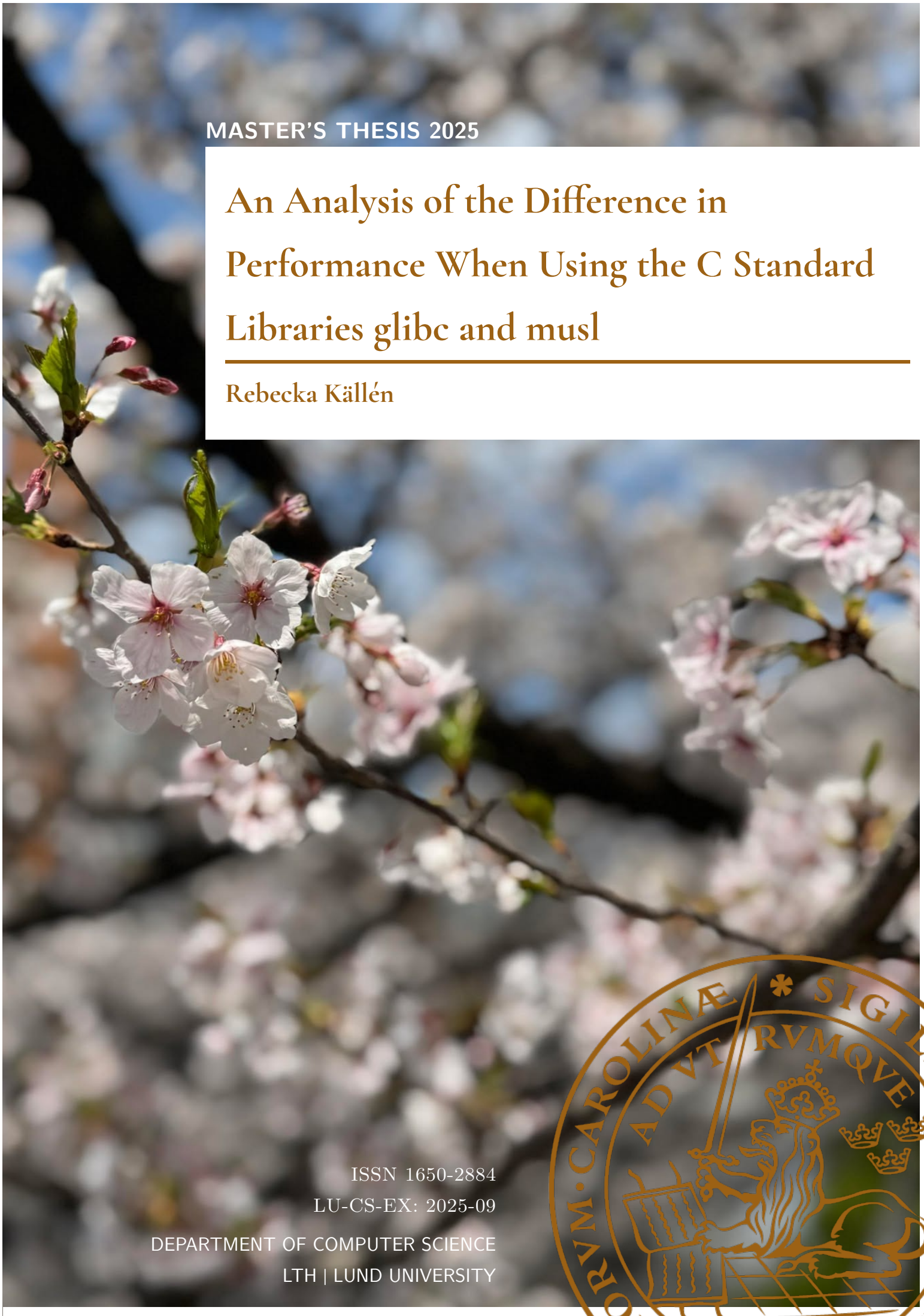

MASTER'S THESIS 2025

An Analysis of the Difference in Performance When Using the C Standard Libraries glibc and musl

Rebecka Källén

ISSN 1650-2884
LU-CS-EX: 2025-09

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-09

An Analysis of the Difference in Performance When Using the C Standard Libraries glibc and musl

Rebecka Källén

An Analysis of the Difference in Performance When Using the C Standard Libraries glibc and musl

Rebecka Källén

rebecka.kallen.093@student.lu.se

May 8, 2025

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Per Andersson, Per.Andersson@cs.lth.se

Abstract

Since there are several well established C standard libraries available for programmers, the decision of which library to use is ultimately up to the individual. A popular choice is the GNU C Library, commonly called glibc, which was released in 1988. More recently, in 2011, musl was released and there are those considering the advantages of switching from the use of glibc to musl. The purpose of this degree project is to compare the performances of these two C standard libraries, and to identify under which circumstances musl would be a preferable choice to glibc.

Seven functions were chosen for the comparisons, and three aspects of their performance were analyzed; the memory size required, the runtime, and the precision of numerical functions. The completeness and correctness of musl was also evaluated with some larger programs. Tybor and PARANOIA were used to evaluate the numerical accuracy, bit precision, and rounding of both libraries. The results showed that both glibc and musl have a reliable numerical accuracy that is in accordance with the IEEE 754 standard. The runtimes required for test programs were measured using Timebase, a C function written at LTH. The use of glibc consistently resulted in shorter runtimes, but the use of musl generally required less memory. For larger programs and benchmarks, PostgreSQL and SPEC CPU2006 were used. The results showed that musl is reliable and robust enough to handle many larger programs, however, some modifications had to be made to musl in order to make it work for some larger programs. There are several additional factors to consider when choosing between the libraries; however, musl appears to be both reliable and more versatile, making it a competitor to glibc in some cases.

Keywords: musl, glibc, C standard library

Acknowledgements

I would like to thank my supervisor Jonas Skeppstedt for his guidance, patience, and support not only during this thesis, but also during his courses at LTH.

Contents

Chapter 1
Introduction

Libraries are an essential part of programming since most languages, such as C, only support the most basic features needed to write a program. Therefore, without using already existing libraries it would be up to every programmer to provide the other necessary functions that are frequently used, which would be a severely time consuming task. Furthermore, there is an official C programming language standard called ISO C, which specifies the types, macros, and functions that should be present in a C standard library. This is why several well written and well used C standard libraries already exist, and a particularly popular one is the the GNU C Library, commonly known as glibc. glibc was released in 1988, and while glibc is fast and reliable, its complex source code can be hard to navigate and hard to compile since it makes use of many GNU C extensions. GCC is also the only compiler capable of compiling glibc without modifying its source code, which demonstrate another limitation of using glibc [1]. More recently, in 2011, another C standard library called musl was released, and there are those considering the advantages of switching from the use of glibc to musl [2].

1.1 Problem statement

The purpose of this degree project is to investigate how these already available C standard libraries' performances differ and why, so that programmers can make a well informed decision when choosing which of the two libraries to use, either in general or for a specific program. This project could also be an opportunity for anyone who wants to write their own C standard library to understand different ways that standard functions could be implemented and the advantages and disadvantages of these.

1.2 Scope

As previously mentioned, this degree project compares functions in the two C standard libraries glibc and musl, and considers no other libraries. In order to further narrow down the scope of the study, the number of functions compared are limited to seven. Since both glibc and musl include many functions not included in the ISO C standard library specification, naturally in order to make the comparison, these seven functions must be present in both glibc and musl. In order to choose functions with well-defined behavior, the functions chosen must also be present in the ISO C standard library. There is a variation of the type of function being examined, and these types include mathematical functions and functions used for string manipulation. Functions such as `malloc()` and `free()` are considered but not thoroughly evaluated as it is relatively easy for programmers to write their own version or to use an open source implementation, and therefore the implementation of these would probably not persuade a programmer to choose any library over another. Three aspects of the performance of the seven chosen functions are analyzed during the comparisons; the memory size required, the runtime, and the precision of numerical functions, specifically the resulting bit pattern. The completeness and correctness of musl is also later evaluated with some larger programs.

1.3 Research questions

The purpose of this research is to verify that musl is a reliable library, and the following research questions should be answered:

1. How does the performance differ between glibc and musl?
2. Under which circumstances would musl be preferable to glibc, and vice versa?

1.4 Outline of the report

This thesis contains the following chapters in order:

1. **Introduction**
   This chapter introduces the research topic, describes the limitations, and presents the research questions that will be answered.
2. **Theoretical framework**
   The theoretical background and concepts needed to understand the chapters that follow are presented in this chapter.
3. **Methodology**
   This chapter describes the methods and tests that were used during the testing phase of this thesis.

4. **Results**
   The resulting data from the tests done are presented in this chapter.

5. **Discussion**
   The results are interpreted and discussed in this chapter. Suggestions for future research is later also presented.

6. **Conclusion**
   The research questions, that were presented in the introduction, are concisely answered in this chapter.

Chapter 2
Theoretical framework

This chapter describes the programs and software that were used during the testing phase of this thesis. The concepts needed to understand the discussion of results and the conclusions made are also presented.

