Comparative Analysis of Linking Efficiency

Evaluating `LLD` and `mold` through Insights into Performance Metrics and Architectural Differences in Software Linking Processes

Anna Szymkowiak[1]

Supervisors: Soham Chakraborty[1]**, Dennis Sprokholt**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Anna Szymkowiak
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Burcu Ozkan

Abstract

This study examines the differences between two modern linkers, LLD and mold, focusing on their efficiency during software development. Although the linking process, which combines multiple object files into a single executable, typically occupies a minor fraction of the total compilation time, optimizing it can significantly enhance the overall build efficiency - particularly during the development of large-scale projects. The research aims to determine codebase-level differences between these linkers and assess their performance across various metrics, including linking time, memory usage, and the time spent on different phases of the linking process, using CMake and Bitcoin Core as benchmarks. Furthermore, the study extends to examining the executables produced by both linkers from the HDiffPatch project, comparing their execution times and sizes. The findings consistently show that mold outperforms LLD in terms of speed and efficiency, which results from its comprehensive utilization of parallel processing techniques. Nonetheless, LLD offers broader applicability by supporting a wider range of file formats and being suitable for both embedded and kernel programming. Therefore, the selection of a linker ultimately depends on the specific requirements of the project and the characteristics of the target machine.

1 Introduction

In software development, writing source code is not sufficient by itself; it must be translated into machine code for a computer to execute it. This process requires a compiler [1]. The compilation process primarily involves two main phases: compiling source files into object files and then linking them into a single executable.

Linking is particularly valuable because it enables developers to split the code into multiple files across a project, avoiding the cumbersome and unmanageable nature of placing everything into a single file [2]. Additionally, it allows for incremental compilation, where only modified files are recompiled, and the application is simply relinked, thus speeding up the development process. Although linking time typically represents a minor part of the overall compilation process (less than 1% as demonstrated in the experiment described in Appendix A), it remains an important aspect in the context of software development, during which it occurs repeatedly. Moreover, the linking stage can become particularly time-consuming when managing a large number of object files. Consequently, improving the efficiency of this phase could reduce the overall build time for large projects that require extended compilation time.

There are several well-known linkers for programming languages such as C, C++, and Rust, including GNU ld[1], GNU gold[2], LLVM LLD[3], and mold[4]. Among these, mold has proven to be the fastest, being 23 times faster than ld when linking MySQL 8.3 and four times faster than LLD for Chromium 124 [3]. Additionally, Rui Ueyama, who developed both LLD and mold, has declared that mold is production-ready for userland programs, though it is not yet suitable for kernel or embedded programming [4].

The existing research on software linking predominantly concentrates on specific aspects rather than exploring the core mechanics of linkers. Studies such as those by [5] and [6] explore the integration of static and dynamic linking strategies, aiming to utilize the strengths of each method while mitigating their respective limitations. Additionally, research like [7] has provided valuable insights into the ELF format and developed specialized linkers tailored to its specifications. The work in [8] explores the methods to optimize the performance of shared libraries, seeking a balance between speed and flexibility.

Furthermore, extensive research has been conducted that directly targets the compilation process and compares it across various compilers. For instance, the study in [9] examines the performance of different compilers such as Clang, G++, and Intel C++ Compiler. This research benchmarks the compilers in terms of execution speeds of compiled C/C++ code, that incorporates OpenMP 4.x for parallelization and vectorization, as well as the compilation time for projects with intensive C++ template usage. Similarly, the research in [10] benchmarks the IPS, GNU, and LLVM compilers, focusing on metrics such as executable sizes, and overall build times.

Despite these contributions, there is a notable gap in direct academic research of existing linkers, with no comprehensive analysis aimed at fully understanding and enhancing their performance and examining codebases directly. Consequently, this study aims to fill this gap by scientifically comparing two contemporary linkers.

This research seeks to explore the current state of the linking process, bringing the topic of linking into academic discussions. The goal is to enhance the visibility of linkers in the field of computer science and to contribute to the development of more efficient linking technologies. Despite the critical role that linking plays in software development, it remains a relatively underexplored area in academic research, as evidenced by the disparity in research output. A search conducted on Scopus[5] within the *Computer Science* subject area and limited to publications in English demonstrates a significant difference in the number of results. A query for "compiler" produces approximately 35,000 results, while "assembler" (another substage of the compilation process) returns over 2,100 results. In contrast, a query for "linker" yields around 1,800 results as of June 19, 2024. Importantly, this figure is an overestimate as it includes unrelated results from cheminformatics or molecular sciences, indicating a narrower academic focus on linkers in the context of computer science.

[1] https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html

[2] https://en.wikipedia.org/wiki/Gold_(linker)

[3] https://lld.llvm.org/

[4] https://github.com/rui314/mold

[5] https://www.scopus.com/search/form.uri?display=basic

Thus, this study will delve into a comparative analysis of LLD and mold, examining their architectures and profiling their performance across different projects. The study assesses various metrics, including linking time and memory usage, and compares the executables produced by each linker. LLD was selected for this comparison as it is recognized as the second fastest linker, according to [3].

Specifically, the main research question this paper aims to answer is: **"Why is LLD not as fast as mold?"**. To address this question, we formulated the following sub-questions:

- What does the linking process look like in LLD and in mold and what are the differences?
- What are the differences in architecture between LLD and mold?
- What factors contribute to mold's performance?

The remainder of this paper is structured as follows: Section 2 elaborates on the linking process and Section 3 outlines the methodologies employed to address the research questions. While Section 4 examines the linking processes and architectures of LLD and mold, Section 5 presents the conducted benchmarks and discusses the results. Section 6 addresses the reproducibility of the methods used and Section 7 analyses the findings. Finally, Section 8 concludes the paper and suggests directions for future research.

