HotLD: a Workload-Aware Method for Global Code-Layout Optimization of Shared Libraries

XUEQIN NING*, Department of Computer Science, National University of Defense Technology, Changsha, China
JUN MA*, College of Computer Science and Technology, National University of Defense Technology, Changsha, China
ZHOUYANG JIA†, National University of Defense Technology, Changsha, China
YUSONG TAN†, Schoole of Comnputer, National University of Defense Technology, Changsha, China
JIE YU, National University of Defense Technology, Changsha, China
PAN DONG, National University of Defense Technology, Changsha, China
JING WANG, college of computer science and technology, National University of Defense Technology, Changsha, China
LIANGHAO SHEN, National University of Defense Technology, Changsha, China

Dynamic linking is an important technique in the process of software development. While dynamic linking can save memory and enhance maintainability, it also incurs performance overhead and hinders the application of profile-guided code layout optimization techniques to third-party libraries. Existing works (such as BOLT) can solve the problem by generating optimized versions of shared libraries for a given workload. Online PGO methods (such as OCOLOS) can further support on-the-fly code replacement to adapt to workload changes. These methods, however, are limited in several aspects: (1) hard to perform global optimization, (2) high memory consumption and performance overhead, and 3) limited usage scenarios. To address these issues, we propose HotLD, a workload-aware method for global code-layout optimization of shared libraries. HotLD can improve the performance of shared libraries while introducing limited performance and memory overhead. The core idea behind HotLD is to create dedicated code copies for each typical application workload during the offline phase, and perform global optimization according to workload characteristics. At runtime, HotLD monitors the running workload of the target program, then dynamically selects and links an appropriate code copy. We conducted experiments using real-world applications to evaluate the effectiveness and efficiency of HotLD. The results demonstrate that HotLD can improve their performance (up to 22.37%) with limited performance overhead (at the millisecond level). Compared with existing works, HotLD uses 1%-46% memory to support 2×-9× workloads.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

*Both authors contributed equally to this research.
†Corresponding author

Authors' Contact Information: Xueqin Ning, Department of Computer Science, National University of Defense Technology, Changsha, China; e-mail: ning77@nudt.edu.cn; Jun Ma, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China; e-mail: majun@nudt.edu.cn; Zhouyang Jia, National University of Defense Technology, Changsha, China; e-mail: jiazhouyang@nudt.edu.cn; Yusong Tan, Schoole of Comnputer, National University of Defense Technology, Changsha, China; e-mail: ystan@nudt.edu.cn; Jie Yu, National University of Defense Technology, Changsha, China; e-mail: yj@nudt.edu.cn; Pan Dong, National University of Defense Technology, Changsha, Hunan, China; e-mail: pandong@nudt.edu.cn; Jing Wang, college of computer science and technology, National University of Defense Technology, Changsha, China; e-mail: wangjing@nudt.edu.cn; Lianghao Shen, National University of Defense Technology, Changsha, Hunan, China; e-mail: shenlianghao@nudt.edu.cn.

ACM 1544-3973/2025/9-ART
https://doi.org/10.1145/3769310

Additional Key Words and Phrases: Dynamic Linking, Shared Library, Profile-Guided Optimization, Workload-Aware

1 Introduction

With the increasing complexity of modern software functionality, dynamic linking has become an indispensable technology in software development and maintenance [1–4]. Unlike static linking that consolidates all code and data into a single executable, dynamic linking compiles code and data into a independent shared library that are loaded into memory at runtime by dynamic loaders. Compared to static linking, dynamic linking provides significant advantages in memory and disk space efficiency [5] and software update/maintenance [6, 7]. However, this flexibility may introduce runtime performance overhead.

Firstly, dynamic linking introduces performance overhead [8]. The load addresses of shared libraries cannot be determined prior to program execution due to the Address Space Layout Randomization (ASLR) mechanism [9–11]. Consequently, programs must rely on the Procedure Linking Table (PLT) and the Global Offset Table (GOT) to resolve the locations of library functions at runtime. This process not only adds extra trampoline code, but also places strain on the cache system [12–14]. To mitigate these overheads, prior work has explored bypassing the PLT and GOT mechanisms [6, 15]. Agrawal *et al.* [15] proposed recording the target addresses of trampoline code using a branch predictor, but their approach requires specialized hardware support. Ren *et al.* [6] proposed a software-based method that modifies call instructions by replacing the target address with the actual function address at load time (Fig. 1b). While effective, this approach essentially transforms shared libraries into static ones at load time, thereby compromising the memory-sharing advantage of dynamic linking.

Secondly, dynamic linking imposes constraints on code layout optimization [16]. Prior studies have shown that tailoring code layout to better utilize hardware resources (e.g., cache hierarchies and branch predictors) can substantially enhance program performance [17–26]. Profile-Guided Optimization (PGO) is the primary technique for this purpose. However, traditional PGO needs source code recompilation, while shared libraries are typically distributed as precompiled binary, limiting its practical applicability. To address this, Panchenko *et al.* introduced BOLT [19], a PGO tool specifically designed for binary files. BOLT performs basic block-level optimizations by rewriting the segment structure of the target binaries. Although BOLT effectively extends PGO techniques to the binary level, its effectiveness heavily relies on profiling data gathered offline. When workload characteristics deviate from the profiling data, the optimization benefits may degrade significantly, potentially leading to performance regressions [26].

In this regard, researchers have proposed several online PGO methods, such as OCOLOS [26], DCM [25], and JACO [27]. They collect profiling data during program execution, dynamically generate optimized code, and replace code on-the-fly. However, these methods are still limited in the following aspects: (1) Introducing runtime performance overhead. In particular, under frequently shifting workload characteristics, the performance gains from online optimization are often offset by the system's overhead; (2) Impacting user experience. In OCOLOS and DCM, code hot-swapping incurs process suspension, and in DCM, the application's response delay can increase by two orders of magnitude during optimization [27]; (3) Limited usage scenarios. Both JACO and DCM are designed for JVM.

Furthermore, all the existing PGO methods perform optimizations within a single binary, preventing global code layout optimization across shared libraries. Moreover, each process typically maintains its own optimized instance of all shared libraries, increasing memory usage and undermining the memory-sharing advantage of dynamic libraries.

In this paper, we introduce HotLD (Hot-Library Dynamic Optimization), a workload-aware method for global code-layout optimization of shared libraries. The core idea behind HotLD is to create dedicated code copies for each typical application workload, and perform global optimization according to workload characteristics.

After that, HotLD monitors the running workload of the target program, then dynamically selects and links an appropriate code copy at runtime. There are three main challenges during the design of HotLD:

- **Memory Consuming:** Generating optimized copies of all shared libraries for each workload and loading them at runtime may lead to significant memory consumption. To mitigate this, we design HotLib, which only aggregates hot code fragments of all shared libraries corresponding to a specific workload. Each workload will have one HotLib, which will be injected into the process space during the loading phase. Meanwhile, the cold code still remains in original shared libraries. Furthermore, the HotLibs are private to each process. We can rewrite their library function call instructions to eliminate the trampoline code. This approach makes function calls within HotLib as fast as those in statically linked binaries.
- **HotLib Linking:** Linking HotLib with both the application binaries and the original shared libraries presents significant challenges. HotLD has to ensure: (1) the application can correctly call the optimized functions in HotLib instead of original shared libraries; (2) functions within HotLib can accurately reference symbols from the original shared library. To address these, HotLD modifies the GOT of the original binary at runtime, redirecting function calls to the corresponding optimized functions in HotLib. Additionally, as the original shared libraries' loading addresses can only be determined after the program starts, HotLD dynamically adjusts HotLib's code pointers to ensure correct symbol referencing.
- **Workload Matching:** To achieve workload-aware optimization, HotLD needs to identify the workload characteristics at runtime, then select and link the corresponding HotLib. This is non-trivial, since it requires online monitoring, and thus has to be lightweight. To address this, we design a lightweight workload matching algorithm based on the similarity of function invocation behavior. The algorithm compares runtime function call patterns with predefined workload profiles. It calculates a weighted similarity score using the call frequency spectrum and execution time distribution, and selects the best-matching HotLib. Experiments show that the workload can be accurately identified with just one second of sampling.

An important design principle of HotLD is to keep non-intrusive for standard dynamic loader (e.g., GNU ld [28]). This feature makes HotLD easy to use. HotLD itself will be compiled as a shared library. It then leverages the LD_PRELOAD mechanism to intercept the *__libc_start_main* entry function. Users only need to pre-run different workloads, and HotLD will take care of everything else.

We evaluated HotLD on six widely used open-source applications: RocksDB, MARISA-Trie, Python interpreter, OpenSSL, Apache, and MySQL. The results demonstrate that HotLD can improve the performance of these applications with limited memory consumption and performance overhead. Under dynamic workloads, HotLD achieves up to a 22.37% performance improvement, whereas BOLT achieves a 14.89% performance improvement. Additionally, HotLD uses 1%-46% memory to support workloads 2×-9× larger. HotLD introduces a performance overhead of 1% to 5% during monitoring (lasting a few seconds), with a switching process lasting only milliseconds and without suspending the application. In contrast, OCOLOS introduces a performance overhead of up to 17.2% during monitoring (lasting several minutes) and requires the application to be suspended for switching.

In summary, this paper makes the following contributions:

- We identify the challenges in applying state-of-the-art PGO techniques, such as BOLT, to optimize code layout in systems with multiple shared libraries: (1) hard to perform global optimization, (2) high memory consumption and performance overhead, and (3) limited usage scenarios. To address this, we propose HotLD, a workload-aware method for global code-layout optimization of shared libraries.
  HotLD can improve the performance of shared libraries while introducing limited overhead.
- We design three main components in HotLD: (1) a Hot-Generator that can collect and optimize hot functions of shared libraries for each workload, and organize them as HotLib; (2) a Hot-Linker that can dynamically link HotLib to application and original libraries on demand; (3) a Hot-Monitor that can trace workload features using a lightweight method, and switch HotLibs when necessary.

- We conducted experiments using real-world applications to evaluate the effectiveness and efficiency of HotLD. The results demonstrate that HotLD can improve their performance (up to 22.37%) with limited performance overhead (at the millisecond level). HotLD uses 1%-46% of the memory compared to existing works to support workloads 2×-9× larger.

The source code and data are publicly available at https://github.com/Hotld/HotLD.git.