2.1 The standard C library

The standard C library is a collection of header files containing functions, types, and macros. There are many implementations of this library available, and two are presented below. Because of the C language's inability to directly perform common operations such as string manipulation and memory management, a library, such as glibc or musl, should be linked to your program when you compile it [17]. The standard functions used in this project are also presented further below together with the header file they belong to.

2.1.1 ISO C

ISO, short for International Organization for Standardization, was created in 1946 and now includes 173 different countries. The organization's goal is to develop standards in a wide variety of sectors that are meant to guarantee safety, efficiency, and quality in different industries. These standards are also important when libraries for programming languages are developed, as standardized coding practices make sure that the software quality is maintained. ISO has developed standards for several programming languages, including C, C++ and SQL [3].

ISO C defines the standardized version of the C language in order to ensure consistency, portability, and reliability for developers across platforms.

The C language standard is continuously revised, and each new standard is named by the year it was released, with C23 being one of the most widely used versions. But since the official documentation written by ISO is not free, most programmers get their information about C language standards from third-party websites. Li et al. investigated four of these websites and discovered that not all popular online documentation conforms with the C99 standard accurately [18].

ISO C specifies:

- the representation of C programs
- the syntax and constraints of C
- the semantic rules of C
- the representation of input data to be processed by C programs
- the representation of output data produced by C programs
- the restrictions and limits imposed by a conforming implementation of C [4]

2.1.2 GNU C Library

The GNU C library, commonly known as glibc, is a standard C library implemented by the GNU Project and has been in development since 1988. glibc contains header files which individually bring definitions and declarations for a collection of facilities belonging together [5].

2.1.3 musl

musl, pronounced like "mussel" or "muscle" is an implementation of the standard C library which was first released in 2011. musl, like glibc, is responsible for implementing library routines necessary when using the C language, this includes functions such as `sin()` and `memcpy()`, but it also does other things such as providing C bindings for the OS interfaces, memory allocation management, and thread creation and synchronization operations, which is defined by the POSIX pthreads API.

One way to use musl with the compiler GCC is to use a wrapper script called `musl-gcc`. This script automatically links GCC with the musl library instead of glibc.

According to the creators of musl the key principles of this implementation of the standard C library are:

- **Simplicity**
  If possible, musl tends to use simple algorithms over more complex ones, minimizing abstractions, and keeping code as self-contained as possible. These things also usually lead to a more readable code.

- **Resource efficiency**
  Due to very little internal coupling between libc components, when using musl with static linking the application will use very little unnecessary code. libc's global data size is also as small as possible.

- **Attention to correctness**
  musl contains some functionality that have not been implemented in any previous standard C library. One example of this is musl being the first Linux libc to have mutexes safe to use inside reference-counted objects.

- **Safety under resource exhaustion**
  musl always checks that enough resources are available before initializing an operation, and backs out if not enough resources are available. In general, musl also does not use dynamic allocation unless necessary.

- **Ease of deployment**
  Binaries statically linked with musl have no external dependencies, and musl's MIT license makes commercial use easy.

- **First-class support for UTF-8/multilingual text**
  It is not necessary while using musl to use external locale files or conversion modules to process UTF-8 or query properties of arbitrary Unicode characters [6].

2.2 math.h

The `math.h` header file declares various macros and mathematical functions, where all functions initially only were able to take a parameter of type `double` as well as return a value of type `double`. However, when C99 was released, new functions were introduced that could take `float` and `long double` instead. These functions which takes different parameter types have names which reflect the type, `sin` takes a `double`, `sinf` takes a `float`, and `sinl` takes a `long double` [20]. The following four functions, which are included in the `math.h` header file, are analyzed during this thesis.

2.2.1 sin

```
#include <math.h>
double sin(double x);
```

The `sin` function calculates the sine of a `double x` and returns a `double`. `x` should be in radians [22].

2.2.2 atan

```
#include <math.h>
double atan(double x);
```

The `atan` function calculates the principle value of the arc tangent of a `double x`. The return value should be a `double` in the interval $[-\pi/2, +\pi/2]$ [22].

2.2.3 sqrt

```
#include <math>
double sqrt(double x);
```

The `sqrt` function calculates the square root of a `double x`, and returns a `double` [22].

2.2.4 log

```
#include <math.h>
double log(double x);
```

The `log` function calculates the the natural logarithm, which has the base `e`. This function takes a `double` and returns a `double` [22].

2.3 string.h

The `string.h` header file declares one variable type (`size_t`), a macro (`NULL`), and a collection of functions which makes it possible to manipulate strings and memory areas. `size_t` and `NULL` are also declared in other header files; however, the C standard library ensures that they are not redefined when more than one of these header files are used by the same C file [22]. The following three functions, which are included in the `string.h` header file, are analyzed during this thesis.

2.3.1 memcpy

```
#include <string.h>
void* memcpy(
      void restrict*          dest,
      const void* restrict    src,
      size_t                  size);
```

The `memcpy` function copies `size` bytes from `src` to `dest` and returns `dest` [22].

2.3.2 strcpy

```
#include <string.h>
char* strcpy(
      char* restrict          dest,
      const char* restrict    src);
```

The `strcpy` function copies the string pointed to by `src` to the array pointed to by `dest`, `dest` is then returned [22].

2.3.3 strstr

```
#include <string.h>
char* strstr(
      const char* s1,
      const char* s2);
```

The `strstr` function searches the string pointed to by `s1` for the first occurrence of the string pointed to by `s2`. If the string is found, the function returns a pointer to the string, otherwise a null pointer is returned [22].

2.4 Memory layout of a C program

A C program is divided into several memory segments, as seen in figure 2.1.

- The **stack** segment is where local variables, parameters to functions, and return addresses are stored.
- The **heap** segment is where objects allocated with `malloc` are stored. As seen in figure 2.1, the **stack** segment and the **heap** segment grow towards each other in the currently unused memory.
- The **data** segment is where variables with static storage duration are stored, this includes global and static variables.
- The **text** segment, also known as the code segment, is where the machine code instructions are stored. The **text** segment can also contain constants and string literals. This segment and the **data** segment both have fixed sizes [22].

The term segment is used in a running program and also in an executable file, such as `a.out`. In relocatable files, with `.o` suffix, the data and text segments are instead called sections, and there can be different sections related to data which will contribute to the data segment of the program. There are also sections which do not end up in a segment.

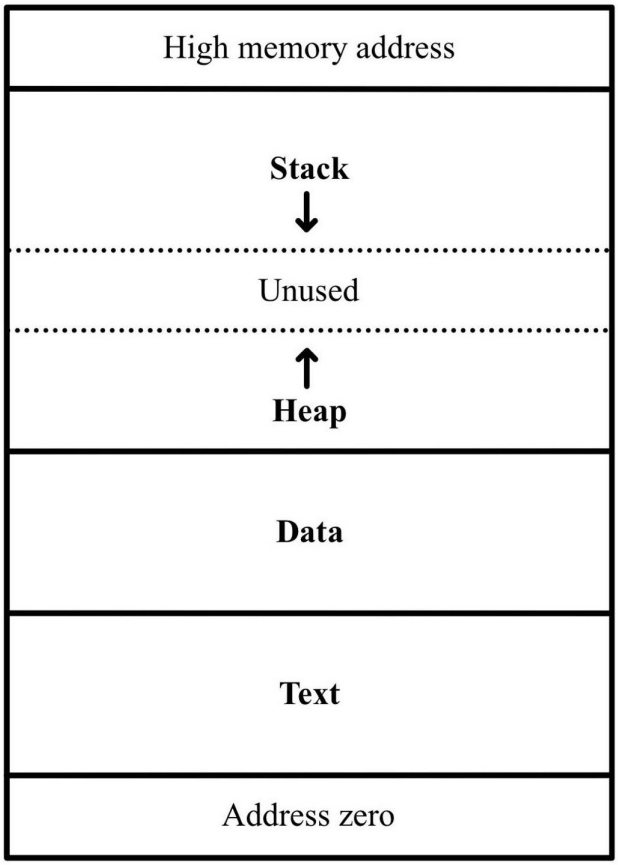

Figure 2.1: Memory segments of a C program [22]

2.5 Linking

When a C program file is compiled, an object file is created containing the binary representation of the C program. A C program file has a `.c` extension and an object files has a `.o` extension. This object file is not runnable, but instead contains machine code that can be linked with other object files and necessary libraries. The linker later combines these files into a single executable file. There are two different types of linking, and each version has its own advantages and disadvantages depending on their usage.

The two types of linking:

- **Static linking**
  In static linking, the linker copies all code from object files and libraries into the final executable file.

- **Dynamic linking**
  In dynamic linking, the executable file instead contains references to library functions, such as functions in a standard library, but the program's own functions are linked to the executable file. The linking then takes place at runtime [22].

In order to compile and link a program `a.c` using GCC in one step, you can write as follows to use dynamic linking:

```
gcc a.c -lm
```

The `-lm` flag links the math library in the used C standard library to the program, and the result is an executable file. In order to use static linking instead, the flag `-static` should be added.

