On Efficient Sorting Through In-Memory Processing

Master's Thesis

Submitted by

Zeno Adrian Weil

Matriculation Number: 9875792

Supervisor

Dr. Manuel Penschuck

Johann Wolfgang Goethe-Universität

Frankfurt am Main, 11th October 2024

Abstract

The growing disparity between processing and memory speed, coupled with increasing data demands, has led to memory accesses being a bottleneck for many modern workflows. An example are sorting algorithms, which are often designed around the constraints set by memory subsytems. In-memory processing (also known as processing in memory, PIM) is an umbrella term encompassing several approaches which offload computational tasks to accelerators in or near the memory itself. In PIM systems designed and manufactured by UPMEM, traditional dynamic random-access memory (DRAM) modules are augmented with general-purpose processors called DRAM processing units (DPUs). These are located next to the memory banks themselves, whereby high memory access speed is accomplished. An UPMEM-based PIM system may contain thousands of DPUs, each capable of additional thread-level parallelism. Although designed for general use, the DPU architecture does come with limitations to its computational prowess.

The scope of this thesis is the design, implementation, and evaluation of sorting algorithms which run on a single DPU. For several sequential and parallel sorting algorithms, we document the engineering process and adaptations to the merits and shortcomings of the DPU architecture. We find that sorting is a suitable task for a DPU, which can be sped up nearly ideally through multithreading. This paves the way for more large-scale sorting algorithms which run on multiple DPUs.

Danksagung

Danken möchte ich meinem Betreuer, Manuel Penschuck, für seine weitreichende Unterstützung in den vergangenen Monaten, die insbesondere in den letzten drei Wochen über alles erwartbare Maß hinausging. Ebenso bedanken möchte ich mich bei meinen Kommilitonen Lukas Geis und Alexander Leonhardt, mit denen ich viele fruchtbare und beflügelnde Gespräche im Laufe des Semesters führen durfte. Zuletzt gilt mein Dank meiner Mutter und meinem Vater, der glücklicherweise seine private Informatikbibliothek aus eigenen Studientagen nie entsorgt hat.

Contents

Chapter 1.

Introduction

For decades, processor and memory technology have developed in opposite directions. In case of processors, particular attention has been paid to speed, whereas it has been capacity for memory. As a consequence, the performance of a processor core has grown by up to 52 % year after year, whilst the latency of dynamic random-access memory (DRAM), for example, has been improving only by 7 % (see Fig. 1.1). The improvements in bandwidth, though not as bad as those in latency, could not keep up with processor improvements as well. The emergence of multicore architectures has aggravated the discrepancies. Thus looms the *memory wall* which gets hit when a processor is forced to idle whilst waiting for data. Techniques like prefetching and multi-level caches have been developed to defer the impact, and yet, idle times make up as much as 60 % of the total runtime [32]. Additionally, despite latency improvements being partially slowed down due to energy concerns [19, p. 130], memory transfers are responsible for much of the total power consumption of a system, accounting for 63 % of the total in consumer devices [6] and 40 % in scientific applications [25].

One approach to tear the memory wall down is *in-memory processing* (also *processing in memory*, PIM), which is further divisible into processing using memory and near-memory processing [18, 32, 34]. *Processing using memory* (PuM) denotes analogue computation through repurposing existing structures within a memory device, like memory storage cells or and peripheral circuitry. This allows mainly for simple bitwise operations but also for random number generation. Even vector-matrix multiplication is possible in memory built from crossbar arrays, enabling more complex tasks like signal processing, compression, and image filtering [28].

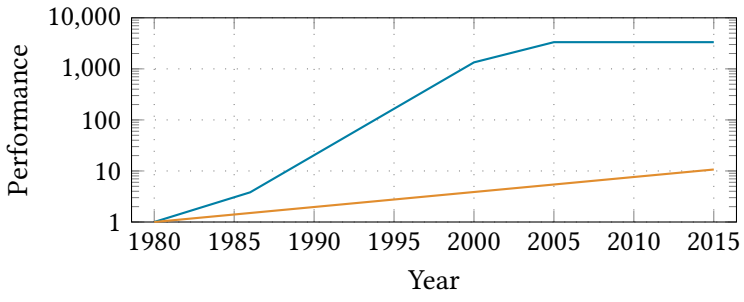


Figure 1.1. *Sketch of the divergent development of the latency of a single DRAM bank (bottom) and the average time difference between memory requests by a single processor core (top), with the performance of the year 1980 as baseline. [19, Figure 2.2]*

Near-memory processing (PnM) denotes digital computation through additional compute engines close to or in the memory device. The memory latency improves because of the proximity to the storage, and the bandwidth scales naturally with the size of the memory. Such separate circuitry allows for more complex applications like weather modelling [40] or personalised recommendation [24].

The PnM solution offered by UPMEM augments regular DRAM modules by adding general-purpose DRAM processing units (DPUs), of which there may be thousands in a fully equipped system. The parallelism potential is elevated by each DPU being capable of thread-level parallelism. Next to the memory of a DRAM module, to which is also referred as Main RAM (MRAM), each DPU possesses a private scratchpad memory called Working RAM (WRAM) which is smaller but faster than the MRAM. Despite the rather limited computational prowess of individual DPUs, the architecture has sparked interest [18, 21, 32, 34] and promising performance in applications like sparse matrix-vector multiplication [16], time series analysis [18], reinforcement learning [17], deep-learning recommendation models [7], and compression [33] make further investigations worthwhile.

One possible new application for PIM is sorting. Sorted data is crucial in many applications, as it allows to search in logarithmic time instead of linear time through a binary search, determine whether a data set is a subset of another one, delete duplicates trivially, or impose a topological ordering on a directed acyclic graph of dependencies for time-forward processing. Consideration of data access patterns plays a vital role in designing sorting algorithms, and the limitations of the various kinds of memory have led to quite sophisticated solutions.

First, Chapter 2 gives a basic overview of the UPMEM hardware and of sorting algorithms in general. Chapter 3 focuses on sequential sorting algorithms for small inputs which fit into the WRAM. Then, Chapter 4 turns to sequential and parallel sorting algorithms for large inputs which are stored in the MRAM because of their size. Chapter 5 concludes this thesis by offering a summary of our findings and giving an outlook on future developments. Appendices A and B contain further measurements in support of Chapters 3 and 4, respectively.

We restrict ourselves to measuring performance on 32-bit and 64-bit integers only. Taking our cue from Axtmann et al. [1], we benchmark our sorting algorithms against the following six input distributions:

Sorted The numbers from 0 to $n - 1$ are generated in ascending order.

Reverse Sorted The numbers from 0 to $n - 1$ are generated in descending order.

Almost Sorted The numbers from 0 to $n - 1$ are generated in ascending order. Then, $\lfloor \sqrt{n} \rfloor$ many random pairs are drawn and swapped sequentially. Pairs may not be disjoint.

Zero-One Every element is set independently to either 0 or 1, each with a probability of 50 %.

Uniform Every element is drawn independently and uniformly from the range $[0, 2^{31} - 1]$.

Zipf's Every element is drawn independently from the range $[1, 100]$, with each value k being drawn with a probability proportional to $1/k^{0.75}$.

The range of possible values in the uniform input distribution is motivated by a simplification of the input generation. There is no impact on the performance on 64-bit integers by their 33 most significant bits being always set to zero. The program code can be downloaded from `https://github.com/s9770652/pim-sorting`.

Chapter 2.

Prerequisites

This chapter lays the necessary groundwork for understanding the design principals of our proposed algorithms. Section 2.1 serves as an introduction to in-memory processing on an UPMEM system mainly in terms of hardware, but major implications for designing code are also covered. Section 2.2 establishes necessary terminology in the context of sorting algorithms. In addition, an insight into some modern sorting algorithms and the reasons for their high performance is provided, along with a discussion on the applicability of the methods to the UPMEM architecture.

2.1. The UPMEM Architecture

Section 2.1.1 provides a short conspectus of the composition of such a system, and Section 2.1.2 presents the components of an individual PIM chip. Section 2.1.3 covers the instruction pipeline, which differs in key aspects from that of a modern central processing unit (CPU), whilst Section 2.1.4 is concerned with the instruction set architecture, as acquaintance with it helps frequently in identifying optimisation potential. Finally, Section 2.1.5 gives an insight into developing programs for an UPMEM system. The four sources for this section are a white paper by UPMEM [43], a talk given by UPMEM's founder, Fabrice Devaux, at the 31st Hot Chips Symposium [11], the official documentation of the UPMEM toolchain [44], and an extensive study by Gómez-Luna et al. [18]. For the sake of clarity, most statements will not be given a specific source.

2.1.1. Overview

The PIM capabilities are realised on modules of regular random-access memory (RAM) or, to be more precise, on Dual In-Line Memory Modules (DIMMs) of Double Data Rate 4 Synchronous DRAM with a transfer rate of 2400 MT/s (DDR4-2400 SDRAM). Therefore, PIM DIMMs can act as replacement for DIMMs already present in existing systems without repercussion for tasks which do not rely on in-memory processing. A PIM DIMM consists of two ranks, each with eight *PIM chips*, that is modified DRAM packages, which contain the memory banks. Each PIM chip, in turn, contains eight *DRAM processing units* (DPUs), so there are 128 DPUs per DIMM. Each DPU is closely situated to one of the memory banks of size 64 MiB. Due to the spatial proximity to its memory bank, a DPU is capable of rapidly accessing data stored on a DIMM.

Depending on the model[1], a DPU possesses either 16 or 24 hardware threads, whose software abstraction are called *tasklets*. Taklets work independently from each other, meaning programs can use different control flows to process different data. An UPMEM system can boast up to 20 PIM DIMMs, setting the total count of DPUs to 2560 and of tasklets to 40 960 or 61 440, respectively. Tasklets of the same DPU communicate using shared memory, whereas DPUs have no direct way to communicate or even share data with each other. Instead, inter-DPU communication is implemented by the host CPU fetching data from one DPU and sending it to another one. Hence, for a task to run well on a PIM system, it not only needs to frequently access the RAM, it also needs to consist of many, fairly independent subtasks. If such a task is indeed on hand, speedups well in the double digits for memory-bound tasks, compared to an execution on a CPU or graphics processing unit (GPU), are possible (compare Gómez-Luna et al. [18]). Next to a faster execution, a gain in power efficiency is also to be expected, since data transfers between the RAM and a host CPU drive the power consumption in regular systems significantly; UPMEM claims a tenfold increase of the power efficiency.

The retention of the general DDR4 architecture comes at a price. A DPU is manufactured using only three layers of silicon, resulting in transistors three times slower than other transistors of the same process node. Also, their density is considerably reduced. In consequence, DPUs are not suitable for computing-intensive tasks (compare also Gómez-Luna et al. [18]).

1. There are two DPU models, v1A and v1B. The former runs at 350 MHz and is equipped with 24 threads, whereas the latter runs at 400 MHz and is equipped with 16 threads. Measurements for this thesis were conducted on v1A.

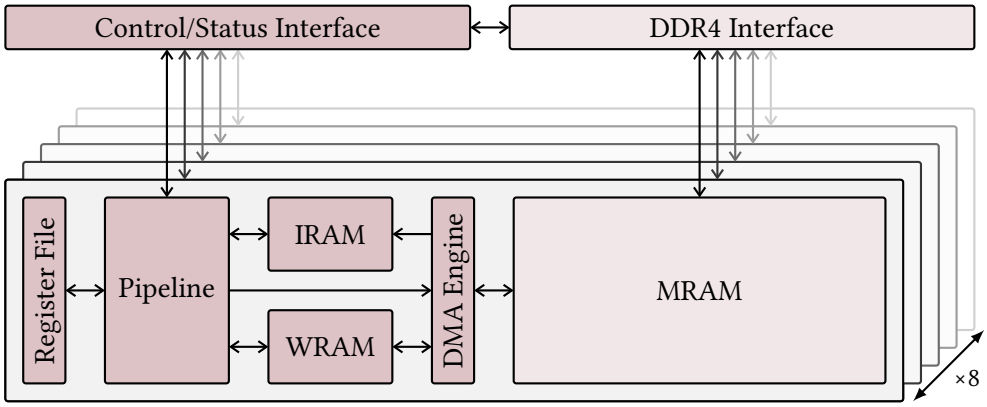


Figure 2.1. *Schematic depiction of a PIM chip. The bright components are part of a standard DRAM package, the dark components are exclusive to PIM chips. [11]*

2.1.2. The Structure of a PIM Chip

A PIM chip (Fig. 2.1) contains eight DRAM banks of 64 MiB each. These are connected with a regular DDR4 interface through which a host CPU can access the memory. Next to each DRAM bank, there is a DPU with a direct connection to it, thus bypassing the DDR4 interface. An access to the memory bank is also called a *direct memory access* (DMA) and is handled by the *DMA engine*. Furthermore, the eight DPUs are connected with a special control interface which, in turn, is connected with the memory controller. This enables the host to communicate with the DPUs but it does not allow DPUs to access DRAM banks other than their own. It is not possible for a DPU and the host to access a DRAM bank concurrently.

DPUs contain several major and minor memories. The memory of the DRAM bank is also referred to as *Main RAM* (MRAM). It is by far the largest memory of a DPU and typically holds the input provided by the host and the output calculated by the DPU. However, the MRAM is also the slowest memory, for each DMA comes with a latency of dozens of cycles. Also, despite the theoretical peak bandwidth of 2.2 GiB/s of DDR4-2400, the empirically measured bandwidth of a DMA is 0.6 GiB/s on a v1A DPU.

The *Working RAM* (WRAM) is based on faster but more expensive and less dense static RAM. For this reason, the WRAM is far smaller, comprising only 64 KiB, yet its latency is practically zero, and the measured bandwidth for a v1A DPU reaches 2688 MiB/s. A typical workflow is, hence, to load input data from the MRAM into the WRAM, process it, and write output data back into the MRAM. The WRAM also contains the stacks of the tasklets, where their local variables are stored. Global variables which are visible to every tasklet may be stored in the WRAM or MRAM, however, any MRAM variable can be processed only when placing it temporarily in the WRAM (see Section 2.1.4). Unlike a CPU, there is no multilevel cache hierarchy with a coherence protocol moving data automatically, and it is in the responsibility of the programmer to ensure that critical data are stored in the WRAM. Still, there is a small number of automatically managed registers. The driver allows the host to access a specific section of the WRAM only if the data has been specifically designated for this purpose, and such transfers are slower than transfers involving the MRAM.

Tasklet	Cycle																		
	21	22	23	24	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
0	J	K	L/A	M/B	N/C	D	E	F	G	H	I	J	K	L/A	M/B	N/C	D	E	F	G
1	I	J	K	L/A	M/B	N/C	D	E	F	G	H	I	J	K	L/A	M/B	N/C	D	E	F
2	H	I	J	K	L/A	M/B	N/C	D	E	F	G	H	I	J	K	L/A	M/B	N/C	D	E
3	G	H	I	J	K	L/A	M/B	N/C	D	E	F	G	H	I	J	K	L/A	M/B	N/C	D

Figure 2.2. *An excerpt from cycles 21 to 39 of an exemplary pipeline with four threads which were issued one cycle apart. The fourteen letters A to N represent the fourteen stages of an instruction. Due to the interleave, every pair of subsequent, final stages N is eleven cycles apart.*

Whilst the WRAM typically holds the data which is processed, the *Instruction RAM* (IRAM) contains the program (also called *kernel*) which a DPU executes. The IRAM has a size of 24 KiB which translates to a maximum of 4096 instructions out of which a kernel has to be built.[2] This memory can be modified only by the host, as the DPU can only read it in usually automated processes.

Next to these major memories, there is also a 256 bits large *atomic memory* whose bits are accessible in a thread-safe way, allowing for mutual exclusion, barriers, and similar. Furthermore, there is a 64 bits large *run memory* through which individual threads can be booted, suspended, and resumed by setting the corresponding status bit.

2.1.3. The Instruction Pipeline

Instructions are executed using *pipelining*, that is, instructions are divided into several stages which are performed one after another, with each stage taking exactly one cycle. Once a stage has been completed, the respective transistors are free to process the next instruction even if the previous instruction has not reached the end of the pipeline yet. The pipeline is *scalar*, meaning there is at most one instruction per stage at any time, and *executes in order*, meaning instructions are statically scheduled and executed in the order as indicated by the compilation. Threads can have only one scheduled instruction in the pipeline. However, all threads use the same pipeline, so a nominal throughput of one instruction per cycle is achieved if enough threads are active for all stages of the pipeline to be continuously performed (Fig. 2.2). The pipeline consists of fourteen stages such as the fetching of the instruction from the IRAM, the reading of the operands from the registers, and the performing of the operation itself while accessing the WRAM if needed. The last three stages can be interleaved, that is performed in parallel, with the first three stages. Thence, the pipeline length is effectively reduced to eleven, meaning only eleven active threads are needed to exploit the computing capabilities of a DPU in full capacity.

Nevertheless, having more than eleven threads active is not detrimental to the throughput, which remains at one instruction per cycle, it only means that individual threads are put into a queue and have to wait for some cycles. This not only may make some parallel task easier

2. In fact, a v1B DPU can hold 2 KiB of data less in its MRAM and 128 instructions fewer in its IRAM since parts of those are 'reserved for production and quality control purposes.' [44, Introduction – DPU chip characteristics]

to program, it can result in a performance gain when DMAs are involved. DMAs are mainly executed by the autonomous DMA engine. Whilst a thread is performing a DMA, it is suspended and removed from the pipeline, freeing a slot up. Therefrom, the employment of more than eleven threads allows to hide DMA latency by keeping the pipeline full.

As concluding remark, it shall be mentioned that there are circumstances under which the execution of an instruction takes twelve cycles instead of eleven. [44, Instruction Set Architecture – Efficient scheduling] This is related to the identifiers of the registers used, however, the compiler usually manages to evade these circumstances. Hence, one can regard a DPU as a *unit-cost machine* where each instruction takes eleven cycles to complete with the seldom exception of some taking twelve cycles and with the exception of DMAs. Counting instructions is, therefore, a valid technique to assess the performance of some piece of code.

2.1.4. The Instruction Set Architecture

Each thread owns several private *general-purpose registers* labelled `r0` to `r23` which can hold arbitrary 32-bit values and are freely readable and writeable by the respective thread. An even Register `r(2i)` and subsequent odd Register `r(2i + 1)` form the 64-bit Register `d(2i)`. Furthermore, there are the read-only Registers `id`, `id2`, `id4`, and `id8`, which hold the identifier of the respective thread, multiplied by 1, 2, 4, and 8. Also, there are special registers for the program counter holding the IRAM index of the next instruction to execute, a performance counter used for measuring the time, a carry bit, and the zero flag. Last but not least, there are four read-only registers which are shared by all threads: the Registers `zero` and `one` hold, as their names suggest, the constants 0 and 1, whereas the Registers `lneg` and `mneg` hold the least negative and most negative 32-bit values, that is -1 and -2^{31}.

A DPU is a reduced instruction set computer (RISC) with mainly 32-bit instructions — most 64-bit operations are pieced together from several 32-bit ones, thereby taking more than eleven cycles. There is no hardware support for multiplication or division, so these are emulated by functions, thereby taking even longer. On top of that, there is no hardware support for floating point arithmetic, requiring costly emulation as well.

Instructions follow a 3-operands design, meaning there can be up to three register arguments to an instruction, with the target register coming first. Next to registers, it is also possible to pass *immediate values*, that is constant values passed directly without a register, and *labels*, which are effectively IRAM indices of instructions. Some examples:

- `move r6, 4` stores the immediate value `4` in Register `r6`.
- `lw r13, r12, -4` loads the 32-bit word which is four bytes away from the WRAM address stored in Register `r12` into Register `r13`. Note that all addresses are physical.
- `add r1, r5, r11` takes the 32-bit integers in Registers `r5` and `r11`, adds them, stores the result in Register `r1`, and sets the carry bit accordingly.
- `addc r0, r4, r10` performs an addition taking the carry bit into account, allowing to perform one 64-bit addition by invoking two 32-bit instructions.
- `jump .LABEL` sets the program counter to the index of the labelled instruction.

Despite their name, some of the general-purpose registers do have conventional uses. The Registers `r0` to `r7` are filled with the arguments of a function before it is called. The return value

of a function is written to the Registers `r0` or `d0`, depending on whether it is 32 bits or 64 bits long. Register `r22` contains the stack pointer, that is the address of the currently last element in the stack of the respective tasklet. When a function is called and it need store data on the stack, it saves the original value of the stack pointer on the stack itself before incrementing the stack pointer, therethrough allocating new memory. When the function terminates, it loads the original stack pointer value back into Register `r22`, therethrough deallocating memory. Register `r23` contains the return address, that is the IRAM index of the instruction whither to jump after the termination of a function. Here, the instruction to load and store 64-bit large double words are of particular use. By invoking `sd r22, <offset>, d22`, the content of both Registers `r22` and `r23` is stored to some position relative to the current stack pointer, whence it can be recovered by invoking `ld d22, -<offset>, r22` later on. Thereby, the bandwidth of the WRAM is effectively doubled and the instruction count reduced.

The capabilities of DPU instructions is substantially enhanced by the plethora of *conditions*, of which there are a total of 51. Conditions are binary flags which are passed as additional arguments to instructions so that those act as either test operation or combo operations. A *test instruction* performs its usual purpose but stores the evaluation of the condition in the target register. For example, the instruction `add r0, r0, -1, pl` takes the content of Register `r0`, decrements it, and checks the condition `pl`. This condition evaluates to true if the result is greater than or equal to zero. Therefore, Register `r0` will contain the value 1 if and only if Register `r0` used to store the number 1 or greater, and will contain 0 otherwise. A *combo instruction* takes a label as yet another argument. The instruction performs its usual purpose, checks whether the result fulfils the condition, and, if yes, performs a jump to the label. An example is the instruction `add r0, r0, -1, pl, .LABEL_LOOP`, where Register `r0` holds a loop index which get decremented. Should Register `r0` now hold a value greater or equal to zero, a jump back to the beginning of the loop body marked by the label `.LABEL_LOOP` is performed. Otherwise, the next line of the compilation is executed. This way, it takes just eleven cycles to update the loop index, check the loop condition, and perform the appropriate action. Such techniques of saving instructions are especially valuable because DPUs are incapable of instruction level parallelism. Although conditions are employed automatically by the compiler for the most part, Chapter 4 includes a manual use of conditions.

2.1.5. Programming a Kernel

Executing tasks on an UPMEM system requires both a program executed on the host CPU and a kernel executed on the DPUs. DPUs are handled in groups of up to 64 DPUs from the same rank of a DIMM. The groups, in turn, are aggregated in a *DPU set*. A typical course of action is the following:

1. Start the host program.
2. Write the input to the MRAM and/or WRAM of all involved DPUs.
3. Boot the DPUs and execute the kernel synchronously or asynchronously.
4. Read the output from the MRAM and/or WRAM.
5. Go back to the second or third step if needed.

When kernels are executed synchronously, the host cannot access the memory of a DPU until all

DPUs in the whole set have finished. But even with an asynchronous execution, the host cannot access the memory until all DPUs in the same rank have finished. Note that data is generally not deleted when a kernel finishes, so subsequent executions can hark back to previous results. Also, any communication between the host and the DPUs must be initiated by the host. The host program can be written in C, C++, Java, or Python. Apart from a few additional functionalities provided by the UPMEM application programming interface (API) for communicating with the DPUs, the host program is a regular executable.

The software development kit includes a simulator which allows to run kernels on machines without UPMEM DIMMs. The kernel has to be written in either C or assembler, but we will focus on the former. Its entry point is the `main` function, thence one can proceed as in any C program. All tasklets execute the same kernel, but their control flow can be changed by simply including conditionals on the tasklet identifiers. Synchronisation between tasklets can be achieved, amongst others, via barriers, mutual exclusion, and semaphores. Communication between tasklets is achievable by defining global variables. The C standard library is only partially available as some compute-intensive functionalities have been not been implemented, for example the entire `math` library.

The biggest changes to a regular C program are in relation to the memory. Any variable resides in the WRAM by default, but creating an MRAM variable is as easy as adding the qualifier `__mram` to the variable declaration. By default, too, any pointer is assumed to point to data in the WRAM, which can be changed by adding the qualifier `__mram_ptr`. The compiler correctly identifies confusion between pointers of different address spaces.

Local WRAM variables are created on the stack of the respective tasklet, and the stack sizes can be set for each tasklet individually at compile time. Nonetheless, it is possible for tasklets to dynamically allocate more space on the WRAM via an allocator similar to the standard C function `malloc`. Although this is called *heap allocation*, the name is misleading. The compiler organises the WRAM such that all tasklet stacks and anything else statically allocated on the WRAM is in the front, so that the free memory comprises a contiguous block in the back of the WRAM. Then, the so-called *heap pointer* is set to the beginning of the free block. When memory is allocated on the heap, the heap pointer is sufficiently incremented to mark the space as reserved. Afterwards, the original position of the heap pointer is returned to the allocating tasklet. In other words: the heap memory is simply a stack memory shared by all tasklets. Indeed, a DPU lacks an equivalent to the standard C function `free` to deallocate heap memory. The only possibility is to reset the entire heap by setting the heap pointer back to its initial position. There is also no model of ownership, so tasklets can write to any memory address, including the stack and heap memory of other tasklets. Heed must be paid when structuring the scarce WRAM, which is subject again in Section 4.2.

As hinted before, transferring data between the MRAM and the WRAM is in the responsibility of the programmer. When only single elements are to be accessed, variables in the MRAM can be treated like normal variables. For example, `var = arr[i]` is valid code no matter whether the array `arr`, the variable `var`, or the index `i` have been declared to reside in the WRAM or the MRAM. However, this still constitutes one or more DMAs on each use, and each DMA comes at a cost. According to measurements [18], reading from the MRAM has an overhead of 77 cycles, whilst writing to the MRAM has an overhead of 61 cycles. The transfer of each byte costs a further 0.5 cycles. This means that DMAs of about 128 B or less are dominated by the

overhead. Therefore, it is recommended to move large blocks of MRAM data into the WRAM, perform calculations there, and move the modified data block back to the MRAM. This way, the overhead is mitigated. Note that, like for WRAM data, the time to access MRAM data is independent of the exact location — only the memory type matters.

To perform such blockwise moves, one calls the C function `mram_read` and `mram_write`, which take a source address, a target address, and the number of bytes to use transfer. There are, however, several limitations.

- The WRAM address must be aligned to 8 bytes. This can be ensured automatically by adding appropriate qualifiers to stack variables or by using heap memory which gets properly aligned automatically.
- The MRAM address must be aligned to 8 bytes. No special functionality exists to this end; it is up to the programmer to organise the MRAM with this limitation in mind and to resort to DMAs to single elements if such an alignment is not given.
- The number of transferred bytes must be at least 8, at most 2048, and a multiple of 8.

Failing to fulfil these constraints can result in missing or corrupt data. The DMA engine works sequentially, meaning data for only tasklet can be transferred at a time. If multiple tasklets call `mram_read` or `mram_write`, some of them will be suspended for longer as they wait for the other DMAs to finish. If DMAs are very frequent, having many active tasklets is especially important to keep the pipeline full.

For a performant kernel execution, it is generally recommended to restrict oneself to simple 32-bit logic as much as possible. Some 64-bit functionalities are executed in eleven cycles, like loads and stores, but most take twice or even thrice as long. Multiplication, division, and floating point arithmetic are emulated in software, so they should be avoided if necessary. In their stead, addition, subtraction, and bitwise logic should be used. Also, due to the unit-cost model, a decrease in the count of instructions translates into a performance gain. Unfortunately, a common issue is a nosediving quality of the compilation, perhaps resulting from a wrong configuration of the LLVM-based compiler. Investigating the compilation and trying different approaches — including even mundane alternatives like reordering independent `if` statements — is paramount when aiming for top performance and, therefore, a recurring theme in this thesis. In our experience, explicitly saving the result of a computation if the value is reused at a later time prevents the compiler from issuing a recalculation which would elsewise hurt due to the compute-boundedness of the architecture. Also, it seems that pointer logic tends to be compiled better than index logic. Lastly, inlining leads to a performance gain oftentimes as the overhead for function calls is quite heavy. Even though the call itself is a mere jump taking elven cycles, several registers must be saved and reloaded on entering and exiting a function; for an empty function with two arguments, we determined a call overhead of 144 cycles. This number can easily rise with heavier register usage. This may also explain why turning arguments into global variables nets a performance gain in some cases as well.

2.2. Fundamentals of Sorting

This section is concerned with algorithm which solve the sorting problem for numbers.

Problem 1 (Sorting). Given an input sequence $\langle a_1, a_2, \dots, a_n \rangle$ of n numeric elements, find a permutation π of the set $\{1, \dots, n\}$ such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$ holds.

Albeit important, the runtime is not the only distinguishing feature of a sorting algorithm. Stability and the memory footprint are remarkable, too. We specify space complexity in the form of the count of indices, pointers, or numbers of the same type as the input, thus ignoring their lengths in bits.

Definition 2. A sorting algorithm works *in place* if it needs auxiliary space of size $O(1)$ to sort an input of n elements. A sorting algorithm works *out of place* if it does not work in place.

Definition 3. A sorting algorithm is *stable* if equal elements remain in the same order in the output as in the input. A sorting algorithm is *unstable* if it is not stable.