2 Background

This section provides essential background on the linking process. Subsection 2.1 outlines linking's role within the compilation process and its general steps. Subsection 2.2 covers the standard format for object files, and Subsection 2.3 explains the differences between static and dynamic linking. Finally, Subsection 2.4 discusses the function and purpose of linker scripts.

2.1 The Linking Process

To translate the human-readable code into machine code, it must go through the compilation process [11]. The compilation process involves several distinct steps, as depicted in Figure 1. Initially, the source code undergoes preprocessing to prepare it for the compiler. Such prepared code is then compiled into assembly language, which is subsequently converted by an assembler into object files containing raw machine language instructions. Finally, a linker combines these object files into a single executable program. It is important to note that this process varies with different programming languages. For example, Java code is compiled into Java bytecode [11]. However, this research will focus on the described compilation flow for languages like C and C++.

The purpose of the linking process is to combine all object files into a single executable that can be run by a computer. This process typically unfolds as outlined in [2]:

1. The linker scans the files specified in the command line from left to right:
    - For object files, it enforces the insertion of symbols (names or identifiers of variables, functions, or data structures) into the global set of symbols. If a symbol already exists, it merges them, which may lead to an immediate resolution of the symbol.
    - For archive files, the linker checks whether any of the archive's symbols resolve an undefined symbol already in the set. If so, it resolves the symbol and includes the archive in the final set of files that will form the executable.
    - For dynamic libraries, the linker notes references to symbols within these libraries but does not incorporate the library code into the output executable.
2. The linker performs a check for any remaining undefined symbols. If any are found, it throws an error; otherwise, it proceeds.
3. Merges all code and data sections, except sections from shared libraries.
4. Assigns runtime addresses to all static symbols and updates all symbol references so they point to their correct runtime addresses. References to symbols within dynamic libraries remain as those will be resolved later.
5. Outputs the final executable.

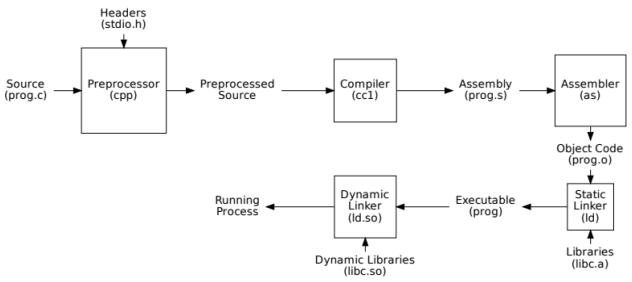

Figure 1: The compiler toolchain [11]

It is important to note that in this flow, which is implemented by traditional linkers as ld, the order in which files are specified as arguments matters. As the linker traverses the files from left to right, if an object file references a symbol in a library that appears earlier in the command line, that symbol will not be resolved.

2.2 The ELF Format

Typically, object files come in various formats depending on the operating system. For instance, Windows utilizes the Portable Executable (PE) format, macOS employs the Mach-O format, while Unix systems use the Executable and Linkable Format (ELF) [2]. This research paper will specifically focus on the ELF format.

An ELF file is generally composed of four main components: the Executable Header, Program Headers (optional), Sections, and Section Headers (optional), as illustrated in Table 1 [12].

The Executable Header indicates that it is an ELF file, specifies the type of ELF file, and points to the locations of other components. Section Headers locate specific sections within the file. While they provide the section view of the

Executable header
Program headers
Sections
Section headers

Table 1: Overview of the ELF structure

file, Program Headers provide a segment view of the file [12]. Lastly, the ELF file includes multiple types of sections, such as `.text`, `.data`, `.symtab`, `.rel.*`, and `.rela.*`.

- `.text` - contains the compiled code of the program.
- `.data` - holds initialized variables.
- `.symtab` - provides a symbol table with symbols that are defined and referenced in the program.
- `.rel.*` / `.rela.*` - contains relocation tables with entries detailing specific addresses where relocation needs to be applied (relocation specifies where the linker must update the location with the actual address of the symbol).

2.3 Static and Dynamic Linking

There are two primary types of linking, as described in [2]: static linking and dynamic linking. Static linking occurs during the compile time and is part of the linking stage to produce an executable. It incorporates static libraries, known as archives (the `.a` file extension). An archive is essentially a collection of object files that are not automatically linked; they only become linked when the linker identifies a reference from other object files being compiled. Conversely, dynamic linking takes place at run time. This method allows some symbols to remain undefined when the program is compiled, with these symbols being resolved only during the run-time of the program [6].

Although static linking is conceptually simpler than dynamic linking, it generally results in larger executables. This is because each program must contain its own copy of any linked libraries. On the other hand, dynamic linking permits the storage of a single copy of a library on disk, which can be shared among multiple programs [6]. By deferring symbol resolution until runtime, dynamic linking not only reduces the size of the executable but also simplifies the process of updating libraries. With dynamic linking, there is no need to relink the entire program when a library is updated. However, dynamic linking can lead to "DLL Hell", where the proper functioning of an executable depends on having the correct versions of libraries [6]. Given these differences, it is important to note that this paper specifically focuses on static linking.

2.4 Linker Scripts

Most linkers incorporate a scripting language that plays a crucial role in specifying the memory layout of the final executable by dictating the allocation of input files' sections in the output file [7]. The linker script language also determines the program's entry point and specifies the regions of memory, including their flags and alignment.