An advantage of using dynamic linking is that the most recent versions of libraries will be used when the program is run, which then automatically includes all security updates of the library that are installed on the computer. However, with static linking, it is guaranteed that no library or library version is missing, which sometimes happens in dynamic linking. Since dynamic linking does not store all library functions in each executable program, using dynamic linking usually results in smaller executable files.

An object file contains a number of sections and some of these, such as the text and data sections, are put in their respective memory segment of the running program. While other sections only have information that is useful during linking, this includes sections for the symbol table and relocation entries. A relocation entry is useful because it contains information about how the data that is copied to a segment should be modified by the linker.

For example, consider two global variables:

```
int x;
int* p = &x;
```

When the program starts, the pointer `p` should contain the address of `x`. In order for this to happen, there is a relocation entry that specifies how many bytes into a data section that `p` is located. There is also information regarding which symbol's address should be used, in this case it should be `x`. The linker then modifies that specific part of the data segment.

Another example is when a function that is not located in that specific object file is called. The linker then calculates the number of bytes that the call instructions should branch forward or backward. In some instruction set architectures all instructions are located on a 4-byte aligned address, so only the number of instructions is written in the machine instruction.

There is one section of such relocation entries for each section that should be put in a segment, therefore there is usually one relocation section for text, and another for data. A compiler can also decide to put constants, strings, and other immutable data in a special data section, called rodata, for read only. When the compiler has collected all the different sections from all included object files, it will look for missing functions and variables, such as `printf` in libraries.

A library is a file which contains multiple object files. If the linker, for example, finds `printf` in one object file, and finds that it is implemented with `fprintf` it will continue searching for `fprintf` in the library, and so on. Eventually, the linker has either found all global variables and functions needed or it concludes that some are missing. If all were found, it can add the sizes of all text sections to calculate the size of the text segment, and similarly it calculates the sizes for all different kinds of data sections that will be put in the data segment.

When debugging is used, the compiler produces special sections with information about types and line numbers. This information is also included in the executable file so that a debugger, such as `gdb`, can interpret the information when it is executed. Sometimes it is also necessary for debugging information to need relocation entries, there are then separate sections for these [21].

2.6 Initialization of a C standard library

An executable program needs data from the operating system before it can start in the main function. This data includes the number of program arguments (`argc` in C), the address of the array with the argument strings, and an array with environment variables such as `HOME` and `PATH`. It also requires specific details about the system, including the page size used in virtual memory and features of the CPU. The `argc` parameter is passed in a register like any other parameter, which is also the case for the `argv` (`char* argv[]`) array of pointers to argument strings.

Furthermore, the argument strings and environment variables also need to be located. At startup, the system provides the initial stack pointer to the new process and puts the value of `argc` and the three arrays after each other in the stack segment of the new process. By using

the stack pointer (register `g1` on Power) the array `argv` is located. After this array, the array of environment variables, which contain values such as `PATH=/usr/bin:/bin`, is located. Lastly, there is an array with elements of type `size_t` i.e. typically 64-bit unsigned integers. This last array is called `auxv` and has values at predefined positions. For instance, one of these elements contains the `pagesize`. Both glibc and musl read the `auxv` to setup internal data structures, including the `pagesize`, which is used by malloc. On a linux system, the contents of the `auxv` for a running program can be found in the file `/proc/self/auxv` which is a file whose contents the Linux kernel creates when a process reads it, as in, the file is not stored on disk [21].

2.7 Combining C standard libraries

Although it might seem reasonable to assume that two different C standard libraries' functions could be combined freely since they must be a part of a standard, this is rarely the case. One cannot assume that, for example, `sin` from musl and `cos` from glibc would work together without issues, or that `sin` from glibc could be used inside musl. The reason why this is not possible, even though they both implement the same mathematical function, could be because of potential binary compatibility issues between them. Several things could cause this, but low-level details such as data being passed between functions in different ways could cause issues, or the types used could be defined differently. Certain types are standardized by ISO C, below are three examples.

```
typedef struct { int quot, rem; } div_t;
typedef struct { long quot, rem; } ldiv_t;
typedef struct { long long quot, rem; } lldiv_t;
```

But there are also types that are not standardized, such as `FILE`. This difference in type definitions could therefore cause issues when combining libraries. Furthermore, there are types that are standardized only for a specific platform, which means a specific combination of operating systems and CPU families.

Despite the fact that most functions cannot be easily replaced, there is at least one function, `malloc`, where replacing the function is supported. The specifications are given below.

```
weak_alias(__simple_malloc, __libc_malloc_impl);

void *__libc_malloc(size_t n)
{
    return __libc_malloc_impl(n);
}

static void *default_malloc(size_t n)
{
    return __libc_malloc_impl(n);
}
```

```
weak_alias(default_malloc, malloc);
```

The developers of musl have chosen a standard `malloc` that suits the purpose of most programs, but there is the choice to relatively easily replace or customize `malloc` to better suit the purpose of your application [21].

2.8 Linux commands

The following two Linux commands are used during this thesis in order to find and compare the sizes of C standard library files.

2.8.1 size

```
size [objfile...]
```

The Linux command `size` lists all section sizes (of sections that will be put in a segment) and the total size of all binary files on its argument list. `[objfile...]` are the files to be examined, and `size` will produce one line of output for each file. `a.out` will be used by default if `[objfile...]` is left empty.

This command results in an output similar to:

```
text    data    bss     dec     hex
4256    32      0       4288    10c0
```

Where the names are explained as:

- **text:** The text segment is the memory section where executable instructions are located.
- **data:** The data segment contains the global and static variables that have been initialized.
- **bss:** The bss segment contains all data with static storage duration that should be initialized to zero by the operating system when the program starts. bss stands for block started by symbol and the name is a historical rest.
- **dec:** This is a summation of the three previous fields, text, data, and bss, as a decimal number
- **hex:** This is the dec converted into hexadecimal number [7]

2.8.2 nm

```
nm [objfile...]
```

The `nm` linux command lists the symbols from the selected object files `[objfile...]`. `a.out` is the default input if no object files are listed [8].

2.9 IEEE 754

IEEE 754 is a standard for floating-point arithmetic which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE) under the leadership of Professor William Kahan at UC Berkley [23]. This is an industry standard which specifies arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments, and two of the floating-point formats specified are `single` and `double` [9]. There are tools available for testing if software conform to the IEEE 754 standard, one such tool was proposed by Verdonk et al. which tests how well a floating-point implementation complies with IEEE 754 [23].

IEEE 754 is often the most efficient way to represent floating-point numbers, and the representation has three components:

1. a 1-bit sign, **s**
2. an 11-bit biased exponent, **e**
3. a 52-bit fraction, **f** [10]

2.10 Floating-point errors

A math library's precision is essential since errors in floating-point computations can severely limit the accuracy of calculations, which is extremely important in many scientific fields. Since floating-point numbers are represented according to the IEEE 754 standard, the representation uses a finite number of bits, and errors due to approximations are unavoidable. This is an issue that was discussed by Qi et al., the intervals where errors were most likely to occur were also identified and an approach to finding the maximum error of mathematical functions was suggested [19].

Benz et al. discussed the possible sources of floating-point errors, the consequences of undetected errors, and presented a dynamic program analysis used to detect different kinds of floating-point errors [16]. The sources of errors that can arise can be divided into three groups:

- **Rounding**
  It is impossible to prevent rounding errors because of the finite precision of floating-point numbers.

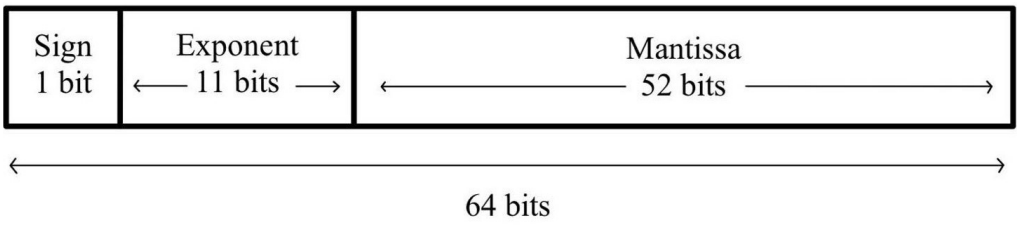

Figure 2.2: IEEE 754 floating-point standard for the type double.

- **Data uncertainty**
  This occurs when there is an error or insufficient precision in the initial input or the result of a previous calculation.

- **Truncation**
  Truncation happens when a numerical algorithm approximates a mathematical function.

The error caused by a single operation is often small and therefore difficult to detect; however, large errors can appear due to error accumulation. There are applications where the IEEE 754 floating-point representation is insufficient, but it has a high enough precision for many applications [16].

2.11 The `double` data type in C

The following data type, `double`, will be used in this thesis during analysis of numerical precision. `double` in C is used to store high precision floating point values. A `double` is 8 bytes long, which is 64 bits, and the value can range from $1.7 * 10^{-308}$ to $1.7 * 10^{+308}$. It should be possible to store values with up to 15 decimal points without any loss in precision.