2 Background

2.1 Procedure Linkage Table Mechanism

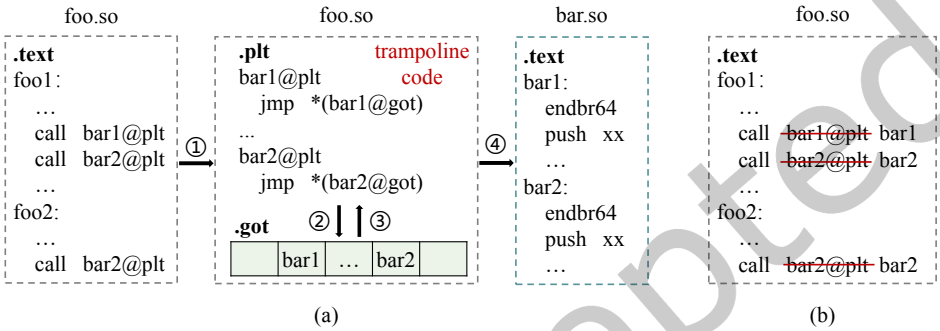

Fig. 1. **Library Function Call Process. (a)** ① Jump to the corresponding PLT trampoline code. ② Query the Global Offset Table. ③ Retrieve the actual address of the target function. ④ Jump to the target function. **(b)** Rewrite the target to the function address to bypass trampoline code.

In dynamic linking, the Procedure Linkage Table (PLT) enables indirect calls to library functions [8, 29], supporting lazy binding which resolves function addresses at runtime instead of load time. As illustrated in Fig. 1a, when a program calls a library function, it first jumps to the corresponding PLT entry containing trampoline code that queries the Global Offset Table (GOT) for the function's address. If the address is already in the GOT, the trampoline code jumps directly to the target function. Otherwise, it calls the dynamic linker to resolve and store the address in the GOT before jumping to the function. Subsequent calls then read the address from the GOT, avoiding repeated lookups.

While this mechanism enhances flexibility, it introduces performance overhead due to trampoline execution. The PLT holds read-only trampoline code, and the GOT stores writable function addresses on separate memory pages, leading to irregular access and cache misses. Rewriting library function calls directly to their target addresses (Fig. 1b) can reduce this overhead. However, modifying read-only code triggers copy-on-write (COW), preventing code sharing across processes.

2.2 Code Layout Optimization

Code layout optimization improves program execution efficiency by arranging code in memory to reduce cache misses, increase instruction cache hits, and lower branch prediction failures. In this work, we focus on three primary code layout optimization strategies: the Function Reordering [23, 30–32], the Basic Block Reordering [33–41] and the Hot-Cold Splitting [19, 42].

2.2.1 Function Reordering. Function reordering adjusts the storage order of functions in binary files so that functions with close calling relationships are placed adjacent to each other. This optimization is guided by a call graph constructed from call relationships and frequencies. The classic Pettis-Hansen (PH) algorithm [30] uses a

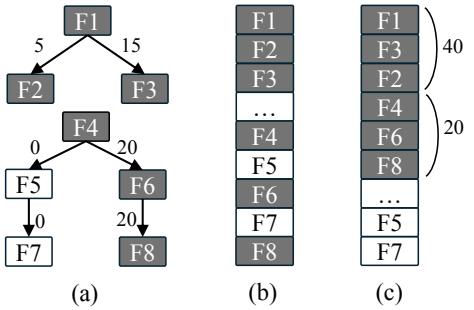

Fig. 2. **Function Reordering Example. (a)** Call graph. **(b)** Original function layout. **(c)** Optimized layout.

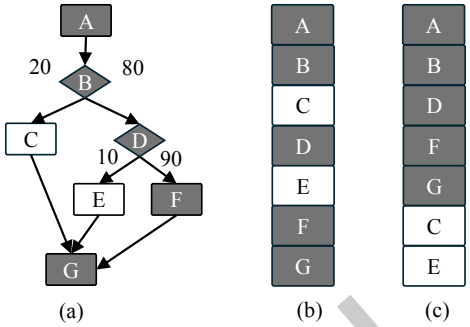

Fig. 3. **Basic Block Reordering Example. (a)** Call graph. **(b)** Original basic block layout. **(c)** Optimized layout.

greedy strategy to group the most frequently called function pairs but does not distinguish between callers and callees, which may result in suboptimal locality. The C3 algorithm [23] improves upon PH by prioritizing callers, reducing long jumps, and enhancing instruction cache and iTLB efficiency. The Hfsort [43] uses hierarchical clustering to group related functions and then applies greedy sorting for a better overall layout. Figure 2 shows an example of Hfsort.

2.2.2 Basic Block Reordering. Basic Block Reordering is a more finer-grained code layout optimization technique that aims to optimize the control flow within a function, as shown in Fig. 3. In the absence of runtime profile data, the compiler relies heavily on static heuristics to infer execution frequency[31, 33, 34, 38, 44]. However, since these methods do not accurately reflect the actual load characteristics, they often lead to sub-optimal code layout. PGO identifies high-frequency execution paths through runtime hot path analysis, and accordingly performs Basic Block Reordering and Function Reordering. Together, these techniques constitute the primary code layout strategies in PGO.

2.2.3 Hot-Cold Splitting. Hot-Cold Splitting aims to separate high-frequency execution code from low-frequency execution code to reduce the interference of low-frequency code to the cache system. It includes three levels: (1) Basic Block: separates hot and cold blocks within functions. (2) Function-Level: stores hot and cold functions separately to avoid loading cold code during hot function execution. (3) Section-Level: places hot and cold code in different sections or pages to improve page table efficiency.

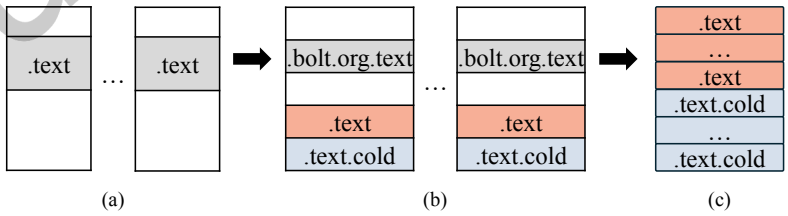

Fig. 4. **Library Code Optimization Process. (a)** Original Shared Libraries. **(b)** BOLT-Optimized Libraries. *.bolt.org.text:* Original *.text* section; *.text:* Hot basic blocks of hot functions; *.text.cold:* Cold basic blocks of hot functions. **(c)** HotLib: Merged hot functions from different libraries with separated hot and cold basic blocks.

2.2.4 Binary Optimization and Layout Tool. BOLT [19, 20] is a post-link binary optimization tool integrated into LLVM [45]. It starts by executing the program to collect profile data, then translates the machine code into LLVM's machine-level intermediate representation (MIR) for analysis. Subsequently, optimizations such as basic block reordering, function reordering, and hot-cold code splitting are performed at the MIR level to produce optimized binaries. In these binaries, cold functions remain in their original section, hot functions are reordered and relocated to a new section, and cold basic blocks are extracted and consolidated into a dedicated cold code section (Fig. 4b). Building on this approach, HotLD further separates hot and cold basic blocks across shared libraries (Fig. 4c), thereby reducing cache pollution caused by low-frequency code.

3 Design Overview

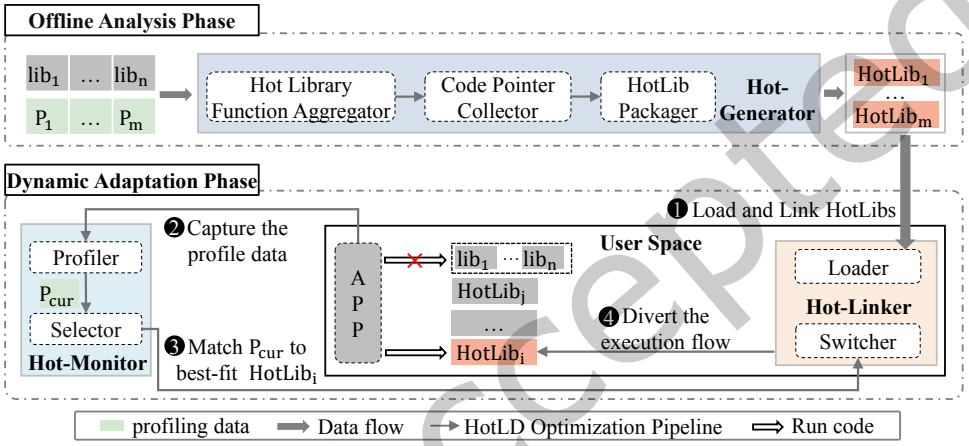

Fig. 5. Overall Architecture of HotLD.

This section introduces the design of HotLD. First, we present the overall architecture of HotLD (Sec 3.1), followed by three key components: Hot-Generator, responsible for generating HotLib (Sec 3.2); Hot-Linker, which links HotLib at runtime (Sec 3.3); and a lightweight Hot-Monitor for tracing program status and switching between HotLibs (Sec 3.4).

3.1 Overall Architecture

The HotLD framework is shown in Fig. 5. It employs a two-phase optimization architecture: offline analysis and dynamic adaptation, to achieve global code layout optimization across shared libraries. During the offline analysis phase, the Hot-Generator collects profiling data from representative workloads and leverages the BOLT [19, 20] toolchain to optimize the code layout of shared libraries. The optimized hot functions of different libraries and code pointers that need to be dynamically patched are packaged into a workload-specific optimization entity HotLib.

In the dynamic adaptation phase, the Hot-Linker loads both the original shared libraries and the generated HotLibs at process startup while preserving the original symbol bindings. The lightweight Hot-Monitor continuously tracks workload characteristics. When the running workload matches a predefined HotLib, the Hot-Linker dynamically redirects function entries in the GOT to the optimized HotLib code. This symbol rebinding enables seamless, transparent switching to the optimized layout without interrupting execution, while maintaining shared memory for cold functions to ensure resource efficiency.

3.2 Hot-Generator

Hot-Generator is the offline optimization module of HotLD. It takes profiling data and shared libraries as input, and generates HotLibs, which contain optimized hot functions across all shared libraries. The main tasks of Hot-Generator include: (1) **Cross-library hot function aggregation:** Collecting optimized hot functions from shared libraries and consolidating them into contiguous memory regions; (2) **Code pointer collection:** Identifying and collecting code pointers within the optimized code and original shared libraries that require dynamic patching to ensure correct linkage; and (3) **HotLib generation:** packaging the optimized code along with the collected code pointers into HotLibs.

