use the described extension to execute multiple variants in one process concurrently: We decouple all primary-variant regions, allowing the user to create one MMView for each desired variant combination. With the described base-mode primitives, the user can load different variants into the MMViews. Thereby, an MMView can combine these variants from the variant-overlay groups.

In Fig. 1, step 5, we see two MMViews α and β, which currently have loaded variant A resp. B. We see that the non-multivariant text and data remain shared and only in the overlay region (0x2000-0x3000) is decoupled. Since MMViews have a synchronized structure and the MELF linker aligned the start address of the multivariant functions, all common symbols (e.g., `call f`) and function pointers (e.g., `gptr`) are globally valid and threads in different views can easily co-operate.

With MMViews in place, a thread can switch variants on function-call granularity, for which call and return edges have to perform inverse MMView switches. For this, the run-time library provides a trampoline function (Lst. 2) that switches to the desired MMView, forwards arguments, restores the previous MMView and returns the return value. As the trampoline is not part of an overlay, it can also transfer the control flow between two multi-variant functions. The call protocol for MELFs defines the following call-chain:

1. Call `call_with_helper` with a variant index, function pointer and its arguments
2. Switch to the variant, save old return pointer and replace it by `call_return`
3. Jump to provided function pointer
4. Return from provided function pointer (now returns to `call_return`)
5. Switch back to the variant before the call-chain started
6. Jump to saved, original return pointer, ending the call-chain

```
_threadlocal variant_id_previous = 0;
_threadlocal variant_return = nullptr;
variant_id = 1;
func_pointer = &do_work;
func_arg = 10;
// 1. Call trampoline.
call_with_helper(variant_id, func_pointer, func_arg){
    asm {
        // 2. Switch view
        push variant_id
        syscall_variant_switch
        // Syscall result is old variant id. Save it.
        xchg %rax, variant_id_previous@threadlocal
        // Load new return pointer "call_return".
        leaq call_return(%rip), %r10
        // Exchange return pointer with "call_return".
        xchgq    %r10, (%rsp)
        // Save old return pointer.
        xchgq   %r10, variant_return@threadlocal
        // 3. Jump to function pointer.
        jmp func_pointer
    }
}

// 4. "func_pointer" will return to this function.
call_return(){
    asm {
        // 5. Load old variant id and switch back.
        mov variant_id_previous@threadlocal, %rax
        push %rax
        syscall_variant_switch
        // 6. Return to original return pointer.
        jmp    variant_return@threadlocal
    }
}
```

Listing 2: Trampoline function `call_with_helper` ensures to call `call_return` at the end of the call chain to switch back to the caller's original application variant.

Since the protocol always requires jumping back to the original application variant, we only demand local quiescence per thread.

3 Case Studies

As the MELF approach is a general semi-dynamic–variability method for compiled languages, we now provide multiple case studies to demonstrate the potential of our approach. We will justify, for each case study, its relevance, describe the usage of MELF, and show its benefits with a quantitative evaluation. Thereby, we will only focus on the MMView mode as we consider it the more interesting application mode for MELF.

Benchmark Setup

We cover the server-centric scenarios with a dual-socket system (Intel Xeon Gold 6252, 2.10GHz, 2×24 physical cores, 2 NUMA nodes, 384 GiB DRAM, hyperthreading disabled). Additionally, we use a smaller machine with more restricted hardware (Intel i5-6400, 4 cores, 32 GiB DRAM, no hyperthreading). On the software side, we used Debian

SQLite 3.39.4 (a29f994989)		*(both views)*	
`-O3`		Required for scalability	
`DEFAULT_MEMSTATUS=0`		Required for scalability	
`PAGE_CACHE_OVERFLOW_STATS=0`		Required for scalability	
`ENABLE_RTREE=1`		Required for workload	
Unify `struct sqlite3_mutex`		Required for MELF compatibility	
Perf. View	`NDEBUG=1`	**Debug View**	`SQLITE_DEBUG=1`
Functions: 1432		Functions: 1726	
.text=1 008.5 K	.rodata=27.4 K	.text=1 294 K	.rodata=51.6 K
.data=2.6 K	.bss=1.2 K	.data=2.6 K	.bss=2.3 K
MELF Overlay			
Aligned Functions: 1310		Padding: 372.3 K (13.48 %)	
VM Size: .text=1 314.3 K	.rodata=61.8 K	.data=2.7 K	.bss=2.5 K

Table 1: Overview over the SQLite case-study binary

GNU/Linux 11 with an MMView-enabled Linux 5.15 kernel with Spectre and Meltdown mitigations enabled.

3.1 Case-Study: SQLite Asserts

With this case study, we demonstrate that MELF is able to overlay multiple handwritten code variants and that we can performance-isolate both variants for thread-contextualized execution (with MMViews). More concretely, we build a MELF binary that contains two variants of the SQLite library: (1) the *debug view*, where `assert()` statements and additional sanity checks are enabled, and (2) the *performance view*, where these are disabled. Within the same process, multiple threads execute read-only SQL queries, either with the debug view or the performance view. We vary the total number of threads and the number of threads in the debug view, as well as the benchmark machine.

Scenario Justification Unlike compiler-based security measures, executable asserts [7], [8] are inserted manually by the developers to test high-level invariants at run time. They are an intrinsic part of debug builds, which often include extended data structures and code paths to check application behavior. Thereby, assertions not only assist the development of safer programs [9], but they are also an active security measure [10]. However, due to their complexity, size and performance impact, they are usually disabled in production in favor of a performance/release build. With MELF, we can provide a more restricted debug view with enabled assertions, for example, for SQL queries that handle user input. Technically, this case-study is of interest as it shows how to manage multiple variants that interpret data differently.

Workload We use a geospatial proximity search, since handling two-dimensional data requires complex algorithms and data structures. On the list of 2856 UK postcodes, we issue SQL queries that find the geographically closest code that is not further away than 25 km for randomly chosen coordinates in the UK. For handling coordinates, we use SQLite's R-Tree plugin.