When a double is stored in memory in C it is divided into three parts and stored as a 64-bit data item. These parts are: the sign bit, the exponent, and the mantissa, this is according to the IEEE 754 standard. The memory is divided as following, and can be seen in figure 2.2:

1. **Sign bit:** 1 bit
2. **Exponent:** 11 bits
3. **Mantissa:** 52 bits [22]

2.12 Test programs

Programs that were used to perform tests during this project are briefly presented in this section.

2.12.1 Tybor

Tybor is a Floating-Point C Extensions (FPCE) test suite that checks C and C++ compilers, headers, and math libraries so that they conform to the FPCE part of C99. With these tests it is possible to check the accuracy of a math library, as well as the accuracy of decimal conversions of the I/O library.

The tests that are available to download for free online are:

- `tflt2int.c`: tests `float` to `int` conversions
- `tint2flt.c`: tests `int` to `float` conversions
- `tsin.c`: tests `sin(355.0)` for accuracy
- `tbin2dec.c`: tests internal binary to external decimal conversions of floating-point numbers
  using `printf("%f")` [11]

2.12.2 PARANOIA

The test program PARANOIA tests the floating-point arithmetic implementation on a computer. The purpose of PARANOIA is to check for errors in floating-point computations, round-off errors, precision loss, and unexpected results in arithmetic operations. The original program was written in BASIC by Professor William Kahan at UC Berkley, who as mentioned earlier chaired the development of the IEEE 754 standard for floating-point arithmetic. [23] Even though the original version was written in BASIC, the program is now available in C, which was the version used during this project [12].

2.12.3 timebase

`timebase` is a C function which was written at LTH. The function uses a hardware register of the POWER instruction set architecture and the lines with `timebase` and `clock` in the file `/proc/cpuinfo` in order to measure runtimes and clock cycles. The information from `cpuinfo` is read only once, and not when running the program being studied. Files in the Linux `/proc` directory are not regular files but instead the Linux kernel produces their contents directly when a process reads them, which therefore gives the current value of the CPU clock frequency. [21].

2.13 PostgreSQL

PostgreSQL is an open source object-relational database management system that uses the SQL language. The development of PostgreSQL started in 1986, and it has since become

a popular choice for many organizations due to its reliability. PostgreSQL requires a standard C library to perform a number of low-level operations, such as memory management and string manipulation. In addition, PostgreSQL also requires POSIX functions to execute system calls. [13]

2.14 SPEC

SPEC, short for Standard Performance Evaluation Corporation, is a non-profit organization that develops standardized benchmark suites and tools to evaluate and compare computer systems. The SPEC CPU benchmarks are updated as needed to account for newly introduced faster computers with increased memory and CPU performance. SPEC CPU has been released in 1989, 1992, 1995, 2000, 2006, and 2017. [14]

2.14.1 008.espresso

008.espresso is a benchmark included in the SPEC CPU89 benchmark suite, and it performs boolean function minimization which simulates realistic allocation and deallocation patterns. The benchmark uses malloc and free to a relatively large extent, and with that measures the performance of malloc and free. [21]

2.14.2 CPU2006

SPEC CPU2006 is a standardized CPU-intensive benchmark suite that can run on many different computer architectures [24]. This benchmark suite contains 31 different tests and was created to measure a computer's performance in different ways [15]. These suites are used in both industry and academia to test the performance of memory systems, CPUs, and compilers. SPEC CPU2006 includes both a number of integer benchmarks and floating-point benchmarks. Many experts have studied SPEC CPU2006, among them, Zou et al. examined the memory access behaviors of the SPEC CPU2006 benchmarks [24].

Chapter 3
Methodology

This section describes how the experimental tests were performed during this degree project. First, the numerical accuracy of musl was evaluated using Tybor and PARANOIA, as these two tests comprehensively test the numerical accuracy of musl when combined. Later, a comparison of runtimes and memory sizes were examined using Timebase and `size`. At last, the completeness and correctness of musl was tested using larger more complex programs. These areas of testing were chosen as they provide a comprehensive assessment of the overall performance of musl.

3.1 Software and hardware

- The following seven functions were the only ones examined during the testing phase of the project.

    1. `sin`
    2. `atan`
    3. `sqrt`
    4. `log`
    5. `memcpy`
    6. `strcpy`
    7. `strstr`

- During testing, the following software versions were used:

    ```
    GCC version: 9.4.0
    ```

```
GNU libc version: 2.40

musl version: 1.2.5
```

- The computer used had IBM POWER8 architecture which is a superscalar processor used in high-end servers.

3.2 Testing numerical accuracy

This section describes how the numerical accuracy of mathematical functions in glibc and musl were tested with Tybor and PARANOIA.

3.2.1 Tybor

`Tybor` was first used when testing the numerical accuracy of functions. Since only free Tybor tests were available, altering an already existing `Tybor` program was the only way to compare the resulting bit patterns from the two libraries with the known correct bit pattern. The test used was `tsin.c` and fortunately, this test was easily altered to fit the different mathematical functions.

The following steps were taken:

1. The mathematical function and an argument was decided.

   For example:

   ```
   log(x) and x = 234
   ```

2. The correct output value that the chosen libraries' resulting bit pattern should be compared to was calculated with a high precision using `matlab`. This value was printed with a precision of 15 decimal digits, and printed to a file for storage. Matlab uses a library for mathematical functions with high numerical accuracy which complies with the IEEE 754 standard. Therefore, 15 decimal digits were deemed a high enough accuracy. This resulted in a file with the resulting numerical value:

   ```
   5.455321115357702
   ```

3. The file `tsin.c` was then altered to test the desired function. The test program needed to be altered in several places to do this.

   The changes that were made:

- Line 75:
  `DBL const correct` was set to the value that was previously calculated with `matlab`.
  For example, for `log(235)`:
  `DBL const correct = 5.455321115357702;`
- Line 93:
  `calculate` was set to the function and value that was to be calculated.
  For example, for `log(235)`:
  `calculate = log(234.0); /* Function Under Test */`
- Line 96 and 154:
  Both these lines needed to be changed to reflect the current function being tested, solely to make the output easier to read.

4. The file that was just altered was then compiled with both `glibc` and `musl` separately.

   For example, for `log(235)` using `glibc`:

   ```
   gcc tsin.c -lm -o glibc_out
   ./glibc_out
   ```

   For example, for `log(235)` using `musl`:

   ```
   musl-gcc tsin.c -static -o musl_out
   ./musl_out
   ```

This process was then repeated for all numerical functions with different arguments.

3.2.2 paranoia.c

The steps taken to run the paranoia test with glibc and musl is described below:

1. In order to run paranoia with glibc, write the following command in the terminal:

   ```
   gcc paranoia.c -lm
   ```

2. In order to run paranoia with musl, write the following command in the terminal:

   ```
   musl-gcc tsin.c -static
   ```

The result was then printed in the terminal.

3.3 Timebase and size

This section describes how Timebase and `size` were used in order to find the runtimes and memory sizes required when running the seven chosen functions in glibc and musl.

3.3.1 musl

In order to run a program using Timebase and musl, the following steps were taken:

1. A program was created that ran the function to be tested a number of times, this number was big enough for the runtime to be at least a few seconds long.

2. A bash script, called `run`, was used to run the tests. This script can be seen in appendix A. The file `run` was altered to include the newly created program and the compiler used was changed to `musl-gcc`.

   `run` does the following things:

   - Compiles your program with a chosen compiler.
   - Runs a performance profiling tool for Linux called `operf`.
   - Produces symbol image summaries with the command `opreport`.
   - Produces source and assembly annotated with profile data, this is done with the command `opannotate`.

3. After running the program, a number of files were created and results were printed in the terminal. Results for clock cycles, runtimes, and library files used could be found among these files.

3.3.2 glibc

In order to obtain the same information for glibc that was found for musl, a few extra steps had to be taken. When finding out which library files were used for the function, the following steps were taken:

1. A program `a.c` was run with gcc and the -c flag, this tells gcc to compile the source code into an object file `a.o`, but not to link it.

   ```
   gcc -c a.c
   ```

2. The following line was written in the terminal:

```
nm a.o
```

This results in a print similar to the text below. By analyzing the text it was possible to find the symbols necessary for finding used library files. From the example below it could be seen that the symbols `atan` and `printf` had to be imported from the glibc library in order to get library file information.

```
                 U atan
0000000000000000 T main
                 U printf
                 U .TOC.
0000000000000000 D v
```

3. The `nm` command was used again with the relevant static archive file, with the filename extension `.a`, inside glibc. This made it possible to identify which object file was needed to run the program with `run` in order to get the sought after information.

```
nm lib/libc.a
```

In the resulting text, this was the relevant results:

```
s_atan.o:
0000000000000038 r a
0000000000000000 T __atan
0000000000000000 W atan
0000000000000000 W atanf32x
0000000000000000 W atanf64
0000000000000030 r b
```

4. From this information it could be seen that the object file `s_atan.o` was necessary. The object file was then found inside the library and copied into the same directory as `run`. `run` was also altered to include the necessary object file when compiling.
5. The previous steps were then repeated for every file needed to get the desired information from timebase.

