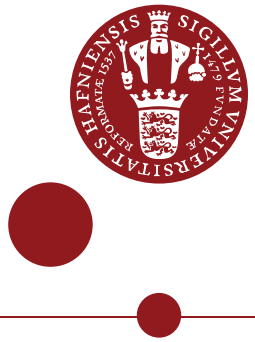
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



Bachelors Thesis

Simon Vinding Brodersen

RISC-V based computers in the data center

Advisor: Philippe Bonnet

June 9, 2024

Contents

1 Introduction

1.1 Context

Data centers are becoming increasingly essential in the IT sector. Whether it is Google's cloud platform, Microsoft Azure, or Amazon's web services, news about new data centres seems like a daily occurrence[6]. With such scale comes an ever-growing need for custom solutions and cutting-edge technologies to both reduce power consumption and improve overall performance.

Historically, solid-state drives (SSDs) served as drop-in replacements for magnetic disks, utilizing similar interfaces to ensure seamless integration. But the use of SSDs came with multiple improvements over the magnetic disks of the past, which were hindered by said interface. Furthermore, flash SSDs were subject to endurance problems, and issues with wear leveling emerged[4, 20]. As such, there was a rapid movement towards Open-channel SSDs that do not have a firmware Flash Translation Layer(FTL) and instead leave the management of the physical SSD to the computer's operating system[19, 15]. This allowed for greater utilization of the SSD, but introduces further data transferring between the Central Processing Unit(CPU) and the SSD. This increases the discrepancy between the amount of memory that the CPU uses to analyze data and the memory it has available[3].

A solution to the problem would be to offload the CPU and provide computation at the SSD level. Such a solution has been described as a computational storage device (CSD). This would involve implementing the most commonly used data manipulations, such as indexing into an SSD or more complex manipulations like sorting. Within this thesis, the issue of implementing a high-performance investigated. Sorting algorithm running on a stand-alone bare-metal[1] processor has been been investigated.

1.2 Problem

For computational storage to be a viable solution for meeting the ever-growing demand for massive data computations, it is essential to investigate whether implementing a processor designed for such a purpose is feasible. Consequently, at least two open questions remain unanswered. (1) What type of computation should be performed by a storage device? (2) Is it possible to implement such computation on a bare-metal processor?

1. **What computation should be handled by a storage device?**

[1] See Section 3.2.

Among the various instances of large data transfers between a CPU and an SSD, sorting a given array stands out as particularly significant. Sorting is fundamental in numerous programming scenarios, serving as a core component of many search algorithms and playing a critical role in data science. Efficient sorting is essential for optimal performance. Due to its time complexity of O(n log n), merge sort was selected for further investigation. Furthermore, parallel implementations of the merge sort algorithm should be feasible on bare-metal.

2. **Is it feasible to implement such a computation on a bare metal RISC-V processor?**
   As the main goal is to offload the primary CPU, it must be investigated whether it is at all possible to create a sorting algorithm on a bare-metal processor.

1.3 Approach

For this thesis, an experimental approach was taken. First, a feasible design for implementing on a bare metal processor is introduced. Secondly, an implementation of said design is presented. Third, the viability and validity of the implementation is evaluated. Lastly, shortcomings and proposed further research are presented. These implementations will be carried out on a QEMU virtual machine where the code is loaded via a general loader. The QEMU virtual machine is compatible with the standard RISC-V ISA. However, a few standard extensions are required for the implementation these including the Control and Status Register (Zicsr), Atomic Instructions(a), Integer Multiplication and Division (m) and Compressed Instructions(c) [21].

1.4 Contribution

For this thesis, the following contributions have been made:

1. Present available design patterns when developing a computational storage device
2. Design and implement a specific merge sort algorithm meant for running on a bare-metal processor as described in Section 3.2.
3. Evaluate the implementation on lists of varying sizes.
4. Evaluate the viability of custom bare-metal applications for later use as a Computational Storage Device.

2 Background

2.1 Accelerator-based Computer Architecture

The notion of offloading has long been established in specialized teams, where each member focuses on their area of expertise. This concept seems inherently logical when discussing day-to-day work environments. Effective communication between entities, with an emphasis on performing tasks best suited to our skills, appears to be the foundation of efficient collaboration. Contrary, computer architecture relies heavily on the CPU for executing various operations. An accelerator serves as a separate substructure designed with distinct objectives compared to the CPU itself. By offloading the CPU, accelerators can optimize performance and reduce energy consumption[16]. A prime example of an accelerator is the Graphics Processing Unit (GPU), a crucial component in contemporary computers. This thesis aims to explore the feasibility of adopting a similar design approach for creating computational storage devices.

2.2 RISC-V

Reduced Instruction Set Computing (RISC), particularly its fifth iteration, RISC-V, represents an Instruction Set Architecture (ISA) designed to simplify the development of custom processors for various applications. Unlike proprietary ISAs created by private companies, RISC-V offers a free and open-source solution that minimizes intellectual property concerns and reduces entry barriers, promoting innovation and affordability in processor development[2].

RISC-V aims to provide a small core of instructions which compilers, assemblers, linkers, and operating systems can generally rely on, while still being extendable for more specialized accelerators. In RISC-V there are two primary base integer variants, RV32I and RV64I, which provide the 32-bit and 64-bit user-level address spaces respectively. However, RISC-V is already in the works with a RV128I variant which would provide the foundation needed for a 128-bit user address space in the future. In general, RISC-V provides standard and non-standard extensions, where standard extensions should not conflict with other standard extensions, and the non-standard extensions are highly specialized.

With the rise of ARM[2] based machines with comparable and in some cases better performance than that of a Complex Instruction Set Comput-

[2]Short for Advanced RISC Machine. ARM is another RISC based ISA. Difference is it is proprietary.

ing(CISC) alternative[1]. RISC-V aims to provide the same benefits in an open sourced environment. With this RISC-V, more specifically the 32-bit version, was chosen as the ISA for development in this thesis.

2.3 Computational Storage

Computational storage can be seen as a subsection of Accelerator-based Computer architecture. Firstly, it aims to offload the host processor as described in Section 2.1 by providing a secondary processors optimized for specific computational tasks. Secondly, it aims to reduce data movement between the storage device and the host processor. This would allow the read and writes to be distributed among multiple RAM sections rather than a single processor. This could be an integral part of the issues presented in Section 1.1, as a computational storage device would be scalable with the ever-growing need for larger volumes of data processing.