As already pointed out in Chapter 1, sorting is a fundamental problem in computer science and is being worked on at least since the middle of the 20th century [14, 20, 47]. Multiple types of sorting algorithms have emerged throughout the decades. For instance, if every input element is of one of k possible values, one may use a CountingSort where the input sequence is scanned and particular counters are incremented when passing elements, thereby achieving a runtime in $O(n + k)$. If the input elements have few significant digits, RadixSort may be useful. RadixSort has k rounds if the input elements have k digits, and in the ith round, elements are distributed amongst buckets according to their ith digit, whereby a runtime in $O(kn)$ is achieved. The arguably most extensive category are comparison-based sorting algorithms, which need a comparison operation establishing a total preorder over the data. Their runtimes can differ greatly from one another and also between input sequences, however, there is a lower bound on the runtime.

Theorem 4. *A comparison-based sorting algorithm cannot be faster than $\Omega(n \log n)$ in the worst case. [30, pp. 91 sq.]*

Proof. Sorting can be understood as identifying the current permutation of the input and applying a new one. The identification process can be modelled through a binary decision tree. Each comparison made corresponds to going one level further down from a parent vertex to a child vertex. When reaching a leaf, the input is fully identified, and the number of comparisons made along the way matches the depth of the leaf. Since there are $n!$ possible permutations, there must be at least $n!$ leaves. A binary tree of depth d has at most 2^d many leaves. Thus, we get

$$2^d \geq n! \implies d \geq \operatorname{lb}(n!) \geq \operatorname{lb}\left((n/2)^{n/2} + 0^{n/2}\right) = n/2 \cdot \operatorname{lb}(n/2) \in \Omega(n \log n)$$

as lower bound on the depth of the decision tree and, consequently, on the number of comparisons in the worst case. □

By arguing that the average depth of leaves in a binary tree is at least $\lfloor \operatorname{lb}(n!) \rfloor$ [4], a lower bound of $\Omega(n \log n)$ is shown for the average case as well. Indeed, there are sorting algorithms

which are asymptotically optimal in the average case and even in the worst case. Nevertheless, suboptimal algorithms can be advantageous, for example, on shorter inputs due to better constant factors hidden in the asymptotic notation or on inputs which display exploitable patterns. A sorting algorithm which combines two or more sorting algorithms is called a *hybrid sorting algorithm*.

Many if not almost all modern sorting algorithms are hybrids and, despite the age, still rely in principle on *HeapSort* (see Section 3.3), which uses a heap as a priority queue to find the next greatest element, *QuickSort* (see Section 3.4), which compares elements against a pivot element to form two partitions of little and great elements, respectively, and *MergeSort* (see Section 3.5), which repeatedly merges sorted subsequences. Whilst QuickSort is usually the fastest of the three, its runtime on some inputs is suboptimal. IntroSort [31] circumvents this issue by switching to HeapSort when QuickSort takes too long since HeapSort has a guaranteed runtime in $O(n \log n)$. Pattern-defeating QuickSort [35] switches to HeapSort even earlier by detecting when the problem size is not diminishing enough. Both hybrids fall back to the asymptotically suboptimal InsertionSort if the problem size is sufficiently reduced — a technique shared with the MergeSort-based TimSort [36]. Fallback algorithms are also useful on DPUs thanks to their reduction of the instruction count, as will be seen in later chapters.

Another source of performance gains arises from *loop transformations*. This includes loop unrolling where the step size is multiplied by an unroll factor x and the loop body is repeated x times (see Section 3.5.1 for a more profound discussion). Not only is this advantageous on CPUs in sorting algorithms like IPS^{4}o [1], S^5 [2] or SkaSort [41] but also on DPUs where reducing the instruction counts in hot loops is critical. Other loop transformations, however, have little to no effect on a DPU or might be even hurting performance through an increased loop overhead. S^5 uses loop fission, that is, it breaks loop bodies apart and puts them into own loops. This can be useful if, for example, one were to calculate `arr[i] = 2 × arr[i] + 1` in a lopp. The loop can be broken apart such that a first loop calculates `arr[i] = 2 × arr[i]` and a subsequent one `arr[i] = arr[i] + 1`. Now, the calculation is eligible for vectorisation, that is the simultaneous exertion of the same operation on multiple elements, of which a DPU is incapable.

Other enhancements are not applicable to DPUs, too. When a CPU encounters a branch, it speculates on whether the branch will be taken and fills its pipeline accordingly before the evaluation of the branch condition is finished. When a misprediction happens, the work was in vain and the pipeline of the CPU has to be flushed. According to information theory, QuickSort is the fastest when the problem size is perfectly halved in each step. However, with a perfect fifty-fifty split, no branch predictor can effectively speculate on the future of individual elements. For this reason, a skewed split can improve the runtime in spite of an increase of the instruction count [22]. BlockQuickSort [12] and IPS^{4}o eliminate branch prediction altogether by writing the indices of wrongly ordered elements to buffers of which is taken care at a later time. They do so by maintaining an index of the current position in the buffer and writing the index of every element to this index unconditionally. The comparison result of the check on whether an element needs to be displaced is cast to either zero or one and added to said index. As a result, the index of an element which needs no displacement gets overwritten upon visiting the next element. Similarly, IPS^{4}o and S^5 use casting to calculate positions in a decision tree branchlessly to quickly distribute elements amongst buckets. It is reasonable to assume that

these techniques would be disadvantageous on a DPU as they do effectively nothing besides increasing the instruction count.

Of course, there are also design decisions which are related to memory accesses. IPS^{4}o was contrived with the parallel external memory model in mind in order to limit I/O on systems with non-uniform memory access. In this model, each thread has a small private cache of size M and can access the large external memory in blocks of size B. Such a consideration is not inappropriate in the context of DPU algorithms, for a single DPU already accesses memory not in a uniform way due to the split of WRAM and MRAM. The non-uniformity is exacerbated when considering algorithms where DPUs access the MRAM of others. Indeed, limiting DMAs is subject of Section 4.3 although our use case is simple enough to omit a theoretical analysis via the external memory model.

Chapter 3.

Sorting in the WRAM

This chapter is concerned with sorting data which fits into the WRAM entirely. The presented sorting algorithms are designed for sequential execution by a single tasklet. Key characteristics of the DPU architecture which are of especial importance in this chapter are the uniform access to WRAM data and the unit-cost of instructions.

Section 3.1 discusses InsertionSort which is a component of all algorithms presented thereafter as it is lightweight and performant on short inputs. Section 3.2 covers ShellSort which is a generalisation of InsertionSort. Sections 3.3, 3.4, and 3.5 deal with HeapSort, QuickSort, and MergeSort, respectively, which are more elaborate algorithms suitable for long inputs. Every section gives a short presentation of its respective algorithm at the beginning, ensued by a discussion of key parameters in their designs. This is usually followed by an insight into non-algorithmic challenges faced during development caused by the compiler whose optimisations are often of suboptimal quality. Finally, an evaluation of the performance of the respective algorithm completes each section. Section 3.6 summarises the findings on the presented algorithms and gives a brief outlook on future improvements.

Appendix A contains a comprehensive collection of measurements but the ones essential for following the content of this chapter are also presented in figures herein. Every measurement was repeated a thousand times with the sorting algorithms in their default configuration unless explicitly noted otherwise. The meaning behind and reasoning for the individual parameters in the configurations are subject in Sections 3.1 to 3.5 but shall be mentioned already for ease of reference:

InsertionSort explicit sentinel value

ShellSort explicit sentinel values; step sizes $h = (1, 6)$ for inputs with at most 64 elements and $h = (1, 4, 17)$ for longer ones

HeapSort top-down for 32-bit integers; bottom-up with swap disparity for 64-bit integers

QuickSort fallback threshold of 18 elements; random medians as pivots; prioritisation of right-hand partitions over left-hand partitions; iterative for 32-bit integers; recursive for 64-bit integers; Handling (5)

MergeSort half-space; starting run length of 14 elements

Measurements are confined to at most 1024 elements. The reason is that 64 KiB of WRAM are available and that most kernels will run with at least 11 tasklets to saturate the instruction pipeline. In consequence, at most 5957 B are allotted to each tasklet. The tasklet stack accounts for 600 B of this amount, leaving space for about 1339 32-bit elements. On that score, 1024 elements is a reasonable cutoff.

3.1. InsertionSort

InsertionSort works by moving the ith element leftwards as long as its left neighbour is greater, assuming that the elements with indices 0 to $i-1$ are already sorted [30, p. 83, 48, Section 2.2.1]. Its asymptotic runtime is above the theoretical minimum of $\Omega(n \log n)$, reaching $\Omega\left(n^2\right)$ not only in the worst case but also in the average case, since any of the $\binom{n}{2}$ pairs of input elements is in wrong order, needing to be swapped at some point in the execution, with probability 50 %. Nonetheless, InsertionSort does have more than one silver lining:

1. If the input array is mostly or even fully sorted, the runtime drops down to $O(n)$.
2. It works in place, needing only $O(1)$ additional space.
3. The sorting is stable.
4. Its implementation is short, lending itself to inlining.
5. The overhead is small.

Especially the last two points make InsertionSort a good fallback algorithm for asymptotically better sorting algorithms to use on short subarrays.

3.1.1. Presentation of Key Aspects

Sentinel Values When moving an element to the left, two checks are needed: Does the left neighbour exist and is it less than the element to move? The first check can be omitted through the use of *sentinel values* [48, p. 93]: If the element with index -1 is permanently set to the least possible value of the chosen data type, it is at least as little as any element in the input array, and the leftwards motion stops there at the latest. Since a DPU lacks branch prediction, the slowdown from performing twice as many checks as needed is quite high and goes up to 30 % for short inputs with 24 uniformly distributed 32-bit elements.

Setting such an *explicit* sentinel value can be omitted by using *implicit* sentinel values [38]. At the start of round i, one can check whether the element with index 0 is at least as little as the element with index i. If so, the former is a sufficient sentinel value, and InsertionSort can proceed as normal. If not, the latter must be the minimum of the first $i+1$ elements. Therefore, one can shift the first i elements one position backwards and place the minimum in the front. For simplicity, the words ‘explicit’ and ‘implicit’ are, henceforth, applied to the word ‘InsertionSort’ directly to imply the type of the sentinel value used.

3.1.2. Investigation of the Compilation

Figure 3.1a shows a naïve implementation of InsertionSort which begins at the very start of the input. Obviously, nothing happens in the first round as a lone element is already sorted, so it is algorithmically sound to let InsertionSort begin at the second element. This optimisation is accomplished in Fig. 3.1b. Surprisingly, it leads to a longer runtime! For instance with sixteen 32-bit integers, the runtime is increased by nine instruction. The same increase happens if one keeps `*i = start` and uses `curr = ++i` instead to begin at the second element.

Looking at the compilation reveals the reason: In the naïve version, the pointer `pred` is optimised away and, in its stead, the pointer `curr` is passed to all load instructions together with a constant offset of `-4`. In the optimised version, the pointer `pred` is used together with

```c
void InsertionSort(int *start, int *end) {
  int *curr, *i = start;
  while ((curr = i++) <= end) {
    int to_sort = *curr;
    int *pred = curr - 1;
    while (*pred > to_sort) {
      *curr = *pred;
      curr = pred--;
    }
    *curr = to_sort;
  }
}
```

(a) *Start at the first element and with predecessor pointer.*

```c
void InsertionSort(int *start, int *end) {
  int *curr, *i = start + 1;
  while ((curr = i++) <= end) {
    int to_sort = *curr;
    int *pred = curr - 1;
    while (*pred > to_sort) {
      *curr = *pred;
      curr = pred--;
    }
    *curr = to_sort;
  }
}
```

(b) *Start at the second element and with predecessor pointer.*

```c
void InsertionSort(int *start, int *end) {
  int *curr, *i = start;
  while ((curr = i++) <= end) {
    int to_sort = *curr;
    while (*(curr - 1) > to_sort) {
      *curr = *(curr - 1);
      curr--;
    }
    *curr = to_sort;
  }
}
```

(c) *Start at the first element and without predecessor pointer.*

```c
void InsertionSort(int *start, int *end) {
  int *curr, *i = start + 1;
  while ((curr = i++) <= end) {
    int to_sort = *curr;
    while (*(curr - 1) > to_sort) {
      *curr = *(curr - 1);
      curr--;
    }
    *curr = to_sort;
  }
}
```

(d) *Start at the second element and without predecessor pointer.*

Figure 3.1. *Four different implementations of InsertionSort in C. Figures 3.1a and 3.1c are compiled the same. Figures 3.1b and 3.1d are compiled differently.*

offsets of `+4` and `0` to fetch the values of `to_sort` and `*pred` at the beginning of each round. Only then, the pointer `curr` is initialised before being used in the inner loop as in the naïve version. Effectively, this initialisation introduces one more instruction.

These changes fully explain the extension of the runtime by nine instructions: The optimised version invokes one instruction at the beginning of the function to advance the starting position and loops 15 times in total, each time initialising the pointer `curr`. The naïve version loops 16 times, invoking seven instructions for naught in the first iteration.

Multiple workarounds exist to sidestep this problem. One of them is to take the unoptimised code and change the starting position via inline assembler. This is trivial for the explicit InsertionSort since one can simply inject an `add` instruction at the beginning of the function to increment the pointer `start`. The implicit and the sentinel-less InsertionSorts need to know the original starting address `start` later on, though, and initialise the actual starting point rather late; injecting inline assembler proves more difficult as a consequence. Moreover, as

InsertionSort is to be used as fallback algorithm by other sorting algorithms which might also need to keep the original value of `start`, inline assembler is a bad choice even for the explicit InsertionSort.

Another workaround is the usage of a wrapper function calling InsertionSort with the arguments `start + 1` and `end`. This works quite well: First, the register holding `start` is incremented, and, then, the inlined code from the actual InsertionSort follows. Doing so makes the runtime drop as expected.

Recall how in the faster version (Fig. 3.1a), the pointer `pred` is always deduced from the pointer `curr` using an offset. This gives the cue for yet another workaround: In Figs. 3.1c and 3.1d, every occurrence of `pred` is replaced with `curr - 1`. As a consequence, the code of Fig. 3.1c compiles the very same as the one of Fig. 3.1a, whilst Fig. 3.1d yields the same compilation as the versions with the wrapper function or the inline assembly. This workaround is clearly the best of the three and, hence, the one used in the rest of this thesis.

Alas, the eternal struggle with the compiler endeth not herewith. A deeper look into the compilation reveals the following sequence:

```
move r3, r0           // copy content of register r0 to r3
jleu r4, r2, .LABEL   // jump to .LABEL if r4 ≤ r2
move r3, r0
```

Without delving further into its significance — the second `move r3, r0` is unneeded as it is impossible to jump directly to it nor to return via `jleu`. Copying the whole assembler code and injecting it as inline assembler but without this second `move r3, r0` pushes the runtime even further down whilst maintaining the correctness of the output. New issues, especially for inlining, are introduced, though, and we deem a proper assembly implementation as out of scope for this thesis.

3.1.3. Evaluation of the Performance

The runtimes of the three InsertionSorts can be compared in the Figs. 3.2, A.1 and A.2. The sentinel-less InsertionSort is consistently worse than the explicit one. For most input distributions, the implicit InsertionSort is also a bit slower, as it effectively performs one check more for each element. Of course, the gap becomes less significant with increasing input length as the other operations of the loops dominate the runtime.

An outlier, however, are the reverse sorted inputs. For 32-bit numbers (Fig. 3.2), the speedup[1] of the implicit InsertionSort over the explicit one drops down to as little as 0.68. This comes as a surprise since both versions effectively execute the same loop body while shifting everything one position backwards, with only the loop condition being different. Due to DPUs being unit-cost machines, a value check on whether the preceding element is less (explicit InsertionSort) and an address check on whether the preceding position is the start of the array (implicit InsertionSort) should take the same amount of time. Yet, even the sentinel-less InsertionSort surpasses the implicit InsertionSort, despite doing both value checks and address checks. For 64-bit numbers (Fig. A.2), the implicit InsertionSort would be expected to perform better than

1. The *speedup* S of an algorithm A over an algorithm B is defined as the ratio $\mathrm{t}(B)/\mathrm{t}(A)$ of their runtimes $\mathrm{t}(A)$ and $\mathrm{t}(B)$. Values below 1 indicate that algorithm A runs slower than algorithm B.

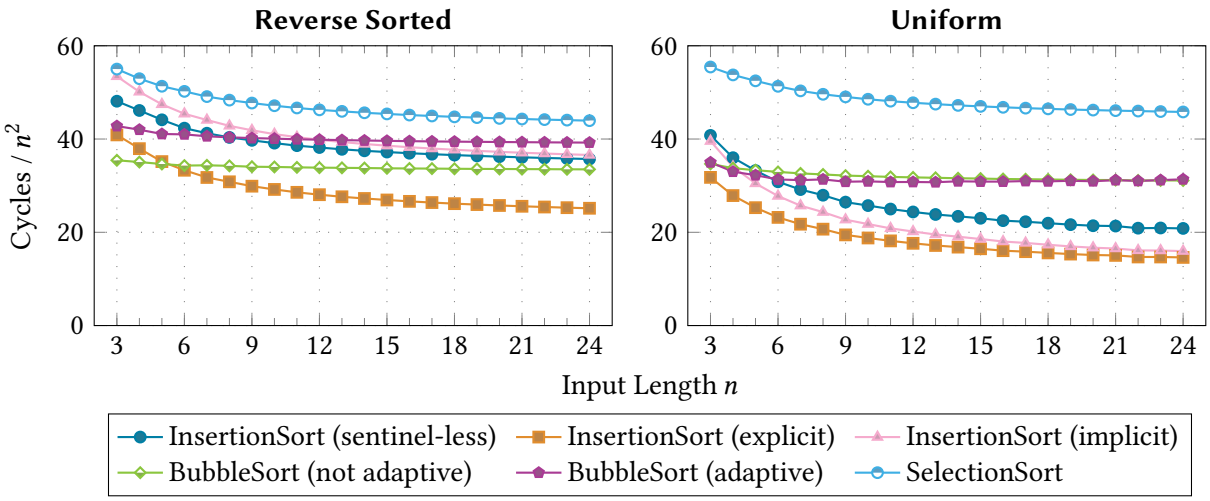


Figure 3.2. *Mean runtimes of sorting algorithms with a runtime in* $O(n^2)$ *on 32-bit integers.*

the explicit one, considering that a value check now takes two instructions and an address check still only one. Nonetheless, the two InsertionSorts tie. This constitutes another case of bad compilation. We did not troubleshoot as the explicit InsertionSort would still be expected to offer superior performance in most cases. The explicit InsertionSort is, therefore, used in the rest of this thesis and referred to plainly as 'InsertionSort' henceforth.

Note. Other known simple sorting algorithm are SelectionSort and BubbleSort. *SelectionSort* [30, p. 83, 48, Section 2.2.2] assumes, like InsertionSort, that the elements with indices 0 to $i-1$ are already sorted in round i. It scans the elements with indices i to n and finds their minimum. Then, it swaps a minimum element with the element with index i. *BubbleSort* [48, Section 2.2.3] scans the elements with indices 0 to $n-i+1$ and swaps each pair of neighbouring elements if they are in the wrong order. An easy extension is adaptive BubbleSort which sorts only up to the position of the last swap.

The average-runtime complexity of SelectionSort and BubbleSort is the same as that of InsertionSort. The asymptoticity, however, hides much higher constant factors such that InsertionSort should always be preferred, as seen in Figs. 3.2, A.1 and A.2. Consequently, they will not be expanded on further in this thesis.

3.2. ShellSort

InsertionSort suffers from little elements in the back of the input, since those have to be brought to the front through $\Theta(n)$ comparisons and swaps. ShellSort [39, 48, Section 2.2.4] circumvents this by doing k passes of InsertionSort with decreasing step sizes: In pass $p = 1, \dots, k$ with step size h_{k-p}, the input array is divided into h_{k-p} subarrays so that the ith subarray contains the elements with indices $(i,\ i + h_{k-p},\ i + 2h_{k-p},\ \dots)$, for $0 \leq i < h_{k-p}$. These subarrays then get sorted individually through InsertionSort. The final step size is $h_0 = 1$ such that a regular InsertionSort is performed. Intuitively, early InsertionSorts are fast as they touch only few elements and little elements in the back are brought forward in large steps. Later InsertionSorts are also fast as elements are close to being sorted. Like regular InsertionSort, ShellSort works in place, but it loses the stability property.

Finding the right balance between the heightened overhead through multiple InsertionSort passes and the shortened runtime of each InsertionSort pass is subject to research to this day [27, 42] and depends on the cost of the operation types (comparing, swapping, looping). Traditionally, step sizes were constructed mathematically, allowing to determine ShellSort's runtime to be, for example, $O(n^{1.2})$ [48, p. 106] or $O(n \log^2 n)$ [42, Section 2], that is better than InsertionSort. Nowadays, well-performing step sizes are identified empirically [8, 27, 42], making a generalisation and, thus, asymptotic analysis more difficult.

3.2.1. Evaluation of the Performance

Let us first focus on short inputs where only two passes with step sizes $h = (1, h_1)$ suffice. The previous results on InsertionSort suggest that a fast ShellSort ought to make use of h_1 sentinel values. Figures 3.3, A.3 and A.4 show that, with the exception of the ShellSort with step size $h_1 = 2$, the additional pass starts to pay off at around 16 elements for both 32-bit and 64-bit values with the uniform random input distribution, reaching a speedup of around 1.15 to 1.2 at 24 elements. In case of the reverse sorted input, the speedup is practically always above 1 even for very short inputs, reaching around 1.25 to 2.1 at 24 elements. On sorted and almost sorted inputs, ShellSort exhibits a slowdown in the benchmarked range of input lengths.

When moving to greater input lengths (Figs. 3.4 and A.5 to A.8), the differences in performance between the two-pass ShellSorts become more pronounced. Between 48 and 64 elements, three passes get worthwhile to consider. On the one hand, the findings are in accordance with the widely used ones by Ciura [8, cf. 42] who, for 128 elements, determined $h = (1, 9)$ to be the optimal pair and $h = (1, 4, 17)$ to be the optimal triplet, which is also the case in Fig. 3.4. On the other hand, the gain from doing three passes is far smaller: Ciura calculated an average speedup of 1.33 over doing two passes, while it is only 1.16 on a DPU. In contrast to his findings, this also makes it unlikely that four passes would yield any gains at this input length. This shows that past findings on non–unit-cost models cannot be applied one-to-one to DPUs.

But would pushing the limits of ShellSort even be rewarding? Greater input lengths require greater steps — presumably well into the three digits for $n \approx 1000$ [8, 42] — and those in turn require more sentinel values. Implicit sentinel values could provide relief since the slowdown from implicitness should vanish for longer inputs, as was the case for InsertionSort. Still, finding the best step sizes for longer inputs would be more complex because the length and, thus, the

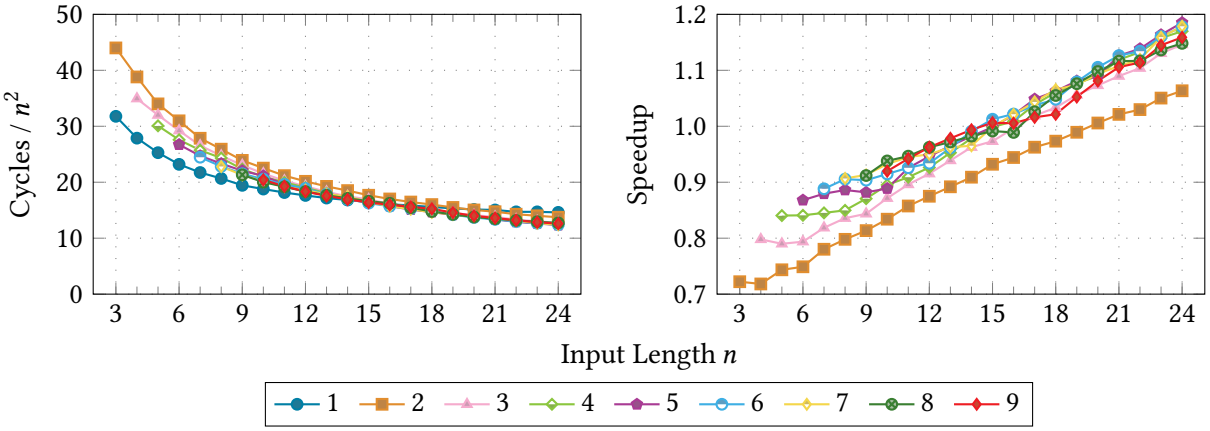


Figure 3.3. *Mean runtimes and speedups over InsertionSort of InsertionSort (1) and various two-pass ShellSorts (2–9), whose step sizes h_1 are indicated by their labels, on uniformly distributed 32-bit integers.*

number of reasonable combinations of step sizes become larger. Unfortunately, longer optimal tuples cannot be constructed straightforwardly from shorter optimal ones. The usefulness of such an endeavour, on the other hand, would likely be niche. ShellSort is outperformed by other algorithms presented hereafter, and those have no use for a ShellSort adjusted to longer inputs. Its only saving grace could be its in-place property (especially when relying solely on implicit sentinel values) combined with its medium speed, as discussed in Section 3.6.

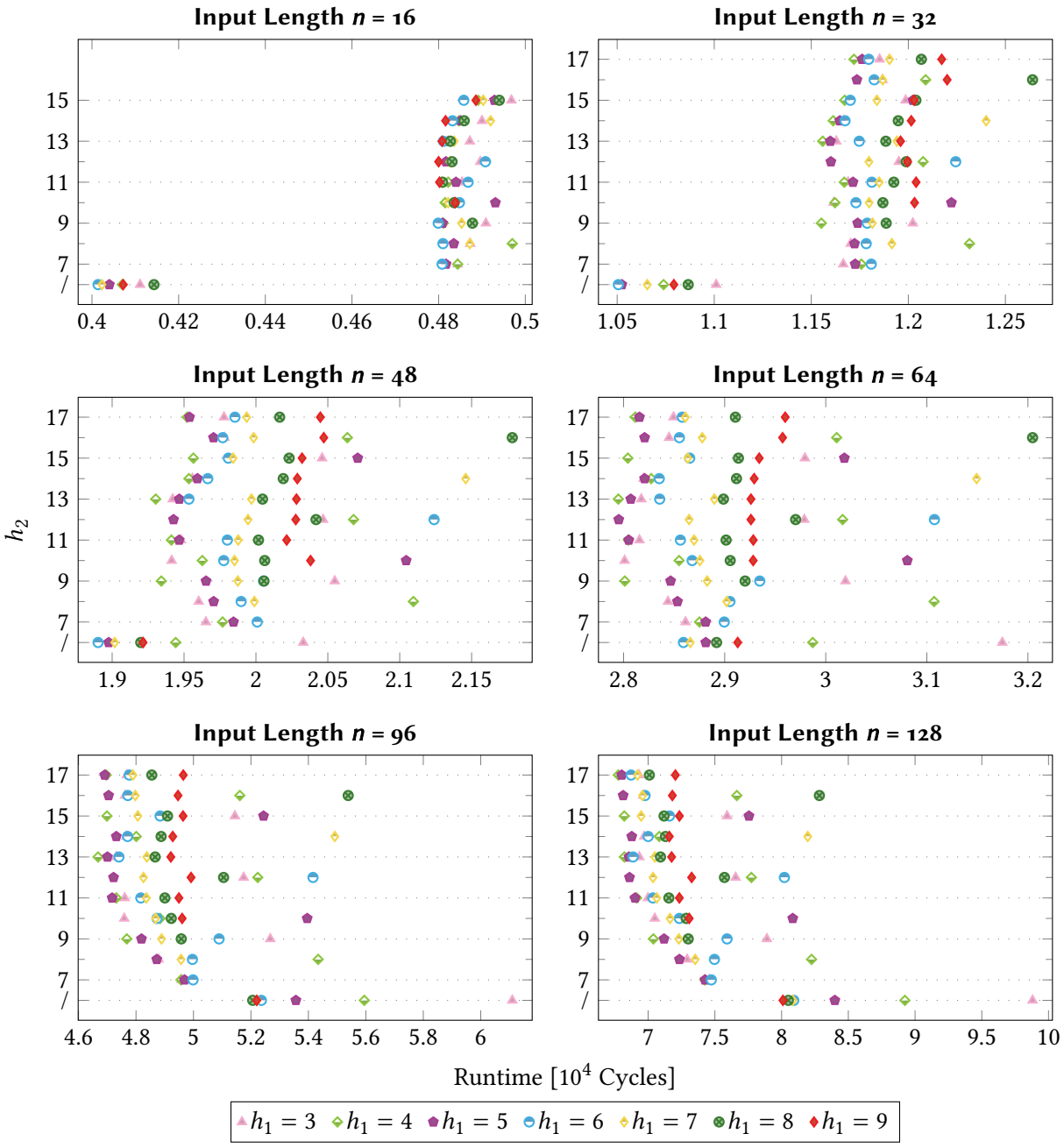


Figure 3.4. *Mean runtimes of two-pass and three-pass ShellSorts on uniformly distributed 32-bit integers. The two-pass ShellSorts are situated on the lowest file, which is labelled '/'. The three-pass ShellSorts are situated on the files above, whose y values indicate the step size h_2. The coloured symbols encode the step sizes h_1.*

3.3. HeapSort

HeapSort [14, 47, 48, Section 2.2.5] makes use of a so-called *heap*, which is a priority queue allowing to retrieve and remove the maximum element stored in time $O(\log n)$. Repeated retrieval and removal of the maximum allows to sort in place in time $O(n \log n)$, although the algorithm is unstable.

A heap (or, more specifically, a *binary max-heap*) is a binary tree of logarithmic depth whose layers are fully filled, that is, the layer of depth i contains 2^i vertices. The only exception is the last layer, which may contain less vertices but must be filled from left to right. In the context of HeapSort, the vertices are identified with the elements to sort. The *heap order* dictates that each parent must be at least as great as its child. Consequently, the root has the greatest value. A heap with n vertices can be represented as an array of length n using a bijective mapping between the vertices and the array indices: If the root is stored at position 1, the children of the vertex with index i have the indices $2i$ and $2i + 1$, whilst its parent has index $i \div 2$, where the obelus ($\div$) denotes an integer division.