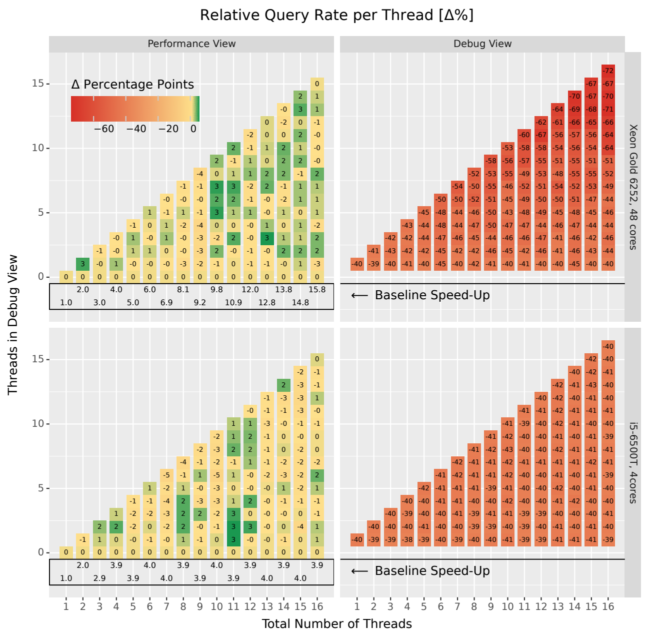

Figure 3: SQLite Performance Measurements

Benchmark In Tab. 1, we provide a comprehensive overview about the used benchmark binary. Since SQLite's default configuration did not scale beyond a single core, we had to disable some statistic features to limit contention. While running both views concurrently worked out-of-the-box for most parts, we had to unify the `struct sqlite_mutex` as the debug view requires additional fields to track mutex ownership. Without a unified data type, the address calculation for array elements differed and provoked a crash. In total, we had to change 30 lines of code.

For a seamless interoperability, MELF aligns 1310 functions by inserting a total of 372.3 KiB of padding, which is 13.48 percent of the combined size in the virtual address space. MELF already optimized the required padding by using 109/202 view-private functions in the performance/debug view as gap fillers. For the mutable global data in (.data, .bss), we align both variants but only use the debug view's data.

Performance Isolation In order show that the MELF approach is able to isolate the impact of the debug view on threads in the performance view, we run the benchmark with 1 to 16 threads, whereby 0 to 16 threads execute permanently in the debug view, while the others execute in the performance view. We also execute the benchmarks on our 4-core and on our 48-core machine in order to determine if core contention has a significant impact. We execute each benchmark for 60 seconds, record the number of completed SQL queries

In Fig. 3, we show the per-thread SQL-query rate and normalize it to the results where all threads execute in the performance view (y-axis = 0), which we consider the baseline for this experiment. For the baseline case, we also show the speedup to confirm that contention within SQLite itself is not the cause of performance degradation but only the usage of MMViews and MELF. As expected, we see near perfect speedup on the 48-core machine, while the speedup caps around 4 on the 4-core machine. Please note the highly asymmetric color scale in this figure.

In the debug view, we see a significant impact of the additional assertions and sanity checks on the query rate. As the slowdown on the 48-core machine (-39 % to -72 %) is significantly worse for more threads in the debug view than on the 4-core machine (-38 % to -43 %), we conclude that the additional sanity checks provoke more contention due to additional state locking.

In the performance view, we see that the number of threads in the debug view has no consistent impact on the other threads, and some results even indicate better indicate a higher performance with using MMViews. Therefore, we take a look at the relative standard deviations for the baseline case to determine if these results stem from SQLite itself. While we cannot derive any conclusions from the relative standard deviation for the 4-core machine (0.3 %–12.4 %), the 48-core machine (rel. stdev.: 0.3 %–1.7 %) suggests that the MELF approach also has a small impact on the performance view. If compared to the observed relative query rates (-4 % to 3.4 %), we conclude that MELF has a negative performance impact of around 1 percent and adds around 2 percent of jitter. Nevertheless, in relation to the impact of globally-enabled assertions, the MELF approach isolates the impact of additional sanity checks in SQLite successfully.

3.2 Case-Study: Thread Pools on Heterogeneous Instruction-Set Machines

With this case study, we show that MELF eases the programming of non-homogeneous multicore machines where cores share a common subset *instruction-set architecture (ISA)* but have additional heterogeneous ISA extensions. More concretely, we provide a thread-pool abstraction (see Fig. 4) that accepts jobs together with a hint on which core type the job will run best. Depending on the current load, the pool schedules the job (preferably) on a hinted core where it uses a MELF-prepared code view that exploits the core-specific ISA extensions or on another core with a different code view that is optimized for that core. Thereby, the thread-pool user fully utilizes her heterogeneous architecture without the need for adapting her code paths for the specific architecture.

Scenario Justification While the first non-uniform multicores (e.g., , ARM big.LITTLE [19]) came with a unified ISA, recent work [20]–[22] investigates on the performance and energy benefits of heterogeneous ISAs. However, ISA diversity poses a programmability challenge as programmer are not keen to distribute their program/data

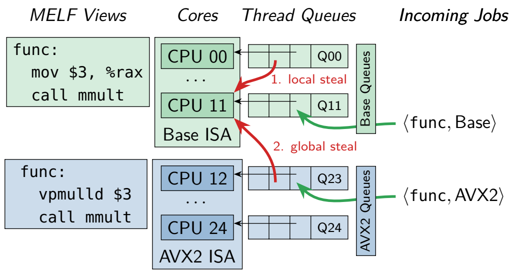
```
func:
  mov $3, %rax
  call mmult
```
```
func:
  vpmulld $3
  call mmult
```

Figure 4: Heterogeneous-ISA Thread Pool

flow manually over different ISAs. Therefore, researchers proposed fault-and-migrate [23], cross-core invocation [24], and multi-kernel [25] methods to manage this variability. With MELF, we take a step towards the seamless integration of heterogeneous ISAs into our programs. Technically, this case-study is of interest as we make use of different cross-cutting compiler options on instruction level, something that is not easily expressible on a language or ifdef level.