2.4 Toolchain

QEMU

QEMU short for quick emulator system emulator, which has the capabilities of emulating both a 32-bit and 64-bit RISC-V processor [7]. With QEMU I am able to create code intended for a processor running the RISC-V instruction set even if my development environment is running a different ISA. For the purposes of this thesis it is the RISC-V 32-bit version of the QEMU virtual machine that will be used. The RISC-V 32-bit virtual machine is compatible with the standard extensions, and throughout this thesis the following standard extensions will be used. The Control and Status Register (Zicsr), Atomic Instructions(a), Integer Multiplication and Division (m) and Compressed Instructions(c).

LLVM and RISC-V GNU Toolchain

The LLVM project is a collection of reusable compiler and toolchain technologies. Most notably for the context of this thesis clang. Clang is a gcc compatible frontend compiler, which aims to provide fast compile times and low memory use. In tandem with the LLVM compiler back end, clang provides a library-based architecture such that the compiler can work together with other tools. This allows for the use of more sophisticated development environments such as an Language Server Protocol(LSP). Generally clang also provides more sophisticated error reports making the overall debugging easier.

At the time of writing, LLVMs LLDB debugger's connection to the RISC-V QEMU machine was unable to connect probably. Consequently, the GNU gdb debugger for the RISC-V target was compiled for use in the implementation section. For less headache with changing platforms, the same was done for the LLVM clang compiler. Additionally, the RISC-V GNU toolchain supplies essential standard library header files, enabling basic algorithm implementations, such as memcpy[3], for the compiler.

[3]Used by default when copying one memory buffer to another.

3 Design

3.1 Requirements

As described in Section 1.2 the design goal is to implement a merge sort algorithm running on a bare-metal RISC-V processor. The implementation should be usable for lists of varying sizes, and should be easily customisable given different hardware specifications within the RISC-V ecosystem. This should all be done without the need of an operating system as described in Section 3.2.

3.2 Development platform

freeRTOS

RTOS stands for real-time operating system. The goal of an RTOS is to provide a small and simple design, which is easy to port to different architectures. FreeRTOS provides fast execution speeds and methods for multi-threading, mutexes, semaphores, software timers, and more. FreeRTOS specifically is a leading open-source RTOS, fitting well with the open-source idea provided with RISC-V as well. Using an RTOS like freeRTOS would allow for creating a similar implementation of a parallelized merge sort as provided in this thesis. It would also facilitate easier portability across various architectures. However, using freeRTOS would always provide a small overhead compared to a complete bare-metal implementation and as a more custom solution meant for a specific architecture was the development goal, many of the benefits of freeRTOS seemed less enticing leading to a more custom approach for this thesis.

Unikernel

A unikernel can be seen as a small footprint single-address space kernel. It allows a single application to run as if a fully-fledged operating system is in the background. Thus, it removes many of the obstacles by developing on bare-metal, as described in Section 3.2. Generally, these are used to create specialized images, which bridge the gap between having a fully-fledged operating system and working directly on bare-metal. This generally provides better performance, while still reducing the amount of issues provided by working on bare-metal. However, in the field of RISC-V, the most prominent seems to be Nanos. Nanos aims to provide an alternative than the Linux operating system when creating images meant for virtual machines. This could be cloud computing where something like Docker might be used

together with Linux today. Although unikernels can be created for specific hardware applications, it seems sparse in the RISC-V development environment and thus not useable for the work within this thesis.

Bare-metal

Also known as embedded system programming, bare-metal programming involves developing applications meant to run without an underlying operating system. This allows for interlinking with specific hardware and enables a more customizable program tailored to the hardware's specifications. However, this approach comes with drawbacks, such as the lack of a standard library. Default memory allocation functions like malloc in C are not implemented on bare-metal systems because they require an operating system to function. Consequently, developers must handle memory allocation and deallocation themselves for more advanced programs. Additionally, standard printing for debugging purposes is left up to the developer to implement, requiring them to understand hardware specifications, such as UART printing[4] [18].

Memory management often leads to unusual errors during debugging. Notably, while gdb[5] typically detects stack overflows as segmentation faults, this is not possible on bare-metal. In a typical environment, the operating system alerts a user-level program when it exceeds its allocated memory. Without an operating system, unless I implement error checking myself, no such alerts occur. Instead, a stack overflow will change the values in different sections without further notice. However, with the benefits of customization, bare-metal was chosen as the development platform in this thesis.

Another challenge lies in the differences between architectures. Bare-metal programming involves direct interaction with hardware, which is highly dependent on the specific hardware it is programmed for. However, RISC-V offers compatibility standards that allow implementation to work across all RISC-V compatible processors, provided the extensions outlined in Section 1.3 are given. Additionally, the toolchain described in Section 2.4 ensures that most C code remains independent of architecture and can be compiled for other architectures. Nonetheless, there are instances where specific RISC-V assembly instructions, such as atomic operations, must be modified to suit different architectures. QEMU's generic loader facilitates loading an .elf file

[4]The QEMU virt machine is compatible with UART, redirecting any output from stdout to the terminal instance running the QEMU virt machine by default. In Section 5.2, this redirection is used for debugging the implementation, with output being redirected to a file instead.

[5]gdb is the GNU Project Debugger later used for debugging in Section 5.1.

onto various cores. To run the implementation on real hardware, one needs to replicate this process by loading the .elf file for each core on the target system as well as modifying the linker script. See Section 3.5 and Section 4.2 for further information on system-specific modifications and linker scripts.

3.3 Single vs Multicore

The merge sort algorithm can be executed both with serial and parallel computing. Merge sort, in itself, has a recursive pattern of dividing the list into subsections and sorting each subsection for itself. Given that parallel algorithms often outperform their serial counterparts, this thesis chose a multicore approach for implementation[5]. This decision introduces new challenges not present in serial computing ranging from race conditions to memory management of multiple cores and threads.