After the heap has been built in place from the input array in time $O(n)$, the sorting works as follows: At the start of round $r = 1, \ldots, n$, the first $n - (r - 1)$ elements of the array represent the heap and the last $r - 1$ elements the end of the sorted output. The root, which is the rth greatest element of the input, gets removed and, since the heap cannot contain holes, a reparation procedure is performed. Since the heap has shrunken by one vertex, the removed root can be stored at index $n - (r - 1)$, that is the freed-up position directly behind the end of the heap.

3.3.1. Presentation of Key Aspects

Sifting Direction Once the heap is built, the *top-down* HeapSort proceeds as follows: At the start of each round, the root and the rightmost leaf in the bottom layer ('last leaf') swap places. The root is now in the correct output position, but the former last leaf may violate the heap order, that is, the root may be less than one or both of its children. The greater of the two children is determined, and the root and the greater child swap places. This sifting of the former last leaf downwards continues iteratively until the heap order is restored.

In contrast, the *bottom-up* HeapSort [45] works as follows: At the start of each round, the root is removed so that a hole sits now at the top of the heap. Then, the hole and the greater of its two children swap places. This sifting of the hole downwards continues iteratively until it becomes a leaf. Now, the last leaf is moved to the position of the hole. Should this former last leaf be greater than its new parent, then the heap order is now violated. It needs to be sifted upwards by iteratively swapping positions with its respective parent until the heap order is restored. At last, the original root element can be put where the former last leaf used to be.

The motivation behind these variants is at follows: In each step where the top-down HeapSort sifts a former last leaf downwards, two value checks (Which child is greater? Is the parent less than the greater child?) need to be done. The leaves of a heap tend to be little so the downwards sift normally lasts awhile. As opposed to this, each step of the bottom-up HeapSort needs only one value check (Which child is greater?). Both HeapSorts sift downwards similarly long so many checks can be saved. Since the last leaf effectively takes the place of another leaf and since both are likely little, the upwards sift should be short-lived and not eat the gain up.

Sifting upwards reverts some of the changes done by sifting downwards. The bottom-up HeapSort can be brought to swap parity with the top-down HeapSort by the following change: The sifting downwards is traced but the vertices are not actually moved. Once the leaf where the hole would end up is reached, the sifting is backtracked until the bottommost vertex which is greater than the last leaf. The position found is where the last leaf would end up after sifting upwards, so all vertices below can stay put and all vertices above move to their parents' positions, that is, thither the swaps from sifting downwards would have put them. This implementation variant makes sifting downwards even cheaper, but it must be sifted upwards all the way to the root then.

Sentinel Values When HeapSort sifts a vertex downwards, it needs to determine the greater of its two children before deciding whether and whither to move. If and only if the heap has an even number of vertices, there is a left child without a right sibling: the rightmost leaf in the bottom layer. Instead of adding some check on whether the right sibling exists, one can rather add the missing leaf and set it to the least possible value each time the heap reaches an even size. Thus, if it has been confirmed that a left child exists, a right one does also exist. Bounds checks on whether a left child exists are still required lest HeapSort loses its in-place property, since there are about $n/2$ leaves of which all would need sentinel children.

Likewise, whenever HeapSort sifts upwards and considers the parent $i \div 2$ of a vertex i, it will only proceed if the parent is less. Since the root has index 1, the formula $i \div 2$ yields 0, so it makes sense to set the element with index 0 to the greatest possible value to stop any upwards sift. The speedup from using sentinel values is about 1.15.

Code Duplication A strategy particularly useful for HeapSort — although employed in other sorting algorithms, too — is *code duplication*. Sifting downwards can be broken down into two steps: 1. Find the greater child. 2. Perform some operations on said child. A natural and concise implementation would determine the greater child and, then, store its index in a variable on which the operations are performed afterwards. However, the quality of the compilation improves if the operations are implemented twice, once for either child, and executed conditionally. This approach led to a speedup of about 1.07.

Base Cases When 15 elements or fewer remain in the heap, InsertionSort is used to sort them. Admittedly, the impact of this one-time use is rather negligible, and ShellSort would not make much of a difference.

3.3.2. Investigation of the Compilation

Under zero-based indexing, the indices of the children of a vertex with index i are $2i + 1$ and $2i + 2$, whilst the one of its parent is $(i - 1) \div 2$. Under one-based indexing, the indices of the children of a vertex with index i are $2i$ and $2i + 1$, whilst the one of its parent is $i \div 2$. The formula $i \div 2$ is computable through a bitwise shift one place to the right, whereas $(i - 1) \div 2$ requires a subtraction before the bitwise shift. Since the bottom-up HeapSorts rely heavily on finding parents during backtracking, one-based indexing is clearly superior.

Consistency alone would suggest one-based indexing for all types of HeapSort. However, the first HeapSort implemented was the top-down HeapSort, which only ever sifts down. The choice is not so obvious when focussing only on that version of HeapSort. The compiler automatically turns multiplications by 2 into a bitwise shift by one place to the left. Next to a regular `lsl` instruction for such bitwise shifts to the left, DPUs also possess an instruction called `lsl_add` which first shifts to the left and then adds a number. This way, the formulæ $2i + 1$ and $2i$ take the same amount of time to compute. Notwithstanding `lsl_add` being indeed employed in the compilation, the zero-based indexing sees a speedup of 0.93 over one-based indexing. The reason is that `lsl_add` takes four arguments: the target register, the addend, the integer to shift, and the number of places to shift. Whilst DPUs have a read-only register permanently storing the number 1 at disposal, read-only registers can only ever be the first register to be passed, not the second one. As a consequence, the compiler has to move the number 1 to a general-purpose register before implementing the calculation of $2i + 1$. However, this general-purpose buffer is immediately overwritten with the result from `lsl_add`, raising the need to move the number 1 the next time again. Hence, the calculation of $2i + 1$ does take twice as long as $2i$ after all.

There are a number of other curious observations. For example, the runtime difference between stopping HeapSort when one element remains in the heap and stopping HeapSort when only three elements remain (which then get sorted by InsertionSort) reduces the runtime by tens of thousand of cycles. Stopping HeapSort even earlier has comparatively little effect. For comparison, sorting just three elements solely with HeapSort barely takes one thousand cycles at worst, including the time to build the heap.

Another curiosity was the following: Building a heap uses downwards sifting, so if the input length is even, a single sentinel leaf must be inserted beforehand. Adding this leaf if the input length is odd makes no difference algorithmically, as it would be a left leaf never to be accessed due to the bounds checks. However, adding an `if` statement determining whether the sentinel leaf has to be placed has dramatic effects compared to placing the sentinel leaf unconditionally. Since the parity of the input length is reused later on, the conditional version is expected to gain one instruction. Yet, when measuring the runtimes on 1024 elements, one can observe anything from a reduction by 5000 cycles over changes within the margin of error to increases by 25 000 cycles, depending on the sifting direction and the input distribution. Adding the sentinel leave outside of the HeapSort functions has no impact on this behaviour. Comparing the compilations reveals minute differences at the beginnings of the HeapSort functions, none of which affecting anything repeatedly executed. The register usage does also not change in such a manner that the execution time of instructions would be prolonged to twelve cycles.

3.3.3. Evaluation of the Performance

The measurements are visualised in Figs. 3.5, A.9 and A.10. In general, the performance of HeapSort is hardly volatile and mostly independent from the input distribution. Reverse sorted inputs are sorted faster than most since they are already max-heaps so building the heap is quick, whereas sorted inputs are sorted the slowest since they are min-heaps, essentially needing inversion. Nonetheless, the reverse sorted inputs get sorted at most 10 % faster than the sorted ones. An extraordinary outlier are zero-one inputs on which the top-down HeapSort achieves a speedup of 1.8 over the bottom-up HeapSort. With zero-one inputs, roughly half of the elements

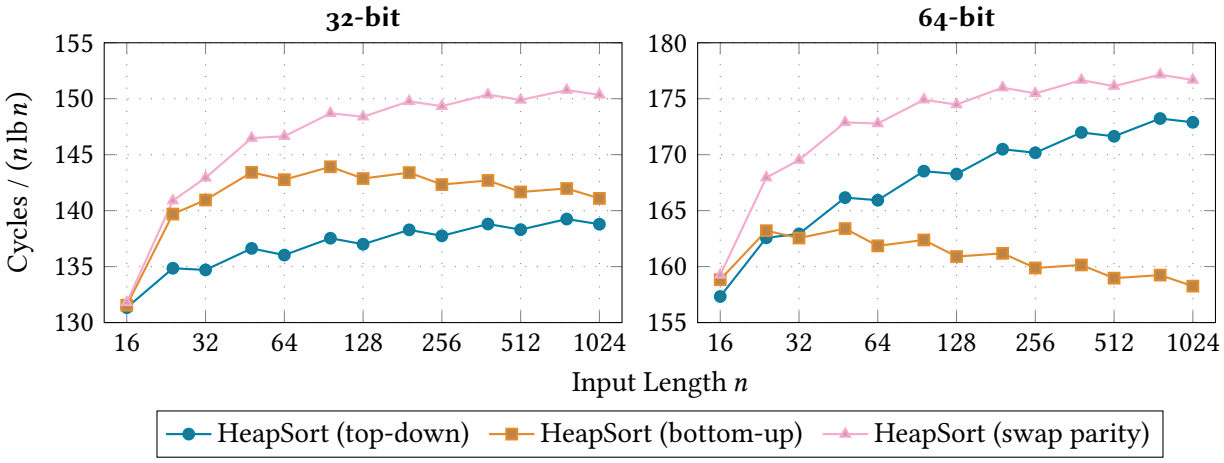


Figure 3.5. *Mean runtimes of all HeapSort implementations on uniformly distributed integers.*

are zeroes and roughly half are ones. After building the min-heap, the first half of the array consists of ones and the second half of zeroes. About half of the ones and about half of the zeroes are already in these respective halves of the input, so building the heap is fairly quick, too. More importantly, however, is the foreshortening of the downwards sift. After about $n/2$ many rounds, only zeroes remain in the heap. Therefore, no last leaf violates the heap order when moved to the top, so sifting downwards terminates immediately. But sifting downwards becomes quicker even earlier as many ones turn into leaves once the zeroes which are their children have been moved to the top and were sifted down to somewhere else. Such ones do not violate the heap order when moved to the top at a later point, too.

Ignoring this outlier, the normalised runtimes of the top-down HeapSort and the bottom-up HeapSort with swap parity show a slight upwards trends, whereas that of the bottom-up HeapSort with swap disparity mostly shows a slight downwards trends. The exception are reverse sorted inputs, where the latter also shows a slight upwards trend. Of interest is their ranking: Value checks on 64-bit integers take two instructions, so that the savings of the bottom-up HeapSort with swap disparity allow it to outperform the top-down HeapSort even for short inputs. Its advantage grows with the input length. This makes sense as roughly 50 % of the vertices are leaves and 25 % are parents of leaves, no matter the total heap size. Therefore, the percentage of former last leaves being sifted down from the top to the bottom remains steady but the travelled distance increases. Value checks on 32-bit integers, on the other hand, take only one instruction, so that the reduction of these is overshadowed by the increased overhead from the longer downwards sift and the added upwards sift. Indeed, at around 2000 elements, the bottom-up HeapSort with swap disparity overtakes the top-down HeapSort because of their inverse trends, but the lead stays meagre even at 6000 elements.

The bottom-up HeapSort with swap parity consistently trails behind. This comes as no surprise since the overhead of its considerably prolonged upwards sift bears no proportion to the few swaps saved. This holds true even for 64-bit integers as moves still cost only one instruction so the savings do not increase. Unrolling the upwards sift proved to be unhelpful.

3.4. QuickSort

QuickSort [20, 30, pp. 88–91, 48, Section 2.2.6] uses partitioning to sort in an expected average runtime of $O(n \log n)$ and a worst-case runtime of $O(n^2)$. It selects a so-called *pivot* element from the input array, then, scans the whole input array and moves elements less or greater than the pivot to the left-hand or right-hand side of the array, that is the *partitions*, respectively. Elements equal to the pivot are allowed to be in either partition. Finally, QuickSort calls itself recursively on the left-hand and right-hand partition. QuickSort sorts out of place, as additional space of size $O(\log n)$ is needed for a call stack. Furthermore, QuickSort is unstable.

The partitioning is implemented using a modification of Hoare's original scheme [20]: After the pivot `p` is selected, two pointers `i` and `j` are set to either end of the array. The left pointer `i` moves rightwards until finding an element at least as great as the pivot (`*i >= *p`). Then, the right pointer `j` moves leftwards until finding an element at most as great as the pivot (`*j <= *p`). If the two elements found are unequal, they need to be swapped to put them in the correct partition. But even if they are equal, swapping them anyway does not put them in an incorrect partition. After swapping the elements, the pointers move onwards as described. This process of repeated swaps continues until the pointers pass each other. The position where the right pointer `j` came to rest marks the end of the left-hand partition, and where the left pointer `i` did marks the beginning of the right-hand partition.

3.4.1. Presentation of Key Aspects

Sentinel Values In order to dispose of many bounds checks on the pointers, the partitioning presented above does not exactly follow Hoare's original scheme where pointers stopped only if `*i > *p` and `*j < *p`. Instead, by stopping if `*i >= *p` and `*j <= *p`, the pivot `p` acts as sentinel value for both pointers as the stopping condition is met there definitely. This means that they cannot leave the array during their very first motion onwards.

If one pointer surpasses the pivot before the other reaches it, the pointer still cannot go out of bounds. If the left pointer `i` reaches the right pointer `j`, then all elements behind pointer `j` are already in the correct partition, that is, they are at least as great as the pivot. An analogous argument can be made for when pointer `j` meets pointer `i`. Of course, they can also stop directly on each other if the value on which they stopped is equal to the pivot. In consequence, only one bounds check is needed during partitioning, namely whether `j <= i` holds whenever both pointers stopped.

A downside to this modification is that elements equal to the pivot are also swapped during partitioning. But even on inputs with many duplicates, this is a price worth paying, as experiments show.

Pivot Positioning By a further modification, one can find the final position of the pivot, so it need not be touched anymore in the future. After the pivot `p` is selected, it is swapped with the last element of the array. Consequently, the right pointer `j` has to begin at the second to last element. Since the right-hand partition contains only elements at least as great as the pivot, the pivot must be the minimum of that partition. Therefore, once the partitioning is over, the last element of the array, that is the pivot, can be swapped with the first element of the right-hand

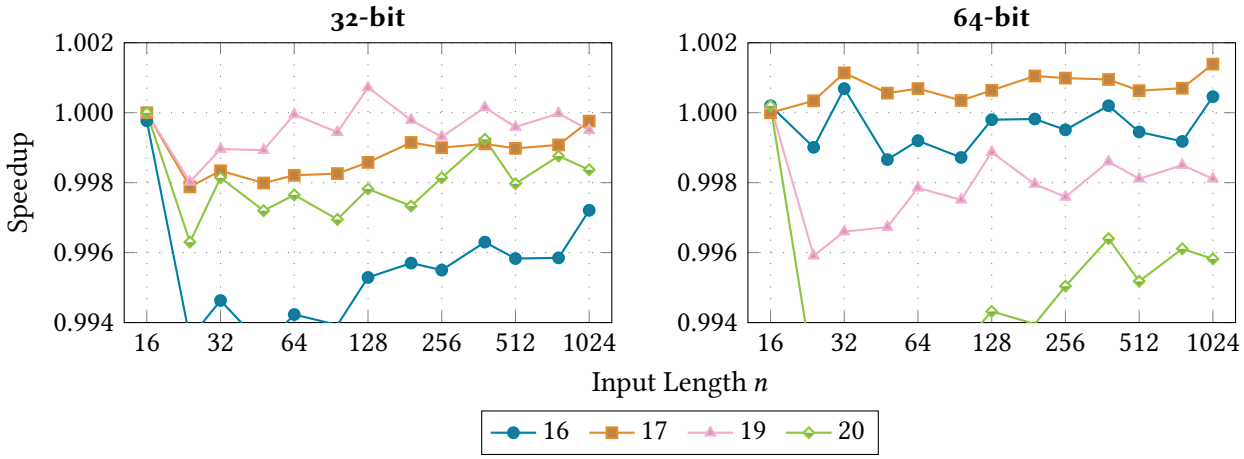


Figure 3.6. *Speedups of QuickSorts with different thresholds (16–20) for when to fall back to InsertionSort over a QuickSort with a threshold of 18 elements, on uniformly distributed integers. Using ShellSort is not beneficial because many partitions undercut the thresholds significantly.*

partition, that is the element with address `i`. The right-hand partition can be shortened to begin at address `i + 1` instead of `i`.

Base Cases When only a few elements remain in a partition, QuickSort's overhead predominates such that InsertionSort lends itself as fallback algorithm. Falling back generates a speedup of up to 1.67. As shown in Fig. 3.6, the optimal threshold for switching the sorting algorithm is 18 elements for 32-bit integers on uniformly distributed inputs. For 64-bit integers, the optimal threshold is 17 elements. Notwithstanding, we set 18 elements to be the default threshold for both data types to simplify matters since the impact is minuscule. For sorted and almost sorted inputs, the threshold is higher since InsertionSort performs well on them, so falling back earlier and, thus, ending the sorting process is better. Because QuickSort's two pointers invert large portions of reverse sorted inputs while partitioning, the same holds true for them, too, even though they represent InsertionSort's worst case.

To avoid unnecessary uses of InsertionSort, another base case is imaginable, namely terminating when a partition contains at most one element. There are tremendous consequences for the runtime depending on the exact implementation of the base cases, as shown later in Section 3.4.2.

Recursion vs. Iteration One might be tempted to assume that the question of whether an algorithm should be implemented recursively or iteratively would come down to convenience. Due to the unit-cost of instructions, jumping to the beginning of a loop or to the beginning of a function costs the same, as does managing arguments automatically through the regular call stack or manually through a simulated one. Furthermore, in case of QuickSort, the compiler turns tail-recursive[2] calls into jumps back to the beginning of the function, so that one partition

2. A recursive call is *tail-recursive* if it is the final operation performed by the callee.

is sorted recursively and the other iteratively. All this would suggest a recursive QuickSort due to its simpler implementation.

Which of these options is better unfortunately hinges on the compiler. Even parts of the algorithms which are independent from the choice between recursion and iteration can be compiled differently, such that there are implementations where being iterative is better than being recursive and the other way around. A detailed analysis is given in Section 3.4.2.

Partition Prioritisation Always sorting the shorter partition first and putting the longer partition on the call stack guarantees that the problem size is at least halved each step, so that the call stack stores $O(\log n)$ elements at most. Unfortunately, this approach proves to be detrimental to the quality of the compilation, as shown in Section 3.4.2. Instead, it is advisable to always prioritise the same side. Whether the left-hand or the right-hand partition is sorted first should not make any difference for the runtime but even this choice changes the quality of the compilation; in this thesis, the right-hand partitions are prioritised.

Pivot Selection Another parameter to tune is the way in which the pivot is selected. The following methods were implemented and benchmarked:

- Using the *last element* is the fastest method, requiring zero additional instructions.
- Taking the *deterministic median* of three elements, namely the first, middle, and last one, is more computationally expensive since the position of the middle element must be calculated, the median be determined, and the pivot be swapped with the last element of the array.
- A *random element* is most efficiently drawn on a DPU when using an xorshift random number generator and rejection sampling [15].
- The *random median* is a combination of the previous two methods by taking the median of three random elements. For simplicity, there is no check on whether an element is drawn twice or thrice. The chances of this happening are low, though, since the partitions are rather long.

Taking a median increases the probability of selecting a pivot that is neither particularly little nor particularly great. This leads to more balanced partitions so that the call stack is less likely to overflow and the base cases are reached faster. In want of branch prediction, a median is expected to be beneficial on a DPU, for branch misprediction cannot hamper the runtime as on CPUs (see Section 2.2). But as long as one of the deterministic methods is used, it is possible to construct inputs where the runtime climbs up to $O(n^2)$ [13]. Such a runtime could occur, for example, when the pivot is always the minimum or maximum of the partition so that everything is moved to the same partition. As a consequence, the problem size would be reduced by only one element (namely the pivot) after each partitioning step. This problematic behaviour is aggravated by the call stack overflowing easily because of the static partition prioritisation.

The random pivots circumvent this issue. Whilst the pivots could, by ill luck, also lead to the same unbalanced partitions as the deterministic pivots, the worst-case expected runtime is in $O(n \log n)$ [3]. The method *median of medians* [5] guarantees a runtime of $O(n \log n)$ but was not implemented because a performant implementation would presumably be quite complex and its benefit for this thesis minuscule.

3.4.2. Investigation of the Compilation

The quality of the compilation is highly erratic to such an extent that — even with the same pivots — one implementation variant may see a speedup of 1.33 over another one where none would be expected. There are small details influencing the runtime. For instance, storing the value of the pivot in a dedicated variable instead of accessing it through a pointer changes the runtime by a few percentage points in both directions, depending on the rest of implementation. But as hinted at in Section 3.4.1, there are four major parameters to examine: handling of the base cases, recursion/iteration, pivot selection, and partition prioritisation. Before the findings are discussed, the first parameter shall be explained in more depth.

Besides falling back to InsertionSort if 18 elements or fewer remain ('treshold undercut'), another base case is imaginable, namely a termination if at most one element remains ('trivial length'). Realistically speaking, it should not be needed to check for trivial lengths because even though it would take just one additional instruction, such tiny partitions are rare, and InsertionSort would terminate after a few instructions anyway. Nonetheless, its inclusion or exclusion can have significant impact as does the position at which the checks for the two base cases happen. The following handlings were benchmarked:

(1) If the length is trivial, terminate immediately. Otherwise, if the threshold is undercut, sort with InsertionSort and terminate. If neither, partition the input and invoke QuickSort on both partitions.

(2) If the threshold is undercut, check if the length is trivial and either terminate immediately or sort with InsertionSort and terminate afterwards. Otherwise, partition the input and invoke QuickSort on both partitions.

- This handling significantly reduces the number of checks on triviality.

(3) If the threshold is undercut, sort with InsertionSort and terminate. Otherwise, partition the input and invoke QuickSort on both partitions.

- This handling forgoes the check on triviality completely at the cost of some unneeded invocations of InsertionSort.

(4) If the threshold is undercut, sort with InsertionSort and terminate. Otherwise, if the length is trivial, terminate immediately. If neither, partition the input and invoke QuickSort on both partitions.

- This handling is nonsensical from a logical point of view, since the length cannot be trivial if the threshold is not undercut. However, it gives the compiler an explicit guarantee that the loop for partitioning does not end immediately, eliminating a possible reason for the varying quality of the compilation.

(5) If the threshold is undercut, sort with InsertionSort and terminate. Otherwise, partition the input. Then, check for either partition if its length is trivial and invoke QuickSort on it if not.

- This handling, as well as the next two, eliminates some unneeded invocations of QuickSort.

(6) Partition the input. Check for either partition if the threshold is undercut and invoke either InsertionSort or QuickSort on it.

(7) Partition the input. Check for either partition if its length is trivial or if the threshold is undercut and invoke either InsertionSort, QuickSort, or nothing on it.

(8) If the threshold is undercut, terminate immediately. Otherwise, partition the input and invoke QuickSort on both partitions. After all QuickSorts are done, sort the whole input array with InsertionSort.

- This handling always does one pass of InsertionSort. For example, the other handlings invoke InsertionSort roughly 91 times on 1024 uniformly distributed elements.

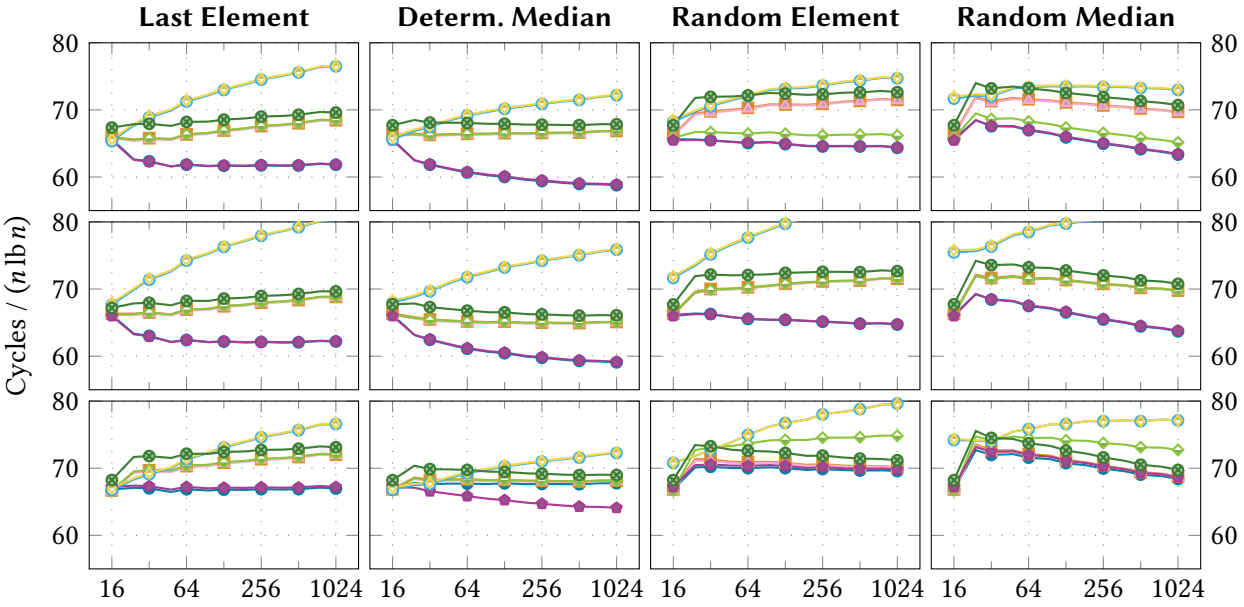
The performances of all benchmarked implementation on 32-bit integers are shown in Fig. 3.7. The measurements were conducted on uniform input distributions, so the deterministic pivots are, in expectation, of the same quality as the random ones.

Even when ignoring the differences between specific handlings for now, the high fluctuations between the plots become immediately apparent. Plots within the same column share the same method to select pivots, plots within the same row share the same prioritisation of partitions. In general, it would be expected that plots within the same column are fairly similar, yet prioritising the shorter partition is almost universally associated with an increase in runtime. When focussing on the top-performing handlings, the speedup can fall below 1 down to 0.87. There is no clear trend between the consistent prioritisations of either side, although the difference can be huge in individual cases as well. However, recursive implementations are more susceptible to the partition prioritisation than iterative ones.

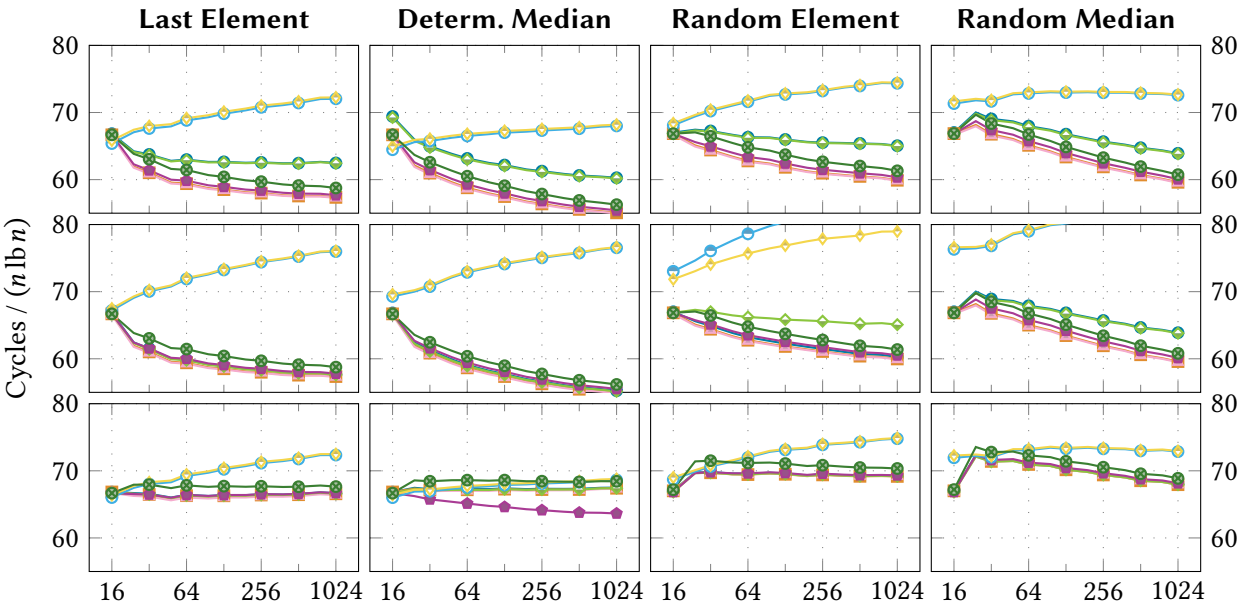
The correlation of recursive and iterative performance is weak. On the one hand, there is, for example, Handling (5) with deterministic medians and prioritisation of shorter partitions where the runtimes are essentially the same. On the other hand, there is Handling (8) with deterministic medians and prioritisation of right-hand partitions where recursion is slower by more than a third. All in all, iterative implementations usually perform better, though, especially when focussing on the top-performers of each pivot selection method.