System Model As we have no heterogeneous-ISA machine at hand, we simulate one by virtually dividing one NUMA node of our 48-core machine into two partitions (see Fig. 4): On the 12 AVX2 cores, modern AVX/AVX2 vector instructions are available, while the other 12 cores lack this ISA extension.

Work Load For our benchmark, we choose two job types that benefit differently from the AVX2 instructions: While jobs with a recursive Fibonacci (n=36, Base/AVX: 57.9 ms) do not benefit at all, the duration of a Matrix-multiplication (565×565) job drops from 58.3 ms to 38 ms on the AVX2 core. For the matrix multiplication, we use the Eigen C++ library (v3.4), which uses explicit ISA specialization according to the given compiler flags. Please note, that we have chosen the parameters such that the base-core execution time match. As work load, we submit 1000 jobs with 0 to 100 percent of the jobs being matrix multiplications (see Fig. 5) and record the end-to-end latency of those 1000 jobs as well as the accumulated job execution time.

Thread-Pool Variants Based on Eigen's non-blocking thread pool, which already implements thread-local queues and work stealing, we build three thread-pool abstractions that all take a function pointer and a scheduling hint as a job description: The **1 Pool, Base only** variant executes all jobs on a single 24-worker thread pool and only uses code without AVX2 instructions; the scheduling hint is ignored. The **2 Pool** variant uses two 12-worker thread pools, one for the base cores and one for the AVX2 cores; each core executes code specialized for its ISA and workers are pinned to its core; *no* stealing happens between the pools; and the scheduling hint selects the thread pool. The **1 Pool, MELF** variant uses a single 24-worker pool that utilizes MELF: Each worker thread is pinned to its core and executes in a MELF code view that is specialized for its ISA. If a worker

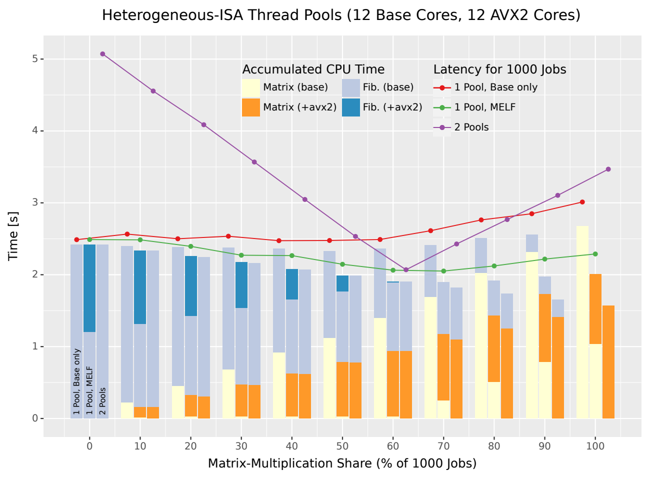

Figure 5: Latency and accumulated processing time for a mixed work load on a 24-core heterogeneous-ISA machine with different thread-pool configurations. To scale latency and processing time, the processing time was divided by 24.

queue runs empty, the worker first tries to steal from workers with the same ISA (see Fig. 4, 1. local steal) before stealing from other ISAs (2. global steal). Please note that stealing from a foreign ISA queue works seamlessly as the local MMView exposes the same functions but implemented with different instructions.

Results In Fig. 5, we show the evaluation results on one NUMA node of our 48-core machine, where we compare the three pool variants with respect to required processing time. Please note, that we divided the accumulated processing time by 24 to match its scale to the end-to-end latency. The remaining difference between latency and processing time stems from pool overheads and execution phases where not all workers execute jobs.

For the Base only variant, processing time is, as expected, only spent in the base code view (less bright colors). The slight increase in both curves stems from the increased cache pressure if 24 cores execute matrix multiplications in parallel compared to executing recursive Fibonacci calls on the (cached) stack. Although the 2 Pool variant spends the least amount of processing time, its end-to-end latency for 1000 jobs is significant at both ends as it only utilizes 12 cores at 0 and 100 percent matrix-multiplication jobs. Also, it achieves its best latency at 60 percent multiplications, which is expected from the ratio between a Fibonacci job (57.9 ms) and an AVX2-Matrix multiplication (38 ms).

Finally, the MELF-enhanced 1 Pool variant, performs better in both dimensions: Compared to Base only, it uses less processing time as it actually utilizes the AVX2 instructions, whereby also its latency is better. Compared to the 2 Pool variant, it always utilizes all cores resulting in a consistently low latency and only when more than 60 percent of the submitted jobs are Matrix-multiplications requires more processing time.

3.3 Case-Study: Profiling in memcached

In this case study, we dynamically en-/disable compiler-introduced function-level profiling on a per-thread basis with MELF and MMViews in `memcached` (v1.6.10). Similar to the SQLite study, we combine two memcached variants in one binary:(1) In the *profiling view*, the compiler (with the `-pg` flag) introduced `mcount()` calls into function prologues that record the invocation, while (2) the *performance view* contains the same functions but without profiling code. On a per-connection basis, worker threads either select the profiling or the performance view. For our benchmark, we gradually change the number of profiled connections and measure the request handling time within `memcached`.

Scenario Justification As developers cannot emulate complex production environments, it is often up to the DevOps team to detect and explain performance anomalies after deployment. For this function-level profiling, as provided by `gprof` [26], would provide precise insights about call frequencies and caller–callee relationships, but its cost prohibits us to have it permanently enabled. Also, in a multi-tiered environment, where only some clients incur a certain anomaly, it is desirable to enable profiling only selectively for certain threads and requests. Because `gprof` consists of both, a compiler instrumentation to modify function translation and a statically-linked profiling library, developers are unable to define exclusive code paths they want to profile. They can either profile the whole application or nothing at all. Thanks to MELF, the DevOps team can enable `gprof`-profiling dynamically and selectively, thus limiting the impact to a minimum.

Technically, this case-study is of interest as we reduce contention on cross-cutting features.