3.2.1 Cross-library hot function aggregation. This step consists of the following two phases:

- **Generating optimized shared libraries:** Consider a target application A that depends on a set of shared libraries $\{lib_1, lib_2, \ldots lib_n\}$, where the libraries are arranged in topological order (i.e., lib_i may depend on lib_j if and only if $j > i$). For a representative workload set $\{W_1, W_2, \ldots W_m\}$ of application A, Hot-Generator collects runtime profiling data P_i for each workload W_i. Subsequently, each pair of (P_i, lib_j) is fed into the BOLT framework to generate a load-sensitive optimized library, the process can be formalized as follows:

$$\text{BOLT}(P_i, lib_j) \rightarrow opt\_lib_{ij}, \quad \forall i \in [1, m], j \in [1, n] \tag{1}$$

  where opt_lib_{ij} refers to the library lib_j optimized for the workload W_i. The optimization implements fine-grained reordering at the function and basic block level.
- **Aggregating cross-library hot functions:** For each workload W_i, Hot-Generator extracts hot basic blocks (located in the *.text* section) and cold basic blocks (located in the *.text.cold* section) of hot functions from the optimized shared library set $\{opt\_lib_{i1}, opt\_lib_{i2}, \ldots opt\_lib_{in}\}$. The hot basic blocks are then aggregated across libraries into a contiguous memory region, based on the topological order of shared library dependencies, while the cold basic blocks are stored together in a separate segment, as shown in Fig. 4c. This design separates the hot and cold basic blocks from different libraries to avoid cache system pollution.

3.2.2 Code pointer collection. To link HotLib and original shared libraries, HotLD needs to dynamically correct two types of critical pointers: 1) **Internal code pointers in HotLib:** instructions in the optimized code segment that reference symbols from the original shared library; 2) **Global Offset Table entries:** pointers to optimized functions in the executable and the original shared library.

- **Internal code pointer collection in HotLib:** Hot-Generator extracts and reassembles code fragments from shared libraries, which disrupts internal instruction references to associated library symbols. Since the load addresses of HotLib and original libraries are unknown before program startup, the offsets between HotLib instructions and their target symbols cannot be determined statically. To overcome this, Hot-Generator introduces cross-library relocation entries to enable dynamic corrections.
  Specifically, it disassembles the code sections of HotLib to identify all instructions that access symbols from the original shared libraries (such as *call*, *jmp*, and *mov*). For each instruction requiring correction, a relocation entry is generated, represented as a quintuple:

$$R = (r_{addr}, t_{addr}, l_{idx}, r_{type}, r_{next}) \tag{2}$$

  where r_{addr} is the offset of the pointer that needs to be corrected within HotLib, t_{addr} is the static offset of the target symbol in the original library, l_{idx} is the index of the original library containing the target symbol, r_{type} indicates the relocation type, r_{next} is the distance between r_{addr} and the next instruction. This process can be formalized as:

$$\forall i \in \text{HotLib\_Inst}, \text{GenReloc}(i) \rightarrow R_i \in S_{\text{reloc}} \tag{3}$$

where S_{reloc} is the set of all internal code pointers that need to be patched.

- **GOT entry collection:** To redirect calls from shared libraries to HotLib, Hot-Generator scans the executable and its dependencies to extract GOT entries whose target functions belong to the hot function set F_{hot}, which includes hot functions from all shared libraries. For each matched entry, it generates a triplet:

$$G = (g_{\text{addr}}, h_{\text{addr}}, l_{\text{idx}}),\ G \in G_{\text{entry}} \tag{4}$$

where g_{addr} is the offset of the GOT entry in the original library, h_{addr} represents the offset of the corresponding function in HotLib, and l_{idx} denotes the topological index of the original shared library. These entries guide the runtime redirection to HotLib's optimized code.

3.2.3 HotLib Generation. HotLib encapsulates cross-library optimized hot functions and their relocation metadata, removing redundant sections from the traditional ELF format to achieve a more compact representation. Its design ensures compatibility with dynamic linking, allowing it to coexist and load alongside the original shared libraries. Additionally, HotLib incorporates an MD5-based verification mechanism to verify the integrity and consistency of shared library versions. The following sections detail HotLib's structure and packaging process.

- **HotLib Format:** The HotLib employs a modular binary structure, comprising three components: the HotLib header, the section table, and the data sections. The header records critical information such as the location of the section table, the number of entries, and the offset and size of the code segment. The section table compactly defines the type, permissions, and layout of each data section.
  The data sections are divided into five types: the *.depend* section, which records the index, MD5 hashes, and hot/cold code boundaries of dependent libraries for load-time dependency verification; the *.h_got* and *.h_rela* sections, which contain GOT relocation entries and internal code pointers for dynamic symbol correction; the *.r_only* section, which stores read-only strings; and the *.text* section, which holds the optimized hot code.
- **HotLib Packaging:** The HotLib packaging process begins with section initialization and memory allocation. MD5 hashes of dependent libraries are then computed, and relevant metadata is stored in the *.depend* section. Next, the S_{reloc} and G_{entry} sets are encoded into their respective sections, followed by insertion of the optimized code and library name strings. Finally, the file header and section table are generated, resulting in a self-descriptive binary structure.

3.3 Hot-Linker

Hot-Linker is the core dynamic loading component of the HotLD framework. It loads all HotLibs at program startup and dynamically switches to the most appropriate HotLib instance based on the workload. Additionally, Hot-Linker rewrites indirect call instructions in HotLib to bypass the execution of PLT trampoline code. Since HotLib is privately owned by one process, it does not need the trampoline code for memory sharing mechanisms.

3.3.1 HotLib loading. Hot-Linker is implemented as a shared library (*hotlinker.so*) and injected into the target process via the *LD_PRELOAD* mechanism. This mechanism allows user-defined libraries to be preloaded at process startup, overriding the original library functions. By intercepting the *__libc_start_main* entry function, HotLD causes the dynamic linker to hand over control to Hot-Linker after completing the loading process, rather than to the application. Hot-Linker performs the following operations:

- **HotLibs Loading:** Hot-Linker loads all HotLibs into the target process's address space and computes the hash values of the program header and section tables of each dependent library. It compares these hashes with the metadata stored in the HotLibs, completing the mapping only if the hashes match.
- **HotLibs Internal Linking:** After completing the loading, Hot-Linker traverses the internal relocation entries (S_{reloc}) in HotLib, calculates the PC-relative offsets based on the actual load addresses of the original

shared libraries and HotLib, and patches the relevant instructions to enable access to the original library's data. At this point, the execution flow still resides in the original shared libraries.

3.3.2 HotLib Switching. Before returning control to the application, Hot-Linker initiates a monitoring thread. This thread listens for control signals sent by the Hot-Monitor, indicating which HotLib should be activated at any given time. The monitoring thread will remain in the blocking state when idle to minimize runtime overhead. Upon receiving a signal to activate a specific HotLib, Hot-Linker triggers the hot-switching mechanism, which wakes the monitoring thread to execute the following tasks:

- **Permission modification:** The monitoring thread temporarily marks the target GOT page as writable. As the GOT page is private to the process, this modification does not affect other processes.
- **GOT update:** The monitoring thread iterates through the GOT redirection entries G_{entry}, replacing the entry values with the absolute addresses of the corresponding hot functions in the HotLib.
- **Permission restoration:** The monitoring thread restores the GOT page to read-only.

For short-lived processes or applications with stable workload patterns, HotLD supports static HotLib binding during initialization to eliminate runtime switching overhead. In this mode, the loading and linking of HotLib occur only during process startup.

3.3.3 Indirect Call Optimization. In response to the indirect jump overhead introduced by the PLT/GOT mechanism, Hot-Linker further identifies the relocation entries related to PLT within HotLib ($S_{\text{plt}} \in S_{\text{reloc}}$), and directly rewrites the target addresses of call and jump instructions, replacing PLT entries with the actual function addresses stored in the GOT, as shown in Fig. 1b. This optimization allows library function calls to achieve performance comparable to static linking. Since HotLib is private to the process, the rewrite does not break the memory sharing properties of shared libraries, thereby avoiding memory redundancy issues encountered in schemes such as iFed [6], and as HotLibs contain hot code extracted from multiple shared libraries, the optimization covers (1) indirect calls within a library's hot code to itself and (2) indirect calls within a library's hot code to other shared libraries.

3.4 Hot-Monitor

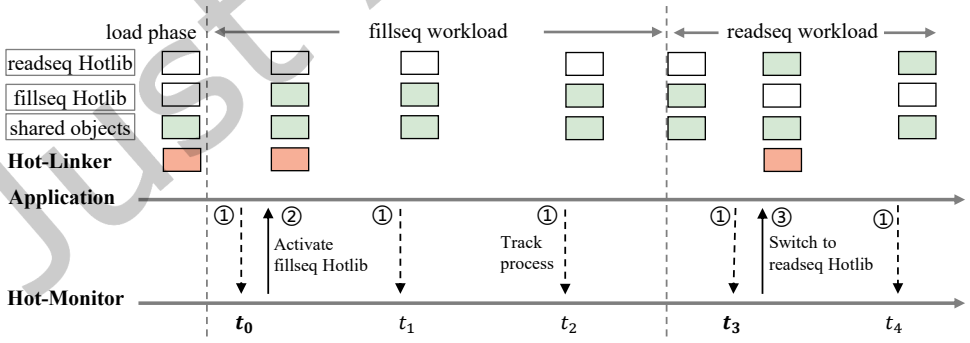

Fig. 6. **HotLD Workflow Example.** RocksDB executes fillseq followed by readseq. During the loading phase, the shared libraries, Hot-Linker, and HotLibs are loaded into the process address space, though HotLibs remain inactive. At **t0**, the Hot-monitor briefly observes program execution and notifies the Hot-Linker to activate the fillseq HotLib. The Hot-monitor then periodically checks execution to determine if a HotLib switch is needed. At **t3**, it instructs the Hot-Linker to switch to the readseq HotLib. The original shared libraries remain active throughout to support access to cold code.

Hot-Monitor is the runtime scheduling engine of HotLD. It begins by using *perf* to sample function-level call frequency and timing data, capturing and analyzing the workload features (Fig. 6 ①), then matches them against pre-collected profiles to identify the most suitable HotLib (Fig. 6 ②). If a better match is found, it signals the in-process Hot-Linker to switch to the new HotLib (Fig. 6 ③). In addition, to support diverse application scenarios, Hot-Monitor provides flexible configuration options, enabling users to adjust parameters such as monitoring frequency and duration based on specific needs.