The ranking of the different handlings is rather incoherent. Handling (5), which does not call QuickSort on trivial partitions, fares the best out of all handlings decidedly, being amongst the top performers across all benchmarked implementations. Handlings (6) and (7), which call QuickSort even less often than Handling (5), are the polar opposite and bring up the rear of the ranking every single time. Recursive implementations of Handling (2), where triviality is checked for only if the threshold is undercut, are often worse than recursive implementations of Handling (1), where triviality is always checked for, whilst it is the other way around for iterative implementations. Interestingly, for all investigated implementations, the compiler turns tail-recursive calls into jumps back to the beginning of the function, no matter how convoluted the logic surrounding the call is.

These observations, however, only apply to 32-bit integers. Figure A.14 shows the same measurements for 64-bit integers. Whilst the general trend for pivots, partition prioritisation, and recursion/iteration hold true, the rankings are vastly different. Handling (5) is not undisputedly the best anymore. Handlings (6) and (7) switch back and forth between being the worst and the best handlings. Even the nonsensical Handling (4) manages to take the lead in a few cases. Most notably, the two top-performing implementations using deterministic and random medians, respectively, are actually recursive. Luckily, both use Handling (5) so we decided to make the only difference between the default configurations of 32-bit and 64-bit QuickSort be the usage



(a) *Recursive Approach*



(b) *Iterative Approach*

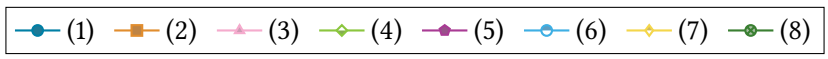


Figure 3.7. *Mean runtimes of QuickSorts using Handlings (1) to (8) and different pivot selection methods on uniformly distributed 32-bit integers. Left-hand partitions are prioritised in the first rows, right-hand ones in the second rows, and shorter ones in the third rows.*

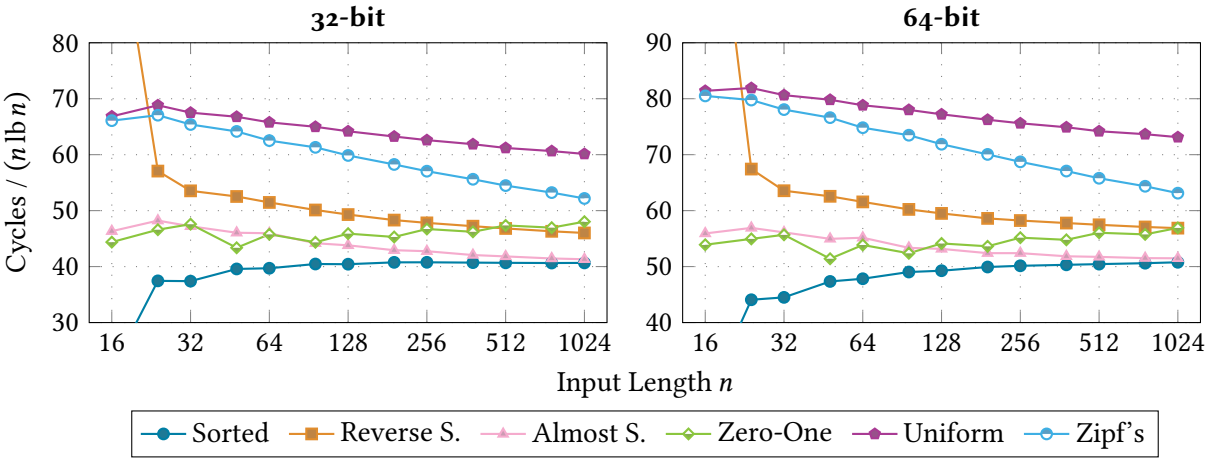


Figure 3.8. *Mean runtimes of QuickSort on all benchmarked input distributions and data types.*

of iteration (32-bit) and recursion (64-bit).

What is causing these huge disparities? There is a great variety in the compilations but some of the common occurrences are …

- … one instruction more before (re-)starting to move the pointers, …
- … one instruction more while moving the left pointer by one element, …
- … one instruction more after the left pointer has stopped, …
- … more stores and loads when entering and exiting the function.

This focus on the left pointer `i` is partially explainable by it being used to calculate the final position of the pivot and, thus, the inner boundaries of both new partitions. As explained earlier under 'Sentinel Values', the right pointer `j` stops ultimately either on the left pointer `i` directly (`j == i`) or on the element behind it (`j == (i - 1)`). Since the pivot is moved to address `i`, the inner boundaries of the partitions are `i - 1` and `i + 1`. Implementations using the right pointer alone or both of them to calculate the boundaries were spot-checked to see whether it alleviates the problems, but the results were mixed: from betterment over indifference to worsening, everything was observable.

3.4.3. Evaluation of the Performance

Before turning to the general performance of QuickSort, we evaluate the ratio between costs and benefits of the pivot selection methods. Looking again at Figs. 3.7 and A.10 shows that the longer the input becomes, the more beneficial a median becomes, achieving small pay-offs for the longest inputs. Moreover, the standard deviations of the runtimes, although not shown in the figures for reasons of clarity, are cut roughly in half. Randomisation slows down noticeably, so random pivots are disadvantageous if the input is known to be fairly random. However, the decrease remains in the single digits percentagewise, supporting the findings by Geis [15] that drawing random numbers on DPUs is quite cheap. For this reason, the random median is used as default method throughout this thesis.

Figure 3.8 shows the runtime of QuickSort in it default configuration. Figures A.11 and A.12 additionally contain the runtimes with deterministic medians as well as the standard errors of the measurements. The mean runtimes are rather close across all input distributions, a consequence of using random medians and of swapping elements equal to the pivot. In fact, it is InsertionSort that primarily causes the discrepancies, as measurements with the threshold set to one element prove. This also explains why QuickSort performs better, the longer an input with Zipf's distribution becomes: Zipf's distribution generates many duplicates, and the longer the total input becomes, the more partitions contain only one or two different values, speeding InsertionSort up significantly.

One might expect QuickSort to perform even better on sorted and reverse sorted input, since everything is either already in the correct position or because the two pointers quickly invert large portions of the inputs. However, a side effect of swapping the pivot twice can be that many elements are displaced by one position from where they should be in the sorted output. Take reverse sorted inputs (starting with element n for easier notation) as an example: Assume that the element $n/2$ is selected as pivot out of the elements n, $n/2$, and 1. The pivot gets swapped with the last element, that is with 1. The array equals now the sequence

$$\langle\, n,\ n-1,\ \ldots,\ n/2+1,\ 1,\ n/2-1,\ \ldots,\ 3,\ 2,\ n/2\,\rangle\,.$$

Thereupon, the two pointers invert the rest of the input, producing

$$\langle\, 2,\ 3,\ \ldots,\ n/2-1,\ 1,\ n/2+1,\ \ldots,\ n-1,\ n,\ n/2\,\rangle\,.$$

Swapping the pivot with the first element of the right-hand partition turns the array into

$$\langle\, 2,\ 3,\ \ldots,\ n/2-1,\ 1,\ n/2,\ n/2+2,\ \ldots,\ n-1,\ n,\ n/2+1\,\rangle\,.$$

If the median is selected from random elements, subsequent partitioning will continue to roughly halve the partitions, bringing 1 and $n/2+1$ closer to their output locations, so InsertionSort will not need too many elements. Still, an implementation without swapping the pivot delivers better performance for such cases, but in exploratory benchmarks, the performance on more random input distributions suffered drastically.

Because of the partition pattern described above, the performance degrades on reverse sorted inputs if the median is selected from deterministic elements. The pivot for the right-hand partition will be selected out of the elements $n/2+2$, $3/4n$, and $n/2+1$. Obviously, the pivot $n/2+2$ barely reduces the problem size, and subsequent pivots exhibit essentially the same behaviour, which is why the respective plots in Figs. A.11 and A.12 leave the charts. Indeed, an overflow of the call stack occurs eventually.

3.5. MergeSort

Two-way bottom-up MergeSort [23, 30, pp. 85 sq., 48, Section 2.3.1] repeatedly compares two elements and *merges* them to form sorted couples. Once only couples (and perhaps one single element) remain, the couples are merged into quadruplets, the quadruples into octuplets and so on until a single sorted sequence remains. Sorted sequences are also referred to as *runs*. MergeSort has a guaranteed runtime of $O(n \log n)$ and is the only stable sorting algorithm with subquadratic runtime presented in this chapter. Its biggest drawback is that additional space of size $\Theta(n)$ is needed.

3.5.1. Presentation of Key Aspects

Starting Runs Instead of starting by merging runs of length 1, that is individual elements, it is beneficial in practice to first subdivide the input and use either InsertionSort or ShellSort on the individual subdivisions. These larger *starting runs* allow to skip a few of the early rounds of merging. For simplicity, all starting runs have the same, predefined length with possible exception of the last one which can be shorter. A substantial downside to ShellSort is that whilst it does allow to sort bigger starting runs quicker, it is not stable unlike InsertionSort. If MergeSort is supposed to sort stably, then we use InsertionSort.

Unlike QuickSort, where each partition naturally acts as sentinel for the subsequent one, it is necessary to temporarily place sentinels values in front of each starting run and later restore the overwritten values of the preceding run. The step sizes used for ShellSort — namely $h = (1, 6)$ for up to 48 elements, and $h = (1, 5, 12)$ for more — have been chosen based on the findings in Section 3.2, according to which these step sizes offer top performance for uniformly distributed inputs and medial performance for reverse sorted inputs. Spot-check inspection suggest no deterioration of InsertionSort's and ShellSort's compilation due to inlining.

Memory Footprint A simple implementation of MergeSort (termed *full-space*) writes all merged runs to an auxiliary array, raising the need for space for n additional elements. After a round is finished and all pairs of runs have been merged, the input array and the auxiliary array switch roles and the merging begins anew. If the final sorted elements are supposed to be saved in the original input array, a final round with a write-back from the auxiliary array to the input array is needed if the number of rounds is odd.

A slightly more sophisticated implementation (termed *half-space*) needs space for only $n/2$ additional elements [26, Section 2.5]: When two adjacent runs are to be merged, the first one is copied to an auxiliary array. Then, the copy and the second run are merged to the beginning of the first run. As a side effect, no write-back is ever needed and, additionally, the merging of two runs can be terminated prematurely once the last element of the copied run is merged, since the last elements of the other run are already in place. To guarantee that the auxiliary array does indeed hold only $n/2$ elements in the worst case, one needs to make sure that the first runs are never longer than the respective second runs. There can only ever be at most one run shorter than the others, namely the last one. For this reason, the formation of starting runs now begins at the end of the input array and proceeds to the front. Likewise, merging pairs of runs also begins with the last pair and works its way forward to the front.

Further optimised, half-space MergeSort would not need to copy the first runs immediately. It suffices to search for the foremost element of the first run which is greater than the first element of the second element. All previous elements are already in the correct position so only the following elements need to be copied to the auxiliary array. This *deferred* copying, although examined during development, was not in use when measuring runtimes since it unfortunately complicates the next optimisation.

Unrolled Flushing There are four common reasons for *flushing*, that is writing consecutive elements unconditionally:

1. When two runs are merged and the end of one of them is reached, that is, when one run is *depleted*, the remaining elements of the non-depleted run can be moved safely to the output location. Especially with the sorted, reverse sorted, and almost sorted input distributions, the number of remaining elements will be high.
2. The number of runs is odd, so the full-space MergeSort moves the last run to the output location unconditionally.
3. The full-space MergeSort may have to write all elements from the auxiliary array back to the input array if the former contains the final sorted sequence.
4. Before each merge of a pair of runs, the half-space MergeSort copies the first run to the auxiliary array.

Therefore, flushing accounts for a considerable part of the runtime, and reducing the loop overhead caused by variable incrementations and bounds checking is helpful. This can be done via *unrolling* by a certain *unroll factor* x: As long as at least x elements still need to be flushed, a loop with step size x is executed, and in each iteration, x elements are moved. Making x a compile-time constant enables the compiler to implement this moving through x instruction which use constant, pre-calculated offsets. Once fewer than x elements remain, an ordinary loop with step size 1, which moves elements individually, is used. In good cases, this approach reduces the loop overhead to an xth, whilst in bad cases, where fewer than x elements are to be flushed, the overhead is increased by only one additional check.

Due to time reasons, we refrained from doing automatic and extensive benchmarks and relied on manual and exploratory benchmarks to come up with the following strategy: When the full-space MergeSort performs a write-back or when the half-space MergeSort copies the first run, the unroll factor is set to the starting run length. In all other cases, the unroll factor is set to 24, which proved to be a sweet spot. This strategy, albeit not optimal, makes the MergeSorts significantly faster. Sorting sorted, reverse sorted, and almost sorted inputs experiences a speedup of up to 1.4, and sorting more random inputs still experiences a speedup above 1 on the whole. In the few cases where unrolling is harmful and the speedup drops below 1, the speedup stays close to 1 nonetheless.

Unrolled Merging The following simple and easily scalable technique, which significantly reduces bounds checks, was employed: Before merging two runs, check whose last element is less, that is, determine which run will be depleted first. This run is referred to as *less run*, whilst the other one is referred to as *greater run* henceforward. As longs as at least x unmerged elements remain in the less run, execute an unrolled loop which merges the next x elements of

both runs. Once fewer than x elements remain in the less run, do the same with $x \div 2$ many elements. Once fewer than $x \div 2$ many remain in the run, execute an ordinary loop with step size 1 which checks after each merge of an element from the less run whether the run gets depleted therethrough.

We found an unroll factor of $\min\{\ell, 16\}$, where ℓ is the starting run length, to be a good choice. A drawback of unrolling is the increased kernel size, since the instruction count of an unrolled loop is increased roughly $x$fold. Larger unroll factors lead to an IRAM overflow.

A more refined unrolled merging can be deduced from InsertionSort not using a dedicated pointer to access a preceding element. Recall how InsertionSort utilises the pointer to the current element together with an offset of `-4`. A similar technique can be used with MergeSort, which maintains two pointers `i` and `j`, each pointing the current element of a run to merge. If, for example, it holds `*i <= *j`, then the next element can be loaded via `*(i + 1)`. Finally, depending on whether `*(i + 1) <= *j` holds, either both pointers `i` and `j` are incremented by `1` or pointer `i` is incremented by `2`. Unluckily, the quality of the compilation suffers from this technique, leading to its dismissal.

3.5.2. Investigation of the Compilation

A significant portion of the runtime is spent on the repeated comparison of elements in a pair of runs, followed by a write of the less element to the output. Figure 3.9a shows a straightforward implementation of an unrolled loop performing such comparisons and writes. The code makes use of two pointers `i` and `j` which are initially set to the first elements of the runs. To get their values, they are simply dereferenced. After the output `out[k]` has been set in iteration `k`, the respective pointer of the less element is incremented.

Despite the succinctness of the C code, the resulting assembler code is of subpar quality. Depending on the run from which an element got merged in the previous iteration, an iteration takes either 7 or 8 instructions. This is a consequence of loading the values of both dereferenced pointers at the beginning of each iteration despite one of the values not having changed since the last iteration. Figure 3.9b shows an alternative implementation, whose compilation results in four instructions per iteration. This was achieved by dereferencing the pointers `i` and `j` before the loop begins and storing the values in dedicated variables. The comparisons and writes use only these dedicated variables, of which only one gets updated per iteration. A more detailed description of the compilations is given in the caption of Fig. 3.9.

It can only be speculated as to the reason for the poor compilation of the simpler implementation. Perhaps the constant reloads are related to the ability of tasklets to write to any WRAM address. Theoretically, it could be that the value obtained by dereferencing the non-incremented pointer changes between two subsequent iterations. This explanation is not fully satisfactory as it would imply yet another load instruction before writing `out[k]`.

Unluckily, only with the full-space MergeSort does this change to dedicated variables make iterations take 4 instructions unanimously. With the half-space MergeSort, some merge iterations take 5 instructions. The reason is that the second pointer `j` is never incremented directly. Instead, whenever an element of the second run is merged, a counter is incremented and, then, the new address of pointer `j` is calculated by taking the address of the first element of the second run and adding the counter. The fix is to change the loop which iterates over the pairs of runs

```
1 #pragma unroll
2 for (int k = 0; k < 16; k++) {
3   if (*i <= *j) {
4     out[k] = *i++;
5   } else {
6     out[k] = *j++;
7   }
8 }
```

```
1  // iteration k
2    lw ri_*, ri, 0                    // load *(i + 0)
3    lw rj_*, rj, 0                    // load *(j + 0)
4    add rtmp, rj, 4                   // tmp ← j + 1
5    jle ri_*, rj_*, .LABEL_k_i        // jump if *i ≤ *j
6    move ri_*, rj_*                   // overwrite ri_*
7    move rj, rtmp, true, .LABEL_k_out // j ← tmp; jump
8  .LABEL_k_i:
9    add ri, ri, 4                     // i ← i + 1
10 .LABEL_k_out:
11   sw rout, 4×k, ri_*                // out[k] ← *i
```

(a) *This code takes 8 instructions per iteration. First, the pointers are dereferenced (lns. 2, 3). Then, the resulting address from incrementing pointer `j` is calculated (ln. 4). If the first run contains the less current element, it is jumped to line 9, where pointer `i` is incremented. Lastly, the less element `*i` is written to the output (ln. 11). If the second run contains the less current element, the register holding `*i` is overwritten with `*j` (ln. 7). Then, a combo operation (ln. 7) finally applies the result from incrementing pointer `j` and jumps to the line where the output is set.*

We do not know why pointer `j` gets temporarily incremented. According to the documentation, an `add` instruction is compatible with the `true` flag, meaning the `add` instruction in line 4 and the `move` instruction in line 7 could be fused.

```
1  int val_i = *i, val_j = *j;
2  #pragma unroll
3  for (int k = 0; k < 16; k++) {
4    if (val_i <= val_j) {
5      out[k] = val_i;
6      val_i = *++i;
7    } else {
8      out[k] = val_j;
9      val_j = *++j;
10   }
11 }
```

```
1  // iteration k (val_i ≤ val_j)
2    jgt ri_*, rj_*, .LABEL_k_j      // jump if val_i > val_j
3  .LABEL_k_i:
4    sw rout, 4×k, ri_*              // out[k] ← val_i
5    add ri, ri, 4                   // i ← i + 1
6    lw ri_*, ri, 0                  // val_i ← *(i + 0)

1  // iteration k (val_i > val_j)
2    jle ri_*, rj_*, .LABEL_k_i      // jump if val_i ≤ val_j
3  .LABEL_k_j:
4    sw rout, 4×k, rj_*              // out[k] ← val_j
5    add rj, rj, 4                   // j ← j + 1
6    lw rj_*, rj, 0                  // val_j ← *(j + 0)
```

(b) *This code takes 4 instructions per iteration. There are 16 cascaded iterations in the assembler code, all of them writing the elements of the first run to the output (top). There is an analogue cascade writing only elements of the second run to the output (bottom). Labels allow to switch between the cascades. First, it is checked whether the cascade should be changed (ln. 2). Then, the output is set (ln. 4), the respective pointer incremented (ln. 5), and the new value from dereferencing loaded (ln. 6).*

Figure 3.9. *Two C implementations of an unrolled loop which merges 16 elements, contrasted with their compilations. Only the assembler codes of one iteration are shown, as all iterations follow the same scheme; a sixteenfold cascade of the given assembler codes yields the whole assembler codes of the loops. The pointers `i` and `j` point initially to the first elements of the runs. The serially numbered registers ('r…') and jump labels ('.LABEL…') were renamed to aid understanding. Note that the data type `int` is 4 B large, which is why all offsets are multiples of four.*

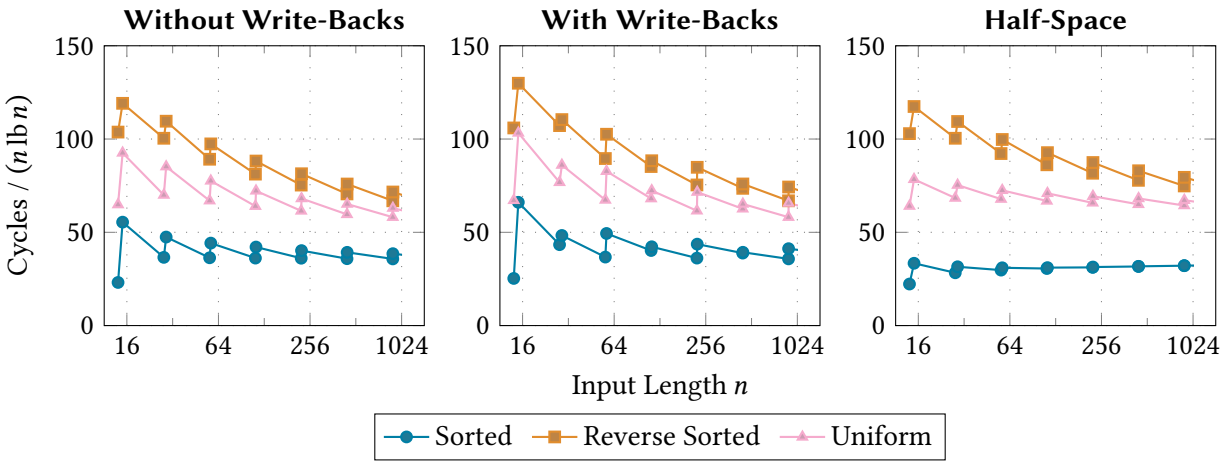


Figure 3.10. *Mean runtimes of the full-space MergeSorts with and without write-backs and the half-space MergeSort on 32-bit integers, for $n = 2^i$ and $n = 2^i + 1$ with $i = 1, \dots, 10$.*

to merge. Rather than using the ends of the second runs as natural loop index, it has to be iterated over the ends of the first runs.

A last mention shall be given to the merge function used by the half-space MergeSort. Passing the copied run as second argument and the uncopied run as the first one nets a noticeably speedup over an implementation with flipped arguments and, of course, flipped logic. Sadly, we could not pinpoint the fundamental cause for this phenomenon.

3.5.3. Evaluation of the Performance

Three implementations using InsertionSort on starting runs of length 14 have been benchmarked: full-space MergeSort without write-backs, full-space MergeSort with write-backs, and half-space MergeSort. The results are shown in Figs. 3.10, A.15 and A.16. Besides the mean runtimes on all benchmarked input distributions, Figs. A.15 and A.16 additionally contain the standard error of the measurements. Note that the benchmarked input lengths have been chosen in such a way that the plots exhibit MergeSort's characteristic zigzagging to the full extent: The merging process can be visualised as binary tree, with the starting runs as leaves and all other runs being inner vertices. Two vertices are siblings if the corresponding runs get merged together; the root contains the final sorted run. This way, the height of the tree is equal to the number of rounds of merging. For $n = 2^i$ input elements, the tree is complete, and the normalised runtime is locally minimal. For $n = 2^i + 1$ input elements, the root has a leaf with one element as child, and the normalised runtime is locally maximal, as the number of rounds increases by one to accommodate for just a single element.

The measurements show that any MergeSort guarantees a runtime of $\Theta(n \log n)$ for the benchmarked input distributions as expected. The differences in runtime between the different input distributions get smaller with increasing input length and are ascribable to InsertionSort and to the differing effectiveness of unrolling flushes. In fact, InsertionSort is the main reason why sorted inputs take the shortest and reverse sorted ones the longest; cases where the

usage of InsertionSort worsened the runtime are unbeknown. The differences across the input distributions become smaller with increasing input length but remain large even for $n \approx 1000$ elements. For the full-space MergeSorts and with $n \approx 1000$, reverse sorted inputs get sorted 85 % slower than sorted inputs of 32-bit integers and 100 % slower than sorted inputs of 64-bit integers. For the half-space MergeSort, these disparities climb to 125 % and 140 %, respectively.

The half-space MergeSort delivers a strong performance despite its smaller memory footprint. On sorted and zero-one inputs of length $n \approx 1000$, it takes the lead over the full-space MergeSort without write-backs for both 32-bit and 64-bit integers, since the second runs need not be flushed and the additional copying of the first runs is unrolled. On almost sorted inputs, they are essentially on par. For the other inputs, the speedup of half-space MergeSort over the full-space MergeSort does not drop below 0.89. The gap to the full-space MergeSort with write-backs is even smaller. This slowdown, albeit small, is likely because of the elements of the first runs being both copied and moved and of most elements of the second runs being moved forwards anyway.

Appendix A.5 contains further measurements on MergeSort, showing why a starting run length of 14 is a good choice. Figures A.17 and A.18 show the average runtimes of starting runs of lengths 12 to 16, all sorted with InsertionSort. Overall, the differences are small, yet a length of 14 delivers a solid performance throughout all benchmarked input distributions. Figures A.19 and A.20 include longer starting runs of lengths between 24 and 96 in order to see whether giving up stability by using ShellSort can yield substantial gains. The disparities between the runtimes of the different starting run lengths are strikingly small despite the wide range of lengths. By and large, however, the speedups, if any, are not big enough to warrant consideration of both a stable and an unstable MergeSort configuration in this thesis.

3.6. Interim Conclusion

This section offers a summary of the findings on the algorithms presented in this chapter. It also gives an outlook on future improvements. Figure 3.11 serves as succinct overview of the runtimes on all benchmarked input distributions and data types.

InsertionSort This algorithm works in place and is stable. It is arguably the best for short inputs with up to 16 elements as it offers the best performance on the benchmarked input distributions and, additionally, exhibits both the stability and the in-place property. Sentinel values enabled significant speedups — a theme shared with most sorting algorithms. However, there is still room for improvement as its compilation is suboptimal, especially in case of the InsertionSort with implicit sentinel values.

As a last point, a strong contender to InsertionSort shall be mentioned, namely sorting networks [9, 26, Chapter 13]. These algorithms work for a fixed input length and swap elements according to a series of predefined comparisons. Testing various code snippets [9, p. 9, 29, 37] suggests a large potential for further speedup.

ShellSort This algorithm works in place but is unstable. It offers a significant speedup over InsertionSort as long as the input is fairly distant from being sorted. Optimising it for long inputs will take some effort, though.

HeapSort This algorithm works in place but is unstable. The runtime is guaranteed to be in $O(n \log n)$ and also proved to be mostly oblivious to the benchmarked input distributions. Nevertheless, it is severely outpaced by QuickSort and MergeSort, as becomes quite clear in Fig. 3.11. Unless the runtime guarantee is absolutely needed and MergeSort is not an option because of its memory footprint, HeapSort should not be used. Its implementation is complicated by the optimal sifting direction being dependent on the data type. Eyebrow-raising observations during development suggest that its compilation can still be improved.

All benchmarked HeapSorts use a binary heap so an obvious endeavour would be to switch to tertiary heaps. Exploratory implementations, however, show that the performance suffers from this change.

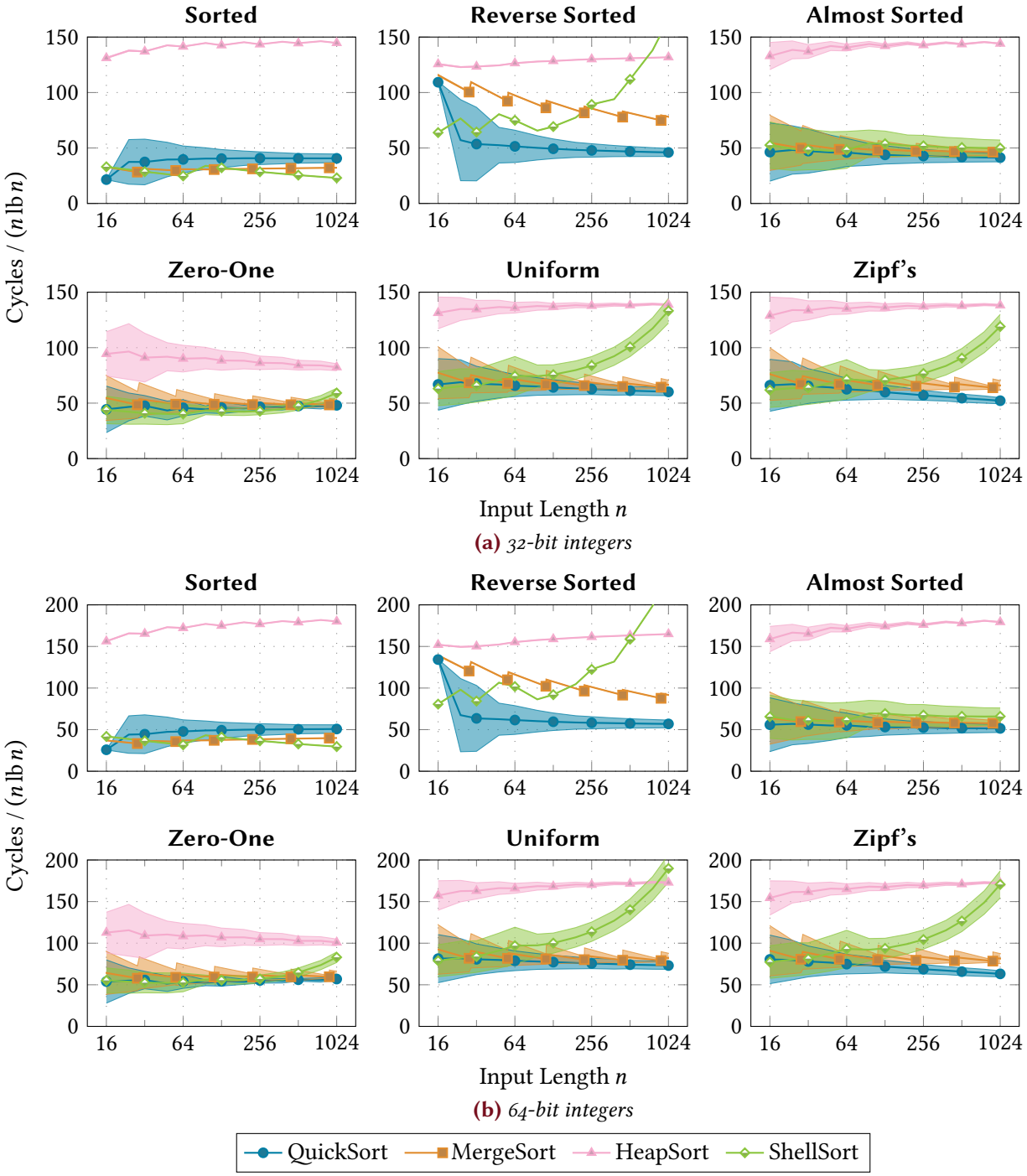
QuickSort This algorithm works out of place and is unstable. Its runtime is in $O(n \log n)$ only in expectation, but the worst-case runtime of $O(n^2)$ is, thanks to random medians being selected as pivots, highly unlikely. QuickSort generally delivers top performance which becomes even better with deterministic medians as pivots if the input is known to be random enough. There is serious work needed to be done, however. So far, there is no prioritisation of shorter partitions, which is needed to prevent an overflow of the call stack, due to problems in the compilation. These problems currently also make both recursion and iteration necessary, depending on the data type, and we cannot rule out other impairments hidden in the compilation.