Work Load As a work load for our multi-variant `memcached` server, we use the `memtier` benchmark, which is a specialized benchmark for key-value databases [27]. On the client side, we use its default SET–GET ratio of 1:10 and start 16 threads with 50 clients each, resulting in 800 clients with 800 active connections to *memcached*. We execute the benchmark on the same machine, but pin `memcached` to one NUMA node, while pinning `memtier` onto the other. We record data request latencies until each view has serviced 100 million requests.

Benchmark Unlike other servers, `memcached` has an event-based design and the worker threads execute a state machine for each connection. Thereby, connections can be easily rebalanced between workers and one worker routinely handles many connections. To match the 16 `memtier` threads, we start `memcached` with 16 worker threads that, together, service all client requests.

For new connections, we decide whether it will execute in the performance or the profiling view, mimicking scenar-

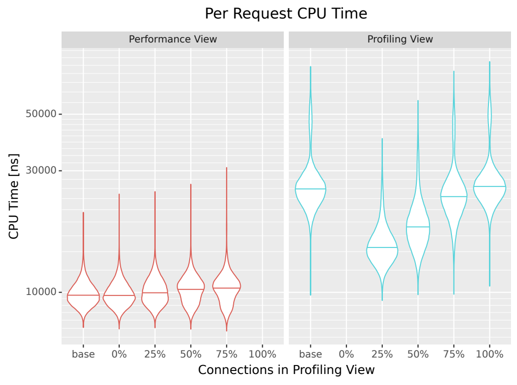

Figure 6: Memcached Performance Measurements for 1M sampled Data Points per Violin

ios where certain IP ranges or customers are profiled. In our benchmark, we use enable profiling for the first N (0 %-100 %) connections to demonstrate the impact of profiling. As long as the profiling percentage threshold is not met, each connection will be profiled. Because `memcached` distributes connections round-robin across workers, each worker thread will serve both, profiling and performance connections. Consequently, every worker thread switches between performance and profiling view continuously across the whole benchmark, which increases TLB pressure since they live in distinct address spaces. Nevertheless, as we will show, this penalty is still better than to enable profiling globally and could even be improved by a profiling-aware connection scheduling.

For our measurement, we record the execution time of the `drive_machine()` function, which is responsible for executing the per-connection state machine, making the function crucial for the request latency. For each benchmark run, we record request latencies until we reach 100M data points for each view. Therefore, for the mixed-mode benchmarks (25 %-75%), we will end up with 200M data points.

Results In Fig. 6, we show violin plots for the `drive_machine()` execution times separated by requests serviced that were serviced in the profiling or the performance view. Please note, that the 25% performance violin on the left matches the 25% profiling violin on the right as both stem from the same benchmark run. Also, we include the *base* variants, which show the results for a `memcached` server with statically enabled or disabled profiling without MELF or MMViews. For preparing the violin plots, we limit each data set (100M data points) to the [0.01%, 99.99%]-interval to remove outliers and randomly sample 1 million representative data points.

By comparing the base variants, we see that profiling has a significant impact on `memcached`'s most important function and increases its execution time by 175 percent. Further, for both views, the violin with the maximum number performance/profiling connections match the results of the base variant (i.e., performance base $\leftrightarrow$ 0% performance). From this, we can conclude that MELF enables us to enable and disable profiling dynamically at run-time without having a continued run-time impact.

For the profiling view, we see that the impact of profiling *per request* increases if more connections use the profiling path. This behavior can be explained by looking at the `gprof`-induced into the code: For each function, `gprof` allocates a counter variable that the compiler-introduced `mcount()` function uses to keep track of the number of invocations. As the activated `memcached` logic is rather small and executed by 16 threads in parallel, all workers in a profiling view access the same small set of per-function counters. Together with the fact that `gprof` does not even cache-align these counters, profiling results in many cross-core cache invalidations. Further, this effect scales with the percentage of threads working in the profiling view as more cache conflicts happen.

For the performance view, we see a slight increase in the median and tail latencies if more connections are shifted to the profiling view. Since workers need to switch views if the currently active MMView does not match the next processes connection, MELF increases the TLB pressure, impacting also workers in the performance view. Also, the frequent cache invalidations in the neighboring profiling view increases the cache-coherence traffic and puts a burden on the memory bandwidth. Nevertheless, even with 75 percent of all connections being profiled, the median over the base variant only increases by 7.61 percent for performance connections, which is far less than activating profiling globally. Inspired by these results, one could restrict profiling to a single worker thread or a small set of connections to get a statistic picture of the whole `memcached` process without inflicting the described cache conflicts from invocation counters.

In total, the MELF approach was able to make the static `gprof` method dynamically and selectively applicable without requiring a restart of the process and without touching the compiler.

3.4 Case-Study: ASAN in MariaDB

With this case study, we demonstrate that MELF is able to handle multiple variants in complex C++ projects. We compile MariaDB (> 20000 functions) once with (`-fsanitize-address`) and once without address sanitizer [5] and use the MELF linker to overlay both variants in the same binary. At run-time, we decide on a per-user basis whether a client's SQL queries are executed with sanitized or unsanitized MMView.

Scenario Justification Sanitizers [28], like Address-Sanitizer [5] and UBSan [29], are often implemented as compiler transformations and they are usually used at development time to find bugs. However, due to their high

MariaDB (10.11)	
ASAN View	**Normal View**
Functions: 21 448	Functions: 20 986
.text=16 777.4 K	.text=4 661.6 K
.rodata=1 082.2 K	.rodata=1 079.2 K
.data=173.9 K	.data=173.9 K
.bss=218.3 K	.bss=218 K
MELF Overlay	
Aligned Functions: 20 615	Padding: 12 487 K (33.87 %)
VM Size: .text=16 934 K .rodata=1 099 K	.data=180 K .bss=224 K

Table 2: Overview of the MariaDB case-study binary

Clients	Normal View		ASAN View	
Normal / ASAN	Median	99%	Median	99%
24 / 0	63 us	73 us	–	–
18 / 6	63 us	74 us	90 us	104 us
12 / 12	63 us	74 us	89 us	102 us
6 / 18	64 us	74 us	90 us	102 us
0 / 24	–	–	89 us	102 us
Base w/o MELF	47 us	55 us	89 us	100 us

Table 3: Query Latency for Sysbench `oltp_point_select` benchmark on MariaDB with and without AddressSanitizer (ASAN).

overheads, they are then disabled in production, although they could provide an additional level of sanity checking for code that handles user input. With MELF, we enable AddressSanitizer, which was found to be the most common sanitizer [28], for individual database users in MariaDB, whereby we mimic a trusted–untrusted customer model. Technically, this case-study is of interest as MariaDB is a multi-threaded, large server application. With ASAN being strongly invasive on the code and data path, it helps to understand how MELF scales for large code bases.