3.4.1 Workload Feature Design. Hot-Monitor periodically collects workload features of the target application. We observe that various workloads can be effectively distinguished by analyzing their function call patterns and execution time distributions. Based on this insight, Hot-Monitor derives two key features from the collected profiling data:

- **Function Call Spectrum (FCS):** FCS captures the set of functions invoked during program execution and characterizes workloads based on function usage patterns (e.g., scientific workloads frequently call numerical routines, while network services rely on encryption functions). It is defined as a binary vector:

$$F = (f_1, f_2, \ldots, f_u), \quad f_i = \begin{cases} 1, & \text{if function } i \text{ is invoked,} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

  where u denotes the total number of library functions.
- **Function Time Distribution (FTD):** FTD quantifies the proportion of execution time consumed by each function, reflecting workload-specific behavior that can impact optimization. For instance, a sorting function may account for 70% of execution time in workload A but only 5% in workload B. FTD is represented as a normalized vector:

$$A = \{\alpha_1, \alpha_2, \ldots, \alpha_u\} \tag{6}$$

  where α_i is the time share of function f_i.

3.4.2 HotLib Selection. Hot-Monitor identifies the optimal HotLib candidate by analyzing the similarity between the current runtime workload features and pre-collected workload profiles using a two-phase process. In the first phase, Hot-Monitor extracts the function call set from collected performance data and computes the similarity between the current function call spectrum F_{cur} and those of each candidate HotLib's corresponding workload. If a clear best match is found, it is selected immediately to avoid extra computational overhead.

If no definitive candidate emerges, the process advances to the second phase, where Hot-Monitor collects function time distribution data and jointly evaluates function call patterns and execution time distributions to determine the best matching HotLib. This selection can be formalized as follows:

$$i^* = \arg\max_i \left[\alpha \cos(\mathbf{F}_{\text{cur}}, \mathbf{F}_i) + (1-\alpha)\beta \cos(\mathbf{A}_{\text{cur}}, \mathbf{A}_i)\right] \tag{7}$$

When Hot-Monitor identifies a HotLib better suited to the current workload, it sends control signals to Hot-Linker to activate the designated HotLib and complete the execution flow transition. If the application's execution pattern changes substantially and no existing HotLib matches, Hot-Monitor issues a fallback signal to revert the process to the original shared libraries, preventing unmatched optimized code execution. These mechanisms ensure the correctness and robustness of HotLib switching, allowing HotLD to dynamically adapt optimization strategies to workload variations and thus maximize system performance.

4 Evaluation

This section aims to address the following research questions:

- What is the effectiveness of HotLD in performance improvement?
- To what extent does HotLD induce performance overhead for monitoring and switching?

- Does switching execution flow between HotLibs and original libraries introduce extra overhead?
- How do HotLD configurations affect the accuracy of workload matching?

The experiments are conducted on a dual-socket Dell PowerEdge 7920 server equipped with two Intel Xeon Gold 6230R processors (x86-64). Each processor socket comprises 26 physical cores, supporting 52 hardware threads, resulting in a total of 52 cores and 104 threads, with a base clock frequency of 2.1 GHz. Each core is equipped with a 32 KiB L1 instruction cache (L1i), a 32 KiB L1 data cache (L1d), and a 1 MiB L2 cache. The two sockets share a 71.5 MiB L3 cache and are configured with 128 GiB of memory. Additionally, each core includes 64 L1 data TLB entries supporting 4 KiB pages and 1,536 L2 TLB entries supporting 4 KiB pages. The operating system used is ubuntu 22.04, running kernel version 6.8.0-51-generic.

4.1 Performance Benefits

4.1.1 Experimental Setup. To evaluate HotLD's performance improvement, we selected six representative applications covering various practical scenarios. The applications, their versions, tools, and workload configurations are summarized in Table 1. For each application, all its workloads were executed sequentially to simulate dynamic workload scenarios. All applications were compiled using their default optimization options to ensure consistency between the testing results and real-world deployment environments. We compared HotLD with three alternative methods:

Table 1. Benchmarked applications, tools, and workloads.

Application	Version	Benchmark Tool / Workloads
RocksDB [46, 47]	9.0.0	*db_bench*: fillseq, fillrandom, readseq, readrandom, overwrite, updaterandom, readrandomwriterandom (read/write ratios: 90%, 50%, 10%)
MARISA-Trie [48]	0.2.6	Modified official test program: build (trie construction), lookup, reverse lookup, prefix search
Python	3.10.16	*pybench2 [49]*: Pi computation, iterative Fibonacci (fib_iter), string concatenation (string_conca)
OpenSSL [50]	3.0.0	*openssl speed*: SHA-256 (hash), AES-128-GCM (block cipher)
Apache	2.4.56	*ab*: keep-alive (long-lived) vs non-keep-alive (short-lived) connections
MySQL	8.1.0	*Sysbench*: read-write mixed (70% reads), read-only, write-only

- **Original:** baseline method using the unoptimized shared library.
- **BOLT-Average:** BOLT optimization based on aggregated profiling data of all workloads.
- **BOLT-Ideal:** BOLT optimization using per-workload profiling data.
- **HotLD-NR:** HotLibs dynamically selected at runtime, without indirect call rewriting.
- **HotLD:** HotLibs dynamically selected at runtime, with indirect call rewriting.

Note that the performance of BOLT-Ideal is theoretical. It does not actually handle dynamic workloads. In our evaluation, BOLT is applied only to shared libraries to align with HotLD's scope. Notably, BOLT can be used to optimize the main program, while HotLD focuses on library, and the two can coexist and complement each other.

In this evaluation, all results were normalized relative to the **Original** baseline to emphasize the optimization effects. During the testing process, HotLD sampled the target processes every 20 seconds, with each sampling period lasting 1 second. HotLD will activate one HotLib when necessary, and we evaluate performance for 30 seconds after the activation. All results are reported as the average of 20 independent runs, with error bars indicating the standard deviation to reflect result stability and reliability.

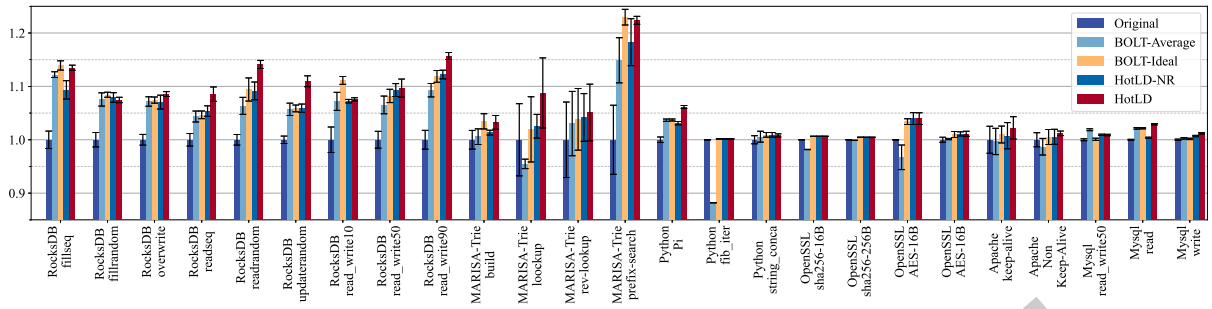

Fig. 7. Performance of HotLD (red bars) compared to BOLT optimized with aggregated profiling data (light blue bars) and workload-specific profiling data (orange bars). All results are normalized to the original non-PGO binaries (dark blue bars) as the baseline.

Table 2. Application characterization data

	.text section of libs(MiB)	avg hot functions	avg indirect call per function	num of HotLibs	memrory overhead(MiB)	
					HotLD	BOLT
RocksDB	8.24	670	19.31	9	9.52	20.82
MARISA-Trie	1.08	43	8.02	4	0.13	10.46
Python	2.02	408	7.87	3	0.95	7.96
OpenSSL	2.54	92	0.85	2	0.132	14.61
Apache	0.11	269	1.63	2	0.137	8.56
MySQL	0.84	256	1.17	3	0.229	13.30

4.1.2 HotLD Result Analysis. As shown in Fig. 7, HotLD yields significant performance improvements in the RocksDB and MARISA-Trie scenarios: the read_write90 workload in RocksDB achieves a 15.69% increase, while the prefix-search workload in MARISA-Trie reaches a maximum gain of 22.37%. In contrast, Pi computation in the Python interpreter and the AES-128-GCM algorithm in OpenSSL show performance improvements of 6.14% and 3.97%, respectively, whereas Apache HTTP Server and MySQL achieve only 1%–3% gains under their respective workloads. Overall, HotLD is most effective for memory-intensive applications or those with pronounced hot code regions, while its impact on lightweight or compute-intensive applications is limited.

To explore the relationship between application characteristics and HotLD effectiveness, we further analyzed the characteristics of shared libraries frequently used by the applications. As shown in Table 2, RocksDB and MARISA-Trie contain a substantial number of dynamic indirect references. Their read, write, and query workloads exhibit significant instruction and data contention, creating greater optimization opportunities for HotLD. In contrast, compute-intensive applications such as OpenSSL rely on fewer hot library functions and dynamic indirect references, which constrains the optimization potential of HotLD. For example, for the SHA-256 workload, HotLD reduces TLB miss events by 44.95%. However, since these events constitute only 0.0034% of the total clock cycles, their impact on overall performance is negligible. The performance discrepancies of Python across the Pi computation, fib_itr, and string_conca workloads further support this theory. In contrast to the Pi computation workload, both the fib_itr and string_conca require more memory for storing intermediate results, thereby diminishing the impact of instruction-level optimizations on overall performance. Unlike the aforementioned applications, Apache and MySQL have smaller hot library sizes and relatively fewer indirect calls, with their

performance bottlenecks primarily stemming from request handling and data access logic. As a result, the impact of HotLD on these applications is limited.

Furthermore, the study reveals an intriguing observation: for the two algorithms in OpenSSL, code layout optimization is more effective when processing 16-byte data blocks compared to 256-byte data blocks. Specifically, the AES-128-GCM algorithm achieves a 3.97% performance improvement with 16-byte blocks, while the improvement decreases to only 1.12% with 256-byte blocks. We hypothesize that this disparity arises because smaller data blocks require more frequent initiation and termination of computation, increasing the cost of context switching. In contrast, larger data blocks can more efficiently utilize CPU caches and parallel processing resources.

4.1.3 Comparison between HotLD and BOLT. **Performance.** The result shows that HotLD outperforms BOLT-Average across nearly all workloads, and performs on par with or better than BOLT-Ideal. Notable gains appear in RocksDB's readrandom, updaterandom, and read_write, MARISA-Trie's lookup, Python's Pi calculation, and OpenSSL's AES-128-GCM. These performance improvements stem from HotLD's global optimization strategy, which rewrites indirect calls into direct ones, enabling HotLib to achieve performance close to static linking when invoking library functions.