A linker script is always in use; if the user does not provide one explicitly, a default script is employed. The ability to use a linker script provides programmers with significant control over the final executable, a feature that proves particularly beneficial in embedded programming where resources are limited and precise control over memory layout and specific management of various code and data sections are critical due to hardware constraints [13].

3 Methodology

This study conducts a comparative analysis of two prominent linkers, `LLD` and `mold`. It aims to elucidate their architectures and unique features by examining their codebases. Additionally, the study evaluates the differences in memory usage and total linking time across projects of various sizes. Furthermore, we profile specific phases of the linking process such as symbol resolution, relocations handling, and file writing.

This study utilizes the built-in options provided by each linker for performance tracing, specifically `--time-trace` for `LLD` and `--perf` for `mold`. In both cases, the codebase incorporates specific markers to record the start and end points of operations within each phase of the linking process. These options allow for direct, reliable measurement of linker performance metrics without the overhead and complexity introduced by external profiling tools.

Initially, we considered tools like `Valgrind`[6] and `perf`[7] to profile the linking stages. However, we found `Valgrind` to be unsuitable due to its lack of support for multicore systems [14], which is a crucial aspect of `mold`, given its multicore efficiency [15]. The `perf` tool was also initially considered; it supports multicore systems and provides detailed performance metrics. Nevertheless, due to potential inaccuracies in assessing the function-specific purpose and the necessity of manual categorization, we decided not to use `perf`.

Although the linking process is critical, it typically occurs only once, after which the focus shifts to the performance of the actual executables produced. Therefore, it is essential to evaluate not only the efficiency of the linking process but also the characteristics of the final executables. We compare the executables linked by `LLD` and `mold` in terms of execution time, file size, and structural differences such as the number and type of sections they contain.

4 Linking Strategies in `LLD` and `mold`

This section will delve deeper into the nuances of both linkers based on the codebases and accompanying documentation. Subsection 4.1 offers an in-depth examination of `LLD`, while Subsection 4.2 focuses on `mold`. Finally, Subsection 4.3 details the differences between `LLD` and `mold`.

4.1 Examination of Linking Techniques in `LLD`

The linking process in `LLD`, as outlined in Section 2, begins by sequentially parsing input files and resolving symbols immediately. After processing the input files, `LLD` aggregates and

[6] https://valgrind.org/

[7] https://perf.wiki.kernel.org/index.php/Main_Page

merges the sections. The final step involves relocating and writing the output sections to the executable, ensuring that all references and links are correctly aligned, which is performed in parallel. Additionally, LLD performs a check for any remaining undefined symbols during the output phase to ensure the integrity and completeness of the final executable.

Unlike traditional linkers such as ld, LLD permits backward and mutually dependent references between libraries. Specifically, LLD lazily adds symbols from archive files to the symbol table — these symbols are not loaded immediately but are extracted as needed. This method enables LLD to remember all symbols from previously encountered archives, facilitating the resolution of undefined symbols by extracting necessary object files instantly [16].

Internally, LLD utilizes a shared symbol table implemented as a map that links each symbol name to an index. Additionally, all symbols are stored in a separate vector and each file maintains its own array of symbols, which provide details about their definition status and other attributes. When resolving symbols, LLD casts symbols at the same memory location and updates their attributes as needed. It utilizes the `class CachedHashStringRef` to store a precomputed hash for each symbol. The internal data structures are specifically designed to minimize the number of lookups, enabling hash table operations to be executed just once per string [17]. Moreover, LLD leverages C++ templates to reduce code complexity and improve performance. For instance, it utilizes templates to determine system characteristics such as endianness and whether the architecture is 32-bit or 64-bit, subsequently invoking the appropriate functions accordingly.

Regarding architecture, LLD maintains separate directories for each file format (ELF, COFF, Mach-O), sharing minimal code between them. This separation is maintained because adding another layer of abstraction is not deemed beneficial enough to justify the associated cost [16]. Moreover, MinGW linker is implemented as a thin wrapper for lld/COFF. The main function determines the target file format and invokes the appropriate driver. Additionally, LLD maintains a separate file for each target architecture, all of which inherit from the `TargetInfo` class, subsequently declaring virtual functions such as relocation-target-specific.

Performance analyses show that while the non-concurrent version of LLD spends considerable time copying files and performing relocations, the concurrent version significantly reduces time spent on these tasks by parallelizing section copying and relocation [17]. Thus, spending the most time adding symbols to the symbol table. They managed to parallelize the section copying and relocation by assigning non-overlapping file offsets to sections. LLD primarily implements parallelization in scenarios where functions modify only individual input files, thus eliminating the need for locks.

4.2 Optimization and Parallelism in `mold`

The linking approach in `mold` differs significantly from that of LLD in terms of input file handling. `mold` processes files in parallel, contrasting with LLD's sequential processing. Instead of automatically resolving symbols while parsing and adding them to a hashmap as LLD does, `mold` first accumulates all symbols in a concurrent hashmap, and then it resolves the symbols and updates their references through a parallel traversal of the files. After symbol resolution, `mold` proceeds to create output sections. While LLD checks for undefined symbols during the file writing stage, `mold` conducts this check earlier for `SHF_ALLOC` sections, raising an error if undefined symbols are found. The sections are then transferred to the output in parallel, and relocations are applied concurrently. Afterward, `mold` performs an additional check for any undefined symbols in `non-SHF_ALLOC` sections before finalizing the executable output.

Similar to LLD, `mold` supports backward and mutually dependent references [18]. It keeps track of symbols that can be resolved from archive files, which allows `mold` to revisit and retrieve object files from archives as needed to resolve remaining undefined symbols.