Work Load As a work load, we use the `sysbench` [30] `oltp_point_select` benchmark. On our 48-core machine, we execute and pin MariaDB to NUMA node 1, while sysbench runs on NUMA node 2. We start MariaDB in the one-thread-per-connection mode, and always have 24 concurrent sysbench connections. To satisfy the mentioned trusted–untrusted customer model, we execute two sysbench instances, each of which connects as different database user. To vary the load between the ASAN/no-ASAN view, we vary the distribution of the 24 connections between both instances. We use the output of sysbench, which records the end-to-end latencies per transaction, as our result data.

Results In Tab. 2, we see an overview of the MariaDB binary produced by the MELF linker. First, we see that the application of ASAN increases the number of functions, as the compiler cannot inline and eradicate some of the smaller functions. We also see that the ASAN variant's text section is 2.6 times larger than the normal text section. Together with the fact that 98 percent of the normal view's functions had to be aligned and, therefore, could not be used for gap filling, explains the larger percentage of padding bytes (33.87 %).

In Tab. 3, we show the end-to-end latency results for the `oltp_point_select` benchmark. First of all, we see that AddressSanitizer has a significant impact on the performance of MariaDB as it increases the median latency by 89 percent. This latency penalty is also inflicted on clients whose queries are processed in the ASAN view. However, also clients in the normal view have a 36 percent increased query latency. This increase can be explained by the fact that the ASAN run-time library still has to intercept and wrap heap allocations, which are known to have a major impact on query performance[31], in order to keep its shadow-memory map up to date. However, in the normal view, MELF only removes the additional checks from the query processing and the additional overhead from the run-time library remains.

4 Discussion

In the following, we discuss limitations and benefits of the MELF approach.

Multi-Variant Data As we have discussed in Sec. 2.3, the MELF approach is currently limited to a strict data-object sharing semantic, where all variants share the data sections of the primary variant. For this, the data and its interpretation have to be compatible in all variants, which can, as we have seen with SQLite (Sec. 3.1), require some manual program modifications.

The following program demonstrates this limitation as it is incompatible in three different dimensions: (1) If a lock is allocated in A, the object would be too small to usage in B, (2) the field `L` has different offsets, and (3) both variants have a different idea about the lock state (`1` vs. `-1`).

```
// Variant A                 // Variant B
struct lock { int L; }       struct lock {int 0; int L;}
#define LOCKED 1             #define LOCKED (-1)
```

Supporting these cases in general is impossible, as it would require complete program understanding on the language level. However, for many cases, one could use semi-automated transformer functions [13], [32], [33], known from dynamic software updates, to synchronize two copies of the data.

For data initialization, we use the variant that is active at the initialization time. Therefore, we use the global data segment and invoke all global constructors in the primary variant. Function-local static variables are, in line with C/C++ semantics, initialized at the first call of the respective function and, thus, in the context of the then active variant.

Besides strict data sharing, the MELF linker also supports a strict data-object partitioning. For this, the linker has to keep all data sections, let each variant only reference its own data sections, and we would run the constructor of all variants at program start. In this use case, the developer has to ensure that objects do not flow (i.e., across the univariant parts of the program) across variant boundaries. This mode could be useful for using multiple incompatible versions of libraries that make heavy use of global state.

MMView Dependency We acknowledge that MELF plays out its benefits particularly in combination with MMViews, which we use to back the same virtual-address range with different contents depending on the active thread of a process. The MMView approach has disadvantages, such as memory overhead and increased TLB pressure [14]. Because the exact runtime overhead of MMViews highly depends on the size of virtual memory and its physical data (plain data in RAM, file-backed mappings), a general overhead cannot be quantified. Furthermore, the measurable effect on the TLB directly correlates to a thread's view switch frequency and memory access patterns, which is individual to every application. For view creation and switching, a mean runtime penalty of 7µs with a standard deviation of 6µs has been measured on earlier benchmarks for memcached and MariaDB [14]. In another recent study of memcached, the cost of MMViews were only measurable for context-switches between different views. In general, however, a transition from one view to another is comparable to a context-switch between two threads of two different processes. As an alternative, multithreaded MELFs could be facilitated through CPU-assisted segmentation [34], such as supported by the IA-32 architecture [35]. With segmentation, we would load every variant into its own segment and each thread could select its variant by setting its code-segment selector register accordingly. On the IA-32 platform, where call and jump instructions implicitly use the code segment, this would be equivalent to MMViews. Without the separate address-space clones, the memory and TLB overheads would be replaced by a minor offset calculation overhead that segmentation entails.

Although segmentation contradicts Linux's flat memory model, MELF binaries could easily be supported if (a) the kernel provides means to initialize and switch hardware segments and (b) if the code-segment register is preserved between thread switches. Unfortunately, segmentation as a virtual memory primitive is currently not in fashion on modern platforms. Particularly, it has been removed from AMD64 [35] and was never available on RISC architectures. Given that segmentation has other advantages, such as safety benefits, we would applaud a renaissance of this virtual memory primitive. However, even without segmentation, we could theoretically implement text variants, using position-independent code coupled with the segmentation remnants in AMD64 (FS/GS register) to facilitate variant-adherence for indirect jumps. However, this would require intrusive linker and compiler modifications.

Switching For both modes (base and MMView), we demand that switching the variant takes only place at function boundaries. This limitation stems from the fact that MELF only aligns function start addresses, but all other intra-function addresses could be unaligned. For example, saved return addresses may not be valid in the other variant. However, with additional compiler support, this quiescence requirement could be weakened: For example, if the compiler would also align call sites and would make the call frames at those call sites compatible across variants, we could switch variants flexibly at every call and return edge. Such an extension could be beneficial for supporting workloads on heterogeneous ISA as it would, for example, ease thread migration between different ISAs without stack rewriting [36].