Working on multiple cores

The first method for implementing a multicore merge sort involved using threading and a scheduler. When splitting a given list into two halves, I would create a thread assigned to sort each half. These two halves would be added to a global queue of available threads, after which the job of merging the two halves could be added to the back of the queue. The merge job would have to check whether the two child threads have finished sorting each half, but otherwise, assuming a round-robin scheduler, the queue would automatically guarantee correct ordering for the parallelized merge sort algorithm. However, this approach introduces multiple race conditions; the primary one being implementing a queue capable of handling concurrent access. The simplest method would be to implement a lock, allowing mutual exclusion when adding to and removing from the queue. The downside of implementing a locking queue is that synchronization means that a core would have to do unnecessary work while waiting on another core. Another method would be to implement a lock-free queue, which should remove any synchronization issues and be a viable solution. Usually such a queue could be implemented with atomic instructions such as the Compare and Set(CAS) operation in the x86 ISA. In RISC-V such an instruction is not present, and instead the operations Load Reserve(lr) and Save Conditional(sc) are present. The lr and sc operations could be substitutes for the CAS instruction, but in such a case they would still have to be wrapped within a loop to guarantee that two threads do not access the same element. Generally, this approach could theoretically produce a clean solution with high performance where it is lock-free in the sense that some core will always make progress. However, there

must still be some sort of synchronization between the cores, and thus was not the chosen method for this thesis.

Instead, the entire thread structure would be initialized before any core begins running computations on a thread job. To simplify initialization, a single core would have the task of splitting the initial array into sublists until every core has a single sublist to work on. While doing the splitting, it would also create threads responsible for merging the sublists once they are finished. This approach eliminates the need for a queue and scheduler since each core, through its own core ID, knows what thread it must complete.

3.4 Context switching

Context switching is an integral part of multithreading. It is the act of storing the state of the process so that it can later be restored and resume execution at a later point. Furthermore, it is possible to modify context, such that execution can start at another point in the program. With this I can create the context needed for sorting a specific part of the list, save it in a thread structure and then a separate core can continue its execution at a later point.

3.5 Memory

Getting system information

To properly use the memory, I need some information about the system I am working on. As this thesis is created on a QEMU system, I am able to get the system information by running the following:

```
qemu-system-riscv32 -machine virt \
-machine dumpdtb=riscv32.dtb
```

This creates a Device Tree Blob (dtb) data file, which contains information about the virt qemu-system-riscv32 virtual machine. This format is generally not seen as human-readable, but by using the Device Tree Compiler (dtc) package, I can convert it from the binary dtb format to a human-readable dts format.

```
sudo apt install dtc
dtc -I dtb -O dts -o riscv32.dts riscv32.dtb
```

Opening the file in a text editor, you will see a lot of information regarding the qemu-system-riscv32 virtual machine. First, note that the Devicetree specification[8] states that the memory node describes the physical memory layout for the system. The memory node has two required sections: first, the device_type, which must simply be 'memory,' and secondly, the reg value. The reg value consists of an arbitrary number of address and size pairs that

specify the physical address and size of the memory ranges [8]. Furthermore, it is stated that the property name reg has the value encoded as a number of <u32> cells required to specify the address and length, which are bus-specific and are specified by the #address-cells and #size-cells properties in the parent of the device node. Looking through our riscv32.dts file, I find the relevant information to be:

```
#address-cells = <0x02>;
#size-cells = <0x02>;

memory@80000000 {
  device_type = "memory";
  reg = <0x00 0x80000000 0x00 0x8000000>
};
```

With the information previously provided, I know that the starting address of the memory section is at address $0x00 + 0x80000000 = 0x80000000$ and has a size of $0x00 + 0x8000000$ bytes, which is equivalent to 128MB.

Memory Layout

As mentioned, all created threads need to have a separate stack for context switching to work. Thus, when creating a thread, I have to allocate some location in RAM to the task the thread has to perform. In Figure 1, this memory area is denoted with the "thread x STACK" area. As a design choice, I chose to separate the thread stacks in the opposite end of RAM from where the core stacks would be allocated. That way, if I ran into a thread stack overflow it would be while servicing a thread job any irregularities would show and vice versa for the core stacks. Generally it is a good custom that the stack size is some multiple of 4 and with some testing 1024 bytes seemed to be large enough for general usage. At the end of the RAM section is where the individual core stacks would be allocated. Again through some general testing a size of 2048 bytes generally worked for small lists. It is important to note, that these values could quickly become too small, and in Section 5.1 this will be further investigated.