3.4 Larger programs and benchmarks

musl and glibc were also used when running some larger programs, this included SPEC CPU2006, 008.espresso, and two versions of PostgreSQL. The PostgreSQL versions were postgres-12 and postgres-17. The SPEC CPU benchmarks were run seven times in order to get the mean, median, and standard deviation of the runtimes.

Chapter 4
Results

As previously stated, the goal of this research was to conduct a comparative analysis of glibc and musl across several key performance metrics. The findings from the performed experiments are presented in this section.

4.1 Numerical accuracy of glibc and musl

Two different tests were used to confirm and compare the numerical accuracy of glibc and musl, Tybor and PARANOIA. The results from these tests are presented below and the findings show that both glibc and musl have a reliable numerical accuracy that is in accordance with the IEEE 754 standard.

4.1.1 Tybor

In order to use the appropriate test available in Tybor to test the accuracy of the resulting bit patterns, the mathematically correct values for the numerical functions were first found using matlab. The correct output with a precision of 15 decimal digits can be seen in table 4.1.

The resulting bit errors for the numerical functions can be seen in table 4.2, and the results show that all tests passed and that both glibc and musl resulted in the same bit pattern with an accuracy of at least 53 bits, resulting in a 0 bit error. Therefore, all 64 bits in the resulting `double` were shown to be correct for both musl and glibc for every test.

Table 4.1: Table showing the correct values for numerical functions with a precision of 15 decimal digits.

Function	Argument	Correct output with a precision of 15 decimal digits
sin	355	-3.014435335948845e-5
	472	0.689719768063795
atan	355	1.567979432837052
	-55	-1.552616511721918
sqrt	7947	89.145947748621751
	0.007	0.083666002653408
log	0.005	-5.298317366548036
	234	5.455321115357702

Table 4.2: Table showing the resulting bit errors out of 53 bits when testing numerical functions with Tybor.

Function	Argument	Bit errors when using glibc	Bit errors when using musl
sin	355	0	0
	472	0	0
atan	355	0	0
	-55	0	0
sqrt	7947	0	0
	0.007	0	0
log	0.005	0	0
	234	0	0

4.1.2 paranoia.c

While running the paranoia.c test, the result were the same for both libraries and no failures, defects or flaws were detected. The resulting output printed ended with the following lines for both glibc and musl:

```
No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754.
The arithmetic diagnosed appears to be Excellent!
```

4.2 Runtime comparisons using timebase

In table 4.3 we can see a comparison of runtimes in seconds for the same program when using glibc and musl respectively. Each program contains only one of the functions, `sin` for example, which is run a number of times with different arguments. Since the exact same program was compiled with glibc and musl respectively, the number of iterations stay the same when using both libraries. glibc turned out to be faster than musl in all tests except for two, `memcpy` and `strcpy`.

Table 4.3: Table showing the result of running the same function a number of iterations with different arguments in order to see the runtime for glibc and musl.

Function	Runtime using glibc (s)	Runtime using musl (s)	Number of iterations
sin	4.38	27.31	100 000 000
atan	26.62	31.93	100 000
sqrt	50.30	50.53	10 000 000 000
log	19.65	23.13	1 000 000 000
memcpy	13.35	10.17	1 000 000 000
strcpy	14.70	10.59	1 000 000 000
strstr	3.37	11.31	1 000 000 000

4.3 Memory usage comparisons using size

This section presents the source code files and the object files needed to run the chosen functions. The result from the Linux `size` command is therefore presented in connection with the object files, and these are the sizes compared at the end of the section.

4.3.1 sin

The source code files needed to run `sin` using glibc and musl respectively can be seen in table 4.4, and the corresponding object files and their sizes can be seen in table 4.5. We can see that musl uses a larger number of object files, but the memory size required is larger for glibc.

Table 4.4: Table showing the source code files needed to run sin.

Library	File name
glibc	`s_sin.c`
musl	`__rem_pio2_large.c`
	`__rem_pio2.c`
	`scalbn.c`
	`__sin.c`
	`sin.c`
	`__cos.c`

Table 4.5: Table showing the object files needed to run sin.

Library	text	data	bss	dec	hex	filename
glibc	4944	16	0	4960	1360	`s_sin.o`
	3520	0	0	3520	dc0	`sincostab.o`
	1868	0	0	1868	74c	`branred.o`
musl	3124	0	0	3124	c34	`__rem_pio2_large.lo`
	1236	0	0	1236	4d4	`__rem_pio2.lo`
	180	0	0	180	b4	`scalbn.lo`
	260	0	0	260	104	`__sin.lo`
	344	0	0	344	158	`sin.lo`
	248	0	0	248	f8	`__cos.lo`

4.3.2 atan

The source code files needed to run `atan` using glibc and musl respectively can be seen in table 4.6, and the corresponding object files and their sizes can be seen in table 4.7. We can

see that only one object file is needed for each library, but that glibc requires more memory than musl.

Table 4.6: Table showing the source code files needed to run atan.

Library	File name
glibc	`s_atan.c`
musl	`atan.c`

Table 4.7: Table showing the object files needed to run atan.

Library	text	data	bss	dec	hex	filename
glibc	14 700	8	0	14 708	3974	`s_atan.o`
musl	864	0	0	864	360	`atan.lo`

4.3.3 sqrt

The source code files needed to run `sqrt` using glibc and musl respectively can be seen in table 4.8, and the corresponding object files and their sizes can be seen in table 4.9. We can see that glibc requires more source code files, object files, and a larger amount of memory. It is noteworthy that `sqrt` requires a small amount of memory as the function is implemented directly in hardware.

Table 4.8: Table showing the source code files needed to run sqrt.

Library	File name
glibc	`e_sqrt.c` `w_sqrt_template.c`
musl	`sqrt.c`

4.3.4 log

The source code files needed to run `log` using glibc and musl respectively can be seen in table 4.10, and the corresponding object files and their sizes can be seen in table 4.11. We can see that glibc requires more source code files, object files, and memory.

Table 4.9: Table showing the object files needed to run sqrt.

Library	text	data	bss	dec	hex	filename
glibc	60	0	0	60	3c	`e_sqrt.o`
	180	0	0	180	b4	`w_sqrt.o`
musl	20	0	0	20	14	`sqrt.lo`

Table 4.10: Table showing the source code files needed to run log.

Library	File name
glibc	`e_log.c`
	`w_log_template.c`
musl	`log.c`

Table 4.11: Table showing the object files needed to run log.

Library	text	data	bss	dec	hex	filename
glibc	180	0	0	180	b4	`w_log.o`
	72	8	0	80	50	`e_log.o`
	732	8	0	740	2e4	`e_log-ppc64.o`
	732	0	0	732	2dc	`math_err.o`
	2 192	0	0	2 192	890	`e_log_data.o`
musl	772	0	0	772	304	`log.lo`

4.3.5 memcpy

The source code files needed to run `memcpy` using glibc and musl respectively can be seen in table 4.12, and the corresponding object files and their sizes can be seen in table 4.13. We can see that glibc and musl both use one source code file and one object file each. The memory required is also similar, but glibc uses less. Another notable point is that glibc uses assembly code, as indicated by the '.S' file extension.

Table 4.12: Table showing the source code files needed to run memcpy.

Library	File name
glibc	`memcpy.S`
musl	`memcpy.c`

Table 4.13: Table showing the object files needed to run memcpy.

Library	text	data	bss	dec	hex	filename
glibc	1 004	0	0	1 004	3ec	`memcpy-ppc64.o`
musl	1 160	0	0	1 160	488	`memcpy.lo`

4.3.6 strcpy

The source code files needed to run `strcpy` using glibc and musl respectively can be seen in table 4.14, and the corresponding object files and their sizes can be seen in table 4.15. We can see that, although both libraries use the same amount of source code files and object files, glibc requires more memory.

Table 4.14: Table showing the source code files needed to run strcpy.

Library	File name
glibc	`stpcpy.c`
	`strcpy.c`
musl	`stpcpy.c`
	`strcpy.c`

4.3.7 strstr

The source code files needed to run `strstr` using glibc and musl respectively can be seen in table 4.16, and the corresponding object files and their sizes can be seen in table 4.17. We can see that glibc uses assembly code for `strstr` as well, and that glibc require more memory.

Table 4.15: Table showing the object files needed to run strcpy.

Library	text	data	bss	dec	hex	filename
glibc	444	0	0	444	1bc	`stpcpy.o`
	124	0	0	124	7c	`strcpy-ppc64.o`
musl	172	0	0	172	ac	`stpcpy.lo`
	72	0	0	72	48	`strcpy.lo`

Table 4.16: Table showing the source code files needed to run strstr.

Library	File name
glibc	`strstr.c`
	`memcmp.S`
	`strlen.S`
	`strnlen.S`
	`strchr.S`
musl	`strstr.c`
	`memchr.c`
	`strchrnul.c`
	`strchr.c`
	`memcmp.c`

Table 4.17: Table showing the object files needed to run strstr.