Applicability and Benefits With our case studies, we have shown that the MELF approach is applicable to a wide range of programs. By covering not only C but also C++ projects, which result in more complex object files (e.g., C++templates are a main user for COMDAT), we have demonstrated that MELF works on multiple programming languages. Further, as our approach only requires a compiler to produce "sectioned" object files, we are in principle language agnostic and widely applicable.

Also, MELF is agnostic to the source of the code modification. As shown, we support automatically-introduced compiler transformations (e.g., profiling) as well as manually-encoded variants (e.g., SQLite). Thereby, MELF covers more scenarios than pure language-based methods, like *aspect-oriented programming (AOP)* [37]. Further, as MELF prepares everything at link time, static binary validation could make MELF safer than dynamic-binary instrumentation (e.g., Intel Pin [38]), which was criticized to ease an exploiter's life [39].

Further, we have shown that MELF's semi-dynamic approach to variability is able to cover a wide range of use cases that are security-related (assertions, ASAN), provide DevOps with deeper insights (profiling), and ease the support of coming hardware generations (heterogeneous ISAs). We believe that especially the DevOps and the heterogeneous-ISA scenario will require semi-dynamic variability, since: (1) We need more dynamically-observed metrics to understand our complex systems down to the individual hot path. (2) Extensible architectures, such as RISC-V platform [40], with its many ISA compatibility levels, will boost the spreading of heterogeneous-ISA machines. (3) In many settings, we simply have not the choice to drop existing applications in favor of from-scratch developed software.

Although three of our four case-studies do not dynamically switch views during runtime, we were able to gain reasonable performance isolation in each application scenario: For profiling in memcached we achieved performance isolation of profiling connections and do dynamically switch a thread's view based on the connection currently being served. In the other case-studies, we were able to obtain: (a) Performance isoliation and improved robustness for dynamic assertions in SQLite. (b) Performance maximization via ISA-specialized function variants with thread-pools. (c) Performance isolation and improved security for ASAN in MariaDB. Additionally, function pointers work "out-of-the-box" for MELF, which eases a programmer's life.

We also imagine that MELF can be used in an embedded setting, where no MMU is available to implement MMViews: For these machines, MELF can prepare variant overlays of in-flash text segments, which then can be exchanged at run-time by partially rewriting the flash memory. Thereby, multiple software variants can be supported in one device without inducing indirection overhead.

Also, we have seen that MELF's function alignment only induces moderate memory overheads (see Tab. 1), while the run-time overhead in combination with MMViews depends on the concrete case study. Nevertheless, even in the `memcached` case study, where threads switch on a regular basis between views, the run-time overhead was limited to less than 8 percent. Furthermore, Fig. 6 shows that the median runtime latency in the performance view is equal for the base and 0% variant.

Summarized, MELF is a cheap, language-agnostic method to lift static code variability to the semi-dynamic level. MELF is widely applicable and provides us with a framework for further explorations of semi-dynamic variability.

5 Related Work

Technically, text overlays [41], [42] are a closely related topic: They were used to reduce a program's primary-storage requirement by loading only the currently used subset of functions into the memory. While overlays have a renaissance [43], [44] for managing complex memory hierarchies, they are fundamentally different as they partition one program to fit it into a smaller memory. In contrast, MELF overlays multiple but similar programs in one binary and, with MMViews, execute those variants concurrently.

On the language level, aspect-oriented programming [45] if applied dynamically [46], [47] is similar to MELF. However, as aspects only add code before, after, or around function (calls), it does not support variants that stem from generic and cross-cutting code transformations.

Function Multiversioning [48] is a GCC extension to generate multiple versions per function, each of which specialized for the availability of different instruction-set extensions. The loader selects one variant on function granularity, which, unlike with MELFs, cannot be changed later on.

Fat binaries support multiple processor architectures by embedding program versions for the different processor types into one executable or library [49]–[54]. The variant to execute can be either selected directly by the operating system [54] or through a polyglot opcode string that is interpretable by both architectures [52]. Nextstep's *Mach-O* format, which was later adopted by Mac OS X, even supports "multifat" binaries that allow more than two different architectures (68K, x86, HP PA-RISC, SPARC) [50], [53], [54]. Going one step further, Cha et al. propose a system for generating multi-architecture binaries that, in contrast to fat binaries, use the same program string which is transformed in a way to be correctly interpretable by multiple processor types [55]. Similarly to the architecture heterogeneity of fat binaries, the *Windows Portable Executable* format has support for multiple platforms as it contains a DOS and Windows program in parallel [56]. Whereas the DOS part is usually just a small stub nowadays, it has been used to ship binaries that work under DOS and Windows in the past. In contrast to MELF, in all these approaches the variant selection covers the whole program, that means it is determined at process start, and cannot be changed later.

6 Conclusion

Multivariant ELF (MELF) is as a binary-level approach for the inclusion of multiple compile-time variants within the same binary and flexible switching between them at run time on function/section granularity. This facilitates the implementation of *semi-dynamic variability*, that is, dynamic switching between feature-variants at run time that are nevertheless generated and known at compile time, whereby even highly cross-cutting compiler features become configurable at run time. In combination with a kernel extension for in-process address spaces, this even works on the level of individual threads.

MELFs are implemented solely on binary level and mostly independent of the employed languages and compilers. Function variants are aligned by the MELF linker to the same virtual address, so that existing pointers or relocations remain valid even in case of a variant switch at run time. Thereby, MELFs are relatively easy to apply to existing projects. We demonstrated this on the example of four case studies, ranging over a wide range of multithreaded C and C++ projects. In all cases, MELF was able to isolate the costs and benefits of the compiler/developer-induced code variants to those threads, that use it at run time.

Acknowledgments

We thank our anonymous reviewers and our shepherd Kenji Kono for their constructive feedback and the efforts they made to improve this paper. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.