Besides fixing these issues, future work could revolve around different partitioning patterns similar to those of dual-pivot [46] or patter-defeating QuickSort. The latter makes use of HeapSort as fallback algorithm in worst cases. Since the maximum input length is rather limited

on DPUs, perhaps a carefully tuned ShellSort may be used instead. For this reason, Fig. 3.11 includes also ShellSort, which starkly contrasts HeapSort despite its yet unoptimised step sizes.

MergeSort This algorithm works out of place but is stable. The runtime is guaranteed to be in $O(n \log n)$ although it fluctuates somewhat, depending on the input distribution and input length. Having said that, the leeway to QuickSort is not too big most of the time, making it unlikely that a stabilised QuickSort with likewise increased memory footprint would be much of a benefit. Deferred copying of runs and fine-tuned unrolling could make the runtime of MergeSort drop even further.

Some allowance on the starting run lengths does not affect the average runtime too much, so the zigzagging of the runtime could be dampened by dynamically adjusting the starting run length. A not-yet-implemented strategy to speed up flushing is the interpretation of two consecutive 32-bit integers as one 64-bit integer which then gets loaded and stored away in just two 64-bit instructions instead of four 32-bit ones. The compiler can be prescribed to do so by casting the involved 32-bit integer pointers to their 64-bit counterparts. Special care, however, is required with regards to the alignment, as the addresses of 32-bit integers are aligned to four bytes, whereas those of 64-bit integers are aligned to eight bytes [44, DPU ABI – Data types]. A trivial implementation of such a flushing yielded both gains and losses for the runtime in the single-digits percentagewise, depending on the MergeSort and the input length. A feature, last but not least, worthwhile to implement should be the detection of natural runs [26, Section 2.6, 48, Section 2.3.2], that is runs already present in the input.



(a) *32-bit integers*

(b) *64-bit integers*

Figure 3.11. *Mean runtimes of the main algorithms presented in this chapter. The tinted areas denote the three-sigma range, that is the 99.7 % confidence intervals.*

Chapter 4.

Sorting in the MRAM

This chapter is concerned with sorting data which resides in the MRAM. Both a sequential sorting algorithm for execution by a single tasklet as well as a parallel sorting algorithm for execution by a whole DPU are presented. We restrict ourselves to varieties of MergeSort since it allows for stable sorting and, also, the performance of MergeSort proved to be competitive in sorting WRAM data in Chapter 3. Next to the unit-cost of instructions, key characteristics of the DPU architecture which are of especial importance in this chapter are the memory hierarchy and inter-tasklet communication through shared memory.

A challenge in designing an algorithm for MRAM data is the management of data transfers between the large MRAM and the small WRAM. A key aspect of the transfer management is the use of the sequential reader, that is a software abstraction provided by UPMEM which simplifies moving data from the MRAM to the WRAM. Section 4.1 presents the advantages of the sequential reader and explains its usage and inner workings. Section 4.2 addresses the segmentation of the WRAM into several buffers which are needed by both sequential readers and MergeSort itself. Section 4.3 deals with the sequential MRAM MergeSort, shows how the sequential reader can be improved upon, and concludes with an analysis of the performance. Finally, the parallel MRAM MergeSort is built from the sequential one and analysed in Section 4.4.

In some instances, we employ `T` to denote the data type of the input and `sizeof(T)` to denote the size of an element of type `T` in bytes. Every measurement presented in this chapter was repeated ten times, which is a sufficient number given the large inputs and the low runtime variance of the MRAM MergeSorts. Appendix B contains further measurements but the ones essential for following the content of this chapter are also presented in figures herein.

4.1. The Sequential Reader

A DMA takes half a cycle per transferred byte. On top of that, each reading DMA comes with an additional overhead of 77 cycles and each writing DMA with an overhead of 61 cycles. To dilute this overhead, MRAM data is preferably processed in blocks using the functions `mram_read` and `mram_write`. The benefit of blockwise processing is so high that, according to Gómez-Luna et al. [18, p. 11], using `mram_read` and `mram_write` remains beneficial compared to accessing single MRAM elements even if only an eighth of the transferred data are actually of interest. However, the functions `mram_read` and `mram_write` do come with some constraints. Both the target and the source address must be aligned to 8 bytes. Also, the number of transferred bytes must be at least 8, at most 2048, and a multiple of 8. Furthermore, programmers are tasked with maintaining a buffer in the WRAM and transferring data at appropriate times, which, given that the WRAM is more than a thousand times smaller than the MRAM, is likely frequent.

Since processing MRAM data consecutively is a common occurrence, UPMEM provides a data structure called *sequential reader*. Through a set of C functions, a sequential reader automates the read-in process and, thereby, removes any need to care for the alignment of addresses or the loading of new data. On top of that, UPMEM claims that '[...] this abstraction implementation has been optimized and will provide better performance than a standard C check of the cache boundaries.' [44, Memory management – Sequential readers]

Of course, a sequential reader still requires a WRAM buffer. Its size in bytes is determined by the compile-time constant `SEQREAD_CACHE_SIZE`, which can be set to either 32, 64, 128, 256, 512, or 1024. The allocation of the buffer on the heap happens through the function `seqread_alloc` and is worth a closer look. Remember that the heap is actually implemented as a never-decreasing stack. This means that new memory is only ever allocated behind the heap pointer, which stores the end of the heap. With `SEQREAD_CACHE_SIZE` $= 2^i$, the i least significant bits of the first byte address in the buffer are required to be zero, for reasons explained shortly. Therefore, padding is introduced by having the heap pointer skip to the next higher multiple of `SEQREAD_CACHE_SIZE` if not already on such a multiple. Due to the nature of a stack, this has the drawback that the padding can never be allocated for something else. After the skip of the heap pointer, a total of $2 \times$ `SEQREAD_CACHE_SIZE` many bytes are allocated, also for reasons explained shortly. All in all, the memory footprint of a sequential-read buffer is at least $2 \times$ `SEQREAD_CACHE_SIZE` many bytes and less than $3 \times$ `SEQREAD_CACHE_SIZE` many.

The function `seqread_init` instructs a sequential reader to load data from a specified MRAM address into its buffer. Conceptually, the whole MRAM is divided into *pages* of size `SEQREAD_CACHE_SIZE`. To load data from the specified MRAM address, the address is rounded down the the next multiple of `SEQREAD_CACHE_SIZE`, which yields the beginning of the page containing the MRAM address. Then, $2 \times$ `SEQREAD_CACHE_SIZE` many bytes are loaded so that the buffer holds two pages. This way, data of some long, compound type at the end of the first page are fully loaded even if extending into the other page.

The function `seqread_init` also returns a pointer to the corresponding position of the specified MRAM address within the sequential-read buffer, to which we will refer as pointer to the *current element*. Due to the page model, this pointer may not be set to the beginning of the buffer. To access the current element, one simply dereferences the pointer. Calling the function `seqread_get` advances this pointer by a given number of bytes, which cannot be

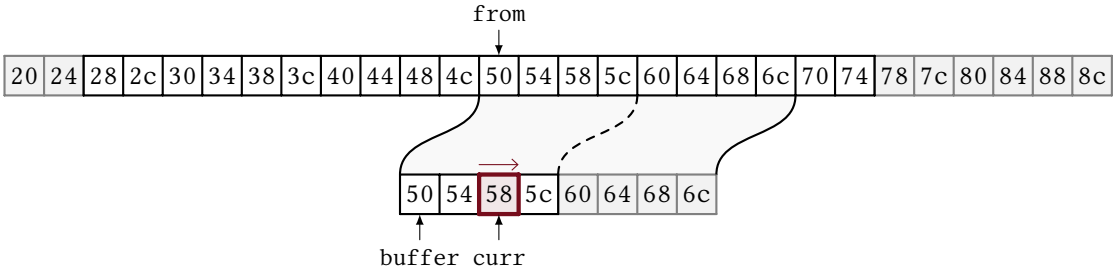


Figure 4.1. *An exemplary sequential reader with* `SEQREAD_CACHE_SIZE` *set to 16 being used to transfer 32-bit elements from the MRAM (top row) into the sequential-read buffer (bottom row). The hexadecimal numbers denote the addresses of the respective elements within the MRAM. Only the elements with addresses from* `0x28` *to* `0x74` *are sought to be read, however, the page model requires that the elements with addresses* `0x20`*,* `0x24`*, and* `0x78` *to* `0x8c` *be also loaded at some point. The pointer* `buffer` *constantly points to the beginning of the sequential-read buffer. The pointer* `from` *points to the MRAM address of the first byte within the buffer. The pointer* `curr` *moves from left to right, 4 bytes at a time. No byte of the second half of the buffer is ever read as the elements fit perfectly within the first half.*

greater than `SEQREAD_CACHE_SIZE`; this way of specifying bytes allows the sequential reader to support arbitrary data types. Once the pointer to the current element ends up in the second half of the buffer, it is set `SEQREAD_CACHE_SIZE` many bytes back so that it points to an address in the first half again. In addition, the MRAM address from which to read is increased by `SEQREAD_CACHE_SIZE` many bytes, and the next two subsequent pages are loaded. This means that the page stored in the second half of the buffer is loaded from the MRAM again but stored in the first half this time. Figure 4.1 visualises an intermediate state of a sequential reader, showcasing its characteristic read behaviour.

The acclaimed speedup through more performant bounds checks happens within the function `seqread_get`, which in turn calls the function `__builtin_dpu_seqread_get`. An inspection of its compilation with `SEQREAD_CACHE_SIZE` $= 2^i$ reveals the use of a combo instruction. The pointer to the current element is advanced by invoking an `add` instruction to increase the stored address. This `add` instruction uses a condition to detect the generation of the ith carry bit. To be more precise, this carry bit is set to `op1[i:0]` + `op2[i:0]`, where `op1[i:0]` and `op2[i:0]` are the $i + 1$ least significant bits of the involved operands, in this case the pointer and the number of bytes to advance. [44, DPU Handbook – Specific Conditions Common To Addition and Subtraction] Thanks to the carefully chosen size and alignment of the buffer, the generation of such a carry bit signifies that the pointer to the current element has left the first buffer half. This means that it takes just one instruction to advance the pointer, check the buffer boundaries and jump over — if needed — the subsequent instructions responsible for updating the whole reader.

4.2. The Triple Buffer

Before beginning to merge, it is yet again beneficial to form starting runs. The formation works by loading a block of MRAM data into the WRAM, sorting it with one of the algorithms presented in Chapter 3, and writing the sorted block back to the MRAM. As those sorting algorithm rely on the data being present entirely within the WRAM, the functions `mram_read` and `mram_write` are used directly. Contrary to the WRAM MergeSorts with starting runs of length 14, the lengths of the starting runs of the MRAM MergeSort go well into the hundreds. The reason is that longer starting runs reduce the number of rounds of MRAM merging and, thus, reduce DMAs, which are relatively costly. However, again similar to the WRAM MergeSorts, it can be beneficial to slightly reduce the starting run length to achieve more balanced and faster rounds. Nonetheless, the impact on the total runtime between 500, 600 and 700 elements per starting run is in the magnitude of one per mille, so for the sake of simplicity, the starting run length is set to the maximum amount of data which a tasklet can store in the WRAM.

This does raise the question how a large WRAM buffer for the starting run formation can be allocated while still leaving memory for the two sequential-read buffers used later during merging. The answer is a *triple buffer* which consists of a general-purpose buffer followed by two consecutive sequential-read buffers. If no sequential reader is in use, the triple buffer can be regarded as one contiguous buffer. To initialise the triple buffer, a tasklet first calls `mem_alloc` to allocate `CACHE_SIZE` many bytes on the heap, with `CACHE_SIZE` being a compile-time constant divisible by 8. This memory is referred to as *cache* and will be used later to store merged runs. Subsequently, the tasklet calls `seqread_alloc` twice. Due to the stack nature of the heap, the two sequential-read buffers are allocated directly behind the cache. To ensure the contiguity of the triple buffer if more than one tasklet is present, a mutex is employed such that only one tasklet initialises its triple buffer at a time. The entire triple buffer has the size `TRIPLE_BUFFER_SIZE` := `CACHE_SIZE` + 4 × `SEQREAD_CACHE_SIZE`, which is, for simplicity, the minimum number of allocated bytes and, therefore, the same for all tasklets even if some calls of `seqread_alloc` introduced padding. Note that padding may appear in front of each first sequential-read buffer only, since the heap pointer is necessarily at a multiple of `SEQREAD_CACHE_SIZE` after any call of `seqread_alloc`. Because of otherwise wasted memory, it makes little sense to set `CACHE_SIZE` to a value which is not a multiple of `SEQREAD_CACHE_SIZE`. The optimal values for `CACHE_SIZE` and `SEQREAD_CACHE_SIZE` are determined in Section 4.3.3.

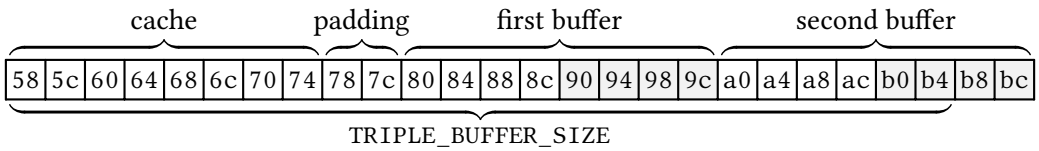


Figure 4.2. *An exemplary triple buffer for 32-bit elements with `CACHE_SIZE` set to 32 and `SEQREAD_CACHE_SIZE` set to 16. The hexadecimal numbers denote WRAM addresses. Note that the padding is still used when regarding the triple buffer as one contiguous buffer. Then, however, the last bytes of the second sequential-read buffer are unused.*

4.3. Sequential MergeSort

The MRAM MergeSort is based on the full-space WRAM MergeSort as presented in Section 3.5 so only the adaptations of the merging to the memory hierarchy are discussed. As precondition for the presented algorithm to work, it is required that run sizes be multiples of eight and that the runs and the output location be aligned to 8 B. This precondition simplifies DMAs and is trivially met for 64-bit elements. To meet it for 32-bit elements, one can insert a single dummy element, set to the maximum possible value, into the input and discard the last element of the sorted output.

4.3.1. Presentation of Key Aspects

The idea underlying MRAM merging is the following: First, initialise a sequential reader on either run. Then, repeatedly compare the current elements, write the less element to the cache, and read the next element. Whenever the cache is full, empty it by writing its content to the output location. Once the end of the less run is reached, stop comparing and empty the cache. Finally, flush the greater run by transferring its remainder from the MRAM to the output location with the help of the entire triple buffer.

During a merge, checks on the depletion of the less run and on the fill level of the cache are needed. To reduce the frequency of the former, the merge procedure consists of two tiers as shown in Algorithm 4.1. The first tier is in operation as long as there are more than `UNROLL_FACTOR` many elements left to merge in the less run. This is verifiable through the function `seqread_tell`, which returns the corresponding MRAM address of an element within a sequential-read buffer (ln. 3). First, an unrolled loop with `UNROLL_FACTOR` many iterations is executed, with each iteration comparing the current elements of both runs, writing the less one to the cache, and advancing the respective pointer (Algorithm 4.2). The less run cannot become depleted during this loop, so depletion checks after each iteration are unnecessary. Afterwards, it is checked whether the cache is filled with `MAX_FILL_LEVEL` many elements (ln. 5), with `MAX_FILL_LEVEL` being a multiple of `UNROLL_FACTOR` and `MAX_FILL_LEVEL`$\times$`sizeof(T)` $\leq$ `CACHE_SIZE` being a multiple of eight. If the fill level is too low, it is jumped back to the beginning of the tier. If, however, the maximum is indeed reached, the cache is emptied before jumping back (lns. 6 to 8). Because the output location is aligned to 8 B and `MAX_FILL_LEVEL`$\times$`sizeof(T)` is a multiple of eight, emptying the cache is possible through a simple call of `mram_write`.

Once there are `UNROLL_FACTOR` many elements or fewer left in the less run, the second tier begins. This one is structurally equal to the first tier with a single exception, for there is no guarantee that the unrolled loop will be executed in full: A depletion check now happens whenever an element of the less run is written to the cache. When it occurs, the cache is emptied and the greater run flushed, completing the merging. To flush the run, its remainder is iteratively transferred from its original position in the MRAM to the output location in blocks of size 2048 B using `mram_read` and `mram_write`. Since the sequential readers are of no use anymore, the whole triple buffer may be used during the transfers in case that `CACHE_SIZE` is below 2048. Emptying the cache requires checking the fill level in case of 32-bit elements as input. Should the number of elements within the cache be odd, then the size of the cache content is not a multiple of eight. In addition, the size of the remainder of the greater run is also not a multiple

Algorithm 4.1. *Two-tiered merging of two MRAM runs, where the first one is the less run. In the event of the second run being less, flip all indices.*

Data : sequential readers readers[2], pointers curr[2] to current elements, pointers ends[2] to last elements, output location out, cache cache
Result: both runs merged together and written to out

```
1  i ⟵ 0                                                  ▷ number of elements in cache
2  early_end ⟵ ends[0] − UNROLL_FACTOR + 1                ▷ no depletion certain until this point
3  while seqread_tell(curr[0], readers[0]) < early_end do  ▷ first tier
4      Merge UNROLL_FACTOR many elements without checking for depletion (Algorithm 4.2).
5      if i = MAX_FILL_LEVEL then
6          mram_write(cache, out, MAX_FILL_LEVEL × sizeof(T))
7          i ⟵ 0
8          out ⟵ out + MAX_FILL_LEVEL
9      end if
10 end while
11 while true do                                          ▷ second tier
12     Merge UNROLL_FACTOR many elements with checking for depletion (Algorithm 4.2).
13     if i = MAX_FILL_LEVEL then
14         mram_write(cache, out, MAX_FILL_LEVEL × sizeof(T))
15         i ⟵ 0
16         out ⟵ out + MAX_FILL_LEVEL
17     end if
18 end while
```

of eight, given that both the sizes of the two runs as well as `MAX_FILL_LEVEL × sizeof(T)` were such. For this reason, the current element of the greater run is moved to the cache in order to bring the size of its content to a multiple of eight, rendering it unproblematic to empty the cache and flush the greater run via `mram_read` and `mram_write`.

4.3.2. Investigation of the Compilation

The most frequently used sequential-reader function is `seqread_get`, followed at some distance by `seqread_tell` and, at even more distance, `seqread_init`. Each use of those functions constitutes a proper call as they cannot be inlined due to being a part of a different translation unit. A function call comes at non-negligible cost since every argument has to be loaded into the respective register, the jump to the function itself be performed, the stack pointer and return address be saved and reloaded, modified registers be restored if need be, and the jump back to the return address be performed. Since the DPU architecture is fundamentally compute-bound, function calls are a serious impediment to performance. In Chapter 3, this has already been an argument in favour of the oft-used InsertionSort whose short implementation lends itself to inlining.

Earlier attempts at reducing function calls included maintaining a counter on the number of elements left to make `seqread_tell` obsolete. This alone yielded prominent speedup while still

Algorithm 4.2. *Merging* UNROLL_FACTOR *many elements. This algorithm is part of Algorithm 4.1, meaning any change to a variable carries over. In the event of the second run being less, flip the indices in the inner* `if` *block and move it down into the* `else` *block.*

Data : sequential readers readers[2], pointers curr[2] to current elements, pointers ends[2] to last elements, output location out, cache cache, number i of elements in the cache
Result: UNROLL_FACTOR many elements merged to cache[i .. i + UNROLL_FACTOR − 1]

```
1  for k ← 1 to UNROLL_FACTOR do                                  ▷ unrolled loop
2      if *curr[0] ≤ *curr[1] then
3          cache[i++] ← *curr[0]
4          if seqread_tell(curr[0], readers[0]) = ends[0] then    ▷ omit in tier 1
5              Empty the cache.
6              Flush the other, nondepleted run.
7              return                                             ▷ stops Algorithm 4.1
8          end if
9          curr[0] ← seqread_get(curr[0], readers[0])
10     else
11         cache[i++] ← *curr[1]
12         curr[1] ← seqread_get(curr[1], readers[1])
13     end if
14 end for
```

being independent of the exact implementation of sequential readers and possible future changes to them. Similarly, calls to `seqread_get` were reduced by manually advancing the pointers to current elements as long as the ends of the first buffer halves were sufficiently far away. Ultimately though, even larger speedups are achievable by implementing an own sequential reader which can be inlined. The simplest way to do so is to duplicate the driver source file `seqread.inc` and have it be part of the same translation unit.[1] The speedup through inlining is significant. For example, with CACHE_SIZE = 1024, SEQREAD_CACHE_SIZE = 512, QuickSort as WRAM sorting algorithm, and 2^{19} uniformly distributed 32-bit integers, a MergeSort with inlined sequential readers achieves a speedup of 1.4 over a MergeSort where sequential readers are used as is, that is with function calls.

A drawback of the two-buffer system is that data are loaded twice. Since it is a precondition for runs to be aligned to 8 B and since elements are either 32 bits or 64 bits large, it is assured that the last element in the first buffer can never extend into the second half. Hence, a natural optimisation is to regard two consecutive sequential-read buffers as a singular one. New data is loaded only when the pointer to the current element reaches the end of the second original buffer. However, the two-buffer system is intrinsic to `__builtin_dpu_seqread_get`, that is the function used by `seqread_get`, and we are unaware of any alternative C function to it. For this reason, inline assembly must be employed to help adapt to the new buffer sizes. This includes changing the carry bit condition, the amount of MRAM data read, and the number of bytes by which a pointer to a current element is reset.

1. The BSD-style licence of the driver permits modification and redistribution of its files given proper credits.

```
1  add rcurr, rcurr, 8, nc10, .LABEL   // curr ← curr + 8; jump if no 10th carry bit
2  add rreader, rstack, -120           // get address of reader in the WRAM stack
3  lw rmram, rreader, 4                // load MRAM address of reader
4  add rmram, rmram, 1024              // MRAM address ← MRAM address + 1024
5  sw rrdr, 4, rreader                 // store new MRAM address in reader
6  lw rwram, rreader, 0                // load buffer address of reader
7  ldma rwram, rmram, 255              // load (255 + 1) × 8 bytes from the MRAM
8  add rcurr, rcurr, -1024             // curr ← curr − 1024
9 .LABEL:
```

(a) *The assembler code generated for `__builtin_dpu_seqread_get`. ´Line 2 is omitted in half of the cases, namely when the address of the reader is already stored in a register.*

```
1  add rcurr, rcurr, 8, nc11, .LABEL   // curr ← curr + 8; jump if no 11th carry bit
2  add rmram, rmram, 2048              // MRAM address ← MRAM address + 2048
3  ldma rwram, rmram, 255              // load (255 + 1) × 8 bytes from the MRAM
4  add rcurr, rcurr, -2048             // curr ← curr − 2048
5 .LABEL:
```

(b) *The handwritten assembler code.*

Figure 4.3. *Comparison of the assembler code of the function `__builtin_dpu_seqread_get` and the improved assembler code, which is handwritten. In both cases, elements are 64 bits large and `SEQREAD_CACHE_SIZE` is set to 1024. The flags `nc10` and `nc11` are true if and only if the respective carry bit is not generated. Only if a flag evaluates to true, a jump to the specified label is performed.*

A closer look at the original compilation, as shown in Fig. 4.3a, reveals more savings potential. Despite being constant, the WRAM address of the sequential-read buffer is loaded from the `struct` representing the reader (ln. 6) whenever new data need to be loaded. The MRAM address stored in the reader is not only loaded (ln. 3) but also stored (ln. 5) after being set to the new value. In exactly half of the instances where this assembler code appears, even the address of the reader `struct` itself first needs to be loaded from the stack (ln. 2), because the register into which it is loaded gets overwritten thereafter. These four load and store instruction can be cut by abandoning `structs` to represent the two readers used and employing two arrays of length 2, one for the buffer addresses and one for the MRAM addresses. As a consequence, these four addresses are kept permanently within registers without ever being overwritten, making the inline assembler code, as shown in Fig. 4.3b significantly shorter — admittedly, the savings are less than the reduced number of lines suggests, for the DMA dominates the runtime in this piece of code.

Next to `seqread_get`, more optimisation potential is hidden in the function `seqread_init`, which is called once for each sequential reader before a new pair of runs is merged. This function checks whether the MRAM address to which a sequential reader is set is already in the buffer. Since sequential readers are always initialised to the beginnings of runs and the runs are too long, this check is always negative and can be omitted. Moreover, recall that the original

function divides the MRAM into pages which begin at multiples of SEQREAD_CACHE_SIZE. This means that a run may begin in the middle of a page so the preceding, uninteresting data must be loaded as well. Since runs are aligned to 8 B, the function `seqread_init` can load from the first byte of the run onwards directly using `mram_read`. We can only speculate as to why the original function `seqread_init` did not simply round the given MRAM address down to the next multiple of eight but instead bothered with computing the page boundaries.

The MergeSort with fully optimised sequential readers achieves a speedup of 1.59 over the MergeSort with regular sequential readers and of 1.14 over the MergeSort with the inlined ones. This little gain despite the halved data transfers is testament to the dominance of computations on the runtime of MergeSort.

To conclude, a bug present in the regular sequential reader shall be mentioned. Recall that the MRAM is divided into pages and that always two whole pages are loaded, which may lead to unneeded data being loaded at the beginning by `seqread_init`. The bug occurs if one accesses data within the very last page of the MRAM since the regular sequential reader attempts to load the following, nonexistent page as well. This results in a DMA fault and an abortion of the execution. That is why the optimised sequential reader retains the page model in spite of unnecessary transfers, for in combination with only one page being loaded, a DMA fault never occurs. Such DMA faults are also a reason why the first tier cannot continue when reaching the address `early_end`, that is, why ln. 3 of Algorithm 4.1 contains a <-sign and not a ≤-sign. Otherwise, it might be the case that all remaining elements of the less run get merged back to back, making even the optimised sequential reader load a non-existent page in the last iteration.

4.3.3. Evaluation of the Performance

With only a single tasklet computing, the optimal parameter choice is clearly to set CACHE_SIZE and SEQREAD_CACHE_SIZE to the maximum value in order to minimise DMAs. Employing eleven tasklets is profitable despite limiting the possible values for the parameters severely, as the pipeline is utilised fully then, allowing to execute more instructions at the same time. Comparing the runtimes of a sole tasklet performing an MRAM MergeSort and of a tasklet with ten other ones working concurrently shows an increase of the runtime below the ten percent mark. This tiny increase is owed to multiple tasklets performing DMAs at the same time, stalling all involved tasklets but one since data transfers happen sequentially. Employing twelve tasklets not only makes simultaneous DMAs more frequent, it also increases the effective execution time of a single instruction from eleven cycles to twelve cycles. The effect is dampened by the latency hiding, as tasklets performing a DMA are suspended. Of course, using more than eleven tasklets may also be more convenient from an algorithmic point of view. For this reason, we will look at the influence of different values for CACHE_SIZE and SEQREAD_CACHE_SIZE with eleven, twelve and sixteen tasklets in this section. Please note that UNROLL_FACTOR is always set to eight and cannot be set greater, for the kernel would, otherwise, not fit within the IRAM anymore. Also, MAX_FILL_LEVEL is always set to CACHE_SIZE / `sizeof(T)`.

Since the main purpose of the sequential MRAM MergeSort is being a component of the parallel MRAM MergeSort, measurements are restricted to the maximum amount of data which the MRAM can hold. The parallel MergeSort is essentially a full-space MergeSort, needing auxiliary space of the same size as the input. As the total size of the MRAM is 64 MiB, the total

Tasklets	CACHE	4 × SEQREAD_CACHE		
		512	1024	2048
11	256	1095	1103	1083
	512	1090	1048	1065
	1024	1065	1051	1018
12	256	1052	1067	1049
	512	1048	1009	1032
	1024	1027	1013	983
16	256	994	969	958
	512	1002	964	945
	1024	971	981	946

(a) *32-bit integers ($n = 2^{23}$/Tasklets)*

Tasklets	CACHE	4 × SEQREAD_CACHE		
		512	1024	2048
11	256	800	796	772
	512	781	731	745
	1024	752	728	693
12	256	785	778	752
	512	765	713	725
	1024	732	708	674
16	256	770	714	683
	512	752	695	662
	1024	707	692	655

(b) *64-bit integers ($n = 2^{22}$/Tasklets)*

Table 4.1. *Runtimes of the sequential MRAM MergeSort in hundred thousand cycles for different values of* CACHE_SIZE *and* SEQREAD_CACHE_SIZE *on* 32 MiB *of uniformly distributed inputs. The input is divided evenly amongst the eleven to sixteen tasklets, which sort their proportion of the input concurrently. Measurements represents the means of the maximum runtimes, that is the wall-clock times.*

input size is capped at 32 MiB. Although the logging buffer of a DPU resides in the MRAM, this buffer is not created if no use of `printf`, `puts`, and `putchar` is detected in the program. Therefore, the entirety of the MRAM is indeed freely usable.