Library	text	data	bss	dec	hex	filename
glibc	2 696	0	0	2 696	a88	strstr.o
	4 284	0	0	4 284	10bc	memcmp-power8.o
	836	0	0	836	344	strnlen-power8.o
	280	0	0	280	118	strchr-ppc64.o
	276	0	0	276	114	strlen-ppc64.o
musl	1 404	0	0	1 404	57c	strstr.lo
	264	0	0	264	108	memchr.lo
	256	0	0	256	100	strchrnul.lo
	92	0	0	92	5c	strchr.lo
	68	0	0	68	44	memcmp.lo

4.3.8 Total memory size needed for each function

The individual sizes of each object file needed for each function has been presented in previous sections and in table 4.18 the summation of all object files for each function can be seen. The results show that musl generally requires the smallest amount of memory.

Table 4.18: Table showing the total memory size needed for each function.

Function	Memory size using glibc	Memory size using musl
sin	10 348	5 392
atan	14 708	864
sqrt	240	20
log	3 924	772
memcpy	1 004	1 160
strcpy	568	244
strstr	8 372	2 084

4.3.9 Memory size of malloc

The size of `malloc.o` for glibc and the size of `malloc.lo` and `free.lo` for musl can be seen below.

glibc

```
$ size malloc.o
text     data      bss      dec      hex    filename
44142    2392      96     46630    b626   malloc.o
```

musl

```
$ size malloc.lo
text    data     bss     dec     hex    filename
6075     0       928     7003    1b5b   malloc.lo

$ size free.o
text    data     bss     dec     hex filename
2500       0       0    2500     9c4 free.lo
```

4.4 Larger programs and benchmarks

This section presents the results when running larger programs, PostgreSQL and SPEC CPU2006, with glibc and musl.

4.4.1 PostgreSQL

The results from running two different versions of PostgreSQL with glibc and musl are shown in this section. When running PostgreSQL it was evident that using only musl was not possible, therefore three additional directories needed to be included.

postgres-12

glibc: Compiled without issues and passed all 192 tests.

musl: In order to compile with gcc-musl, three directories needed to be included with the following commands:

```
cp -r /usr/include/linux /opt/musl/1.2.5/include
cp -r /usr/include/powerpc64le-linux-gnu/asm /opt/musl/1.2.5/include
cp -r /usr/include/asm-generic /opt/musl/1.2.5/include
```

This still resulted in two errors out of 192.

postgres-17

glibc: Compiled without issues and passed all 221 tests.

musl: The same three directories that were included when running postgres-12 were included when running this version, this resulted in zero failed tests out of 221.

pgbench (17.0) When postgres-17 was compiled a program called `pgbench` which counts the number of transactions per second (tps) was created. The resulting speed for glibc was around 2 580 tps and the speed for musl was around 2 415 tps, showing that glibc is faster.

4.4.2 008.espresso

From running the 008.espresso benchmark it was estabilised that the amonut of samples for malloc and free are much higher for musl than for glibc, the results can be seen below. The results also showed that malloc and free in musl are slower than malloc in glibc.

glibc

```
samples  %       linenr info  symbol name
114      2.9156  malloc.o:0   _int_malloc
97       2.4808  malloc.o:0   _int_free
51       1.3043  malloc.o:0   malloc_consolidate
```

musl

```
samples  %       linenr info  image name     symbol name
172      3.8861  malloc.c:0   musl-espresso  alloc_slot
166      3.7506  free.c:0     musl-espresso  nontrivial_free
109      2.4627  free.c:0     musl-espresso  get_meta
```

4.4.3 SPEC CPU2006

When running the SPEC CPU2006 benchmark suite seven times, only one serious performance degradation with musl was discovered, but in general, they perform similarly for the benchmarks run with glibc. The resulting mean, median and standard deviation of the runtimes are shown in table 4.19 and 4.21. Table 4.19 contains benchmarks testing floating point performance, while table 4.21 contains benchmarks testing integer performance. Table 4.20 and 4.22 show the difference in percentage for the mean runtimes when using glibc and musl. We can see that there are differences in some benchmarks of up to 37.0 % for `464.h264ref` and it is therefore advisable to compare the libraries first for critical applications.

Table 4.19: Table showing the mean, median and standard deviation from running SPEC CPU2006 floating point benchmarks seven times.

Benchmark	Library	Mean (s)	Median (s)	Standard deviation (s)
433.milc	glibc	503.7	502	3.4
	musl	518.0	515	3.8
470.lbm	glibc	253.4	253	0.7
	musl	253.6	253	0.9
482.sphinx3	glibc	508.9	508	1.4
	musl	510.0	509	1.4

Table 4.20: Table showing the difference in runtimes for SPEC CPU2006 when using musl and glibc.

Benchmark	Runtime difference (%)
433.milc	3.0
470.lbm	0.1
482.sphinx3	0.2

Table 4.21: Table showing the mean, median and standard deviation from running SPEC CPU2006 integer benchmarks seven times.

Benchmark	Library	Mean (s)	Median (s)	Standard deviation (s)
401.bzip2	glibc	549.3	549	0.5
	musl	538.3	538	0.5
403.gcc	glibc	239.4	239	0.5
	musl	284.9	285	0.8
429.mcf	glibc	181.3	181	1.2
	musl	185.6	186	2.1
445.gobmk	glibc	482.3	482	0.5
	musl	485.3	485	0.5
456.hmmer	glibc	258.3	258	0.5
	musl	290.3	290	0.5
458.sjeng	glibc	640.6	640	1.8
	musl	642.7	642	2.1
462.libquantum	glibc	285.6	289	6.6
	musl	283.3	282	6.8
464.h264ref	glibc	560.6	560	0.9
	musl	767.6	767	0.7

Table 4.22: Table showing the difference in runtimes for SPEC CPU2006 when using musl and glibc.

Benchmark	Runtime difference (%)
401.bzip2	-1.9
403.gcc	19.0
429.mcf	2.4
445.gobmk	0.6
456.hmmer	12.4
458.sjeng	0.3
462.libquantum	-0.8
464.h264ref	37.0

The benchmark `464.h264ref` is composed of a number of programs and the results below were acquired when the most time consuming part was run manually.

Results when using musl:

```
operf ./h264ref_base.cc -d sss_encoder_main.cfg
```

```
9441596  41.5098  memcpy.c:6          libc.so          memcpy
5022692  22.0821  mv-search.c:306     h264ref  SetupFastFullPelSearch
2242940  9.8610   mv-search.c:948     h264ref  FastFullPelBlockMotionSearch
1288465  5.6647   mv-search.c:1284    h264ref  SubPelBlockMotionSearch
593664   2.6100   mv-search.c:1017    h264ref  SATD.part.0
586718   2.5795   block.c:876         h264ref  dct_luma
539269   2.3709   refbuf.c:44         h264ref  FastLine16Y_11
446958   1.9650   refbuf.c:169        h264ref  FastPelY_14
280890   1.2349   biariencode.c:164   h264ref  biari_encode_symbol
```

Results when using glibc:

```
4945309  31.4922  mv-search.c:306     h264ref       SetupFastFullPelSearch
2604418  16.5852  (no location info)  libc-2.31.so  __memcpy_power7
2206912  14.0538  mv-search.c:948     h264ref       FastFullPelBlockMotionSearch
1294800  8.2454   mv-search.c:1284    h264ref       SubPelBlockMotionSearch
596041   3.7956   mv-search.c:1017    h264ref       SATD.part.0
585873   3.7309   block.c:876         h264ref       dct_luma
547638   3.4874   refbuf.c:44         h264ref       FastLine16Y_11
442046   2.8150   refbuf.c:169        h264ref       FastPelY_14
281091   1.7900   biariencode.c:164   h264ref       biari_encode_symbol
198867   1.2664   refbuf.c:49         h264ref       UMVLine16Y_11
181519   1.1559   mb_access.c:629     h264ref       getLuma4x4Neighbour
```

This result shows that the use of `memcpy` is the reason why it takes longer to run the benchmark with musl than with glibc.

Chapter 5
Discussion

This section provides an analysis of the results gathered during the testing phase of this research and suggestions for future research are also presented.

5.1 Numerical accuracy

As demonstrated in the results section, musl performed equally as well as glibc in terms of numerical accuracy and rounding. Both musl and glibc conformed to the IEEE 754 standard for all test cases during the performed tests for numerical accuracy. This result is significant as it is the first step to showing that musl is a reliable C standard library.

As previously mentioned, the mathematical functions used when testing for numerical accuracy were limited to `sin`, `atan`, `sqrt`, and `log`, and only two arguments per function were used when testing the numerical accuracy with Tybor. The chosen mathematical functions exhibit a smooth and continuous behaviour across their domain, and therefor it may not be necessary to test an excessive amount of arguments. And although all eight test cases confirmed that musl has high accuracy, it is important to note that the number of test cases were very limited. Therefore, a broader range of arguments and functions would provide stronger evidence of musl's reliability. Trigonometric and logarithmic functions have certain special arguments which could possibly have been useful when determining the numerical accuracy, as these special cases of arguments are close to boundary conditions and could have given unpredictable results. For trigonometric functions, $\pi/2$ is one such example, and for logarithmic functions, 0 is one. But testing with that many arguments could have been an unnecessarily time consuming task, and therefore the chosen arguments were picked at random in the function's domain as representative. As this was only the first step in verifying that the numerical functions did not result in any bit errors for at least 53 bits, it was deemed enough.