`mold` leverages Intel Threading Building Blocks (TBB) for parallel execution and employs several concurrent containers [19]. Key features include `parallel_for_each` and `concurrent_hash_map`. It also utilizes templates with `requires` and `constexpr` for data structures and attributes. The symbol table in `mold` is defined as a concurrent hashmap that maps symbol names directly to symbols, with each file also maintaining a separate list. Unlike LLD, `mold` does not categorize symbols into classes such as `Defined`, `Undefined`, `SharedSymbol`, `CommonSymbol`, or `LazySymbol`. Instead, it employs a unified class for all symbol types.

It is important to note that `mold` utilizes two processes to enhance its efficiency [19]. The main process forks a child process, which carries out the bulk of the linking work. As the child process writes to the filesystem, it signals the main process, allowing `mold` to terminate from the user's perspective, even though the child process continues to finalize tasks. This approach helps `mold` appear to complete its execution earlier than it technically does. Additionally, creating a new file and filling its contents using `mmap` is slower than writing to an existing file in the buffer cache, thus `mold` overwrites an existing executable file if one exists [19].

Notably, `mold` does not support traditional linker scripts. Although linker scripts allow detailed file layout specifications and insertion of arbitrary bytes between sections, most of these functions can be performed by post-link binary editing tools like `objcopy` [19]. However, tasks like mapping input sections to output sections and applying relocations cannot be done post-link. Therefore, `mold` provides only a basic set of linker script features necessary for essential operations, like reading `/usr/lib/x86_64-linux-gnu/libc.so` on Linux [18]. Instead of expanding its support for traditional linker scripts, `mold` aims to simplify the process by introducing simpler alternatives, such as the `--section-order` command line option, that allows specifying addresses for sections and program headers [4].

`mold`'s exceptional speed is attributed to its use of faster algorithms and efficient data structures compared to other linkers, as well as its high degree of parallelism [15]. Furthermore, the decision to abandon support for linker scripts is based on the belief that supporting them complicates the design and slows down the linker [4].

Feature	LLD	mold
Supported Architectures	Supports AArch64, AMDGPU, ARM, Hexagon, LoongArch, MIPS (32/64 little/big-endian), PowerPC, PowerPC64, RISC-V, SPARC V9, x86-32 and x86-64 [20]. Does not support: m68k, s390x, SH-4, and DEC Alpha.	Supports x86-64, i386, ARM64, ARM32, 32/64 little/big-endian RISC-V, 32-bit PowerPC, 64-bit big-endian PowerPC ELFv1, 64-bit little-endian PowerPC ELFv2, s390x, 64-bit/32-bit LoongArch, SPARC64, m68k, SH-4, and DEC Alpha [3]. Does not support AMDGPU, Hexagon, and MIPS (32/64 little/big-endian).
File Formats Supported	Supports ELF (Unix), PE/COFF (Windows), Mach-O (macOS), and WebAssembly, with varying degrees of completeness.	Supports only the ELF format.

Table 2: Comparative analysis of supported architectures and file formats between LLD and mold

4.3 Theoretical comparison between LLD and mold

Although the primary function of both LLD and mold is to assemble object files into an executable, these linkers exhibit significant differences in their capabilities and approaches. Table 2 offers a comparison of LLD and mold, focusing on their support for different target architectures and file formats.

The diagrams in Figure 2 and Figure 3, illustrate simplified and high-level representations of the linking processes in LLD and mold, respectively, with a focus on the parallel or sequential aspect of each step. A key distinction between the two is the extent of parallelization: mold implements extensive parallelism, whereas LLD primarily processes tasks sequentially. LLD processes files sequentially and resolves symbols as they are encountered. This approach ensures immediate resolution of symbols but does not exploit potential parallelism. In contrast, mold adopts a highly parallel approach, parsing files simultaneously and utilizing a concurrent hashmap for symbol addition, which is followed by parallel symbol resolution. Yet, both linkers leverage parallelism when writing the final executable and applying relocations.

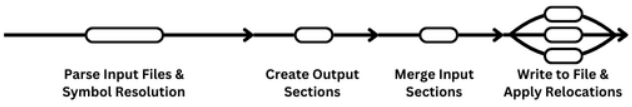

Figure 2: Simplified overview of the linking process in LLD, highlighting parallel or sequential processing

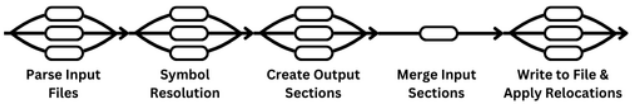

Figure 3: Simplified overview of the linking process in mold, highlighting parallel or sequential processing

Despite differences in processing files, both linkers support backward and mutually dependent references, ensuring that they are not sensitive to the order of input files. Moreover, mold simplifies its architecture by implementing a single class for symbols, whereas LLD employs multiple symbol classes to manage different types of symbols. In terms of file handling, mold categorizes files primarily as SharedFile and ObjectFile, while LLD uses a more diverse classification, including InputFile, SharedFile, BinaryFile, and BitcodeFile.

Regarding section garbage collection, LLD supports this feature through a mark-sweep process that removes unreferenced sections going through the input graph. mold, however, enhances this process by utilizing multiple threads to mark sections concurrently, improving efficiency and speed of identifying and discarding unnecessary sections [19].

A notable performance advantage of mold over LLD lies in its utilization of all available CPU cores during the execution, a feature that LLD does not leverage. This parallelism allows mold to achieve significant reductions in linking time compared to LLD [15].