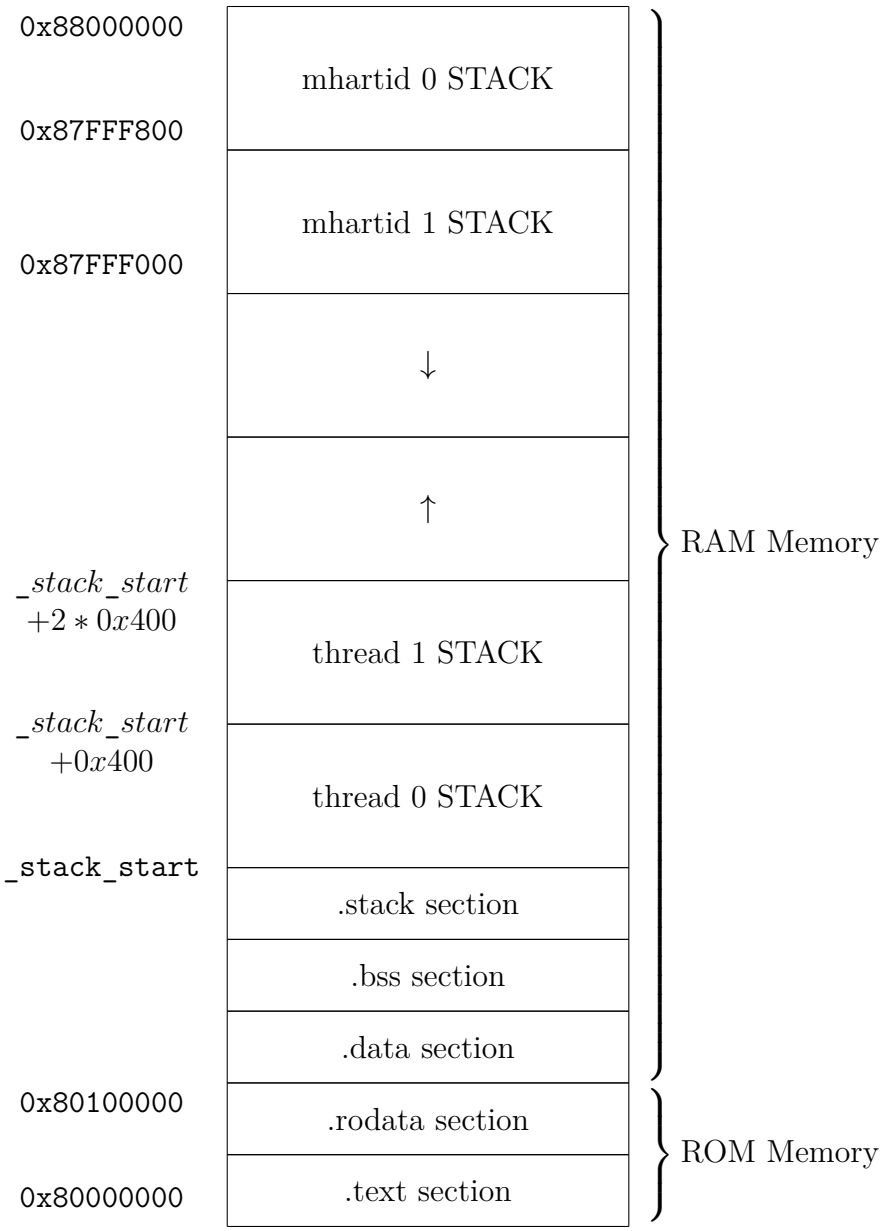


Figure 1: Memory layout of the QEMU virt machine with a core stack size of 2048 bytes and a thread stack size of 1024

4 Implementation

The implementation aims to carry out the design presented in Section 3. First I will go through the Dependencies needed for development. Next, I highlight a few components of the implementations, such that the reader can get a better understanding of the implementation. Note, the elements presented within this thesis is just a part of the implementation and more parts make up the entire program.

4.1 Dependencies

QEMU

Listing 1: Installing QEMU

```
git clone git clone https://github.com/qemu/qemu  # Clone the QEMU repo
cd qemu
./configure --target-list=riscv32-softmmu  # Configure the 32-bit RISC-V target
make -j $(nproc)  # build the project with all num cores jobs
sudo make install
```

Following the instructions from RISC-V's getting started guide, I can build the QEMU RISC-V system emulators by running the code provided in Listing 1[9].

LLVM and RISC-V-gnu-toolchain

Listing 2: Installing LLVM compiler infastructure with RISC-V 32-bit as native target.

```
# Dependencies
sudo apt-get -y install \
  binutils build-essential libtool texinfo \
  gzip zip unzip patchutils curl git \
  make cmake ninja-build automake bison flex gperf \
  grep sed gawk python bc \
  zlib1g-dev libexpat1-dev libmpc-dev \
  libglib2.0-dev libfdt-dev libpixman-1-dev

git clone https://github.com/riscv-collab/riscv-gnu-toolchain
cd riscv-gnu-toolchain  # change directory
./configure --prefix=/opt/riscv --with-arch=rv32gc -disable-linux --enable-llvm
sudo make -j$(nproc)
cd ..
popd
```

Although the LLVM Clang compiler comes with an available cross compiler, I found that it often caused issues with missing header files compatible with my implementation. Furthermore, the LLDB debugger was unable to provide a working debugger for multicore remote debugging on QEMU. These are issues which might only be affecting me, as information revolving around the issues was scarce. As such, the following steps of building LLVM and the RISC-V 32-bit GDB may be obsolete but are left here as a known working toolchain. Running the code in Listing 2 installs a RISC-V compatible Clang compiler and GDB debugger in the /opt/riscv/ directory. For use outside this folder, make sure to add it to PATH.

libucontext

Libucontext is an open-sourced library that provides the ucontext.h C API. Most notably for the project of this thesis, it can deploy on bare metal RISC-V 32-bit with newlib[6]. Building the library from scratch led to some issues on my end, and as such, the necessary files were copied and linked together with my implementation upon building. With this, I am able to use getcontext, makecontext and setcontext, which allows me to do the necessary context switching described within Section 3.

4.2 Creating a linker script

The linker script is used to inform the linker which parts of the file to include in the final output file, as well as where each section is stored in memory. As I am working on an embedded system, I have to deviate from the default and create my own linker script. Clang uses the LLVM lld linker, which is compatible with general linker scripts implementations of the GNU ld linker[12].

Listing 3: Memory area defined in linker script

```
OUTPUT_ARCH('riscv')
ENTRY(_start)

MEMORY
{
/* Fake ROM area */
rom (rxa) : ORIGIN = 0x80000000, LENGTH = 1M
ram (wxa) : ORIGIN = 0x80100000, LENGTH = 127M
}
```

First, I must specify that I want the RISC-V architecture and designate the entry point of the program at a function named '_start', which I will define

[6]Newlib is a C library intended for use on embedded systems.

later. Second, I define the MEMORY area to consist of both a writable memory region and a read-only memory region. I name these regions 'ram' and 'rom,' respectively. With that, I move on to define the SECTIONS element of the linker script.

Listing 4: Linker scripts SECTIONS.

```
SECTIONS {
  .text :
  {
    *(.text .text.*)
  } > rom

  .rodata :
  {
    *(.rdata)
    *(.rodata .rodata.*)
  } > rom

  .data :
  {
    *(.data .data.*)
    . = ALIGN(4);
    PROVIDE( __global_pointer$ = . + GLOBAL_STACK_SIZE );
  } > ram

  .bss :
  {
    *(.bss .bss.*)
  } > ram

  /* It is standard to have
  the stack aligned to 16 bytes*/
  . =
  _end = .;

  .stack :
  {
    . = ALIGN(4);
    PROVIDE(_stack_start = .);
    PROVIDE(_stack_top = ORIGIN(ram) + LENGTH(ram));
  } > ram
}
```

The text, rodata, data and bss sections follow the same general procedure. I match any data for the given section, and then save it either in rom or ram. By specifying the > rom, I tell the linker to save the given section in the ROM section and the same is true for the > ram. From the Figure 1 a memory diagram shows how the linker places elements into memory.[7]

In the data section, I also provide a global pointer, which is used to access global variables within our later code implementation. The global pointer is

[7]ROM stands for Read-Only Memory, and RAM stands for Random-Access Memory. Generally it is not necessarily needed to split the two up as done here, but it is a good practice to separate what can change and what cannot change in memory.

used together with an offset to save global variables. For more customisability this value can be passed in the Makefile.

The last section is the .stack section. I align the starting of the _stack_start with 4 bytes. Generally a good rule of thumb is to guarantee alignment to the word size, which in RISC-V is 32-bit. This is the point from where each thread stack will be allocated. Next, I specify that the _stack_top will reside at the end of the RAM section such that I can allocate a stack for each core.