As illustrated in the theory chapter, a `double` variable in the C language has 64 bits. This

standard includes 1 sign bit, 11 exponent bits, and 52 mantissa bits, and it conforms to the IEEE 754 standard for floating-point arithmetic. Even though a `double` variable in C has 64 bits, only a 53 bit precision is necessary to guarantee good results. This is because the term '53 bits' refers to the number of significant bits in the mantissa, not the total bit length of the number. All of the numerical functions that were tested only accept the type `double`. The choice to only test functions with the type `double` as input was chosen because it is the most common datatype used and default when doing numerical calculations. Therefore, double can be seen as the most important type, but for higher accuracy, other data types such as long double could have been tested. float could have been tested as well since its precision is sufficient in some computations.

PARANOIA uses a wide range of arguments and tests for a wide variety of floating-point errors, including undeflow, overflow, and incorrect rounding. The program also checks if there is any precision loss. Because of the wide variety of tests performed by PARANOIA, Tybor was partly used as additional verification to see if Tybor and PARANOIA would agree on musl and glibc's numerical accuracy.

In conclusion, both musl and glibc are equally reliable when it comes to numerical accuracy, and this should not be a deciding factor when choosing a library.

5.2 Memory usage

Table 4.18 displays the memory used, that is, how many bytes that end up in RAM, by glibc and musl respectively when compiling the chosen functions. The sizes shown are sums of all the object files memory requirements needed when compiling a function, and these files vary depending on the library used. When comparing the resulting memory sizes, it is clear that musl has a smaller memory footprint, which is one of the advantages of musl according to its creators.

The developers of musl also claim that the size of the global data used in musl is minimized, which is a claim that seems to be accurate. The global data is stored in the data segment, and as evidenced by the tables in the result section, musl consistently uses a smaller amount of memory in the data segment than glibc. The bss section is zero for all functions when using both musl and glibc, but there is a difference in the text segment size for several functions, which is where most of the memory used is located. Therefore, musl generally performs better than glibc in minimizing memory used in both memory segments.

Both segments are important for embedded systems and systems with memory constraints, and reducing the size of each can help optimize memory usage. In general-purpose applications, the text segment and data segment are more important, but the bss section size can still have an impact on overall runtime memory usage. While the tests performed during the testing did not stress the memory limits of the used computer, a reduced memory usage can be important when memory limitations need to be taken into account. Therefore, musl could be the preferable choice when needing basic functionality and a small memory size.

Unnecessary memory usage could also slow down the system, but results show that glibc in general was faster than musl.

Another difference observed when looking at the object flies used was that musl consistently only uses C code in their library files, while glibc while implementing some functions uses assembly code. This can be seen in the tables, `.c` files mean that C code is used, and `.S` files indicate that assembly code is used. The fact that some functions were implemented in assembly code does not appear to have a positive effect on the memory size used for those functions.

It can be noted that some object files, including the files used when running `sqrt`, were significantly smaller than the object files used by other functions. The reason for this is because IBM POWER completely implements these functions in hardware.

5.3 Runtimes

When examining the results of the runtime comparisons, glibc generally turned out to be the faster choice. The runtime difference varied greatly depending on the function, but only two of the examined functions runtimes were found to be faster when using musl. It is not unexpected that a smaller memory footprint did not result in shorter runtimes, as the memory footprint only becomes affected if there are enough instruction and L2 cache misses.

5.4 Larger programs and benchmarks

After examining numerical accuracy, runtimes, and memory usage above, we can see that musl is up to the same standard as glibc, and therefore no convincing reason to choose either library has yet been found. However, both libraries have been shown to have different strengths in terms of memory usage and runtimes. One thing that all of the previous tests have in common is that they have been performed on small, simple programs. In order to find out if running musl with a larger and more complex program would cause any unexpected problems, both libraries were used when running larger programs: PostgreSQL and C programs from SPEC CPU2006. The results showed that musl is reliable and robust enough to handle many larger programs; however, some directories needed to be added to musl in order for it to work as well as glibc. Therefore, glibc could be the preferable choice for larger programs.

When musl and glibc were used when running the benchmark SPEC CPU2006, no significant difference in performance for most benchmarks was observed, but glibc was shown to be faster. The largest difference in performance was found for the benchmark `464.h264ref`. The reason for this difference is most likely the faster implementation of `memcpy` in assembly code used in glibc. musl's implementation of memcpy is complicated and the source code can be seen in appendix B. The source code used in glibc is not included in this report as it is not easily readable, but it is of note that it uses SIMD-instructions which results in higher

performance. As the SPEC benchmarks are frequently used in the software industry, confirming that musl could perform well enough to run SPEC CPU2006 is a significant result. This means that musl could potentially be used in the same way that glibc has been used in real-world applications, which would be critical in real-world software development.

A more apparent difference was observed when running two different versions of PostgreSQL with glibc and musl; the versions tested were postgres-12 and postgres-17. When glibc was used, both versions tested resulted in no compilation complications or failed tests. However, when musl was used, some adjustments had to be made when compiling both versions of PostgreSQL to make musl compatible. As we can see in the results section, both PostgreSQL versions require you to include three different directories in order to compile with musl. This resulted in postgres-17 working without issues, but two tests still failed when running postgres-12. Although running postgres-17 appears to have been successful after including these three directories, it is not certain that the included directories are completely compatible with musl. However, the absence of errors suggests that at least the parts of the directories that were utilized seem to be compatible. This is reasonable, as POSIX defines a common interface, and Linux complies with this standard while also including additional features not included in POSIX. C standard libraries used by the software industry also often need to comply with these POSIX standards, making it likely for the directories to be compatible with musl.

By using the program pgbench, we could determine the number of transactions per second resulting from running postgre-17 with musl and glibc. With the resulting rates, it was determined that glibc outperformed musl, with a rate of 2 580 tps compared to musl's rate of 2 415 tps, which is a difference that could have an impact in real-time applications. While the reason for this rate difference has not been definitely determined, the use and speed of `malloc` and `free` could be partly responsible. Running the 008.espresso benchmark revealed that the number of samples for malloc and free were somewhat higher for musl compared to glibc. However, since `malloc` and `free` are some of the easiest methods for the programmer to replace with another implementation if they want, this might not be enough to persuade the programmer to choose glibc over musl.

5.5 Additional considerations

There are other considerations to take into account when deciding between musl and glibc, and since these factors were not noticeable during testing, they have not yet been discussed. One advantage of choosing musl is that you can easily use compilers other than GCC, while glibc would require source code changes in order to be compatible with a compiler like clang, which is a popular choice. Although no peer-reviewed articles testing if glibc can be used with clang could be found, it has been suggested online that they are not compatible, which was confirmed during this research. glibc is also dependent on GNU C extensions, making it the less versatile option. Whole-program compiler optimization can provide more efficient programs that result in shorter runtimes and less energy being used, and it is convenient to be able to use other compilers than GCC, like clang, to do this [21].

Another advantage of using musl is that its source code is smaller, easier to navigate, and easier to understand. Programmers who wish to easily add new functions to their C standard library could therefore benefit from using musl.

5.6 Future Work

During this thesis, many differences in the performance of musl and glibc have been discovered, but a detailed analysis of the reasons why the functions differ is outside the scope of this report and could be considered for future work.

Another topic that could be considered for future work is whether glibc could be replaced by musl in order to decrease the energy used by data centers. Since compilers other than GCC are able to compile musl, it would be beneficial if a more advanced compiler than GCC could be used for whole-program optimization in data centers. Better compiler optimizations could lead to faster programs, which leads to having to use the CPU for a shorter amount of time, and therefore less energy is used. This could be beneficial not only for companies, but also for the environment.

If you want to use inlining from libraries, including C standard libraries, but you wish not to use GCC and prefer not to modify glibc's source code, then musl is the only choice of library. This can result in even smaller files which can save disk space and memory, resulting in more efficient products. This could also be evaluated in future work.

Another topic that could be explored is the implementation of memcpy adapted to CPU-pipeline details, similar functions including memmove, memcmp, and strcpy could also be included. It could also be interesting to rewrite memcpy as a trivial for-loop and let an optimizing compiler generate the fastest version possible.

Chapter 6
Conclusion

This section presents concise answers to the research questions that were outlined in the introduction. These answers are based on the results and discussions presented in the previous sections.

RQ1 How does the performance differ between glibc and musl?

When examining the three areas of interest within the scope of this thesis, the performance differences between glibc and musl were easily identifiable. The results from examining the numerical accuracy of musl and glibc showed no disparity, and both libraries were reliable. However, glibc has consistently resulted in shorter runtimes, and musl has required less memory. When running larger programs, glibc is the more reliable option, but musl could be used with some modifications.

RQ2 Under which circumstances would musl be preferable to glibc, and vice versa?