Finally, a critical difference lies in linker script support. LLD supports traditional linker scripts, allowing for detailed file layout specifications and other advanced configurations [21]. mold, on the other hand, does not support linker scripts, thus it is not yet suitable for kernel or embedded programming [4].

In summary, while LLD offers broad file format and target architecture support, and traditional linker script capabilities, mold excels in speed and efficiency by leveraging parallelism and focusing on ELF format optimization. The choice between the two often depends on the specific needs of the project and the target environment.

5 Performance Comparison of LLD and mold

This section provides a detailed analysis of both linkers, specifically comparing LLD and mold in terms of total execution time in Subsection 5.1, memory usage in Subsection 5.2, and the time spent on different phases of the linking process in Subsection 5.3. Finally, Subsection 5.4 compares the produced executables against different factors. All analyses were conducted on the same machine, equipped with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (6-core/12-threads) and 16.0 GB of RAM with an SSD drive, running Linux Ubuntu 22.04.4 LTS.

The comparison involves benchmarking `Ubuntu LLD 14.0.0` and `mold 2.31.0` against two significant projects: `CMake`[8] and `Bitcoin Core`[9], specifically targeting their largest executables: `cmake` for `CMake` and `bitcoind` for `Bitcoin Core`. The resulting executable sizes for these projects while compiling using default options are 19 MiB and 201 MiB, respectively. We selected these repositories due to their complexity, a big number of object files (1418[10] for `CMake` and 667 for `Bitcoin Core`), and because they compile into a single relatively large executable, which makes them suitable for performance analysis. Additionally, their build processes allow for easy extraction of verbose output necessary to understand how `gcc`[11] invokes the linker, as we invoked each linker directly rather than through `gcc` with the `-fuse-ld=` option enabled.

For the comparison of the output files, the `HDiffPatch`[12] repository was used. We chose this repository primarily because it facilitates easier measurement of program execution time. In this case, invoking the linker directly is not critical, so the `-fuse-ld=` option was used to specify the linker.

5.1 Execution Time Analysis

We used the command `time` to measure the execution time for linking. The initial measurement was discarded as it was significantly higher, likely due to caching effects, and thus considered an outlier. We ran the test five times, and the average real-time execution was calculated. As presented in Figure 4, the linking time for `mold` was consistently faster than for `LLD` for both executables.

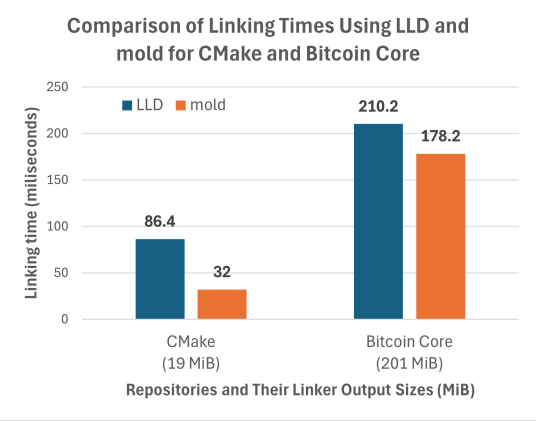

Figure 4: Comparison of linking times between `LLD` and `mold` (lower is better)

Furthermore, having established that `mold` performs faster on relatively large projects, we performed an additional benchmark on a simple `Hello World!` C program with a size of 16 KiB to determine if the multicore capabilities of `mold` also benefit smaller programs. The program was linked 100 times with `LLD` and `mold`, and the average execution times were calculated. The results showed that the average linking time for `LLD` was 15.85 milliseconds, whereas for `mold` it was significantly lower at 4.62 milliseconds, indicating that `mold` provides substantial performance advantages even for minimal programs, likely due to its comprehensive parallelization strategy that effectively manages files, sections, and symbols, thereby optimizing performance across all program sizes.

5.2 Memory Usage Comparison

We also conducted the memory usage analysis for the same repositories using `Valgrind` with the `--tool=massif` option. The initial run was discarded, and data from only the subsequent run was considered. The memory usage peak for both linkers during the linking process is depicted in Figure 5. In the case of `mold`, two processes were run; however, the memory usage of the first process was minimal — 94.4 KiB for `CMake` and 95.7 KiB for `Bitcoin Core` — and thus was considered too small to include in the graph. The results demonstrate that `mold` also excels in terms of memory efficiency.

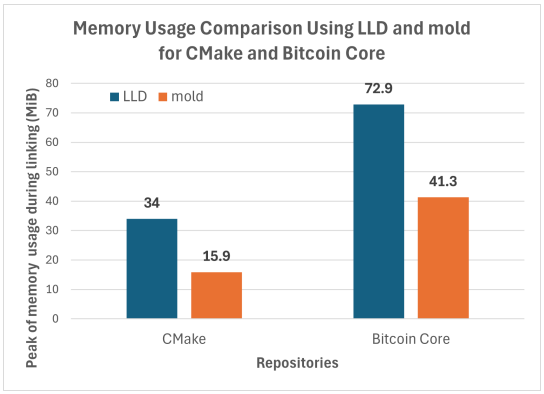

Figure 5: Comparison of memory usage between `LLD` and `mold` (lower is better)

5.3 Profiling Different Phases

As with previous benchmarks, the initial run was disregarded, and data from the second run were utilized for analysis. This study aimed to evaluate the real-time execution of specific phases in the linking process, including:

- Parsing files and symbol resolution, which encompasses adding symbols to the symbol table.
- Merging sections, copying input sections to output sections, writing to the final executable file, and applying relocations.
- Scanning relocations, which entails traversing all files to review and update relocations as needed, including the creation of dynamic relocations.
- Splitting sections, which involves dividing sections into smaller pieces to later merge duplicates from different files, reducing redundancy.