4.3 Getting into the main function

In the linker script, I specified the entry point of our program as '_start'. Next up is implementing this entry point in assembly. In a new assembly file, I add the following:

Listing 5: Assembly code for getting to main function.

```
1  .extern main
2  .extern secondary_main
3  .globl _start
4  .type _start,@function
5  #include "../include/defines.h"
6
7  _start:
8    .cfi_startproc
9    .cfi_undefined ra
10   .option push
11   .option norelax
12   la gp, __global_pointer$
13   .option pop
14   // load _stack_top into the sp register
15   la sp, _stack_top
16   csrr a0, mhartid
17   bnez a0, 2f
18   1:
19     // argc, argv is 0 and jump to main
20     li  a0, 0
21     li  a1, 0
22     jal main
23   1:
24     // loop
25     j 1b
26   2:
27     la t1, STACK_SIZE
28     li t0, 0
29   1:
30     andi sp, sp, -16
31     beq a0, t0, 1f
32     sub sp, sp, t1
33     addi t0, t0, 1
34     j 1b
35   1:
36     // argc, argv is 0 and jump to main
37     li  a0, 0
38     li  a1, 0
```

```
39      jal secondary_main
40    1:
41      // loop
42      j 1b
43      .cfi_endproc  // This point should never be reached
```

Lines 1-5 provide the setup for the assembly file. I specify that a main and secondary_main label will be defined outside of the file, that _start is a global label and that _start is a function type. At last, I include the defines.h file, which includes definitions of the STACK_SIZE.

On lines 7-13 I provide call frame information (CFI) for there being no return address and that the process begins here. Then when initializing the global pointer, I must specify options push, norelax, and pop as described in GNU binutils documentation[10]. After linker relaxation, this would produce the anticipated code:

```
auipc gp, %pcrel_hi(__global_pointer$)
addi gp, gp, %pcrel_lo(__global_pointer$)
```

On lines 15-17, I load the value of _stack_top, which the linker provides through the linker script defined previously, and save it into the stack pointer (sp) register. I read the current machine hart identifier (mhartid), which contains a unique identifier for each core on the processor. This is what allows for differentiation between the different cores. Line 17 moves execution to line 27 if the machine hart identifier is not 0. As such, only mhartid 0 will be allowed to continue execution to line 22, where it jumps to the main function.

All other cores continue execution at line 27, where they load the value STACK_SIZE[8] into the temporary register t1. I also load immediately(li) the value 1 into the temporary register r1. Line 30 aligns the current value in the stack pointer to 16. Afterwards, I compare the value in register a0, which holds the value of the mhartid, to the value in t0. If they are equal, I jump to line 35, which goes to the externally defined secondary_main function. Otherwise, I continue on line 32, where I subtract the register t1 (STACK_SIZE) from the value stored in the sp register. I increment the value in t0 by one, and jump back to line 30. With this loop, I am setting up a stack of size STACK_SIZE for all the different cores defined, such that I get the desired memory layout shown at the top of Figure 1.

4.4 Initializing the thread jobs

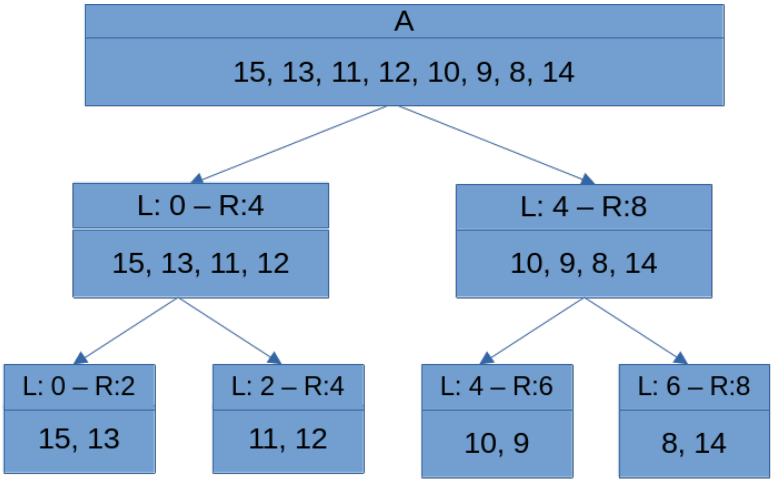
A pseudocode implementation for initializing the parallel merge sort algorithm is provided in Algorithm 1. It should be noted that it is assumed

[8]Defined either via. the Makefile or defined in defines.h

Algorithm 1 Initialization of the threads

Require: threads[] # Empty thread array

```
1: procedure INITIALIZE_THREADS(A)
2:     depth ← MostSignificantBit(NUM_CORES)
3:     idx ← 0
4:     for i = 0, i < depth do
5:         if i == 0 then
6:             # Top level thread
7:             ct ← threads[idx]
8:             mid ←Length(A)/2
9:             thread_create(ct, parallel_merge)
10:            ct.l = 0
11:            ct.mid = mid
12:            ct.r = Length(A)
13:            idx++
14:            continue
15:        end if
16:        k ← 2^k
17:        for j = 0, j < k do
18:            parent_thread ← threads[(idx − 1)/2]
19:            ct ← thread[idx]
20:            if j%2 == 0 then
21:                ct.l = parent_thread.l
22:                ct.r = parent_thread.mid
23:            else
24:                ct.l = parent_thread.mid
25:                ct.r = parent_thread.r
26:            end if
27:            ct.mid = ct.l + (ct.r − ct.l)/2
28:            if i == depth − 1 then
29:                # Just regular merge sort
30:                thread_create(ct, merge sort)
31:            else
32:                # Merge given section
33:                thread_create(ct, parallel_merge)
34:            end if
35:            idx++
36:        end for
37:    end for
38: end procedure
```



THREADS := [(A, 0, 8, MERGE), (A, 0, 4, MERGE), (A, 4, 8, MERGE), (A, 0, 2, MERGESORT), (A, 2, 4, MERGESORT), (A, 4, 6, MERGESORT), (A, 6, 8, MERGESORT)]

Figure 2: Example of partitioning a random list using Algorithm 1

NUM_CORES is a multiple of 2. Consequently, I can calculate the depth desired for our merge tree by employing the number of cores. This may be achieved by taking $\log_2(NUM\_CORES)$ or, equivalently, by examining the most significant bit of NUM_CORES since it is a multiple of 2. At line 5, an early escape mechanism is implemented which takes care of the single thread that will have no parent thread. If I am not positioned at the very top level of the merge tree, I assign the variable 'k' as the number of threads for the given level and partition the list to create threads targeting a specific sublist. Lines 20 - 27 address the task of determining which subsection of the list each thread is responsible for. Finally, line 28 ensures that I do not generate more active threads than what the available number of cores can effectively handle. Upon completion of the algorithm, a threads array will be produced containing $2 \cdot NUM\_CORES - 1$ different threads, each assigned with a specific subsection of the list to either merge or perform merge sort on.