References

[1] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1107–1116. DOI: 10.1109/IPDPSW.2013.207.

[2] C. Cowan, C. Pu, D. Maier, *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM '98)*, San Antonio, Texas: USENIX Association, 1998, p. 5.

[3] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguardtm: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM '03)*, Washington, DC: USENIX Association, 2003, p. 7.

[4] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[6] C. Courbet, "NSan: A floating-point numerical sanitizer," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21)*, Virtual, Republic of Korea: Association for Computing Machinery, 2021, 83–93, ISBN: 9781450383257. DOI: 10.1145/3446804.3446848.

[7] S. Saib, "Executable assertions - an aid to reliable software," in *1977 11th Asilomar Conference on Circuits, Systems and Computers, 1977. Conference Record.*, 1977, pp. 277–281. DOI: 10.1109/ACSSC.1977.748932.

[8] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992. DOI: 10.1109/2.161279.

[9] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray, "Assert use in github projects," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 755–766. DOI: 10.1109/ICSE.2015.88.

[10] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 866–879. DOI: 10.1109/SP.2015.58.

[11] *Intel® oneapi math kernel library*, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (visited on 12/23/2022).

[12] J. Corbet, *Smp alternatives*, Dec. 2005. [Online]. Available: https://lwn.net/Articles/164121/ (visited on 12/22/2022).

[13] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, Ottawa, Ontario, Canada: ACM, 2006, pp. 72–83, ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991.

[14] F. Rommel, C. Dietrich, D. Friesel, M. Köppen, C. Borchert, M. Müller, O. Spinczyk, and D. Lohmann, "From global to local quiescence: Wait-free code patching of multi-threaded processes," in *14th Symposium on Operating System Design and Implementation (OSDI '20)*, Nov. 2020, pp. 651–666.

[15] F. Rommel, C. Dietrich, M. Rodin, and D. Lohmann, "Multiverse: Compiler-assisted management of dynamic variability in low-level system software," in *Fourteenth EuroSys Conference 2019 (EuroSys '19)*, (Dresden, Germany), New York, NY, USA: ACM Press, 2019, ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303959.

[16] *CMake – Cross platform make*, http://www.cmake.org/, visited 2023-01-03. [Online]. Available: http://www.cmake.org/.

[17] *Elf(5) - format of exectuable and linking format (ELF) files*, Linux Progammer's Manual, Mar. 2021. [Online]. Available: https://man7.org/linux/man-pages/man5/elf.5.html.

[18] *LLD - the LLVM linker*. [Online]. Available: https://lld.llvm.org/ (visited on 01/03/2023).

[19] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms," *white paper, ARM Ltd*, 2011.

[20] S. K. Bhat, A. Saya, H. K. Rawat, A. Barbalace, and B. Ravindran, "Harnessing energy efficiency of heterogeneous-isa platforms," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 65–69, 2016.

[21] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association

for Computing Machinery, 2021, 187–199, ISBN: 9781450382878. DOI: 10.1145/3433210.3453112.

[22] W. Lee, D. Sunwoo, C. D. Emmons, A. Gerstlauer, and L. K. John, "Exploring heterogeneous-isa core architectures for high-performance and energy-efficient mobile socs," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17, Banff, Alberta, Canada: Association for Computing Machinery, 2017, 419–422, ISBN: 9781450349727. DOI: 10.1145/3060403.3060408.

[23] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416660.

[24] S. Cho, H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 187–198. DOI: 10.1109/ISCA45697.2020.00026.

[25] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the programmability gap in heterogeneous-isa platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: Association for Computing Machinery, 2015, ISBN: 9781450332385. DOI: 10.1145/2741948.2741962.

[26] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, 120–126, 1982, ISSN: 0362-1340. DOI: 10.1145/872726.806987.

[27] RedisLabs, *Memtier benchmark on github*, https://github.com/RedisLabs/memtier_benchmark, visited: 2023-01-02.

[28] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[29] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in llvm," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 633–647, 2017.

[30] A. Kopytov, *Sysbench – scriptable database and system performance benchmark*. [Online]. Available: https://github.com/akopytov/sysbench.

[31] D. Durner, V. Leis, and T. Neumann, "Experimental study of memory allocation for high-performance query processing.," in *International Conference on Very Large Databases (VLDB)*, 2019, pp. 1–9.

[32] I. Lee, "Dymos: A dynamic modification system," Ph.D. dissertation, University of Wisconsin-Madison, 1983. [Online]. Available: www.cis.upenn.edu/~lee/mydissertation.doc.

[33] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic software updating," *SIGPLAN Not.*, vol. 36, no. 5, 13–23, May 2001, ISSN: 0362-1340. DOI: 10.1145/381694.378798.

[34] J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *Journal of the ACM*, vol. 12, no. 4, pp. 589–602, 1965, ISSN: 0004-5411. DOI: 10.1145/321296.321310.

[35] *Intel® 64 and ia-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4*, Intel Corporation, 2022. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[36] K. Makris and R. A. Bazzi, "Immediate multithreaded dynamic software updates using stack reconstruction," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX '09, San Diego, California: USENIX Association, 2009, pp. 31–31.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, pp. 59–65, Oct. 2001.

[38] *Pin - a dynamic binary instrumentation tool*, Intel Corporation, Santa Clara, California, USA, 2022. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/98650/Pin/doc/html/index.html.

[39] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "Pwin – pwning intel pin: Why dbi is unsuitable for security applications," in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds., Cham: Springer International Publishing, 2018, pp. 363–382, ISBN: 978-3-319-99073-6.

[40] A. Waterman and K. Asanović, Eds., *The risc-v instruction set manual, volume i: Unpriviledged isa – document version 20191213*, Dec. 2019.

[41] R. Cytron and P. G. Loewner, "An automatic overlay generator," *IBM Journal of Research and Development*, vol. 30, no. 6, pp. 603–608, 1986. DOI: 10.1147/rd.306.0603.

[42] R. J. Pankhurst, “Operating systems: Program overlay techniques,” *Commun. ACM*, vol. 11, no. 2, 119–125, 1968, ISSN: 0001-0782. DOI: 10 . 1145 / 362896 . 362923.