[8] https://github.com/Kitware/CMake
[9] https://github.com/bitcoin/bitcoin
[10] Counted using the `find . -name *.o | wc -l` command
[11] https://gcc.gnu.org/
[12] https://github.com/sisong/HDiffPatch

Details on the categories assigned to each phase are available in Appendix B. For detailed analysis, we employed built-in options – `--time-trace` for `LLD` and `--perf` for `mold`. Data for `mold` were initially recorded in seconds from the real-time column, while data for `LLD` were in microseconds; both were converted to milliseconds for consistency.

The diagram in Figure 6 presents the linking time comparison for `CMake` and `Bitcoin Core` between `LLD` and `mold`. Therefore, it is evident that for both linkers, the most time-consuming task during the linking process is writing to a file and applying relocations. Moreover, while for `CMake` `mold` is significantly faster than `LLD` in writing to a file, this advantage is significantly smaller for `Bitcoin Core`. Generally, `mold` tends to outperform `LLD` in all categories for both executables. However, surprisingly, it is slower in splitting sections for `Bitcoin Core`. Despite these differences, the general pattern in time distribution across various linking phases remains similar between `LLD` and `mold`.

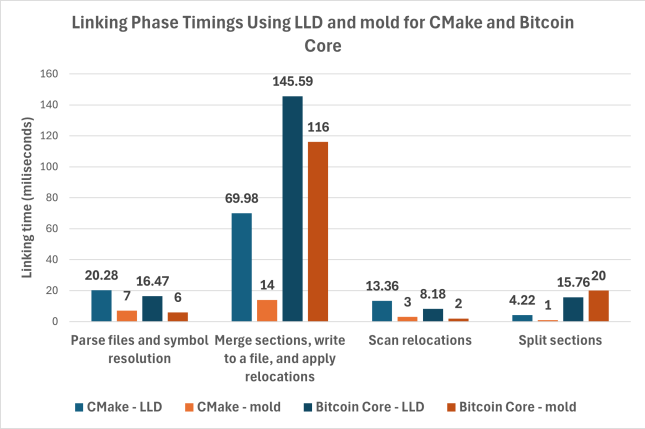

Figure 6: Linking time comparison across different phases between `LLD` and `mold` for `CMake` and `Bitcoin Core` (lower is better)

It is important to note that the comparison may not be entirely fair, as some phases inherently require more time due to their conceptual complexity. However, the aim of this comparison was to identify the general areas where linking consumes the most time and to assess whether there are significant differences between the linkers in how they handle these phases.

5.4 Binary Comparison

The executable `hdiffz`, produced by `LLD`, `mold`, and the default linker `ld`, was evaluated with respect to executable sizes, execution times, and the number of section headers, as shown in Table 3. For evaluating execution times, the `hdiffz` program was tested using PDF files containing 376,205 and 512,233 words, respectively. The execution times were measured using the `time` command, with data collected from the "real" category. To ensure reliability, the program was executed 100 times; both the average execution time and the standard deviation were recorded.

The results indicate that `mold` produces the largest executable, while the executable linked by `LLD` shows minimal size difference compared to the default `ld`. Based on the analysis of execution times, it was found that the execution times are influenced by the choice of linker, with a statistically significant effect ($F(2, 297) = 13.394$, $p < 0.001$, $\eta^2 = 0.083$), indicating a moderate effect size according to η^2. However, post hoc analysis revealed that the linker `ld` does not show a statistically significant difference to `mold` ($p = 0.788$), whereas `LLD` significantly differs from both `ld` and `mold` ($p < 0.001$). Additionally, Table 4 provides a detailed comparison of the section differences for each linker's output. Notably, `mold` generates a significantly higher number of sections than other linkers, raising questions about their necessity and potential for optimization by stripping redundant sections.

Linker	Executable Size (KiB)	Execution Time (ms) ± Standard Deviation (ms)	Section Headers
`ld`	762	1744.31 ± 38.61	33
`LLD`	764	1765.54 ± 28.97	35
`mold`	795	1740.92 ± 40.72	42

Table 3: Comparison of the executables for the `HDiffPatch` project, produced by different linkers, detailing executable sizes, execution times, and section headers

`ld`	`LLD`	`mold`
.plt.got	.got.plt	.plt.got
.plt.sec	.plt.sec	.got.plt
	.bss.rel.ro	.rodata.cst{4/8/16}
	.tm_clone_table	.tm_clone_table
		.copyrel.rel.ro
		.relro_padding
		.rodata.str1.{1/8}
		.copyrel

Table 4: Section differences for executables produced by each linker

6 Responsible Research

This section details the responsible research practices adopted in this study, specifically addressing ethical considerations in Subsection 6.1, the reproducibility of the methods in Subsection 6.2, and the representativeness of the repositories used for benchmarking in Subsection 6.3.

6.1 Ethical Considerations

Both repositories and software used during experiments are publicly available, thus ensuring not violating any proprietary systems or data. This also further aids in the reproducibility of the findings.

6.2 Reproducibility

To ensure the reproducibility of the findings, we provided detailed information about the tools, methodologies, and environments used in the experiments. Specific commit hashes for the software versions used are recorded:

- Last commit hash for LLD: `af36fb00e32e43101b68b142cfc938af68ad5ffe`
- Last commit hash for mold: `20fa8d56f5e0c47d1f4bbf7b829c12d3f43298e1`

Appendix B contains a comprehensive list of all phases and their categorizations to further aid the reproducibility of benchmarks against profiling specific phases of the linking process.