4.5 Reducing synchronization

The idea behind initializing the threads array, as described in Section 4.4, is that I can reduce the need for synchronization between threads once the partitioning of the list is done. An example can be seen in Figure 2 with the corresponding finished THREADS array. This example would be on a system

with 4 cores. As seen in Section 4.3, each core has a unique **id**, mhartid, from 0 to 3. With this, each core can index into the list with the number of jobs it has already completed and the length of the array list such that:

$$index = LENGTH(A) - \text{NUM\_COMPLETED\_JOBS} \cdot id \qquad (1)$$

With this, each thread is capable of retrieving a thread job without having to consider the state of any other thread, as the job is preassigned during the initialization. Each merge must still wait on the child thread (the direction of the arrow in Figure 2) to finish before starting a merge. However, as there is no queue, there is no need for synchronization beyond that between the threads.

5 Evaluation

5.1 Testing

Testing has been accomplished by providing a random_numbers.py file. With three separate integer parameters, it creates a random list using Python's standard random library. The inputs include a lower and upper bound as well as a length. Once the list is created, running make will create a .elf file, which contains the parallel merge sort algorithm with the unsorted list hard-coded within. Once the .elf file is loaded on a RISC-V processor, it will immediately begin sorting the hard-coded list. The implementation also needs the value NUM_CORES defined within the Makefile, where it both defines a constant NUM_CORES for the .elf file to use, and the same value is used for running the QEMU virtual machine.

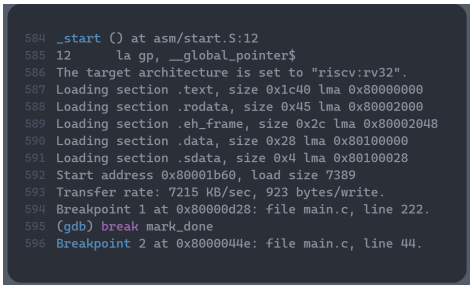
This gives the following work flow for creating and running a test:

- Change directory to src/bare_metal.
- Run random_numbers.py to generate alist.c with an unsorted list.
- Change the NUM_CORES variable in the Makefile to the desired number of cores.
- Run "make clean" to remove all files built with previous settings.
- Run "make test" to generate the .elf file and host QEMU. This step will create a test.txt file, as the stdout of QEMU is redirected to said file.
- Run "python3 validate.py". This reads the test.txt file, first reading the unsorted array and sorting that with pythons standard sorting algorithm. Then it reads the sorted array from the test.txt file and compares it with pythons sorted array.

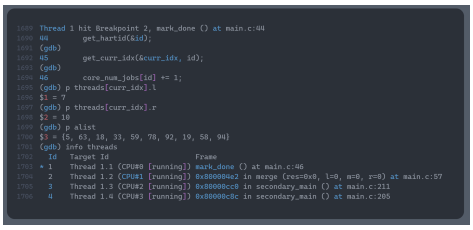
Debugging

To access and debug the multicore system, QEMU provides a way for a remote gdb-server to connect at the start of execution[7]. In a system where each core is equivalent, gdb will automatically detect the cores, but will display them as threads. This allows one to debug the QEMU virtual machine with the same methodology used when debugging multithreaded execution. The workflow shown in Figure 3 is as follows:

- Change directory to src/bare_metal.



(a) Connecting gdb by running "riscv32-unknown-elf-gdb" and breaking at mark_done"



(b) Hit breakpoint, where alist is sorted by core 1 from index 7 to 10 (10 exclusive).

Figure 3: Debugging the QEMU virtual machine with 4 cores.

- Initialize the alist.c file with an array and a given size.
- "make clean" to remove any previous build.
- Run "make debug" to initialize the QEMU virtual machine.
- In a separate terminal instance run "riscv32-unknown-elf-gdb". This will automatically run the commands in the ".gdbinit" file and connect to the QEMU virtual machine.
- Break at mark_done. This function is run whenever a given thread job is finished.
- Run "info threads" to get an overview of all threads.

As an example, Figure 3 shows a list of size 10, where core 1 has sorted the subsection of the list from index 7 to 10. This section corresponds to one fourth of the list, as the number of cores available is 4.

5.2 Validation

When running a test, the .elf file first prints the unsorted array, and then once sorting is done, prints it again. When executing "make test", the QEMU virtual machine outputs the stdout to a file called test.txt. Subsequently, invoking "python3 validate.py" reads this file to ascertain if sorting was performed correctly. In Table 6, an overview of some tests I conducted can be seen. On the left, the formatting is specified as the lower bound of randomly selected numbers, the upper bound, and the number of random elements in the list. A value of 'pass' signifies that the validate.py file executed without

Table 1: Table of tests run

	NUM_CORES					
Lower:Upper:Number	2	4	8	16	32	64
-100:0:100	pass	pass	pass	pass	pass	pass
0:100:100	pass	pass	pass	pass	pass	pass
-50:50:100	pass	pass	pass	pass	pass	pass
-50:50:10	pass	pass	pass	fail	fail	fail
-1000:1000:1000	pass*	pass*	pass*	pass*	pass*	pass*
-1000:1000:20000	pass*	pass*	pass*	pass*	pass*	pass*
-1000:1000:100000	pass*	pass*	pass*	pass*	pass*	pass*

*This run initially failed due to stack overflow. After increasing the different stack sizes following Appendix A it passed.

throwing an assertion error. A value of 'fail' signifies that validate.py threw an error when running. The tests were initially performed with a global stack of 0x800,[9] a STACK_SIZE of 2048 bytes and a THREAD_STACK_SIZE of 1024 bytes. If a failure occurred, modifications were made to these three values in an attempt to pass the test.

In Table 1 one can see that when the number of cores is greater than the number of elements in the list it fails executing the sorting algorithm. Generally it would not seem relevant to ever run a merge sort algorithm with more cores than there are elements in the list due to the tree structure created with merge sort. The error is caused by the merge function being told to merge a single element, which is undefined behaviour in the current implementation. I have decided not to leave a check for this, and instead assume that the number of cores is never larger than the length of the input list.