[43] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha, “A performance model and code overlay generator for scratchpad enhanced embedded processors,” in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 287–296.

[44] C. Jang, J. Lee, B. Egger, and S. Ryu, “Automatic code overlay generation and partially redundant code fetch elimination,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 2, pp. 1–32, 2012.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)*, (Finland), M. Aksit and S. Matsuoka, Eds., ser. Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Jun. 1997, pp. 220–242.

[46] A. Popovici, T. Gross, and G. Alonso, “Dynamic weaving for aspect-oriented programming,” in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD ’02)*, (Enschede, The Netherlands), G. Kiczales, Ed., ACM Press, Apr. 2002, pp. 141–147.

[47] R. Tartler, D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk, “Dynamic aspectc++: Generic advice at any time,” in *Proceedings of the 2009 Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT ’09)*, (Prague, Czech Republic), H. Fujita and V. Marík, Eds., ser. Frontiers in Artificial Intelligence and Applications, Amsterdam, The Netherlands: IOS Press, 2009, pp. 165–186, ISBN: 978-1-60750-049-0. DOI: 10.3233/978-1-60750-049-0-165.

[48] V. Rodriguez, A. Duenas, and E. Stupachenko, *Function multi-versioning in gcc 6*, Jun. 2016. [Online]. Available: https://lwn.net/Articles/691932/ (visited on 01/10/2023).

[49] A. C. Inc., *Creating fat binary programs*, 1997. [Online]. Available: https://developer.apple.com/library/archive/documentation/mac/runtimehtml/RTArch-87.html (visited on 01/11/2023).

[50] A. Singh, *Mac OS X Internals: A Systems Approach: A Systems Approach*. Addison Wesley, 2016, ISBN: 0134426541.

[51] *Fatelf: Universal binaries for linux*. [Online]. Available: https://icculus.org/fatelf/ (visited on 01/11/2023).

[52] B. Wilkinson, *Something common about ms-dos and cp/m*, 1999. [Online]. Available: https://www.heco.wxwilki.com/commscpm.html (visited on 01/11/2023).

[53] A. Tevanian, M. DeMoney, K. Enderby, D. Wiebe, and G. Snyder, *Method and apparatus for architecture independent executable files*, 1995. [Online]. Available: https://patents.google.com/patent/US5432937A/en.

[54] A. Tevanian, M. DeMoney, K. Enderby, D. Wiebe, and G. Snyder, *Method and apparatus for architecture independent executable files*, 1997. [Online]. Available: https://patents.google.com/patent/US5604905/en.

[55] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton, “Platform-independent programs,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS ’10, Chicago, Illinois, USA: Association for Computing Machinery, 2010, 547–558, ISBN: 9781450302456. DOI: 10.1145/1866307.1866369.

[56] B. et al., *Pe format - win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 01/11/2023).

A Artifact Appendix

Abstract

This artifact includes all tools and a documentation to run the evaluation for multivariant ELFs, a binary-level, language-agnostic approach to semi-dynamic variability. It details how to run and modify four case studies using the provided artifact package, which includes benchmark scripts, a virtual disk file and the MMView kernel and its initrd. Users run the virtual machine via QEMU and execute scripts, which allow to generate and display benchmark data for each case-study. The four case studies are: (a) MariaDB ASan, (b) SQLite assertions, (c) Heterogeneous-ISA thread-pool, and (d) memcached profiling to prove the wide applicability of MELFs. Each case study includes instructions on running, exporting results, and modifying the benchmark. Next to the case-studies, users can also modify the provided MELF linker, the crucial component in creating MELFs, to examine the generation and placement of variant generation.

Scope

Users investigating the MELF benchmarks are able to verify that the only dependencies and changes to-be-made to make use of MELFs in applications are: (1) Express the existence of multiple application variants inside an application-specific linker script file (2) Extend existing application code to declare and load variants. (3) Use the llvm-based MELF linker to link application modules (object files) to the final MELF executable.

For each individual case-study, users can reconstruct the claimed benefits of MELFs described within the paper, which is mainly performance isolation and increased binary size depending on the workload. All evaluators, however, need to keep in mind that running those benchmarks in a virtualized environment will not provide the same results we were able to get for our paper. Your final result highly depends on the hardware your host machines use. But the main concept of performance isolation shall be visible in each benchmark executed.

Contents

This archive provides the user with every evaluation resource needed to run our evaluation setup. Namely, this archive consists of:

- run.sh. A script that starts a virtual machine via QEMU.
- hda.qcow2. The virtual disk file of the virtual machine which includes the whole artifact evaluation, based on debian 11.7.
- linux-mmview-vmlinuz-5.15. A Linux kernel fork of version 5.15 which includes the operating system abstractions for MMViews.
- initrd-linux-mmview-vmlinuz-5.15. The corresponding initrd of the MMView kernel fork.

- README.txt. A documentation file giving a detailed explanation of every benchmark setup and how to build, modify and draw benchmark data.

To start artifact evaluation, the user has to have QEMU installed onto their execution environment and to start run.sh. After the VM booted, the user can get access to the system by logging in either as the user ”user” or as ”root”. The user has a Makefile inside his home directory which contains a target for each benchmark to generate the data and export that data into different formats.

Hosting

The artifact evaluation archive is hosted on the domain of our institution and can be downloaded from there: `https://sra.uni-hannover.de/Publications/2023/melf-usenix-atc23/`

Requirements

Most of our artifacts do not require specialized hardware. For the heterogeneous-ISA artifact, we execute code making use of AVX512 instructions. In order to run this artifact your host machine has to support AVX512, but most of modern hardware does that by default. Otherwise, the list of requirements is:

- Modern CPU with at least 16 cores. If you have less you have to adjust the run.sh script and the benchmarks inside the VM.
- At least 8GB RAM, the more the better.

For software requirements, you need to have installed:

- KVM module installed and loaded on your host machine.
- QEMU virtualization software stack.

Users can deviate from the given requirements, but doing so requires manual modification of run.sh and the benchmark inside the virtual machine.