Table 4.1 shows the runtime of the sequential MergeSort for different values of CACHE_SIZE and SEQREAD_CACHE_SIZE. The number of tasklets is eleven, twelve, and thirteen, and each tasklet is assigned the same proportion of the input, which is 2^{23} and 2^{22} elements for 32-bit and 64-bit integers, respectively. Whilst the measurements were conducted on a uniform input distribution, we confirmed that the general observations hold true for the other input distributions as well. We opt to show the mean of the maximum runtimes of individual tasklets, that is the *wall-clock times*, since tasklets have to wait for each other in the parallel MergeSort. The difference between maximum and average runtimes is barely noticeably, however, being in the magnitude of one per mille for deterministic input distributions and in the magnitude of one per myriad for the random ones.

In most cases, it holds that the larger caches and buffers are, the faster the execution is. This suggests that being possibly stalled for longer is outweighed by the fewer occurrences of DMA overheads, especially when many tasklets are present. Also, the larger one of the parameters is, the greater the gain from enlarging the other one tends to be. The speedup has a tendency to be greater for 64-bit integers than for 32-bit integers. Still, some exceptions to these observations appear. For example, with sixteen tasklets, CACHE_SIZE = 1024, and 32-bit integers, doubling the combined size of two sequential-read buffers from 512 B to 1024 B increases the runtime by around 1 %. In summary, though, it is reasonable to set all parameters to the largest values allowed for by the limited WRAM, that is CACHE_SIZE = 1024 and SEQREAD_CACHE_SIZE = 512.

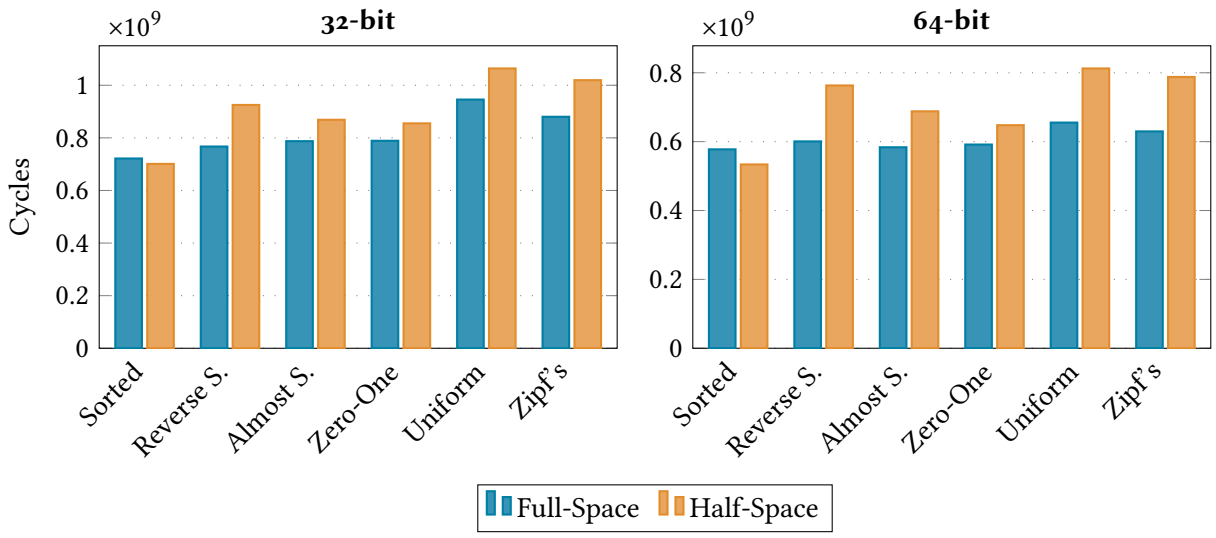


Figure 4.4. *Mean of the wall-clock times of concurrently executed MRAM MergeSorts on all benchmarked input distributions and data types. Sixteen tasklets were employed, each sorting 2^{19} many 32-bit integers and 2^{18} many 64-bit integers. Starting runs are formed using QuickSort.*

As final remark to Table 4.1, it should be noted that the reduction of the runtime through more tasklets is solely owed to the log-linear runtime of MergeSort and individual tasklets having to sort fewer elements. Indeed, the effect of the overfull pipeline becomes apparent either when computing the normalised runtimes of Table 4.1 or, simpler, when looking at Table B.1 where the number of elements sorted by each tasklet is kept constant.

Next, we look at the performance of the sequential MergeSort, both in a half-space and a full-space variant, across all benchmarked input distributions. Even though the parallel sorting algorithm presented in Section 4.4 is a type of MergeSort, its performance is improved by forgoing stability. Thence, it makes sense to analyse the performance of the sequential MRAM MergeSort with both QuickSort and MergeSort as WRAM sorting algorithms used during the formation of the starting runs. With QuickSort, Fig. 4.4 shows that like its WRAM counterpart, the full-space MRAM MergeSort works the fastest on sorted inputs, which is because one of the runs in each pair of runs is flushed in its entirety without overhead from comparisons and elementwise moves. Reverse sorted inputs are sorted slower despite the same flushing pattern. This is explicable through the worse performance of QuickSort — or, rather, InsertionSort — on reverse sorted inputs. The same phenomenon occurs with WRAM MergeSorts, but there, the influence of InsertionSort on the total runtime is much higher because of the shorter inputs, leading to this input distribution being their worst case. Almost sorted inputs are also sorted slower, however, this is because more elements need to be compared before non-depleted runs can be flushed: When a pair of runs is merged, one of them contains mainly little elements, whereas the other one contains mainly great elements. If the former run contains just one great element, it will be merged only almost entirely at first. Upon reaching the one greater element, some elements of the latter run are merged before the last element of the other run can finally be merged and a cheap flush be performed. Late flushes become more frequent with

uniform and Zipf's input distributions, explaining why they former constitutes the worst case. The ranking of the input distributions stays about the same when using MergeSort instead of QuickSort to form the starting runs with a notable exception. For the WRAM MergeSort, reverse sorted inputs constitute the worst case by such a wide margin that they become the worst case for the MRAM MergeSort, too, as shown in Fig. B.1.

For the half-space MergeSort, the picture is different. Only on sorted inputs does the half-space MergeSort manage to beat its full-space counterpart, and it falls behind on all other input distributions decisively. This is because copying the first runs has become too costly due to the DMAs, and not having to flush the second runs yields a considerable advantage only on sorted inputs. In conclusion, merging WRAM data is the fastest with the half-space variant, while merging MRAM data is the fastest with the full-space variant.

4.4. Parallel MergeSort

A simplistic way to parallelise MergeSort is the following: Let the number of tasklets be a power of two and have the tasklet identifiers start from zero. The whole input array is divided into as many shares of equal length as there are tasklets, and each tasklet sorts a share sequentially using the MRAM MergeSort of Section 4.3 to form starting runs. Once finished, Tasklet t with $t \bmod 2 = 1$ informs Tasklet $t-1$ that it is finished with sorting its share and suspends itself. Tasklet $t-1$ merges its own share and that of Tasklet t into a bigger run. Once finished, Tasklet t with $t \bmod 4 = 2$ informs Tasklet $t-2$ that it is finished with sorting the run and suspends itself. Then, Tasklet $t-2$ merges its run with that of Tasklet t. This scheme continues until the last round where the two remaining runs are sorted by Tasklet 0.

The bottleneck is the sequential execution of each merge which eventually leads to a single active tasklet. Even with infinite many processors, this simplistic parallel MergeSort can achieve a theoretical parallel speedup[2] of at most $\Theta(\log n)$. We implement an alternative by Cormen et al. [10] whose maximum theoretical parallel speedup is $\Theta(n/\log^2 n)$. Advantageously, Algorithm 4.1 for merging MRAM data can be reused without fundamental changes when adapting this parallel MergeSort to DPUs. Also, the number of synchronisation points is logarithmic in the number of tasklets only, and the time which each synchronisation takes is insignificant compared to the total runtime.

4.4.1. Presentation of Key Aspects

The parallel MRAM MergeSort is essentially a full-space MergeSort, meaning it needs an auxiliary array of the same size as the input array and the two arrays switch roles after each round. The parallel merge procedure (Fig. 4.5) operates on arbitrary runs, that is, it is no longer required that two runs be neighboured when merging. Likewise, the output location is arbitrary now, too, and not related to the indices of the two runs. Please note that the algorithm presented here is not stable, but we will propose a stabilised variant in the outlook.

Suppose that there are but two tasklets and both have formed a starting run using the sequential MRAM MergeSort. One of the tasklets is now temporarily suspended, whilst the other one determines which of the runs is longer. Then, it determines the median of the longer run, which will act as *pivot element* dividing the longer run into a front half and a back half. The pivot is used to find an index i which separates the shorter run into a front half and a back half such that any element with index $i' < i$ is not greater than the pivot and any element with index $i' \geq i$ is at least as great as the pivot. Finding such an index i can be implemented with a binary search. The elements in both front halves are at most as great as the pivot so they can go to the front of the output run. This means that the position of the pivot can be calculated by taking the output location and offsetting it by the combined length of the two front halves. Now, the front halves of both runs can be merged to the positions in front of the pivot using a sequential merge procedure, and the back halves can be merged to the positions behind the pivot. Since these two merges affect distinct elements and addresses, they can be performed in parallel by the two tasklets.

2. The *parallel speedup* S of a parallel algorithm A is defined as the ratio $t_1(A)/t_p(A)$ of its wall-clock time $t_1(A)$ when run with one processor and of its wall-clock time $t_p(A)$ when run with p processors.

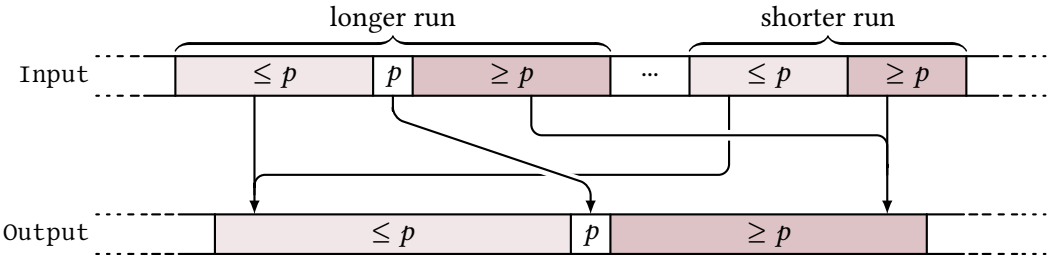


Figure 4.5. *A parallel merge of two runs. The first run is the longer one, so its median p is chosen as a pivot which is used to divide the shorter run. Afterwards, the pivot is moved to its output location. The two front halves of the runs are assigned to one tasklet and are merged to the positions in front of the pivot. At the same time, the two back halves of the runs are merged by another tasklet to the positions behind of the pivot. [10, Figure 27.6]*

The two tasklets do not merge the same number of elements necessarily, but the workloads cannot differ by a factor greater than three. The longer run has a length of at least $n/2$ elements and is divided into two halves with at least $n/4$ elements each due to the pivot being the median. Thus, both tasklets are guaranteed to merge at least $n/4$ elements. The shorter run has a length of at most $n/2$ elements. In the worst case, the pivot is either strictly less or strictly greater than all elements in the shorter run, meaning the shorter run is divided at its beginning or end, respectively, and is merged in its entirety by one of the tasklets. This means that each tasklet merges at least 25 % and at most 75 % of the elements.

The parallel MergeSort can be generalised to work with more than two tasklets. If there are, for example, four tasklets, then Tasklets 0 and 1 merge their runs in parallel, as do Tasklets 2 and 3. Then, Tasklet 0 partitions the resulting two runs and assigns their back halves to Tasklet 2, so that both Tasklet 0 and 2 can partition their particular halves again and merge in parallel with Tasklets 1 and 3, respectively. For simplicity, the number of tasklets is always a power of two in this Section 4.4.

Communication & Synchronisation The communication network can be visualised as a forest of binomial trees. During the first parallel merge, Tasklet 1 informs Tasklet 0 about being finished with sorting (*bottom-up communication*), and Tasklet 0 informs Tasklet 1 about being finished with partitioning (*top-down communication*). Likewise, Tasklet 3 and 4 communicate, Tasklets 5 and 6, and so on. At the beginning of the second parallel merge, Tasklets 1, 2, and 3 inform Tasklet 0 about being finished with sorting, and Tasklet 0 informs Tasklet 2 about being finished with partitioning. Then, both tasklets partition again and inform Tasklet 1 and 3, respectively. This bidirectional communication scheme is expanded for the third and fourth parallel merge if those exist. We implemented the identification of communication partners through bitwise logic on tasklet identifiers.

To inform tasklets on *which* elements they have to sort, there is a global WRAM array whither the division points indices are written by partitioning tasklets. To inform tasklets on *when* they are finished with sorting or partitioning, tasklets employ *handshakes*. Handshakes allow for bilateral communication, which is enough for parallel MergeSort. A tasklet can

call `handshake_wait_for(id)` and gets suspended until Tasklet `id` calls `handshake_notify()`. Likewise, if Tasklet `id` calls `handshake_notify()`, it gets suspended until some other tasklet calls `handshake_wait_for(id)`. If two or more tasklets wait for the same tasklet, an error is thrown and the execution halts. Handshakes render bottom-up communication straightforward: Each non-root of a communication tree calls `handshake_notify` when done with sorting, whereas the root calls `handshake_wait_for` for all of its successors. Due to workload imbalances, some tasklets will try to shake hands earlier than others and will have to wait. Because they are suspended while waiting, they free up the pipeline, thus accelerating the remaining tasklets. Top-down communication is also straightforward: After having shaken hands, each non-root immediately calls `handshake_notify` again to get suspended once more. The root repeatedly partitions the runs, writes the division points to the global WRAM array mentioned earlier, and calls `handshake_wait_for` to resume the next tasklet.

Binary Search The binary search is conventionally implemented, meaning elements are loaded individually from the MRAM and there is no mechanism to load a block of data via `mram_read` for a search within the WRAM once the search interval has been narrowed down enough. Whilst we did implement such a two-tier binary search, the speedup is below measurement uncertainty. The reason is that the binary search is executed a few times in total only, so its impact on the total runtime is minuscule. For the sake of code simplicity and kernel size, the WRAM search tier has been removed. Reducing the kernel size is a valid concern since the many unrolled loops bloat the kernel and only a handful of bytes in the IRAM remain free.

Alignment In Section 4.3, it is required that all sizes and positions be multiples of eight even if the input consists of 32-bit elements. This can be ensured during the formation of starting runs by dividing the input accordingly and introducing dummy variables if need be. Thereafter, however, there is no control over any alignment whatsoever because the sizes of run halves are arbitrary. This raises the need for modifications to the sequential merge procedure.

Before beginning the first tier, the alignment of the output location must be checked. If it is unaligned, the less of the first elements of both runs is written to the output location. As a result, the updated output location is aligned to 8 B again, meaning the first tier can proceed as normal since emptying the cache through `mram_write` is unproblematic.

At first, the second tier proceeds as normal, too. Once the less run is depleted, the cache may contain an odd number of elements, so the current element of the greater run is written to the cache before emptying it. However, if there are elements still remaining in the greater run, flushing the remainder becomes more complicated than in Section 4.3. There, it was sufficient to loop over the remainder in the MRAM, write it to the cache, and move it to the respective output location. Now, the current output location is still aligned to 8 B but it may very well be that the first element of the remainder has an unaligned address. This indicates a mismatch, for all unaligned elements must be transferred to an aligned address and all aligned elements to an unaligned one. When calling `mram_read` and `mram_write`, the alignment of elements within the MRAM and within the WRAM must be the same. If such an instance is detected, the remainder is loaded blockwise from the MRAM into the cache. There, a loop shifts each element forward by one position, resolving the mismatch. Afterwards, the shifted elements can

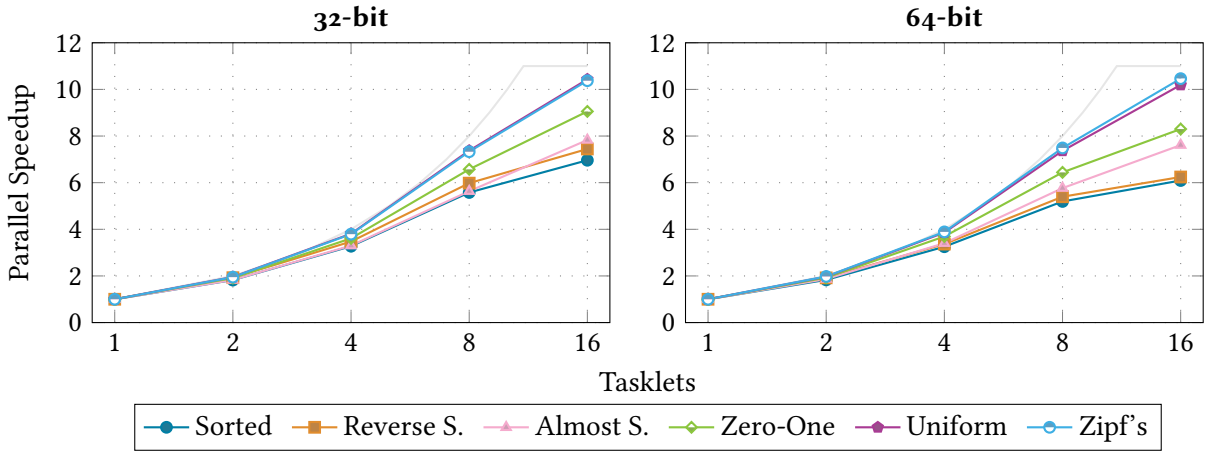


Figure 4.6. *Mean parallel speedups of MergeSort on all benchmarked input distributions and data types with* 32 MiB *of data. The grey line indicates an ideal, linear speedup which is capped at 11.*

be written to the output location. Since only an even number of elements can be moved via `mram_write`, it may be necessary to transfer a single item individually at the end in case that the remainder had an odd length.

Writing single 32-bit elements to the MRAM is not threadsafe, since, internally, eight bytes are read to a hidden WRAM cache, partially modified, and written back to the MRAM. Therefore, an atomic write, which utilises costly virtual mutexes, must be performed.

4.4.2. Evaluation of the Performance

The parallel speedup of the MergeSort on 32 MiB of data is shown in Fig. 4.6. An ideal parallel speedup would be linear in the number of tasklets but capped at 11 or, rather, slightly above because of DMA latency hiding. For inputs with a uniform or Zipf's distribution, the measured speedup is very close to the optimum, reaching values above 10 for both 32-bit and 64-bit integers. This is owed to workloads tending to be balanced naturally and tasklets being removed from the pipeline once they are finished. For all other inputs, the parallel speedup is approximately between 7 and 9 with 32-bit integers and between 6 and 8 with 64-bit integers — a consequence of workloads becoming more unbalanced. This shall be illustrated by sorted inputs: In the last round, two runs remain. When Tasklet 0 performs the first partitioning step, the pivot divides the longer run into two equally long halves. However, the pivot is strictly less or greater than any element in the shorter run, meaning the shorter run keeps its length of about $n/2$ many elements as the division point is at one of its ends. Such unbalanced divisions carry on to further partitioning steps. When the number of tasklets is sixteen, the ratio between the least and the greatest number of assigned elements in the last round of parallel merging is about 2.3 for zero-one inputs and 4 for the three kinds of sorted inputs.

Figure 4.7 shows the wall-clock times. The measurements are subdivided into the three phases of the parallel MergeSort, that is the sequential WRAM phase, the sequential MRAM phase, and the parallel MRAM phase. On the one hand, most results are unsurprising. The

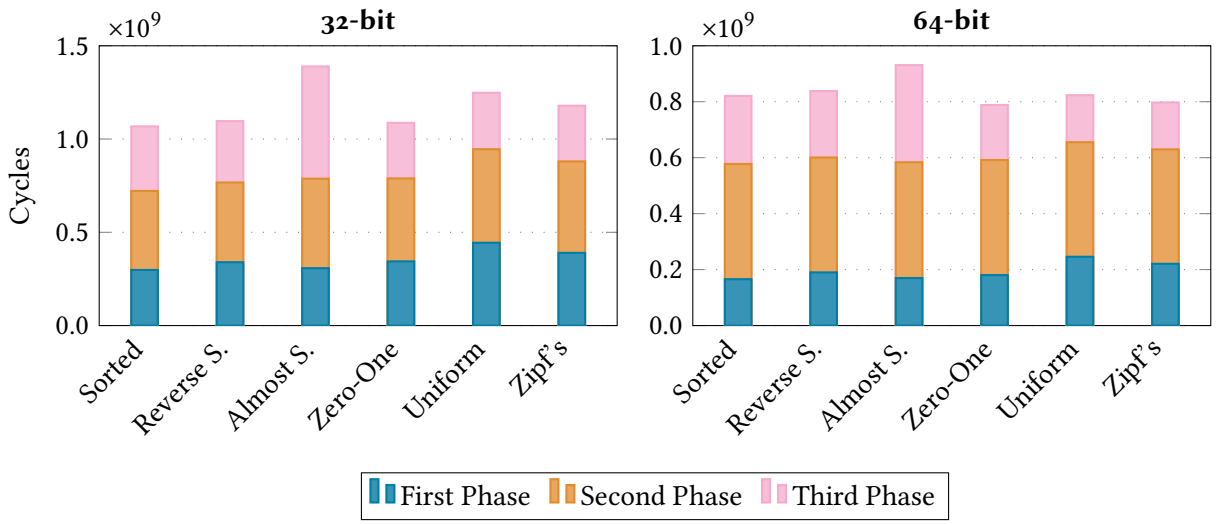


Figure 4.7. *Mean wall-clock times of the parallel MergeSort, broken down into its three phases, on all benchmarked input distributions and data types with* 32 MiB *of data. The first phase comprises sequential sorting in the WRAM, the second one sequential sorting in the MRAM, and the third one parallel sorting in the MRAM.*

sorted, reverse sorted, and zero-one input distribution are sorted quickly as they lead to short first and second phases, whilst the uniform and Zipf's input distribution make these two phases last longer. The third phase always takes approximately the same amount of time, implying that workload imbalances are cancelled out by earlier flushes, although the effect is weaker for 64-bit integers where computation is more costly.

Almost sorted inputs, on the other hand, are clear outsiders because of the remarkably long third phase, making them the worst case amongst all benchmarked ones. Despite the long runtime, the parallel speedup is about the same as for sorted and reverse sorted inputs. Indeed, they all share equal or nearly equal workload imbalances. The explanation for the third phase being so long is the same as the one given in Section 4.3.3 with respect to the sequential MergeSort: A few great elements placed in a run of mostly little elements delays flushes, and the longer the runs become, the likelier it is for a run to contain an overly great element.

Chapter 5.

Conclusion

In this thesis, we engineered several sequential and parallel integer sorting algorithms utilising in-memory processing (PIM) on UPMEM-based DRAM processing units (DPUs). DPUs are threaded general-purpose processors which reside next to the memory banks of DRAM modules. Due to this spatial proximity, the latency of memory accesses is greatly reduced. However, the computational capability of a DPU is low. The memory hierarchy of a DPU is three-tiered, entailing registers, a small but fast scratchpad memory called Working RAM (WRAM) and a larger but slower Main RAM (MRAM).

In Chapter 3, we focused on sequentially sorting data which fits entirely into the WRAM. A crucial means to optimise the performance of a DPU algorithm proved to be the reduction of the instruction count, as all instructions are executed in the same amount of time and WRAM accesses are uniform. Often, this included the exploitation of sentinel values to reduce bounds checks and loop unrolling to lessen loop overheads. We found QuickSort to deliver a well-rounded performance across all benchmarked input distributions. The performance by MergeSort is strong as well, mostly following suit and in some cases even surpassing QuickSort. HeapSort proved to be uncompetitive. Both QuickSort and MergeSort make use of InsertionSort, which we found to perform well on short inputs with about 16 elements or fewer.

In Chapter 4, we focused on sorting data which have to be stored in the MRAM for size reasons. We adapted MergeSort for sequential execution by a single tasklet and parallel execution by an entire DPU, designing an elaborate two-tier merge procedure to lower the instruction count. An additional challenge was the reuse of allocated memory and the management of data transfers between the MRAM and the WRAM. For the latter, UPMEM provides a software utility called sequential reader, which we optimised to gain a further speedup of 1.6. In case of the parallel sorting algorithm, communication and synchronisation were implemented using shared memory. The parallel speedup achieved ranges from nearly hitting the optimum to reaching just half of it, depending on the input distribution.

There is still room for improvement of the proposed sorting algorithms. The compiler produces suboptimal code in quite a few instances of which we have noted several throughout this thesis. QuickSort suffers the most from this and its current implementation has shortcomings whose removal may culminate in the use of inline assembler. A more optimised QuickSort might manage to beat MergeSort more clearly. The compiler issues were an additional reason not to design a QuickSort which works on MRAM data. We conjecture that such an MRAM QuickSort would be very performant due to the uniform cost of instructions and memory accesses. Our optimised sequential reader could aid with the latter, as it is trivially expanded to support the two different directions in which QuickSort reads data. Furthermore, own cursory experiments

have shown that sorting networks are a strong contender to InsertionSort. More suggestions concerning the sequential sorting algorithms can be found in Section 3.6. In case of the parallel sorting algorithms, there are two main areas needing improvement: Load imbalances lead to a worse speedup on some input distributions, and the stability property is lost despite being based on MergeSort. Therefore, we propose the following methods as possible future changes to the parallel merge method.

Load Imbalances Recall that the two halves assigned to a tasklet contain between 25 % and 75 % of the elements of the respective runs, which can lead to workload imbalances. A more even workload of 50 % would be achieved if the median of the merged run were chosen as pivot and the runs divided accordingly. Finding the two positions where the runs should be divided according to this common median requires a modification to the binary search. Two search intervals are set up, one for either run. Then, the medians of both runs are determined. If this does not produce valid division points, then one of the runs has more little elements in its front half than the other run. This means that the front half of this run must become longer and the front half of the other run shorter. Therefore, the search intervals can be narrowed down to the righthand side of the run with the less elements and to the lefthand side of the run with the greater elements. The process can now be repeated until two valid division points are found.

Stability The parallel merge method is unstable because one or both runs may contain the pivot value multiple times and a division point may separate the duplicates. Therefore, it must be ensured that all duplicates of a run remain within the same half. To do so, once the median is determined, it is checked if the left neighbour of the median has the same value. If so, there may be even more duplicates so a binary search is employed to find the earliest occurrence of the pivot value within the longer run. This earliest occurrence marks the division point for the longer run. Similarly, the binary search in the shorter run now has to find the earliest possible division point, too.

Currently, the implementations of the sorting algorithms support only 32-bit and 64-bit integers. Supporting shorter integers or compound data types would probably complicate memory alignment, but ultimately, the performance should remain largely the same. Supporting floating point numbers, however, would likely hurt the performance irremediably. The reason is that DPUs have no hardware support for floating point arithmetic, so comparisons between floating point numbers require emulation in software.

Then next greater step in sorting integers would be the employment of multiple DPUs. New challenges for communication and synchronisation open up, as there is no direct communication channel between DPUs, much less shared memory. Instead, any inter-DPU communication must be overseen by a host CPU. Moreover, DPUs are grouped together, and the host cannot access a DPU until all DPUs of its group have finished executing.

Appendix A.

Further Measurements on Sorting in the WRAM

This appendix contains a comprehensive collection of measurements expanding the content of Chapter 3. Every measurement was repeated a thousand times with the sorting algorithms in their default configuration unless explicitly noted otherwise:

InsertionSort explicit sentinel value

ShellSort explicit sentinel values; step sizes $h = (1, 6)$ for inputs with at most 64 elements and $h = (1, 4, 17)$ for longer ones

HeapSort top-down for 32-bit integers; bottom-up with swap disparity for 64-bit integers

QuickSort fallback threshold of 18 elements; random medians as pivots; prioritisation of right-hand partitions over left-hand partitions; iterative for 32-bit integers; recursive for 64-bit integers; Handling (5)

MergeSort half-space; starting run length of 14 elements

Some figures include tinted areas around the plots. These denote the three-sigma range, that is the 99.7 % confidence intervals.