5.3 Future work

The implementation proposed in this thesis is more a proof of concept, and as such comes with a few shortcomings. Firstly, with the current implementation there is a heavy reliance on the GLOBAL_STACK_SIZE. The array to be sorted is hard-coded and stored on the global stack, which can occupy substantial space when dealing with large arrays. Furthermore, the array of

[9]defined in ram.ld. Equates to 2048 bytes.

thread jobs is also saved on the global stack. With the current implementation each thread type has a size of 848 bytes. Thus, a solution that reduces overall usage of the global stack, or a better method of detecting the size of the global stack would allow for a more reliable solution.

Whenever a merge sort splits the array into two halves, there is created a new thread with an allocated stack size of THREAD_STACK_SIZE. The top-level thread job, which has to sort the entire array, must first create a copy of the array before it can sort in place. The same goes for all other thread jobs, but the amount of the array they need to copy is halved for each level moved down in the merge sort tree shown in Figure 2. In theory, this allows for halving the size of the allocated thread stack for each level change. This does not take into account the constant space needed for the declared variables, so the thread stack size might instead be represented by:

$$\text{THREAD\_STACK\_SIZE} = \text{INITIAL\_STACK\_SIZE}/2^i + C \qquad (2)$$

Where i is the depth level, and C is some constant used to store the variables needed during computation, this constant C would then be different depending on whether the given thread has to perform merge sort or just merge over the array. Experiments determining the size of C could be carried out to optimize memory better.

Furthermore, with the current implementation core 0 is the only thread writing to the global threads array. This core is tasked with initializing all the threads jobs, which means that all other cores must wait until this single thread has finished its initialization. As the number of cores grows, so does the time it takes to initialize the threads. This is inherently inefficient. As such future work could propose a solution where the work of initializing the threads array could be spread among multiple cores instead.

As the implementation is carried out using QEMU, it is not possible to do performance testing. Firstly, the implementation is implemented on bare-metal, and would need some sort of custom timing implemented to get the time at the start and then at the end. Secondly, as QEMU is a virtual machine, it mimics multiple cores by creating threads on the host pc. These threads provide an illusion of multicore, but the host operating system is still in charge of scheduling, which performance vice removes any theoretical performance by creating the bare-metal design in the first part. For performance testing the implementation must be carried out on actual hardware, such that it is not just a virtualisation of the bare-metal platform.

The implementation also requires that the number of cores is a power of 2. This is due to how the initialization builds the threads up. This issue has a few ways it can be fixed. Firstly, instead of stopping at the same level for

all the merge sorts, it could theoretically split only a few of the sublists given, such that instead of having to be a power of 2, it would instead only require that it is a multiple of 2. This would still allow for all cores to work on a sublist at the same time. Another approach would be to allow a subset of the number of cores to busy loop until the two child threads finish as described in Section 4.4. The former approach seems more promising, but is left as something which can later be addressed.

As it stands, the only method of loading an array onto the processor is by hard coding the list into the .elf file. Although this works to show the entire sorting algorithm, it would not be useful in an actual data center unless some sort of communication protocol is established between the host CPU and the computational storage device. I have left this communication up for future work.

6 Related work

Marcelino et al. [14] evaluate three hardware sorting units implemented with specific Field Programmable Gate Arrays (FPGAs). One of these being the FIFO[10]-based merge sorting machine, where they present a merge sorting structure for a FIFO embedded system merging. This implementation assumes two sorted input lists and they later propose a hybrid solution using Insertion and FIFO based merge sorting. With this they find that the FPGA hybrid insertion and FIFO-based merge sorting sees speed-ups between 1.6 and 15 time compared to a quick sort pure software solution. Such a hybrid solution could be implemented in tandem with the implementation provided in this thesis, and might lead to a similar performance increase from what they show within their paper.

Jackson et al. [11] look for faster sorting algorithms used for flash memory embedded devices. They provide a merge sort for sorting with minimal memory usage which aims to reduce the number of WRITES to flash memory. This implementation would only need two memory buffers. They find that when sorting large data sets with small memory the proposed algorithms reduces I/Os and execution time by about 30%. A design choice in CSD might be to reduce the need for large amount of memory, and as such a memory efficient sorting algorithm could be better suited.

Lobo et al. [13] compare the performance of different types of merge sort algorithms. Within their testing they find that the serial and parallel merge sort discussed have a similar amount of resource utilization, but find the delay of the parallel merge sort to be a lot smaller than the serial counterpart. As such, they theorize that this implies the parallel execution to be much faster than the serial. This finding supports the theory that a parallelized merge sort algorithm, such as the one provided in this thesis, could increase the performance over a serial counterpart.

[10]Short for First In First Out

7 Conclusion

7.1 Summary of results

This thesis discusses the viability of implementing custom-made solutions for computational storage devices. In the first part of the thesis, the problem is presented along with the background. Next, the approach of implementing a bare-metal merge sort application on the QEMU virtual machine is proposed. A design for a merge sort algorithm that does not make use of any queue is suggested to reduce the need for synchronization. With this, I discuss a method of memory allocation and the design is carried out by making use of context switching. I tested the implementation of the merge sort algorithm with varying list sizes as well as a varying number of cores utilized. Finally, possible improvements are presented in future work and left as open research questions.

7.2 Takeaways

In Section 1.2 two questions were proposed and aimed to be answered throughout this thesis. As such, these will build the foundation of the following conclusion.

What computation should be handled by a storage device?

This thesis demonstrates that intricate computations, such as sorting, are indeed feasible to implement on a bare-metal storage device. However, this endeavor does not come without its share of complications. These include creating a custom linker script and devising custom assembly scripts, in addition to other obstacles left unmentioned within the scope of this thesis. These challenges may deter the average consumer; nonetheless, when viewed from the perspective of large-scale data centers, the potential for computational storage remains bright. As the CPU increasingly becomes a bottleneck in data transfer between a storage device and its host, computational storage has the potential to go beyond just sorting capabilities. Consequently, this thesis proposes that it is not the complexity of the computation at hand but rather the magnitude of demand from large data centers that will ultimately determine the success of the computational storage device.

Is it feasible to implement such a computation on a bare-metal RISC-V processor?

The bare-metal approach presents numerous challenges that are not encountered when utilizing an underlying operating system. Despite these limitations, with perseverance, it is possible to overcome them and optimize a system specifically tailored to meet certain demands. In the context of large data centers, developing a custom computational storage device is both viable and could be advantageous. Nevertheless, it remains uncertain whether the performance enhancements justify the investment in implementing such systems.