On the other hand, BOLT-Average applies aggregated optimizations across all workloads. While this enables reuse of optimized libraries, conflicting workload needs can lead to performance regressions. For example, in Python's fib_iter workload, the L1 i-TLB miss rate increases by 1–2× after optimization, degrading cache performance. These results indicate that aggregated optimization may lead to negative optimization in some cases.

Memory Usage. HotLD demonstrates superior memory efficiency. When loading HotLibs of 9 workloads in RocksDB, HotLD consumes only 9.53 MiB of memory, which is less than half of the memory required by BOLT for one workload. In MARISA-Trie, loading 4 HotLibs uses just 0.13 MiB, compared to BOLT's 10.46 MiB for one workload. This efficiency comes from HotLD optimizing only hot functions, which typically comprise a small portion of the application—3.6% in RocksDB and 0.83% in MARISA-Trie, while BOLT maintain a full copy of the shared libraries.

4.1.4 Impacts of Micro-Architectural Metrics. We used perf to analyze micro-architectural metrics and understand the reasons behind HotLD's performance gains. Figure 8 shows normalized results for RocksDB under different workloads. The evaluated metrics include: 1) **L1-iTLB-load-misses:** L1-iTLB misses but hits in the L2 TLB; 2) **TLB-load-misses:** TLB-load misses that trigger page table walks; 3) **L1-icache-load-misses, L2-load-misses, LLC-load-misses:** Cache misses at L1, L2, and LLC levels; and 4) **Branch-misses:** Branch prediction failures.

HotLD demonstrates significant improvements in front-end performance, including TLB behavior, cache utilization, and branch prediction. The reduction of over 90% in L1-iTLB-load-misses is primarily due to basic block reordering, which compresses the memory distribution of hot code and increases TLB hit rates. The effect on overall TLB-load-misses is limited, and BOLT-Average even increases misses in the *readseq* workload due to cross-workload optimization conflicts. In the cache hierarchy, HotLD reduces L1-icache-load-misses by 17.56%–25.7% and significantly lowers L2-load-misses in the *fillseq* and *read_write* workloads, while LLC-load-misses improvement is mainly seen in *fillseq*, indicating that optimization impacts differ across cache levels. For branch prediction, HotLD achieves the lowest Branch-misses rate, even surpassing BOLT-Ideal, by eliminating trampoline code in dynamic library calls: indirect calls are rewritten as direct calls, reducing BTB entry usage and conflicts.

4.1.5 Multi-Tenancy and Scalability Analysis. To evaluate the optimization and scalability of HotLD in a multi-tenant environment, we conducted experiments with concurrent applications under two deployment strategies: (1) multiple instances of the same application and (2) heterogeneous applications, including memory-constrained workloads. The setup included three RocksDB instances with the *fillseq* workload (write-intensive services),

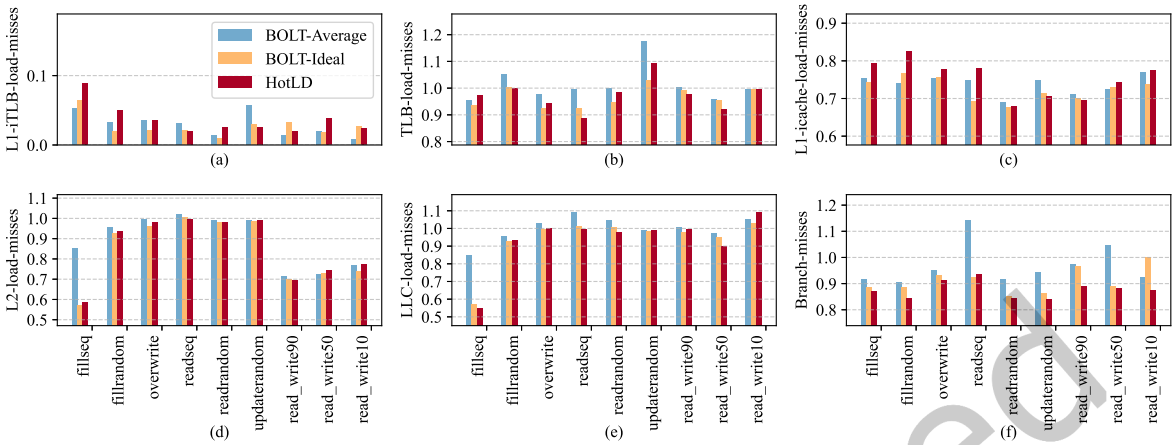

Fig. 8. The Impact of Different Methods on Micro-Metrics of RocksDB (Lower is Better). All results are normalized to the original non-PGO binaries as the baseline.

Table 3. Performance Gain (%) and Memory Usage of HotLD in Multi-Tenant Scenarios

	MARISA-Trie build1	MARISA-Trie build2	RocksDB fillseq1	RocksDB fillseq2	RocksDB fillseq3	MySQL read_write	MySQL read_only	MySQL write_only
Perf. Gain (%)								
HotLD Alone	3.15	3.15	11.89	11.89	11.89	0.80	2.82	1.18
HotLD Conc.	3.07	1.94	10.00	10.76	10.83	0.63	1.22	1.63
Max RSS (MiB)								
Original	1432.54	1429.60	140.74	145.38	143.31	469.11	473.92	488.41
HotLD	1432.16	1434.45	153.02	148.83	143.53	475.97	473.07	486.07
HotLib	0.03	0.03	0.76	0.76	0.76	0.22	0.22	0.22

Note: Perf. Gain is reported as a ratio relative to the baseline (higher is better). Max RSS indicates the peak resident memory in megabytes (MiB). HotLib represents the memory overhead of loaded HotLibs.

two MARISA-Trie instances with *marisa build* (memory-intensive indexing), and three MySQL instances with read_only, write_only, and read_write workloads (diverse transactions). All applications were executed concurrently in the HotLD-enabled environment, and their performance and memory usage were compared with the baseline, as summarized in Table 3.

In single-application execution, RocksDB achieved the highest performance gain of approximately 11.9%, while MARISA-Trie and MySQL gained 0.8%–3.0%. Under concurrent execution, performance gains decreased slightly (e.g., RocksDB: 10.0%–10.8%, MySQL read_only: 1.22%), indicating modest impact from resource contention, yet all workloads maintained positive improvement. Memory overhead was minimal: peak resident memory (Max RSS) changed negligibly, with a maximum increase below 13 MiB, and HotLib modules added only 0.2–0.76 MiB per instance, confirming that HotLD is both effective and memory-efficient in multi-tenant scenarios.

Table 4. Runtime overhead of HotLD.

Application	workload	load HotLib time	switc Hotlib time	code pointer sites	Hot-monitor select HotLib time
RocksDB	overwrite	2.87ms	0.559ms	38007	0.363s
	readrandom	1.62ms	0.505ms	33900	0.353s
MARISA-Trie	lookup	0.705ms	0.119ms	846	0.195s
	prefix_search	0.751ms	0.132ms	1076	0.187s
Python3.10	pi	1.645ms	1.141ms	13982	0.159s
	fib_iter	1.345ms	1.214ms	10817	0.162s
OpenSSL	sha256	0.612ms	0.167ms	1419	0.172s
	AES-128-GCM	0.651ms	0.189ms	1792	0.169s

Summary

The results show that HotLD outperforms BOLT in dynamic workloads, achieving significant memory savings, particularly in scenarios with fewer hot library functions. Moreover, it demonstrates significant performance improvements in workloads characterized by high instruction and data contention.

4.2 Performance Overhead of HotLD

4.2.1 Experimental Setup. In addressing the second research question, we evaluate the runtime overhead introduced by HotLD. For each application, two representative workloads were selected. In each test run, HotLD loads the corresponding HotLib at program startup. Once the application reaches a steady execution state, Hot-Monitor performs dynamic sampling and sends an activation signal to Hot-Linker to switch the control flow to the target HotLib. Each workload is executed 20 times, and table 4 reports the maximum observed value to reflect the worst-case overhead.

4.2.2 Results and Analysis. **Load Time Overhead.** Initially, HotLD incurs additional time during program startup to load HotLibs into the process space. Table 4 presents the time taken to load the HotLibs during the execution of the overwrite and readrandom workloads. Compared to the standard startup method, HotLD's loading time increases by only 2.87 milliseconds, which requires patching 38,007 code pointers. This overhead is acceptable, as it is one-time effort.

Runtime Overhead. HotLD performs a 1-second sample every 20 seconds, which introduces a performance fluctuation ranging from 1% to 5% for monitoring. The performance impact during the HotLib switching process, which involves code pointer rewriting and GOT patching, is negligible, as the process is completed very quickly. Table 4 displays that HotLD only requires millisecond-level time to complete the patching of all code pointers.

Summary

HotLD introduces acceptable time overhead during both loading and execution. Its impact on the overall performance of the application is very limited.

4.3 Control Flow Switching Analysis

4.3.1 Experimental Setup. To address the third research question, we selected RocksDB as the evaluation target due to its diverse workload and broad coverage of various scenarios. We used two typical workloads, fillseq and

readseq, to observe the program execution flow distribution after enabling HotLD. In each run, the application first uses the fillseq workload to write 50,000,000 keys, then switches to the readseq workload for read operations. Hot-Monitor samples once every 20 seconds, with each sampling lasting 1 second (we will evaluate the sampling time in Section 4.4). The experiment was repeated ten times and Fig. 9 shows the detailed behavior process of one run.

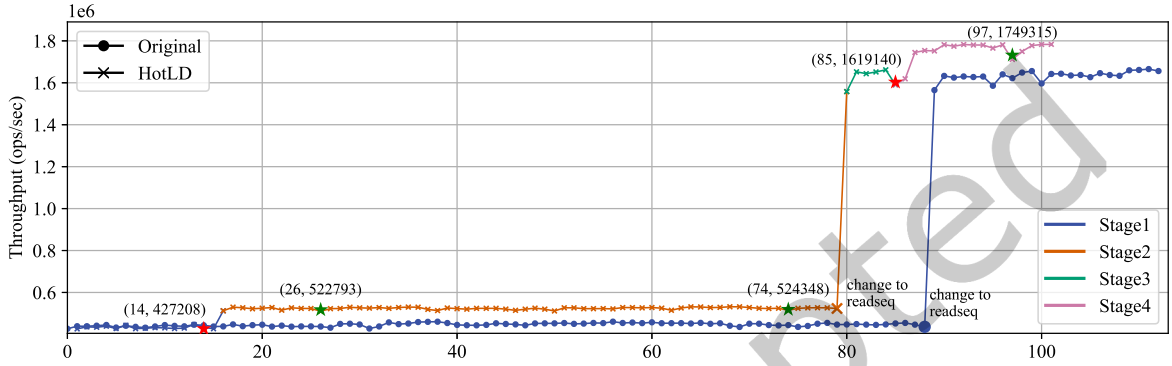

Fig. 9. **Performance Comparison of RocksDB With and Without HotLD Enabled (Higher is Better).** RocksDB executes fillseq followed by readseq. Stage 1: HotLibs loaded but inactive. Stage 2: fillseq HotLib activated. Stage 3: workload switched to readseq, but still using fillseq HotLib. Stage 4: readseq HotLib activated.