When deciding which C standard library to use, it is important to consider the available resources and priorities. It would probably be preferable to choose musl when memory resources are limited or when greater flexibility is desired in terms of compiler compatibility. musl might also be the preferred choice if you are looking for a library with a more readable source code that can be more easily altered. However, if you are searching for a library that shortens the runtime required for both larger and smaller programs without whole-program optimization, then glibc could be the better choice. glibc should also be chosen if you require a library with more advanced features on Linux. For larger programs, glibc could also be the preferable choice due to its reliability without modifications.

The results of this research are significant since they show that musl is a viable option in

many situations, and would in some cases be the preferable choice over glibc. Individuals considering a transition to musl may therefore find this research a compelling reason to explore its use.

References

[1] URL: https://www.gnu.org/software/libc/.

[2] URL: https://musl.libc.org/.

[3] URL: https://www.iso.org/home.html.

[4] URL: https://www.iso.org/standard/82075.html.

[5] URL: https://sourceware.org/glibc/manual/latest/html_mono/libc.html#Introduction.

[6] URL: https://musl.libc.org/about.html.

[7] URL: https://man7.org/linux/man-pages/man1/size.1.html.

[8] URL: https://linux.die.net/man/1/nm.

[9] URL: https://standards.ieee.org/ieee/754/6210/.

[10] URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html.

[11] URL: http://www.tybor.com/.

[12] URL: https://people.math.sc.edu/Burkardt/c_src/paranoia/paranoia.html.

[13] URL: https://www.postgresql.org/about/.

[14] URL: https://www.spec.org/.

[15] URL: https://www.spec.org/cpu2006/.

[16] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. "A dynamic program analysis to find floating-point accuracy problems." In: *Proceedings of the 33rd ACM SIGPLAN Conference Programming Language Design amp; Implementation* (2012), pp. 453–462. ISSN: 9781450312059.

[17] Jens Gustedt. *Modern C*. Manning, 2024. URL: https://inria.hal.science/hal-02383654.

[18] Ruishi Li et al. “The inconsistency of documentation: a study of online C standard library documents”. In: *Cybersecurity* 5.1 (July 2022), p. 14. ISSN: 2523-3246. DOI: 10.1186/s42400-022-00118-9. URL: https://doi.org/10.1186/s42400-022-00118-9.

[19] Hongyuan Qi, Jinchen Xu, and Shaozhong Guo. “Detection of the maximum error of mathematical functions”. In: *J. Supercomput.* 74.11 (Nov. 2018), pp. 6275–6290. ISSN: 0920-8542. DOI: 10.1007/s11227-018-2552-x. URL: https://doi.org/10.1007/s11227-018-2552-x.

[20] Anton Rydahl et al. “Precision and Performance Analysis of C Standard Math Library Functions on GPUs”. In: *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 892–903. ISBN: 9798400707858. DOI: 10.1145/3624062.3624166. URL: https://doi.org/10.1145/3624062.3624166.

[21] Jonas Skeppstedt. *Personal communication*. Personal communication. 2025.

[22] Jonas Skeppstedt and Christian Söderberg. *Writing efficient C code: a thorough introduction*. Amazon, 2020. ISBN: 978-1659599206.

[23] B. Verdonk, A. Cuyt, and D. Verschaeren. “A precision- and range-independent tool for testing floating-point arithmetic. I. Basic operations, square root, and remainder.” In: *ACM Transactions on Mathematical Software* 27.1 (2001), pp. 92–118.

[24] Qiang Zou et al. “Temporal characterization of SPEC CPU2006 workloads: Analysis and synthesis.” In: *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC), Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International* (2012), pp. 11–20. ISSN: 978-1-4673-4881-2.

Appendices

Appendix A

run

```
rm -f a.out
gcc -g -static atan_tb.c timebase.c tbr.s s_atan.o -o a
operf ./a

opreport -t 1 -l -g a
opannotate -s -a a > glibc.annotate

rm -f a.out
musl-gcc -g -static atan_tb.c timebase.c tbr.s -lm -o b
operf ./b

opreport -t 1 -l -g b
opannotate -s -a b > musl.annotate
```

Appendix B

musl's memcpy implementation

```
#include <string.h>
#include <stdint.h>
#include <endian.h>

void *memcpy(void *restrict dest, const void *restrict src, size_t n)
{
 unsigned char *d = dest;
 const unsigned char *s = src;

#ifdef __GNUC__

#if __BYTE_ORDER == __LITTLE_ENDIAN
#define LS >>
#define RS <<
#else
#define LS <<
#define RS >>
#endif

    typedef uint32_t __attribute__((__may_alias__)) u32;
    uint32_t w, x;

    for (; (uintptr_t)s % 4 && n; n--) *d++ = *s++;

    if ((uintptr_t)d % 4 == 0) {
        for (; n>=16; s+=16, d+=16, n-=16) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
```

```
            *(u32 *)(d+4) = *(u32 *)(s+4);
            *(u32 *)(d+8) = *(u32 *)(s+8);
            *(u32 *)(d+12) = *(u32 *)(s+12);
        }
        if (n&8) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
            *(u32 *)(d+4) = *(u32 *)(s+4);
            d += 8; s += 8;
        }
        if (n&4) {
            *(u32 *)(d+0) = *(u32 *)(s+0);
            d += 4; s += 4;
        }
        if (n&2) {
            *d++ = *s++; *d++ = *s++;
        }
        if (n&1) {
            *d = *s;
        }
        return dest;
    }

    if (n >= 32) switch ((uintptr_t)d % 4) {
    case 1:
        w = *(u32 *)s;
        *d++ = *s++;
        *d++ = *s++;
        *d++ = *s++;
        n -= 3;
        for (; n>=17; s+=16, d+=16, n-=16) {
            x = *(u32 *)(s+1);
            *(u32 *)(d+0) = (w LS 24) | (x RS 8);
            w = *(u32 *)(s+5);
            *(u32 *)(d+4) = (x LS 24) | (w RS 8);
            x = *(u32 *)(s+9);
            *(u32 *)(d+8) = (w LS 24) | (x RS 8);
            w = *(u32 *)(s+13);
            *(u32 *)(d+12) = (x LS 24) | (w RS 8);
        }
        break;
    case 2:
        w = *(u32 *)s;
        *d++ = *s++;
        *d++ = *s++;
        n -= 2;
        for (; n>=18; s+=16, d+=16, n-=16) {
```

```
            x = *(u32 *)(s+2);
            *(u32 *)(d+0) = (w LS 16) | (x RS 16);
            w = *(u32 *)(s+6);
            *(u32 *)(d+4) = (x LS 16) | (w RS 16);
            x = *(u32 *)(s+10);
            *(u32 *)(d+8) = (w LS 16) | (x RS 16);
            w = *(u32 *)(s+14);
            *(u32 *)(d+12) = (x LS 16) | (w RS 16);
        }
        break;
    case 3:
        w = *(u32 *)s;
        *d++ = *s++;
        n -= 1;
        for (; n>=19; s+=16, d+=16, n-=16) {
            x = *(u32 *)(s+3);
            *(u32 *)(d+0) = (w LS 8) | (x RS 24);
            w = *(u32 *)(s+7);
            *(u32 *)(d+4) = (x LS 8) | (w RS 24);
            x = *(u32 *)(s+11);
            *(u32 *)(d+8) = (w LS 8) | (x RS 24);
            w = *(u32 *)(s+15);
            *(u32 *)(d+12) = (x LS 8) | (w RS 24);
        }
        break;
    }
    if (n&16) {
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    }
    if (n&8) {
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    }
    if (n&4) {
        *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
    }
    if (n&2) {
        *d++ = *s++; *d++ = *s++;
    }
    if (n&1) {
        *d = *s;
    }
    return dest;
```

```
#endif

    for (; n; n--) *d++ = *s++;
    return dest;
}
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2025-03-28

EXAMENSARBETE An Analysis of the Difference in Performance When Using the C Standard Libraries glibc and musl
STUDENT Rebecka Källén
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Per Andersson (LTH)

glibc vs. musl: Is newer really better?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Rebecka Källén**

Think of a C programmer like a carpenter; they both need the right tools to work efficiently. And just like you can't expect every carpenter to craft their own hammer, every C programmer can't be expected to write their own basic functions from scratch.

That's where the C standard library comes in, offering essential functions, like calculating the square root of a number, so that programmers can focus on more complex tasks.

But with several C libraries available, how do you choose the right one? The answer depends on your specific needs and preferences, which makes it important to compare the libraries available. The goal of this research was therefore to compare two libraries, glibc and musl, to make it easier for programmers to choose based on their needs.

glibc is a very popular library that was released in 1998, but since it has been around for a long time, it has a lot of baggage that some programmers want to avoid. musl, on the other hand, was released more recently in 2011, and the developers of musl make a lot of promises about its performance. But how does it actually compare to glibc?

Three aspects of the performance of glibc and musl were analyzed during the comparisons: the memory required, speed, and how accurate their numerical functions are. The reliability of musl was also evaluated with some larger programs. During this research, I discovered that both musl and glibc are reliable, but they work best in different situations. musl required less memory, but glibc was faster. glibc was more reliable for large programs, but the numerical accuracy, on the other hand, was similar for both musl and glibc. Therefore, programmers should consider the available resources and priorities before deciding which library to use.