References

[1] Blem et. al. "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 architectures". In: *University of Wisconsin - Madison* (2013). URL: https://research.cs.wisc.edu/vertical/papers/2013/hpca13-isa-power-struggles.pdf.

[2] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[3] Antonio Barbalace and Jaeyoung Do. "Computational Storage: Where Are We Today?" In: *Conference on Innovative Data Systems Research*. 2021. URL: https://api.semanticscholar.org/CorpusID:232328691.

[4] Simona Boboila and Peter Desnoyers. "Write Endurance in Flash Drives: Measurements and Analysis." In: Feb. 2010. URL: https://www.usenix.org/event/fast10/tech/full_papers/boboila.pdf.

[5] Darko Božidar and Tomaž Dobravec. *Comparison of parallel sorting algorithms*. Tech. rep. University of Ljubljana, 2015. DOI: 10.48550/arXiv.1511.03404.

[6] Carly Davenport et al. *Generational gorth; AI, Data centers and the comini US power demand surge*. URL: https://www.goldmansachs.com/intelligence/pages/gs-research/generational-growth-ai-data-centers-and-the-coming-us-power-surge/report.pdf.

[7] QEMU Project Developers. *QEMU RISC-V System emulator*. 2023. URL: https://www.qemu.org/docs/master/system/target-riscv.html.

[8] devicetree.org. *Devicetree Specifications*. v0.4. 2023. URL: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.4.

[9] RISC-V Foundation. *RISC-V Getting Started Guide*. 2018. URL: https://risc-v-getting-started-guide.readthedocs.io/en/latest/.

[10] *GNU Binutils RISC-V Directives*. URL: https://sourceware.org/binutils/docs-2.34/as/RISC_002dV_002dDirectives.html.

[11] Riley Jackson and Ramon Lawrence. "Faster Sorting for Flash Memory Embedded Devices". In: *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*. 2019, pp. 1–5. DOI: 10.1109/CCECE.2019.8861811.

[12] llvm. *Linker Script implementation notes and policy*. URL: https://lld.llvm.org/ELF/linker_script.html.

[13] Joella Lobo and Sonia Kuwelkar. "Performance Analysis of Merge Sort Algorithms". In: *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*. 2020, pp. 110–115. DOI: 10.1109/ICESC48915.2020.9155623.

[14] Rui Marcelino, Horácio Neto, and João M.P. Cardoso. "Sorting Units for FPGA-Based Embedded Systems". In: *Distributed Embedded Systems: Design, Middleware and Resources*. Ed. by Bernd Kleinjohann, Wayne Wolf, and Lisa Kleinjohann. Boston, MA: Springer US, 2008, pp. 11–22. ISBN: 978-0-387-09661-2. URL: https://link.springer.com/chapter/10.1007/978-0-387-09661-2_2.

[15] *Open-Channel Solid State Drives*. URL: https://openchannelssd.readthedocs.io/en/latest/.

[16] Sanjay Patel and Wen-mei W. Hwu. "Accelerator Architectures". In: *IEEE Computer Society* (2008). URL: https://users.cs.duke.edu/~lkw34/papers/accelerators-ieee2008.pdf.

[17] LLVM Project. *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[18] SIDSA. *UART 16550 IP*. 2000. URL: https://caro.su/msx/ocm_de1/16550.pdf.

[19] *Taking control of SSDs with LightNVM*. Apr. 2015. URL: https://lwn.net/Articles/641247/.

[20] "The Bleak Future of NAND Flash Memory". In: *10th USENIX Conference on File and Storage Technologies (FAST 12)*. San Jose, CA: USENIX Association, Feb. 2012. URL: https://www.usenix.org/conference/fast12/bleak-future-nand-flash-memory.

[21] Andrew Waterman, Krste Asanovic, and SiFive Inc. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Mar. 2017. URL: https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

Glossary

CISC	Complex Instruction Set Computing.
CPU	Central Processing Unit.
CSD	Computational Storage Device.
dtb	Device Tree Blob.
dtc	Device Tree Compiler, can convert a .dtb file to human readable format.
GCC	GNU Project Compiler. Compiler to make executable file.
GDB	GNU Project Debugger. A program used for debugging.
GPU	Graphics Processing Unit.
ISA	Instruction Set Architecture.
LLVM	Originally for Low Level Virtual Machine, but now provides a collection of modular and reusable compiler and toolchain technologies[17].
QEMU	Used to create a virtual machine running the RISC-V 32-bit ISA. Short for Quick Emulator.
RISC	Reduced Instruction Set Computing.
RTOS	Real time operating system.
SSD	Solid-state drives.

A Choosing Stack Sizes

When selecting the stack size, one can make an educated guess on the size needed. Firstly, we divide it into three distinct stack types: Thread, Core, and Global.

Core stack

The Core stack is the simplest. Generally, the entire computation done on a given core will always be the same, no matter the size of the list. This is due to how the context switching works. Whenever we switch context, we are using a different stack, and since we always do a context switch before copying data for a merge operation, then the stack size needed will generally be the same. From testing I found that a STACK_SIZE of 2048 bytes was enough.

Thread stack

Each thread job will execute at least one merge operation where the merge operation has to copy the entire section it currently is working on. Thus, in the worst case a thread will have to copy the entire list when doing a merge operation. Following this rule of thumb, we have:

$$\text{THREAD\_STACK\_SIZE} = \text{LIST LENGTH} * 4 + C \tag{3}$$

Where 4 denotes the size of an integer in bytes, and C is some constant extra size needed for all the default variables. Throughout my testing, I found that having a C value of 2048 generally was enough for the THREAD_STACK_SIZE.

Global Stack

The global stack stores both the entire list and the thread jobs. Within GDB, I printed the size of the thread_t struct, which told me its size was 848 bytes. Thus, the Global Stack size generally can be set to:

$$GLOBAL\_STACK\_SIZE = \text{LIST LENGTH} * 4 \tag{4}$$

$$+ (2 * \text{NUM\_CORES} - 1) * 848 + C \tag{5}$$

Where 4 represents the size of an integer in bytes, 848 is the size of the struct in bytes, $(2 * \text{NUM\_CORES} - 1)$ denotes the number of thread jobs and C stands for some constant needed for the different variables. A value of 2048 for C was generally sufficient.

B Link to implementation

The implementation is hosted on github, and is publicly available via the link: https://github.com/svbrodersen/bachelor.