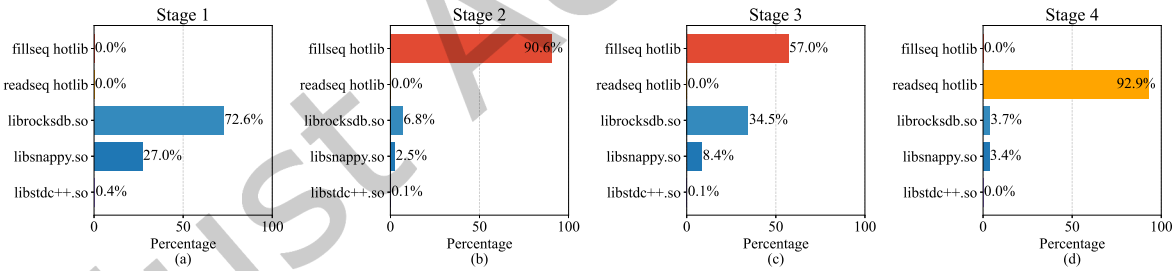

Fig. 10. Code access ratios between HotLibs and original shared libraries across stages.

4.3.2 Results and Analysis. As shown in Fig. 9 and Fig. 10, after enabling HotLD, the application execution process can be divided into four stages.

- **Stage 1 (0 to 14 s):** During this stage, although HotLibs have been loaded into the process space, they remain inactive. The application continues to use the original unoptimized libraries, showing baseline performance.
- **Stage 2 (15s to 79s):** At 14s, the Hot-Monitor samples and identifies the current load as fillseq, successfully matches, and activates the corresponding HotLib, resulting in approximately 14% performance gains.

- **Stage 3 (80s to 85s):** The workload switches to readseq, but Hot-Monitor has not yet completed identification. Around 57% of library function calls go to the fillseq HotLib, while 43% fall back to the original libraries, resulting in a slight overall performance improvement (approximately 1% to 2%).
- **Stage 4 (86s to 101s):** Hot-Monitor successfully matches and activates the readseq HotLib, with about 92.9% of library function accesses redirected to this HotLib, achieving a approximately 7.5% performance improvement.

Summary

Experimental results show that falling back to the original shared libraries does not introduce additional performance overhead. In the worst case, system performance returns to the original baseline level.

4.4 Accuracy of Workload Matching

4.4.1 Experiment Setup. Five typical RocksDB workloads (fillseq, fillrandom, overwrite, readrandom, read_write90) were used to evaluate HotLib selection accuracy. For each run, all five corresponding HotLibs were loaded, but only one workload was executed. Each workload was repeated five times. Sampling was performed every 10 seconds for 1 second, and the HotLib most similar to the sampled performance profile was then selected. We also compared the impact of different sampling durations on the similarity metrics.

Table 5. HotLib Selection Accuracy and Performance Speedup

Metric	fillseq	fillrandom	overwrite	readrandom	read_write90	Average
FCS Accuracy	28/36 (77.8%)	20/40 (50%)	23/40 (57.5%)	26/37 (70.27%)	30/40 (75%)	127/193 (65.8%)
FTD Accuracy	36/36 (100%)	32/40 (80%)	22/40 (55%)	35/37 (94.6%)	37/40 (92.5%)	162/193 (83.9%)
Total Accuracy	36/36 (100%)	35/40 (87.5%)	22/40 (55%)	28/37 (75.68%)	38/40 (95%)	159/193 (82.38%)
Speedup (Dynamic)	11.68%	6.84%	8.11%	11.45%	13.78%	10.37%
Speedup (Matched)	11.89%	6.93%	7.89%	13.26%	13.56%	10.71%

4.4.2 Results and Analysis. Table 5 demonstrates that our HotLib selection mechanism achieves high accuracy on most workloads, with an average of 82.38%. Even for the overwrite workload, which has the lowest accuracy (55%), performance improvement is still close to that of the correctly matched HotLib. Similarity analysis indicates that overwrite and fillrandom share highly similar runtime behaviors, with average similarity scores of 93.50% and 93.08%, respectively, suggesting considerable overlap in hot paths. As a result, even mismatches often lead to the selection of HotLibs with similar performance, demonstrating the robustness of our method under mixed workloads.

4.4.3 Effect of sampling time on similarity. Figure 11 shows the similarity of five workloads between dynamic sampling and offline collected data. Even with a short sampling time of 0.10 seconds, the similarity remains high—over 85% in the worst case (read_write10). As sampling time increases, similarity stabilizes. At 1 second, Hot-Monitor can accurately identify nearly all workloads. Therefore, Hot-Monitor adopts 1 second as the default. This confirms that short-duration sampling effectively captures consistent workload characteristics, offering strong practical value.

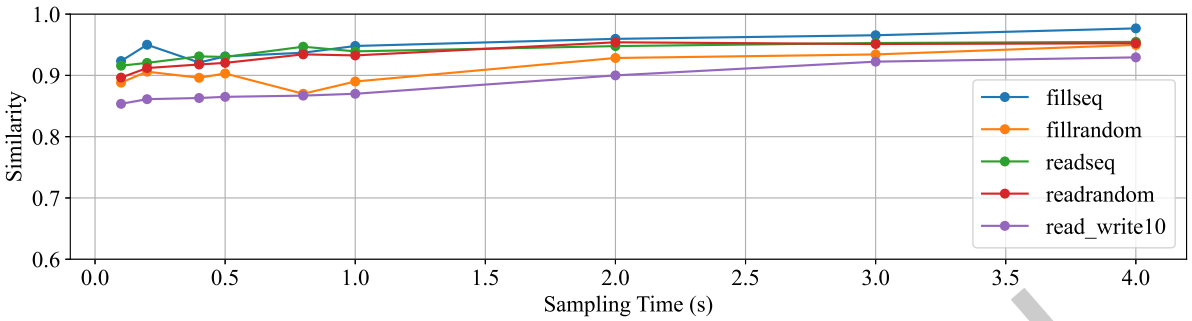

Fig. 11. Impact of Sampling Time on Performance Data Similarity.

Summary

HotLD achieves efficient and robust workload matching by leveraging both function call spectrum and function time distribution similarities, ensuring accurate HotLib selection even under short-term sampling.

5 Discussion

5.1 Dynamic Addition and Removal of HotLib

In the current HotLD architecture, the Hot-Linker loads all HotLibs designed for typical workloads into the target process space, while the Hot-Monitor dynamically selects and activates the appropriate optimizations according to the runtime workload. However, during execution, new workloads may arise, and existing HotLibs may fail to optimize them effectively. Thus, extending the capabilities of the Hot-Linker and Hot-Generator to generate and dynamically integrate new HotLibs tailored to emerging workloads is a necessary enhancement.

Specifically, when the Hot-Generator detects a low similarity between the current workload and existing HotLibs, it can invoke the perf tool to conduct long-term sampling and collect profiling data. Using this data, Hot-Generator can generate a HotLib tailored to the current workload, then Hot-Monitor signals Hot-Linker to map it into the address space of current process. The target process only needs to insert and activate the new HotLib through the Hot-Linker without pausing execution. Additionally, when a user decides to unload a specific HotLib, the Hot-Monitor also can signals the Hot-Linker to remove it.

5.2 Workload Monitoring

The effectiveness of HotLD in maximizing the performance benefits of a target process largely depends on Hot-Monitor's ability to accurately match the current workload with the most suitable HotLib. To this end, Hot-Monitor provides a general matching strategy that compares the similarity between the profiling data used to generate HotLibs and the runtime profiling data to select the optimal HotLib.

In practice, a range of techniques can enhance the accuracy of workload identification [51, 52]. For instance, log analysis, which is widely used in system reliability assurance, including log mining and anomaly detection [53, 54], can also aid workload identification. Additionally, real-time sampling and statistical analysis are extensively researched and applied, often incorporating machine learning algorithms or data mining techniques for automatic classification and workload pattern recognition [55]. These approaches can be used independently and easily integrated into the HotLD framework to further improve workload identification accuracy and efficiency.

5.3 Differences in Dynamic Optimization Between Executable and Shared Libraries

The most closely related work to HotLD is OCOLOS [26]; but, the two differ fundamentally in application scenarios and technical challenges. OCOLOS optimizes individual executables by modifying code segments and the stack to redirect control flow, without considering inter-process code sharing. In contrast, HotLD performs global optimization across multiple shared libraries by updating GOT entries to redirect function calls to optimized code. Thus, the two approaches are complementary in their optimization targets and mechanisms.

HotLD is mainly designed for optimizing shared libraries and performs well in applications where core execution logic resides in these libraries. Its effectiveness may decrease when key logic is embedded in the executable. To broaden HotLD's applicability, two strategies can be considered. First, applications can be compiled with the *–enable-shared* option, enabling the migration of core logic from the executable to shared libraries. Second, lightweight refactoring can be applied to convert the original *main* function into a regular function and compile the executable as a shared library. A new external *main* function can then be introduced to invoke the original entry point. This approach retains the original program semantics while enhancing compatibility with HotLD.

6 Related Work

6.1 Dynamic Indirect Reference Optimization

In dynamically linked programs, indirect references are typically introduced through mechanisms such as the Global Offset Table (GOT) and the Procedure Linkage Table (PLT). Although these mechanisms facilitate modularity and dynamic loading, they introduce additional instruction overhead and indirect jump costs. Many studies have focused on mitigating these overheads to enhance program execution efficiency.

Huang and Lilja [56] proposed a basic block reuse technique that treats PLT trampoline code as an independent basic block. After the initial execution, its input and output are recorded, allowing direct reuse of the result in subsequent calls without re-executing the trampoline code. Kistler and Franz [57] investigated the physical layout optimization of PLT/GOT entries by placing frequently accessed entries in close proximity, thereby reducing CPU cache misses and improving data locality. Nevertheless, this approach remains dependent on the PLT mechanism and fails to completely eliminate additional jump overhead. As a result, its performance benefits are relatively limited in scenarios involving high-frequency function calls.