A.1. InsertionSort

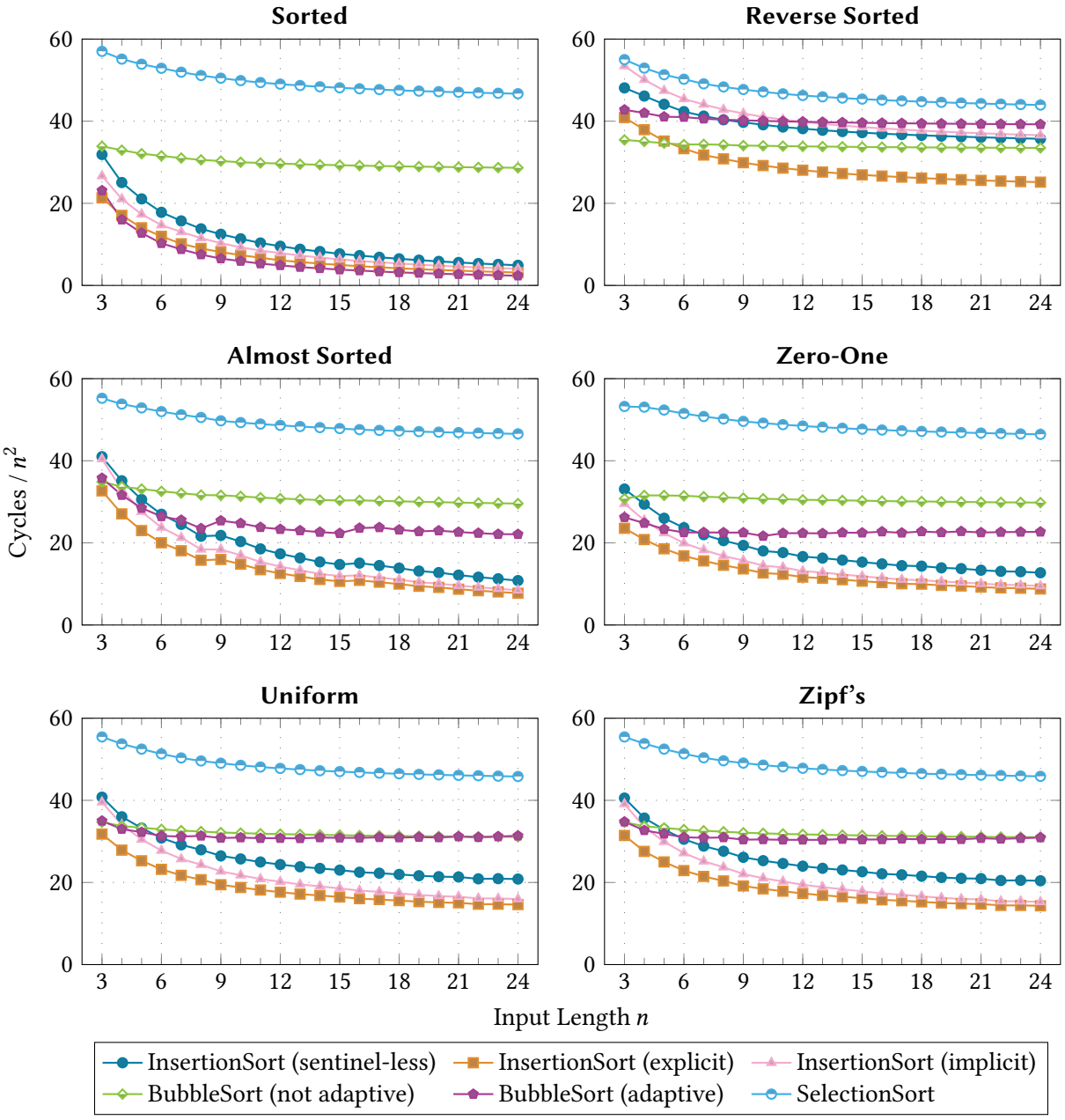


Figure A.1. *An extension to Fig. 3.2. The date size is 32 bits.*

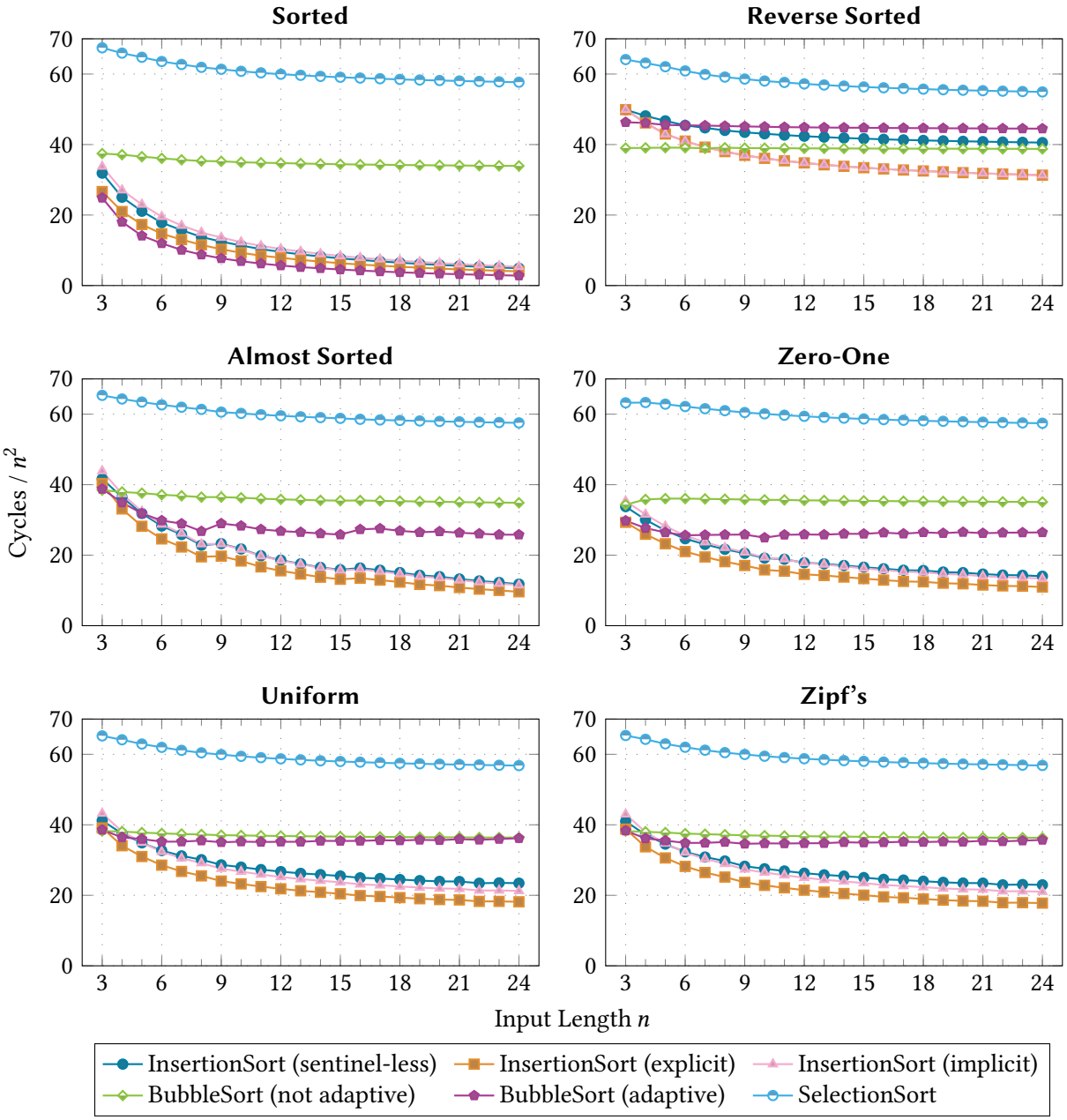


Figure A.2. *An extension to Fig. 3.2. The date size is 64 bits.*

A.2. ShellSort

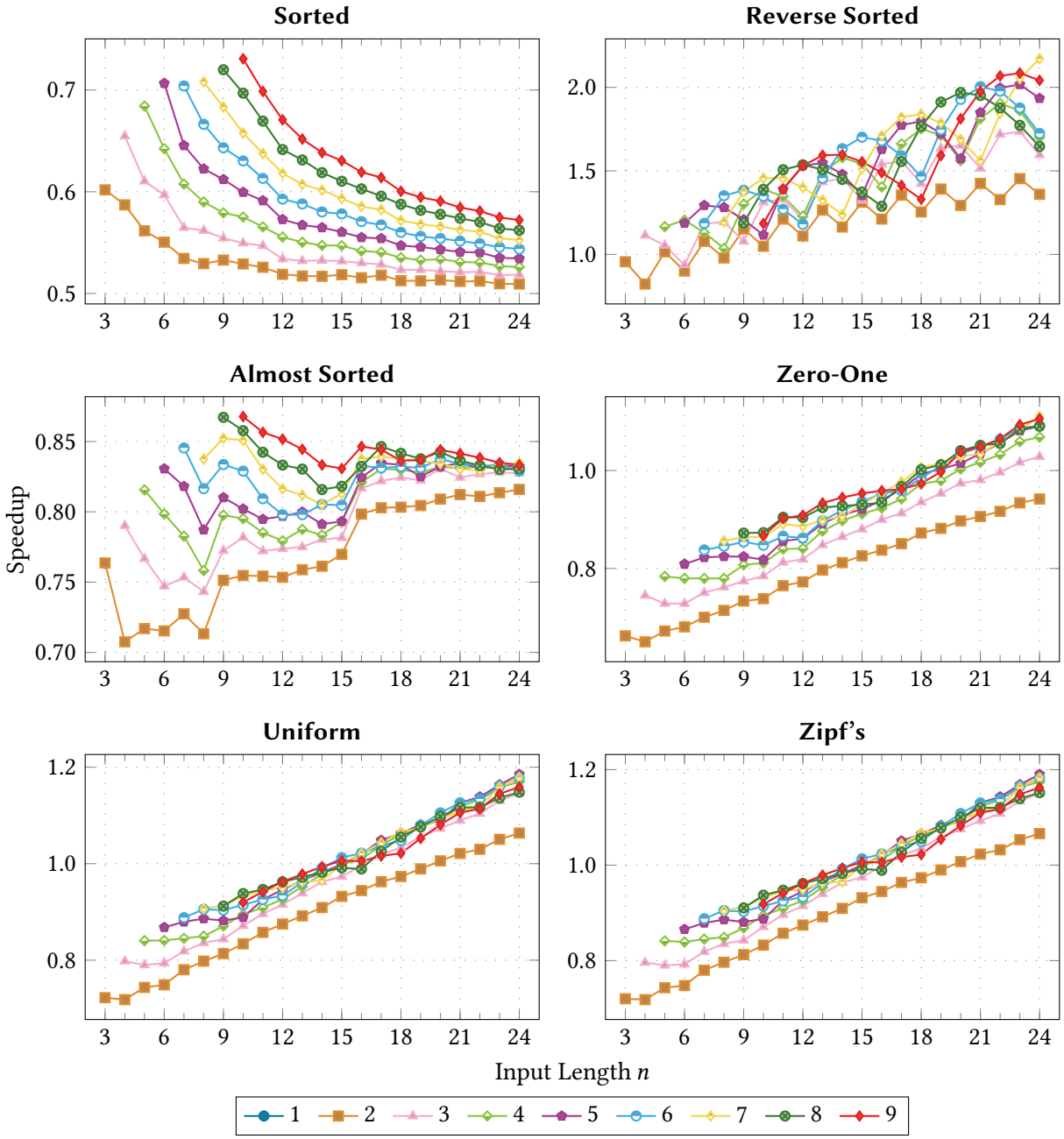


Figure A.3. *An extension to Fig. 3.3. Instead of total runtimes, the speedups over InsertionSort are given for better clarity. The date size is 32 bits.*

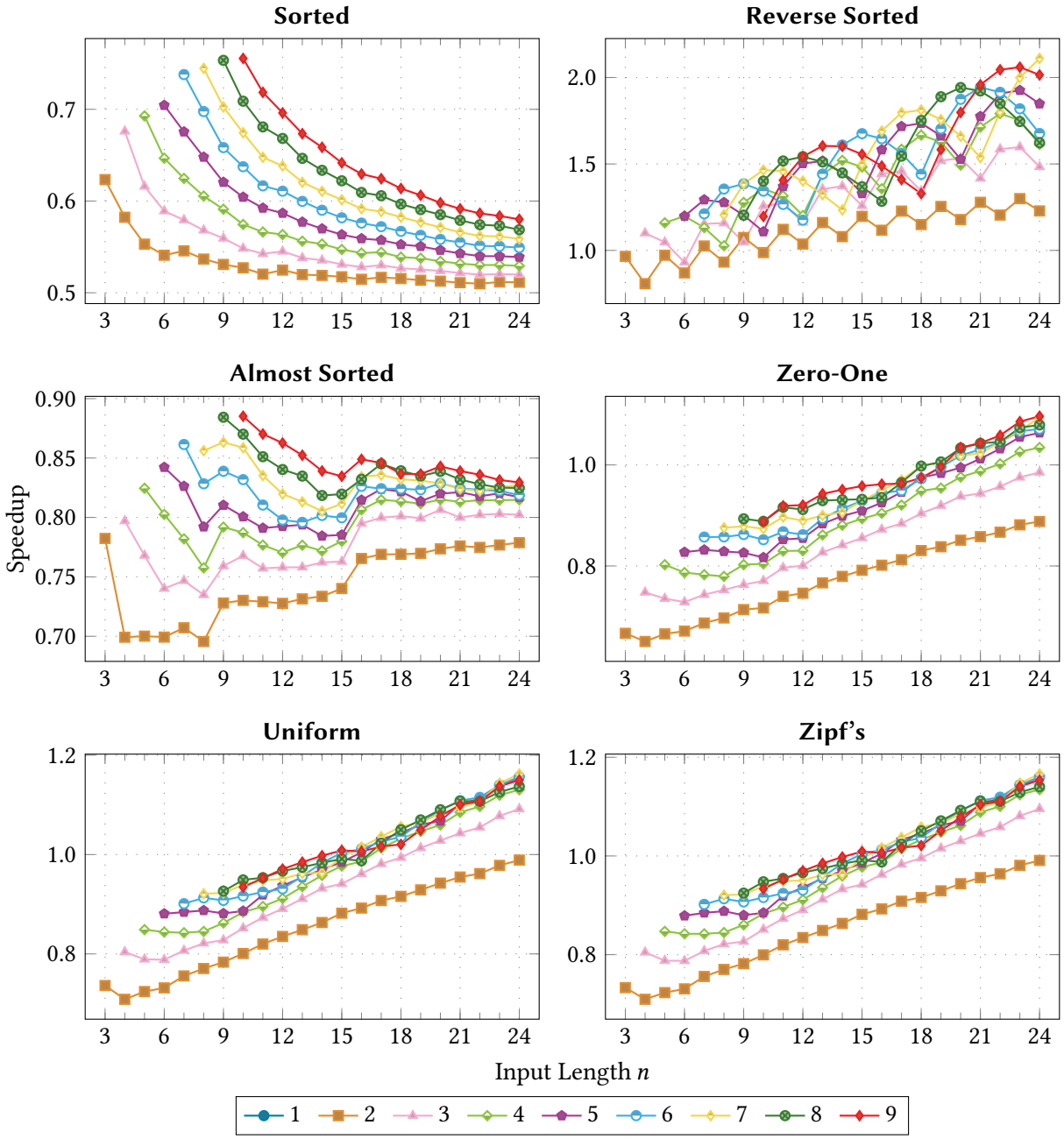


Figure A.4. *An extension to Fig. 3.3. Instead of total runtimes, the speedups over InsertionSort are given for better clarity. The date size is 64 bits.*

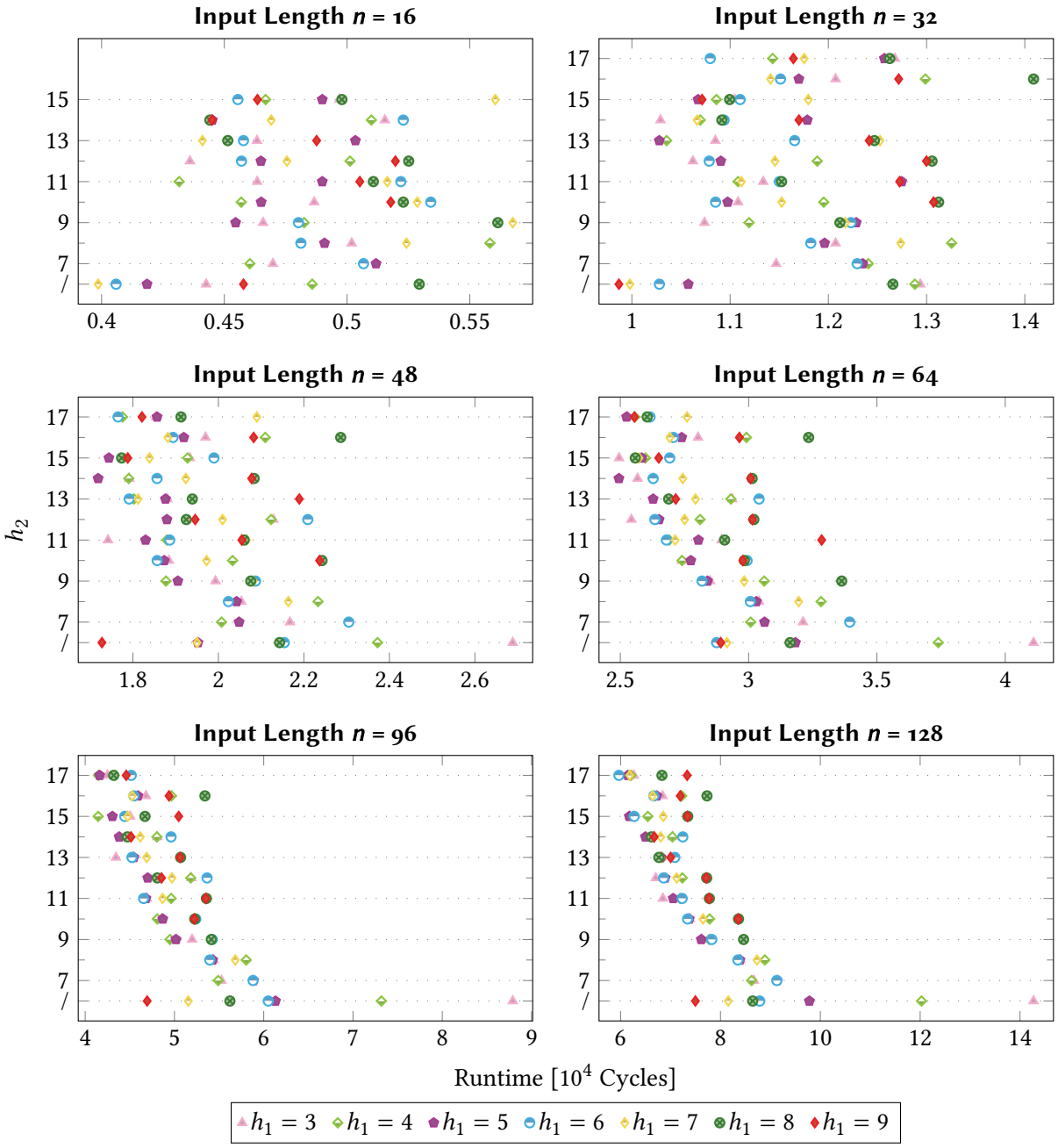


Figure A.5. *An extension to Fig. 3.4. The date size is 32 bits. The input distribution is the reverse sorted one.*

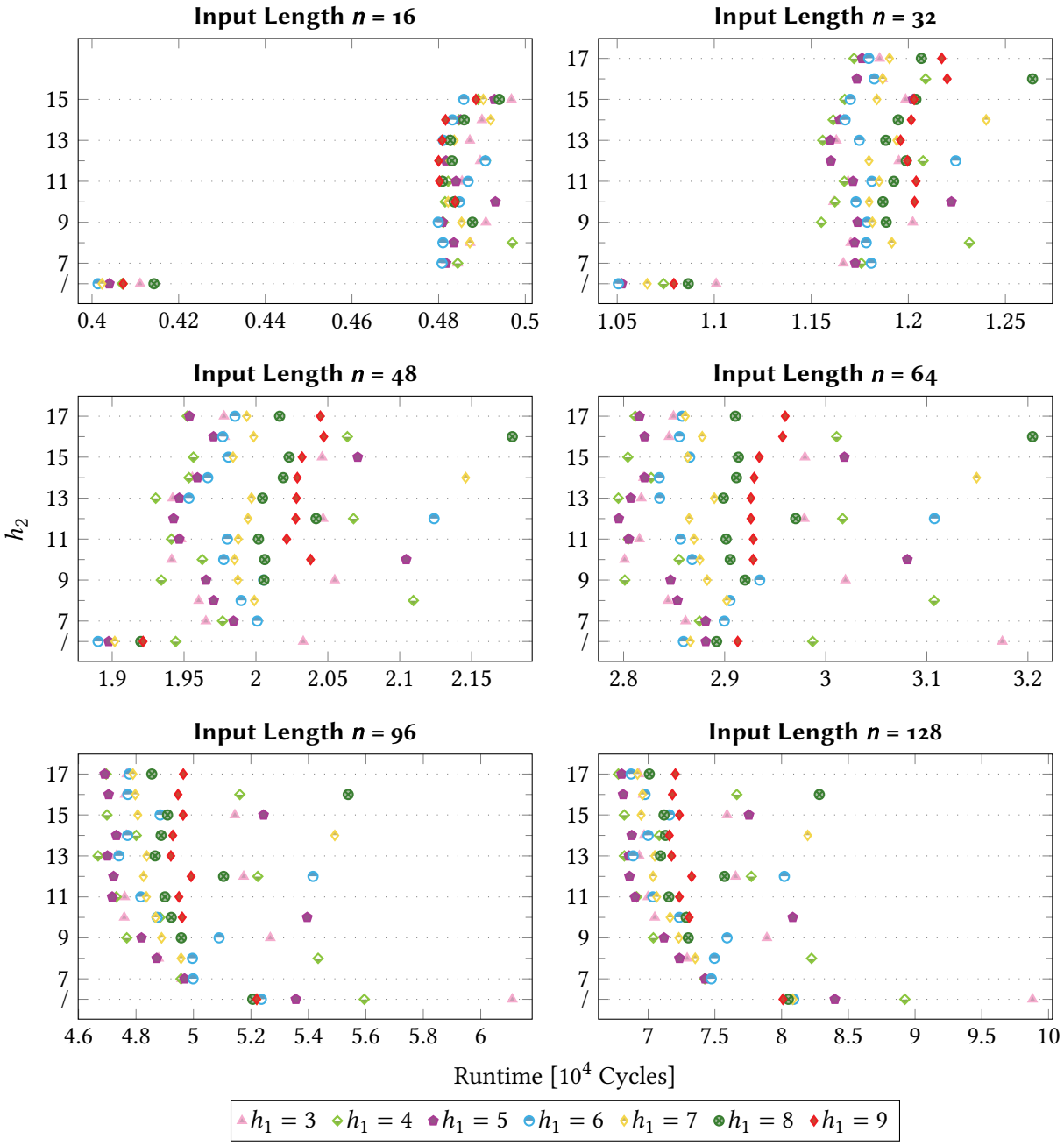


Figure A.6. *A repetition of Fig. 3.4. The date size is 32 bits. The input distribution is the uniform one.*

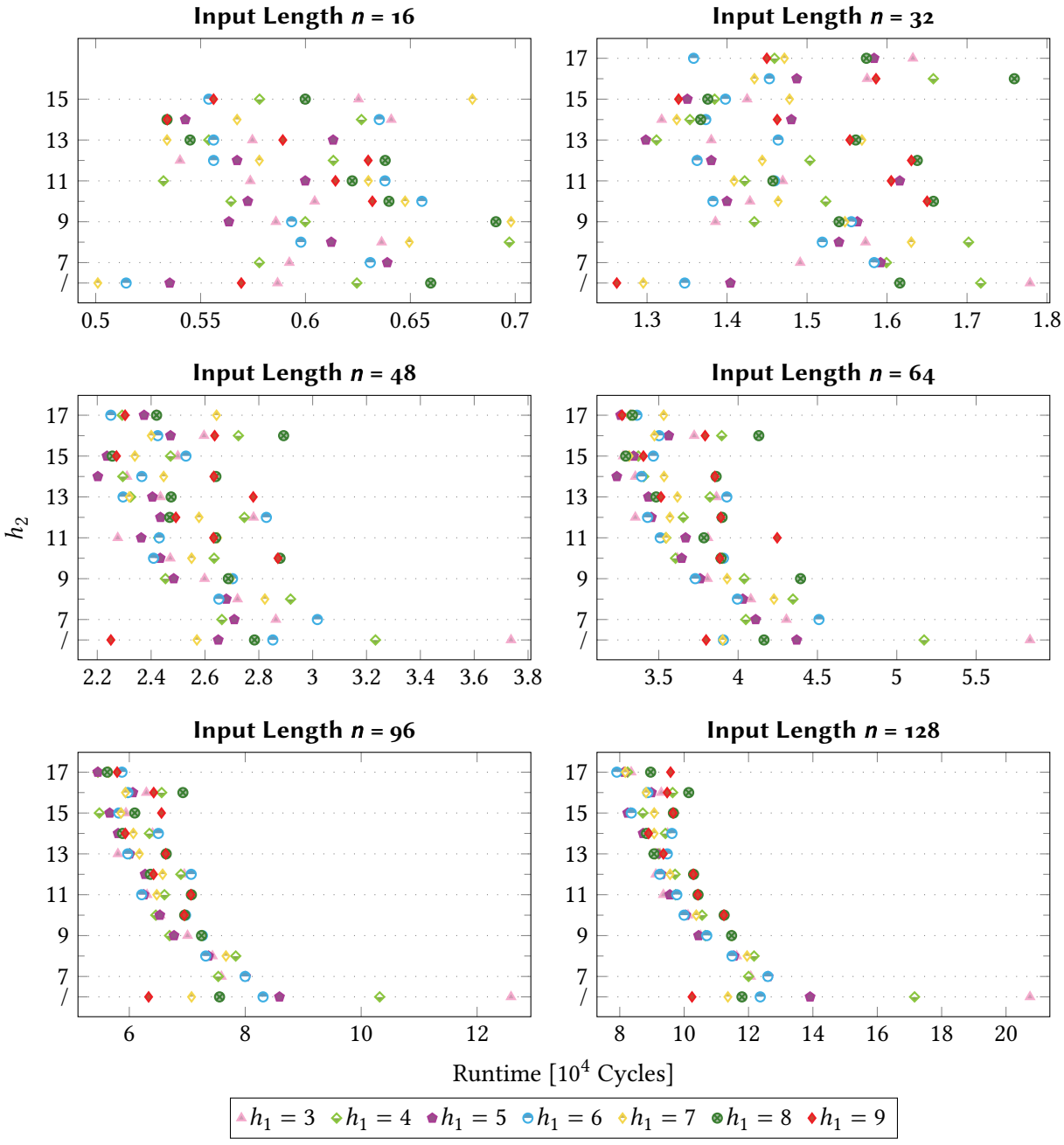


Figure A.7. *An extension to Fig. 3.4. The date size is 64 bits. The input distribution is the reverse sorted one.*

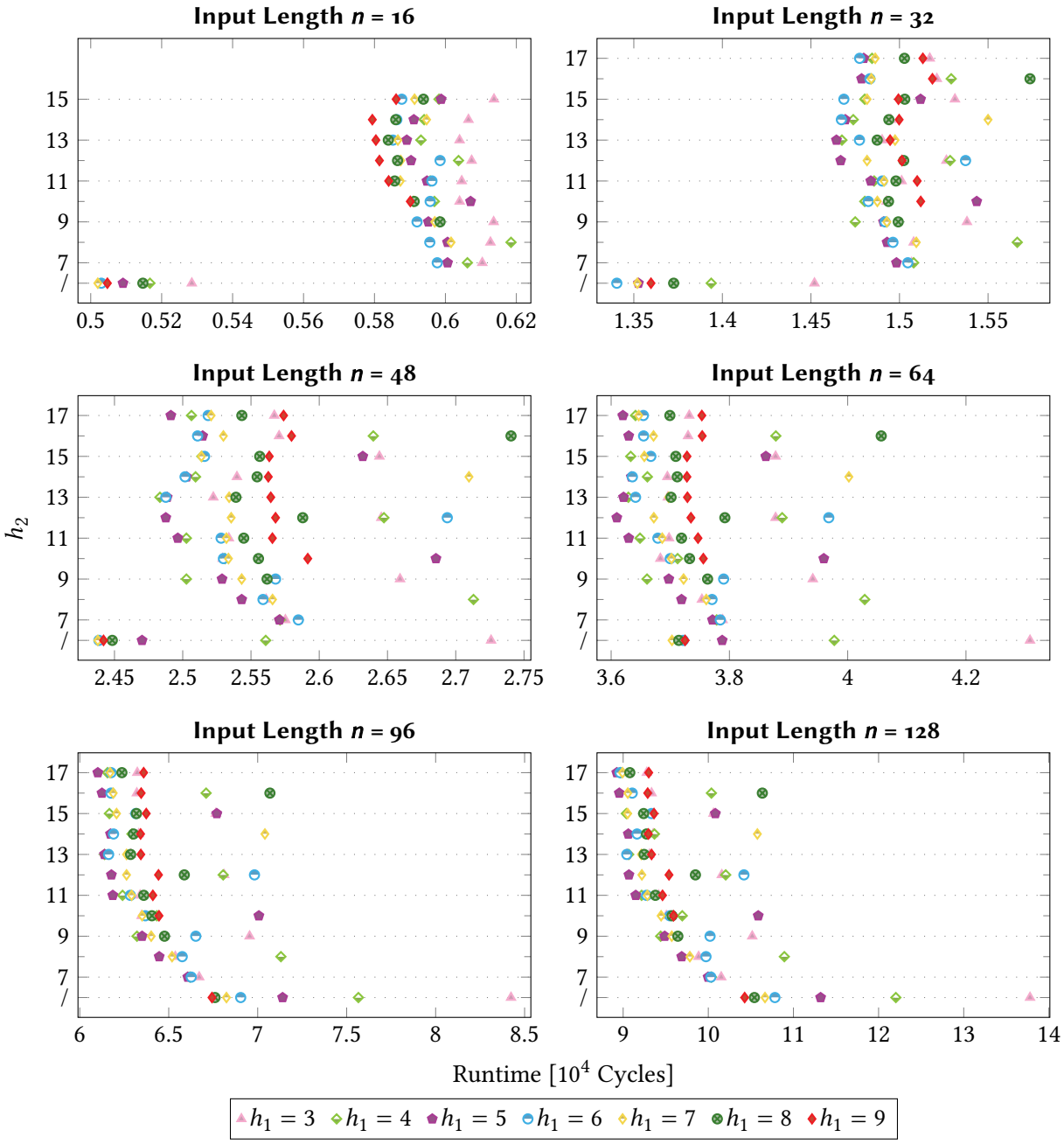


Figure A.8. *An extension to Fig. 3.4. The date size is 64 bits. The input distribution is the uniform one.*