6.3 Representativeness

Ensuring that the sample code used for testing is representative of typical C and C++ codebases is crucial. This study employs two significant projects, CMake and Bitcoin Core, as benchmarks. These projects were chosen because of their complexity and the number of contributors, making them representative of real-world software development. Specifically, the CMake repository has 1350 contributors, and the Bitcoin Core repository has 955 contributors as of June 19, 2024. Using these projects ensures that the results are relevant and reflect true performance improvements across typical development environments.

In addition, we employed the HDiffPatch repo for binary comparison. Although this repository has only 8 contributors as of June 19, 2024, and 30 object files, its inclusion was strategic. We chose this project primarily for its simplicity in comparing execution times. However, it is acknowledged that future research might benefit from selecting a project that presents a more complex and demanding benchmarking scenario.

7 Discussion

The benchmarks performed in this study aimed to assess and compare the performance of LLD and mold across various metrics, particularly focusing on speed and memory usage. It was observed that mold generally excels in both areas. The use of external tools for measurement could potentially introduce overhead, however, since the same tools were utilized for both linkers, the comparisons remain valid despite potential distortions in absolute performance data.

Profiling was conducted using the built-in options provided by each linker, which may measure time slightly differently. Additionally, each linker may define the phases of the linking process differently, potentially skewing the results. While these factors could affect the absolute accuracy of the findings, for comparative purposes, this approach is sufficient to highlight performance differences between the linkers.

Linking speed, while crucial during software development, ultimately represents a one-time cost, making the performance of the resulting executable a primary concern. The analysis highlights noticeable differences in execution speeds, with the executable produced by LLD underperforming compared to those from other linkers. Additionally, mold tends to produce larger executables with more sections, the functions and purposes of which are not immediately apparent. This aspect requires further investigation.

The research findings are dependent on the current state of the linkers' codebases. Significant refactoring could alter the performance characteristics and the validity of the results. Hence, the reported findings are representative only as of the last commit hash documented.

A notable difference between LLD and mold is the extent of parallelization. mold applies parallel processing extensively, even in scenarios requiring data locking and atomic operations. In contrast, LLD limits parallelization to scenarios that do not involve complex data interactions. The absence of comprehensive parallel processing in LLD is often attributed to its support for linker scripts, which is deemed to introduce complexity and slow down the process [4]. The presence of linker scripts in LLD and their absence in mold highlight different priorities in terms of functionality and performance optimization. While linker scripts provide detailed control over the linking process, their complexity may hinder the implementation of more efficient, parallel processing algorithms, however, their impact remains unclear. While the benefits of parallelization in mold are evident, it is essential to consider potential overheads and whether these negate the performance gains.

The analysis of code responsibilities required making assumptions that may not be entirely accurate, adding a layer of uncertainty to the evaluation. Similarly, assumptions were necessary when categorizing tasks for profiling purposes.

The reason for the reduced performance of mold when writing to a file for Bitcoin Core, compared to CMake, is not immediately apparent. One possible explanation is the significantly smaller number of files in Bitcoin Core — 516[13] input files (505 object files + 11 shared files) compared to the 1143 input files (1135 object files + 8 shared files) in CMake. This disparity suggests that the benefits of parallelization may not be as obvious when fewer files are involved. The limited number of files in Bitcoin Core might also contribute to mold's slower performance in splitting sections and could explain why the overall linking time advantage of mold is dramatically reduced compared to its performance with CMake.

In terms of memory usage, one notable difference is how mold and LLD manage their symbol tables. mold maintains a simpler structure, utilizing only a hashmap that maps names to symbols, with each file then holding a vector of pointers to its symbols. In contrast, LLD uses a more complex system involving both a hashmap of symbol names to indices and a separate list of pointers to symbols. Additionally, each file in LLD maintains its own array of pointers to symbols. Moreover, LLD utilizes a greater variety of classes throughout its codebase to differentiate between various symbol types and file types. This additional layer in LLD's data management could contribute to its higher memory usage compared to mold.

8 Conclusions and Future Work

This research aimed to initiate an academic discussion about linkers, potentially sparking interest in the development of a new linker that combines the speed of mold with support for linker scripts. It has also prompted broader considerations regarding general optimizations for linkers.

[13] Counted by directly debugging the mold repository and checking the length of the input files list

The primary difference identified between LLD and mold lies in their approach to parallelization. mold extensively utilizes parallel processing capabilities, which significantly contributes to its performance advantage. In terms of their linking processes and architecture, the two linkers are quite similar, with no significant visible differences. However, mold lacks support for traditional linker scripts, which LLD accommodates. In summary, mold's remarkable performance can be attributed to its extensive use of parallelization across most stages of the linking process, the implementation of more efficient algorithms, and attention to performance-enhancing details such as utilizing two processes and overwriting an existing executable file.

As discussed in Section 7, there are unresolved questions about why LLD has not implemented more aggressive parallelization strategies. Further research into the architectural constraints imposed by linker scripts and other traditional features within LLD might reveal opportunities for significant improvements.

The benchmarks conducted as part of this study focused on single executable outputs, which leads to questions about how mold, given its extensive parallelization, would perform in scenarios with projects producing multiple executables, such as kernel modules. Additionally, considering mold's reliance on multicore capabilities for its remarkable performance, it raises the question of whether mold would still outperform LLD while utilizing only a single core. Although this research briefly addressed the executables generated by the linkers, a more extensive and detailed comparison, especially for larger executables, is necessary. Furthermore, we recommend conducting a broader set of benchmarks on a wider variety of repositories, with different sizes, to obtain a better and more comprehensive understanding of the performance of both linkers.