Varun Agrawal et al. [15] proposed an optimization strategy leveraging modern processor branch predictors. Their approach utilizes the existing structure of the branch target buffer to store the actual addresses of library functions directly in table entries, rather than the PLT trampoline addresses. As a result, the branch predictor can directly determine the target addresses of library functions, circumventing the execution of PLT code. While this method effectively reduces PLT access latency, its effectiveness is contingent on the implementation of the CPU's branch predictor. Ren [6] and Chris Porter et al. [7] adopted a software-level optimization approach. During the program loading phase, they employed the dynamic loader to modify the target addresses of indirect reference instructions, redirecting them to the actual function address instead of the PLT trampoline addresses. This method eliminates PLT jump overhead at runtime. However, because this method directly modifies the shared library's code segment, it compromises the library's shareability, triggering the Copy-on-Write mechanism and incurring memory overhead.

Beyond optimizations targeting the PLT/GOT, several studies have explored reducing the overhead of indirect references through compiler and runtime modifications. ShortCut [58] modifies call instructions in the compiler and leverages a hardware table to allow object accesses to jump directly to the correct handler, or even simplify to a single load or store, thereby reducing redundant branches and dispatch overhead. Reusable Inline Caching (RIC) [59] enhances the V8 inline caching mechanism by storing type information collected during execution in a context-independent manner and reusing it in subsequent runs, avoiding repeated runtime checks and reducing

startup latency. Software Multiplexing [60] leverages the compiler to package specific programs along with their dependent shared libraries into a single binary, thereby reducing overall storage and memory usage while eliminating indirect references.

6.2 Profile-Guided Optimization

Profile-Guided Optimization (PGO) reorganizes code based on a program's actual execution characteristics to enhance performance. It improves execution efficiency by making hot code paths more compact, reducing instruction cache misses, enhancing branch prediction accuracy. PGO can be classified into two categories based on how profiling data is collected and applied: offline and online optimization. Offline optimization is performed during program development and deployment, while online optimization dynamically adjusts code layout at runtime to accommodate evolving execution patterns.

6.2.1 Offline Optimization Methods. Offline optimization incorporates collected profiling data into different stages of the compilation toolchain to improve performance. This data can be utilized during compilation, linking, or post-linking to influence the final binary layout.

- Compilation-Time Optimization: Compilation-time optimization techniques, such as AutoFDO [17], LLVM PGO [45], and GCC PGO [28], leverage profiling data during code generation to guide compiler decisions. For example, AutoFDO identifies program hot spots from sampled profiling data and directs the compiler to apply function inlining, loop unrolling, basic block reordering, and branch prediction optimizations.
- Link-Time Optimization: Link-time optimization techniques, including Propeller [18], LIPO [22], and HFSort [23], restructure code layouts during the linking stage. Propeller integrates static profiling data with sampled profiling data to reorganize functions and basic blocks at link time, enhancing instruction cache utilization. LIPO enables cross-module PGO, improving the code layout of large applications. HFSort arranges frequently executed functions close to one another, increasing instruction cache hit rates.
- Post-Link Optimization: Post-link optimization operates directly on binary files, eliminating the need for source recompilation. Representative tools include BOLT [19, 20] and Ispike [61], which restructure code segments using disassembly and dynamic instrumentation. For instance, BOLT significantly enhances CPU instruction cache hit rates by utilizing binary rewriting to reorganize functions and basic blocks.

6.2.2 Online Optimization Methods. Online optimization methods differ from offline techniques by dynamically adjusting the code layout during program execution to accommodate variations in application execution characteristics. These methods typically rely on Dynamic Binary Translation (DBT) or real-time profiling data for optimization.

- Dynamic Binary Translation Tools: Dynamic Binary Translation tools, such as Dynamo [62], DynamoRIO [63], and StarDBT [64], generate and modify code during program execution. However, these tools often introduce significant runtime overhead, which can degrade performance. Dynamo, an early dynamic optimization tool, improves execution efficiency by dynamically translating and optimizing hot code. In contrast, DynamoRIO provides a more flexible framework for dynamic instrumentation and optimization.
- Continuous Optimization Frameworks: Recent advancements have led to the development of continuous online optimization methods, such as OCOLOS [26], DCM [25], and JACO [27], which enable ongoing optimization during program execution. OCOLOS, built on the BOLT framework, continuously collects and analyzes profiling data to ensure that the code layout remains aligned with the application's execution patterns. However, this method requires application pauses during optimization to update code and switch control flow, potentially affecting user experience. DCM introduces a function reordering technique in the Java Virtual Machine (JVM), enabling continuous optimization of code layout during Java application execution. In contrast, JACO employs an optimization method that eliminates downtime. By allowing old

and new code blocks to coexist and ensuring transparent code optimization through progressive switching, JACO avoids runtime stalls.

7 Conclusion

This paper presents HotLD, a workload-aware global code layout optimization method for shared libraries. It aims to reduce the performance overhead of dynamic linking and address the challenges of library code layout optimization. HotLD generates optimized code copies tailored to typical workloads and dynamically selects the best version at runtime, significantly improving performance. Experiments show that HotLD delivers up to 22.37% performance gains on real-world applications like RocksDB, MARISA-Trie, Python, and OpenSSL, with only millisecond-level overhead. Compared to state-of-the-art tools like BOLT [19, 20], HotLD significantly reduces memory usage, requiring only 1% to 46% of the memory needed to support 2×-9× workloads. Future work will focus on enabling dynamic addition/removal of HotLibs and integrating advanced load monitoring strategies.

8 Data-Availability Statement

The source code and data are publicly available at https://github.com/Hotld/HotLD.git.

References

[1] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A study of modern Linux API usage and compatibility. In *Proceedings of the Eleventh European Conference on Computer Systems*. DOI: http://dx.doi.org/10.1145/2901318.2901341

[2] Michael Franz. 1997. Dynamic Linking of Software Components. *Computer* 30, 3 (1997), 74–81. DOI: http://dx.doi.org/10.1109/2.573670

[3] W. Wilson Ho and Ronald A. Olsson. 1991. An Approach to Genuine Dynamic Linking. *Softw. Pract. Exp.* 21, 4 (1991), 375–390. DOI: http://dx.doi.org/10.1002/SPE.4380210404

[4] Wei Dong, Chun Chen, Xue Liu, Jiajun Bu, and Yunhao Liu. 2009. Dynamic Linking and Loading in Networked Embedded Systems. In *IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, MASS 2009, 12-15 October 2009, Macau (S.A.R.), China*. IEEE Computer Society, 554–562. DOI: http://dx.doi.org/10.1109/MOBHOC.2009.5336957

[5] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 291–304. DOI: http://dx.doi.org/10.1145/1950365.1950399

[6] Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye, Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, and Xinwei Hu. 2022. From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 649–666.

[7] Charly Castes and Adrien Ghosn. 2023. Dynamic Linkers Are the Narrow Waist of Operating Systems. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems, PLOS 2023, Koblenz, Germany, 23 October 2023*. ACM, 26–33. DOI: http://dx.doi.org/10.1145/3623759.3624548

[8] R. A. Gingell, M. Lee, X. T. Dang, and et al. 1987. Shared Libraries in SunOS. *AUUGN* 8, 5 (1987), 112.

[9] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2003. Transparent Runtime Randomization for Security. In *22nd Symposium on Reliable Distributed Systems (SRDS 2003), 6-8 October 2003, Florence, Italy*. IEEE Computer Society, 260. DOI: http://dx.doi.org/10.1109/RELDIS.2003.1238076

[10] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*. IEEE Computer Society, 339–348. DOI: http://dx.doi.org/10.1109/ACSAC.2006.9

[11] Vivek Iyer, Amit Kanitkar, Partha Dasgupta, and Raghunathan Srinivasan. 2010. Preventing Overflow Attacks by Memory Randomization. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. IEEE Computer Society, 339–347. DOI: http://dx.doi.org/10.1109/ISSRE.2010.22

[12] Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. 2007. VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, Dean M. Tullsen and Brad Calder (Eds.). ACM, 424–435. DOI: http://dx.doi.org/10.1145/1250662.1250715

[13] Pierre Michaud, André Seznec, and Richard Uhlig. 1997. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*, Andrew R. Pleszkun

and Trevor N. Mudge (Eds.). ACM, 292–303. DOI: http://dx.doi.org/10.1145/264107.264211

[14] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998*, Dileep Bhandarkar and Anant Agarwal (Eds.). ACM Press, 307–318. DOI: http://dx.doi.org/10.1145/291069.291067

[15] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. 2015. Architectural Support for Dynamic Linking. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 691–702. DOI: http://dx.doi.org/10.1145/2694344.2694392

[16] Sean Bartell, Will Dietz, and Vikram S. Adve. 2020. Guided linking: dynamic linking without the costs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 145:1–145:29. DOI: http://dx.doi.org/10.1145/3428213

[17] Dehao Chen, Xinliang David Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 12–23. DOI: http://dx.doi.org/10.1145/2854038.2854044

[18] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 617–631. DOI: http://dx.doi.org/10.1145/3575693.3575727

[19] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 2–14. DOI: http://dx.doi.org/10.1109/CGO.2019.8661201

[20] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, Aaron Smith, Delphine Demange, and Rajiv Gupta (Eds.). ACM, 119–130. DOI: http://dx.doi.org/10.1145/3446804.3446843

[21] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile inference revisited. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–24. DOI: http://dx.doi.org/10.1145/3498714

[22] Xinliang David Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight feedback-directed cross-module optimization. In *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli (Eds.). ACM, 53–61. DOI: http://dx.doi.org/10.1145/1772954.1772964

[23] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 233–244. http://dl.acm.org/citation.cfm?id=3049858

[24] Alex Ramírez, Luiz André Barroso, Kourosh Gharachorloo, Robert S. Cohn, Josep Lluís Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. 2001. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001, Göteborg, Sweden, June 30-July 4, 2001*, Per Stenström (Ed.). ACM, 155–164. DOI: http://dx.doi.org/10.1145/379240.379260

[25] Xianglong Huang, Brian T. Lewis, and Kathryn S. McKinley. 2006. Dynamic code management: improving whole program code locality in managed runtimes. In *Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE 2006, Ottawa, Ontario, Canada, June 14-16, 2006*, Hans-Juergen Boehm and David Grove (Eds.). ACM, 133–143. DOI: http://dx.doi.org/10.1145/1134760.1134779

[26] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. 2022. OCOLOS: Online COde Layout OptimizationS. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 530–545. DOI: http://dx.doi.org/10.1109/MICRO56248.2022.00045

[27] Wenhai Lin, Jingchang Qin, Yiquan Chen, Zhen Jin, Jiexiong Xu, Yuzhong Zhang, Shishun Cai, Lirong Fu, Yi Chen, and Wenzhi Chen. 2023. JACO: JAva Code Layout Optimizer Enabling Continuous Optimization without Pausing Application Services. In *IEEE International Conference on Cluster Computing, CLUSTER 2023, Santa Fe, NM, USA, October 31 - Nov. 3, 2023*. IEEE, 295–306. DOI: http://dx.doi.org/10.1109/CLUSTER52292.2023.00032

[28] GNU Project. Year. *GNU Documentation Title.* https://www.gnu.org/doc/ Accessed: 2024-10-20.