Therefore, the future research directions would involve investigating the impact of linker script complexity on the performance and parallelization capabilities of LLD, as well as assessing mold's performance in environments that output multiple files to determine if its parallelized nature offers substantial benefits in such contexts. Additionally, it is crucial to assess how mold performs when utilizing only a single core. A deeper examination of the executables produced by these linkers is also recommended to provide a comprehensive assessment of their effectiveness and efficiency.

This study has established a foundation for further exploration into linker technologies and set the stage for future innovations that could enhance the efficiency and functionality of these essential tools.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Introduction," in *Compilers: Principles, Techniques, and Tools*, pp. 1–3, Addison-Wesley, 2nd ed., August 2006.

[2] R. E. Bryant and D. R. O'Hallaron, "Linking," in *Computer Systems: A Programmer's Perspective*, pp. 705–750, Pearson, 3rd global ed., 2016.

[3] R. Ueyama, "mold: A Modern Linker." https://github.com/rui314/mold?tab=readme-ov-file#mold-a-modern-linker, 2024.

[4] R. Ueyama, "Can the mold linker be /usr/bin/ld?," in *Proceedings of the FOSDEM Brussels 2024*, (Brussels, Belgium), FOSDEM, Feb. 2024.

[5] W. Dietz and V. Adve, "Software multiplexing: share your libraries and statically link them too," *Proceedings of the ACM on Programming Languages*, vol. 2, Oct. 2018.

[6] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "SLINKY: Static Linking Reloaded," in *USENIX Annual Technical Conference*, pp. 309–322, 2005.

[7] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: explaining ELF static linking, semantically," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 607–623, Association for Computing Machinery, Oct. 2016.

[8] D. B. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg, "Fast and Flexible Shared Libraries," *USENIX Summer*, pp. 237–252, June 1993.

[9] V. Kasliwal and A. Vladimirov, "A Performance-Based Comparison of C/C++ Compilers," *Colfax International*, 2017.

[10] R. Hebbar S R, M. Ponugoti, and A. Milenković, "Battle of Compilers: An Experimental Evaluation Using SPEC CPU2017," in *2019 SoutheastCon*, 2019.

[11] D. Thain, "A quick tour," in *Introduction to Compilers and Language Design*, pp. 5–10, University of Notre Dame, 2nd ed., 2021.

[12] D. Andriesse, "The ELF Format," in *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*, pp. 31–56, No Starch Press, 2019.

[13] M. Kalaycı, "An Introduction to Linker Files: Crafting Your Own for Embedded Projects." https://medium.com/@mkklyci/an-introduction-to-linker-files-crafting-your-own-for-embedded-projects-60ad17193229, 2023.

[14] Valgrind™ Developers, "Support for Threads." https://valgrind.org/docs/manual/manual-core.html#manual-core.pthreads, 2020.

[15] R. Ueyama, "Why is mold so fast?." https://github.com/rui314/mold?tab=readme-ov-file#why-is-mold-so-fast, 2024.

[16] LLVM Project, "The ELF, COFF and Wasm Linkers." https://lld.llvm.org/NewLLD.html, 2024.

[17] R. Ueyama, "lld: A Fast, Simple and Portable Linker." https://llvm.org/devmtg/2017-10/slides/Ueyama-lld.pdf, 2017. Presentation at the LLVM Developers' Meeting 2017.

[18] R. Ueyama, "mold(1) – a modern linker." https://github.com/rui314/mold/blob/main/docs/mold.md, 2024.

[19] R. Ueyama, "Design and implementation of mold." https://github.com/rui314/mold/blob/main/docs/design.md, 2021.

[20] LLVM Project, "LLD - The LLVM Linker." https://lld.llvm.org/, 2024.

[21] LLVM Project, "Linker Script implementation notes and policy." https://lld.llvm.org/ELF/linker_script.html, 2024.

A Linking Time in the Compilation Process

This section presents the methodology used to differentiate the compilation and linking times for the `CMake` and `Bitcoin Core` repositories. The `time` command was used alongside `make` to measure the total real-time spent on compilation and linking. To isolate the linking time, all produced executables were deleted, and `time make` was executed again. This approach provided the linking time exclusively. Table 5 summarizes the results.

Repository	**Total Compilation Time (s)**	**Linking Time (s)**
`CMake`	821.89	4.77
`Bitcoin Core`	1741.92	8.57

Table 5: Comparison of total compilation time and linking time for `CMake` and `Bitcoin Core` repositories

While this experiment provides an approximation of the time spent on linking, it serves primarily for comparative and visualization purposes rather than precise measurement.

B Categorization of Linking Process Phases

This section details the categorization of the linking process phases for both `mold` and `LLD`. Tasks within each phase are grouped according to their classification during the experimental analysis.

B.1 Classification of Phases for `mold`

1. Parse files and symbol resolution
   - read_input_files
   - resolve_symbols
2. Merge sections, write to a file, and apply relocations
   - open_file
   - copy
   - close_file
   - create_output_sections
3. Scan relocations
   - scan_relocations
4. Split sections
   - split_section_pieces

B.2 Classification of Phases for `LLD`

1. Parse files and symbol resolution
   - Total Parse input files
   - Total Load input files
2. Merge sections, write to a file, and apply relocations
   - Total Write output file
   - Total Add symbols to symtabs
   - Total Finalize .eh_frame
   - Total Aggregate sections
   - Total Assign sections
   - Total Finalize synthetic sections
   - Total Sort sections
   - Total Create output files
   - Total Merge/finalize input sections
3. Scan relocations
   - Total Scan relocations
4. Split sections
   - Total Split sections