[29] The Linux Foundation. 1995. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.* https://refspecs.linuxfoundation.org/elf/elf.pdf Accessed: 2024-10-20.

[30] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 16–27. DOI: http://dx.doi.org/10.1145/93542.93550

[31] Angelica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: static profiling for binary optimization. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. DOI: http://dx.doi.org/10.1145/3485521

[32] Guilherme Ottoni. 2018. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 151–165. DOI: http://dx.doi.org/10.1145/3192366.3192374

[33] Brad Calder, Dirk Grunwald, Michael P. Jones, Donald C. Lindsay, James H. Martin, Michael Mozer, and Benjamin G. Zorn. 1997. Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Trans. Program. Lang. Syst.* 19, 1 (1997), 188–222. DOI: http://dx.doi.org/10.1145/239912.239923

[34] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. 2005. Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction. In *Advances in Computer Systems Architecture, 10th Asia-Pacific Conference, ACSAC 2005, Singapore, October 24-26, 2005, Proceedings (Lecture Notes in Computer Science)*, Thambipillai Srikanthan, Jingling Xue, and Chip-Hong Chang (Eds.), Vol. 3740. Springer, 336–352. DOI: http://dx.doi.org/10.1007/11572961_27

[35] Bhargava Kalla, Nandakishore Santhi, Abdel-Hameed A. Badawy, Gopinath Chennupati, and Stephan J. Eidenbenz. 2017. A Probabilistic Monte Carlo Framework for Branch Prediction. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 651–652. DOI: http://dx.doi.org/10.1109/CLUSTER.2017.29

[36] Yonghua Mao, Junjie Shen, and Xiaolin Gui. 2018. A Study on Deep Belief Net for Branch Prediction. *IEEE Access* 6 (2018), 10779–10786. DOI: http://dx.doi.org/10.1109/ACCESS.2017.2772334

[37] Easwaran Raman and Xinliang David Li. 2022. Learning Branch Probabilities in Compiler from Datacenter Workloads. *CoRR* abs/2202.06728 (2022). arXiv:2202.06728 https://arxiv.org/abs/2202.06728

[38] Stephen Zekany, Daniel Rings, Nathan Harada, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. 2016. CrystalBall: Statically analyzing runtime behavior via deep sequence learning. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 24:1–24:12. DOI: http://dx.doi.org/10.1109/MICRO.2016.7783727

[39] Andy Newell and Sergey Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Trans. Computers* 69, 12 (2020), 1784–1794. DOI: http://dx.doi.org/10.1109/TC.2020.2982888

[40] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K. Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A. Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 816–829. DOI: http://dx.doi.org/10.1145/3466752.3480124

[41] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh-Shahri, Akshitha Sriraman, Niranjan K. Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 742–756. DOI: http://dx.doi.org/10.1145/3470496.3527430

[42] Robert S. Cohn and P. Geoffrey Lowney. 1996. Hot Cold Optimization of Large Windows/NT Applications. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*, Stephen W. Melvin and Steve Beaty (Eds.). ACM/IEEE Computer Society, 80–89. DOI: http://dx.doi.org/10.1109/MICRO.1996.566452

[43] G. Ottoni. 2025. hfsort: a Tool to Sort Hot Functions. GitHub Repository. (2025). Available at https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort (Accessed: March 13, 2025).

[44] Youfeng Wu and James R. Larus. 1994. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, USA, November 30 - December 2, 1994*, Hans Mulder and Matthew K. Farrens (Eds.). ACM / IEEE Computer Society, 1–11. DOI: http://dx.doi.org/10.1109/MICRO.1994.717399

[45] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. DOI: http://dx.doi.org/10.1109/CGO.2004.1281665

[46] Facebook. 2025. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. (2025). https://github.com/facebook/rocksdb.git Accessed: 2025-03-25.

[47] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[48] PyTries. 2025. MARISA-Trie. (2025). https://github.com/pytries/marisa-trie.git Accessed: 2025-03-25.

[49] Lucian Marin. 2025. pybench. (2025). https://github.com/lucianmarin/pybench.git Accessed: 2025-03-25.

[50] OpenSSL Software Foundation. 2025. OpenSSL: The Open Source toolkit for Secure Sockets Layer (SSL) and Transport Layer Security (TLS). (2025). https://www.openssl.org/ Accessed: 2025-03-25.

[51] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. 2016. Workload Characterization: A Survey Revisited. *ACM Comput. Surv.* 48, 3 (2016), 48:1–48:43. DOI: http://dx.doi.org/10.1145/2856127

[52] Roozbeh Aghili, Qiaolin Qin, Heng Li, and Foutse Khomh. 2024. Understanding Web Application Workloads and Their Applications: Systematic Literature Review and Characterization. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 474–486. DOI: http://dx.doi.org/10.1109/ICSME58944.2024.00050

[53] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 61–71. DOI: http://dx.doi.org/10.1109/SANER.2018.8330197

[54] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2022. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 54, 6 (2022), 130:1–130:37. DOI: http://dx.doi.org/10.1145/3460345

[55] Mustafa M. Al-Sayed. 2022. Workload Time Series Cumulative Prediction Mechanism for Cloud Resources Using Neural Machine Translation Technique. *J. Grid Comput.* 20, 2 (2022), 16. DOI: http://dx.doi.org/10.1007/S10723-022-09607-0

[56] Jian Huang and David J. Lilja. 1999. Exploiting Basic Block Value Locality with Block Reuse. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, Orlando, FL, USA, January 9-12, 1999*. IEEE Computer Society, 106–114. DOI: http://dx.doi.org/10.1109/HPCA.1999.744342

[57] Thomas Kistler and Michael Franz. 2003. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.* 25 (2003), 500–548.

[58] Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. 2017. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 494–506. DOI: http://dx.doi.org/10.1145/3079856.3080237

[59] Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable inline caching for JavaScript performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 889–901. DOI: http://dx.doi.org/10.1145/3314221.3314587

[60] Will Dietz and Vikram Adve. 2018. Software multiplexing: share your libraries and statically link them too. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 154 (Oct. 2018), 26 pages. DOI: http://dx.doi.org/10.1145/3276524

[61] Chi-Keung Luk, Robert Muth, Harish Patil, Robert S. Cohn, and P. Geoffrey Lowney. 2004. Ispike: A Post-link Optimizer for the Intel®Itanium®Architecture. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 15–26. DOI: http://dx.doi.org/10.1109/CGO.2004.1281660

[62] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 1–12. DOI: http://dx.doi.org/10.1145/349299.349303

[63] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*, Richard Johnson, Tom Conte, and Wen-mei W. Hwu (Eds.). IEEE Computer Society, 265–275. DOI: http://dx.doi.org/10.1109/CGO.2003.1191551

[64] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Maurício Breternitz Jr., Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC 2007, Seoul, Korea, August 23-25, 2007, Proceedings (Lecture Notes in Computer Science)*, Lynn Choi, Yunheung Paek, and Sangyeun Cho (Eds.), Vol. 4697. Springer, 4–15. DOI: http://dx.doi.org/10.1007/978-3-540-74309-5_3

A Experiment 1: Comparison with Static Linking

A.1 Experiment Setup

To further evaluate the effectiveness of HotLD, we compare its performance with static linking and *"Architectural Support for Dynamic Linking"* (ASDL) [15]. Since ASDL's source code is not publicly available and its implementation requires specific hardware support or major modifications to the system's dynamic loader, direct reproduction is infeasible. Given that ASDL aims to "retain the benefits of dynamic linking while achieving performance comparable to static linking," if HotLD outperforms static linking, it can be inferred that HotLD also surpasses ASDL. As a result, we employ an indirect evaluation methodology, using static linking as the performance baseline. Notably, ASDL's primary design objective is to *"...providing the benefits of dynamic linking with the performance of static linking. [15]"* Therefore, if HotLD demonstrates superior performance compared to static linking, it follows that HotLD would also outperform ASDL under equivalent conditions.

Our experimental evaluation covers two representative applications: RocksDB and OpenSSL. For RocksDB, we test common workloads including *fillseq*, *fillrandom*, *overwrite*, *readseq*, *readrandom*, *updaterandom*, and mixed read-write workloads with ratios of 10%, 50%, and 90%. For OpenSSL, we benchmark *sha256* with 16B and 256B

input sizes, as well as *AES* with 16B and 256B input sizes. Each workload is executed 20 times, and the average speedup is reported relative to the baseline dynamic linking performance.

A.2 Results and Analysis

Table 6. Performance Comparison of Static Linking and HotLD

	RocksDB									OpenSSL			
Method	fillseq	fillrandom	overwrite	readseq	readrandom	updaterandom	read_write10	read_write50	read_write90	sha256-16B	sha256-256B	AES-16B	AES-256B
Static	1.092	1.053	1.063	1.059	1.060	1.074	1.058	1.036	1.064	0.992	0.994	1.032	1.015
HotLD	**1.119**	**1.069**	**1.079**	**1.079**	**1.124**	**1.099**	**1.070**	**1.088**	**1.136**	**1.006**	**1.005**	**1.038**	1.011

Table 6 shows that HotLD outperforms static linking in almost all workloads. For RocksDB, HotLD achieves higher speedups than static linking across all workloads. For OpenSSL, HotLD performs better in *sha256* and *AES-16B*, while the difference is negligible in *AES-256B* (less than 2%). HotLD outperforms static linking due to its ability to not only eliminate indirect library function calls within hot functions, but also optimize the code layout of these functions based on runtime performance profiling, thereby further enhancing execution efficiency.

Interestingly, static linking already provides significant performance improvement for RocksDB. We attribute this to two main reasons: 1) **Function call rewriting:** Hot functions in librocksdb.so contain an average of 19.31 indirect library call, which can be resolved into direct calls under static linking since the target addresses are known at link time. 2) **Compiler optimization differences:** RocksDB is compiled with *-O3*. Under static linking, the compiler may perform additional function reordering, improving instruction locality and hardware utilization.

Summary

Across all workloads of RocksDB and OpenSSL, HotLD consistently achieves performance comparable to or better than static linking, while avoiding the disadvantages of static linking such as loss of library sharing.

Received 2 July 2025; revised 10 September 2025; accepted 14 September 2025