A.3. HeapSort

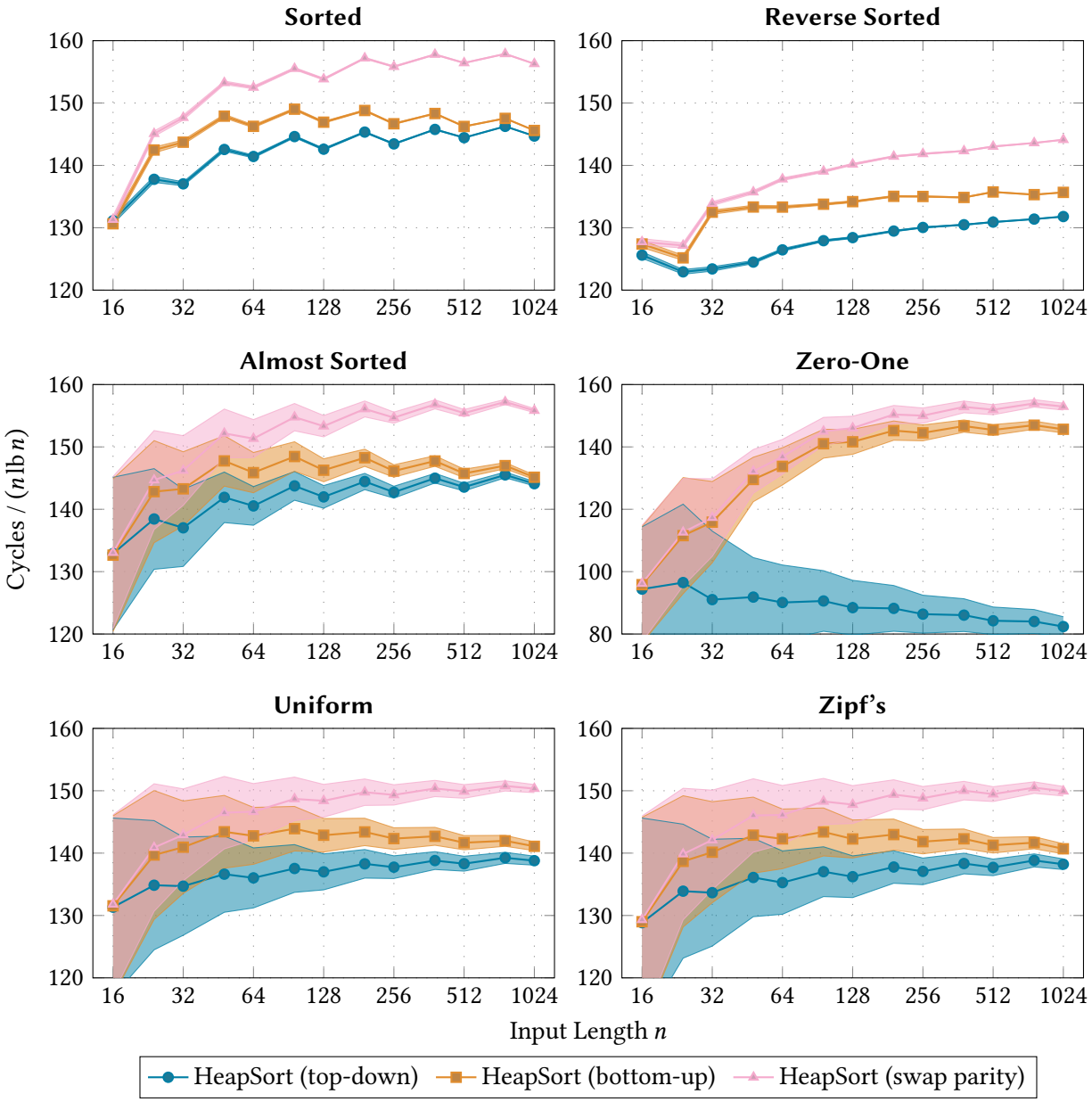


Figure A.9. *An extension to Fig. 3.5. The date size is 32 bits. Beware of the different y axis for the zero-one input distribution.*

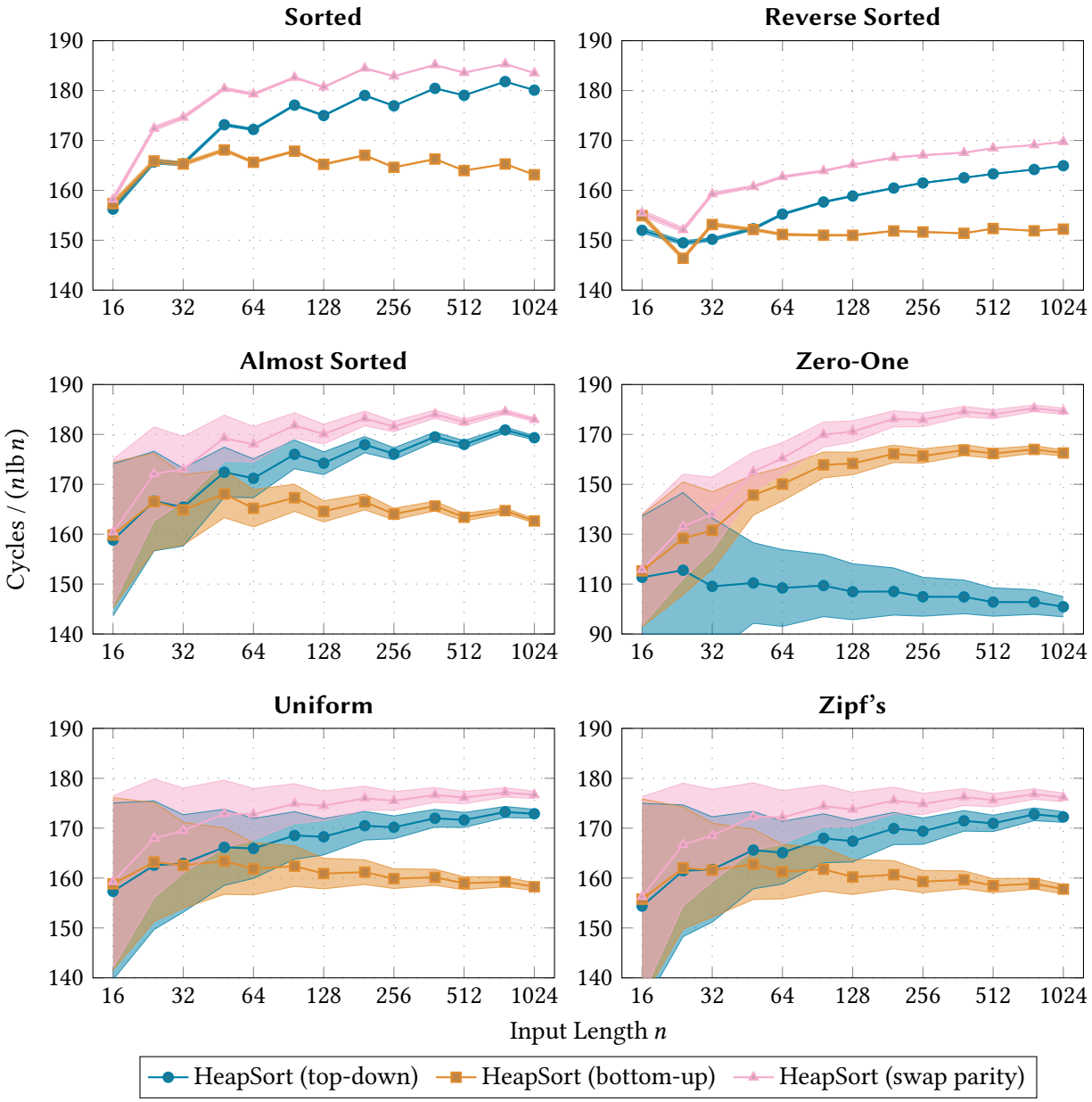


Figure A.10. *An extension to Fig. 3.5. The date size is 64 bits. Beware of the different y axis for the zero-one input distribution.*

A.4. QuickSort

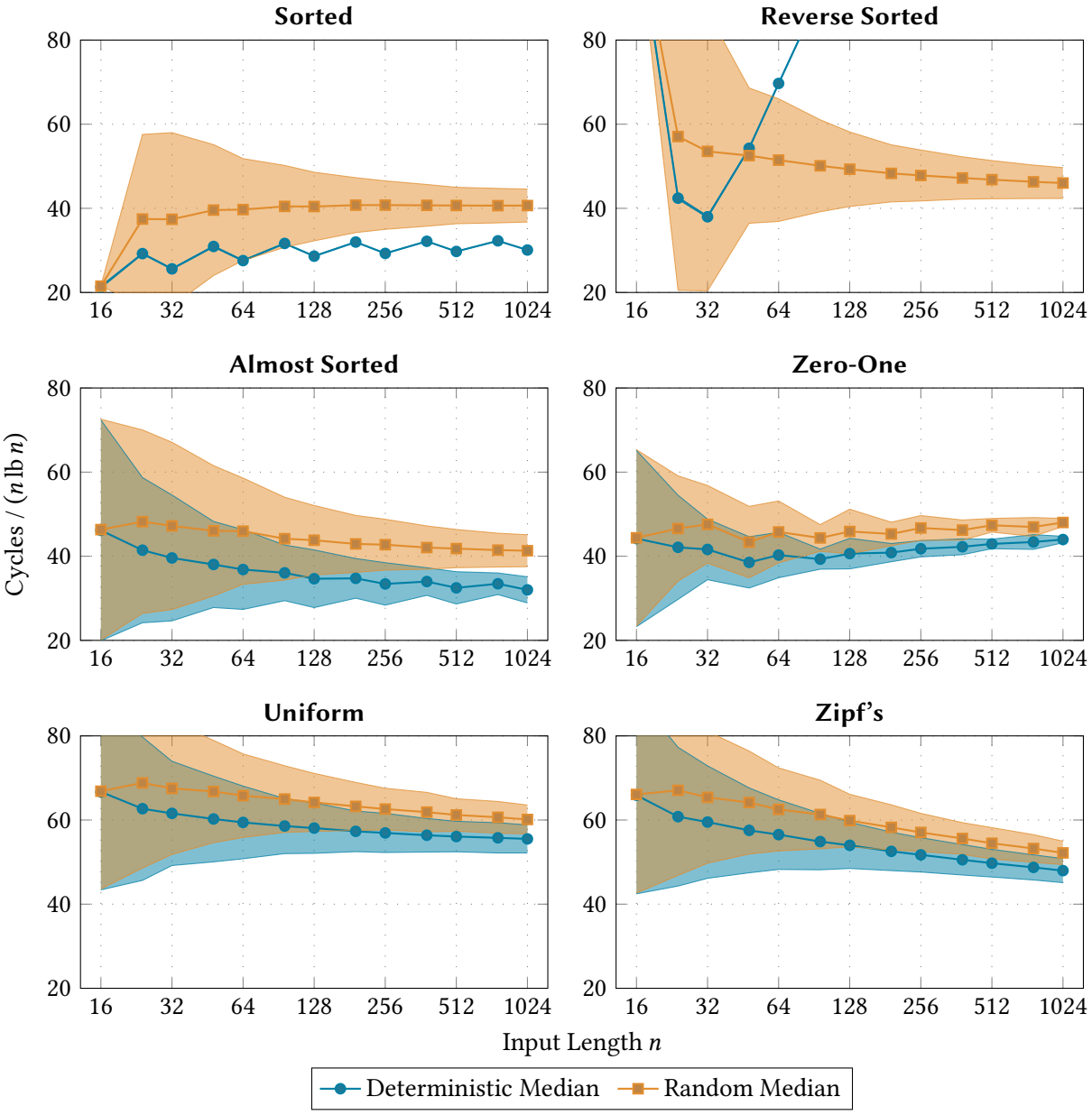


Figure A.11. *An extension to Fig. 3.8 with two different methods to selects pivots. The date size is 32 bits.*

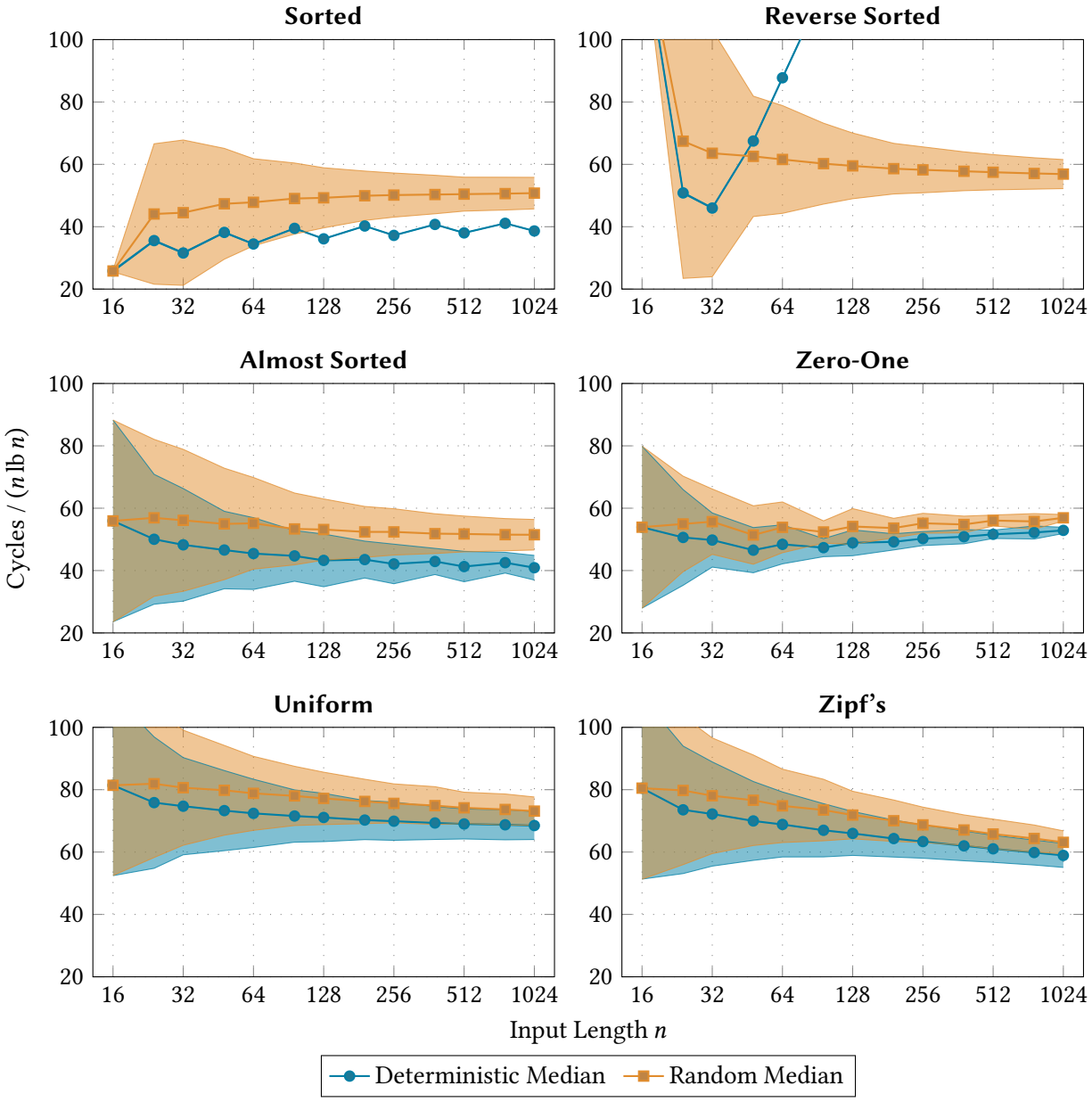
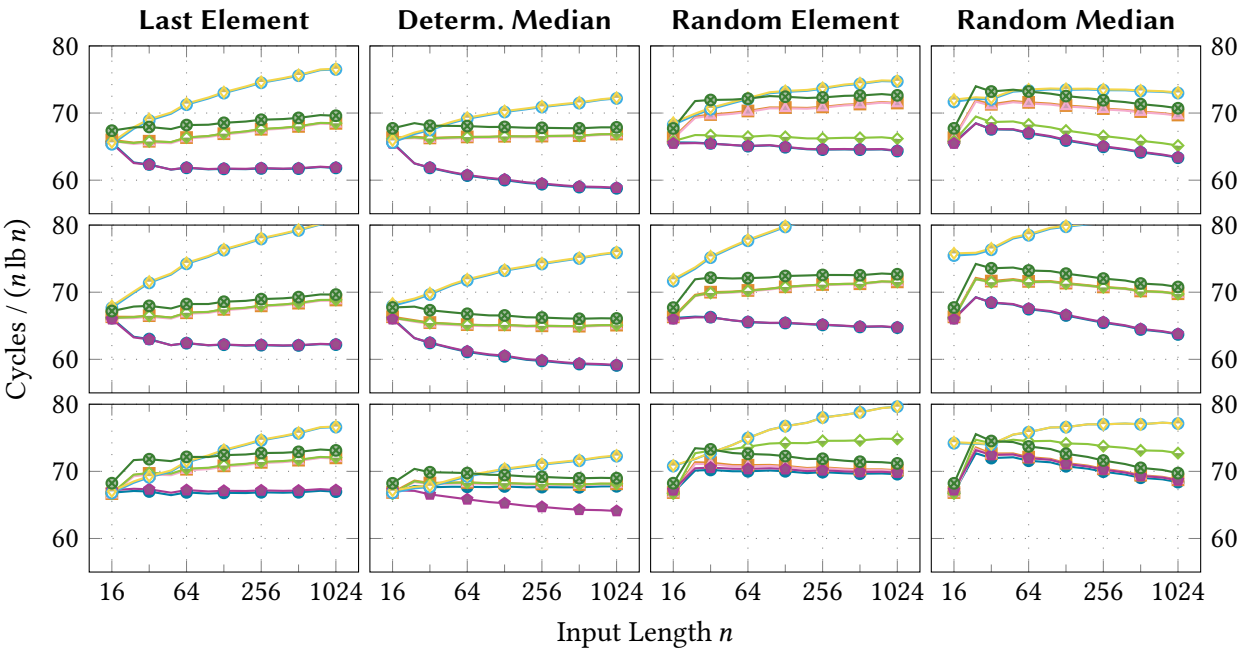
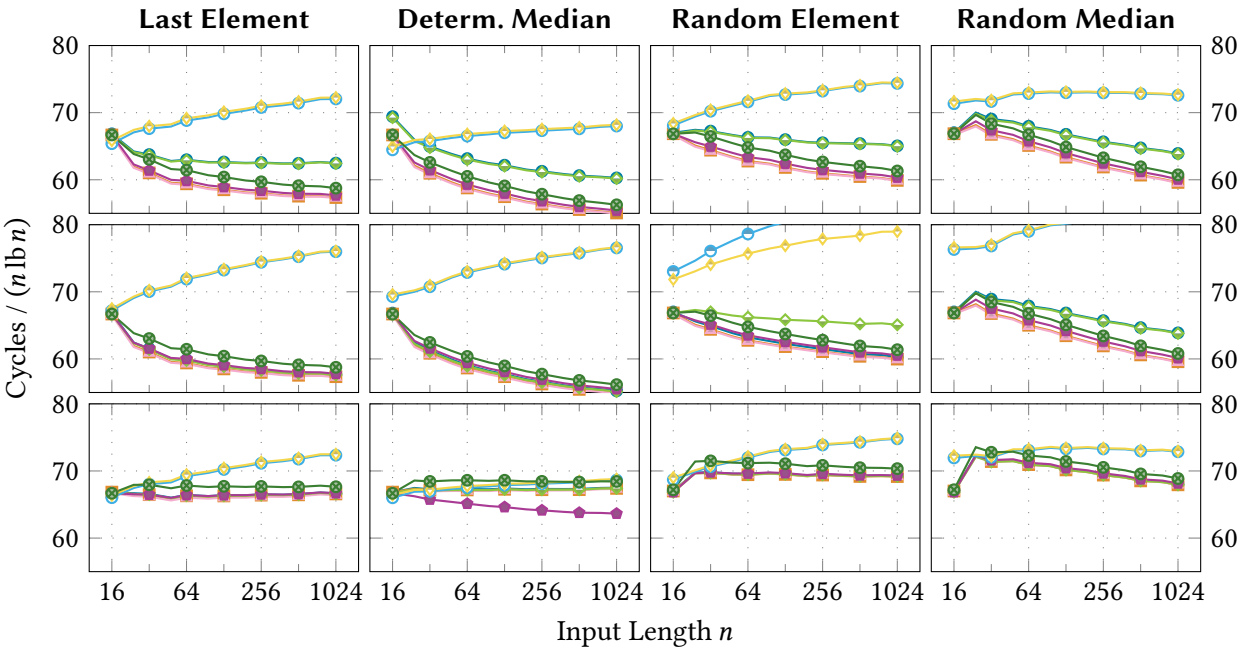


Figure A.12. *An extension to Fig. 3.8 with two different methods to selects pivots. The date size is 64 bits.*



(a) *Recursive Approach*



(b) *Iterative Approach*

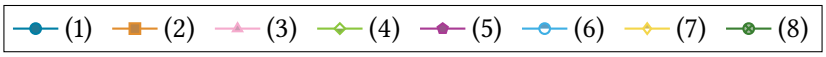
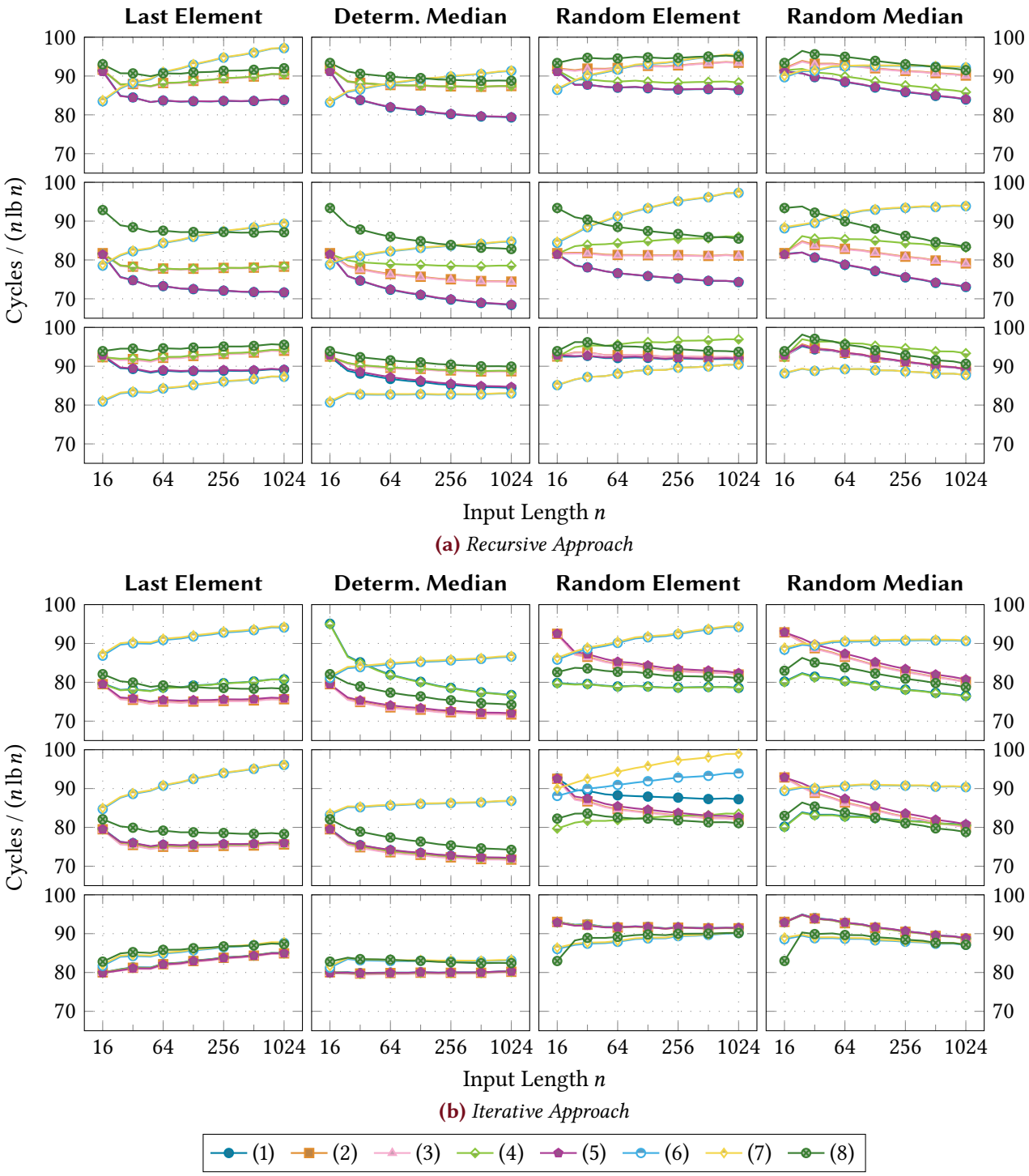


Figure A.13. *A repetition of Fig. 3.7. The date size is 32 bits.*



(a) *Recursive Approach*

(b) *Iterative Approach*

Figure A.14. *An extension to Fig. 3.7. The date size is 64 bits.*

A.5. MergeSort

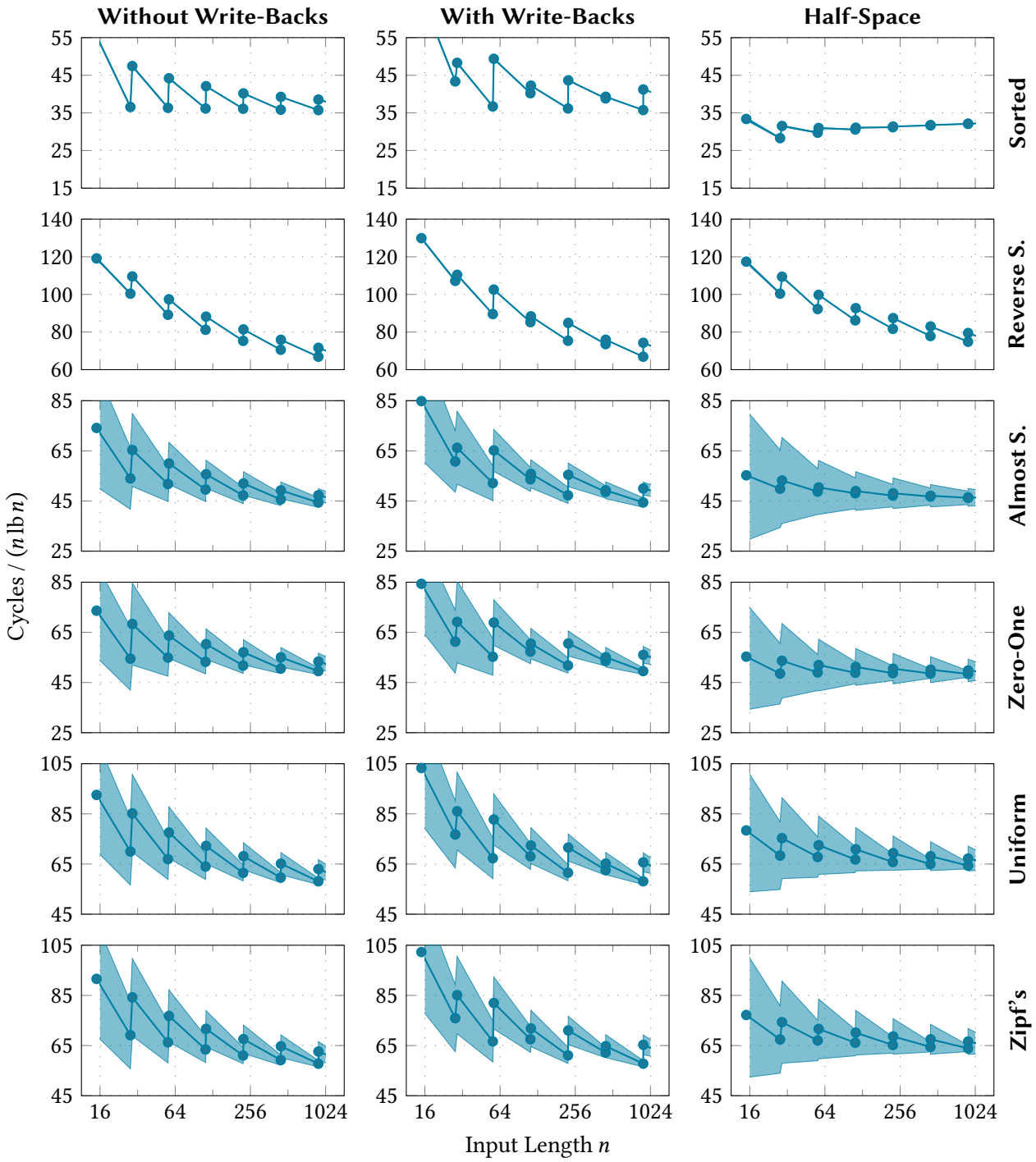


Figure A.15. *An extension to Fig. 3.10. The date size is 32 bits.*

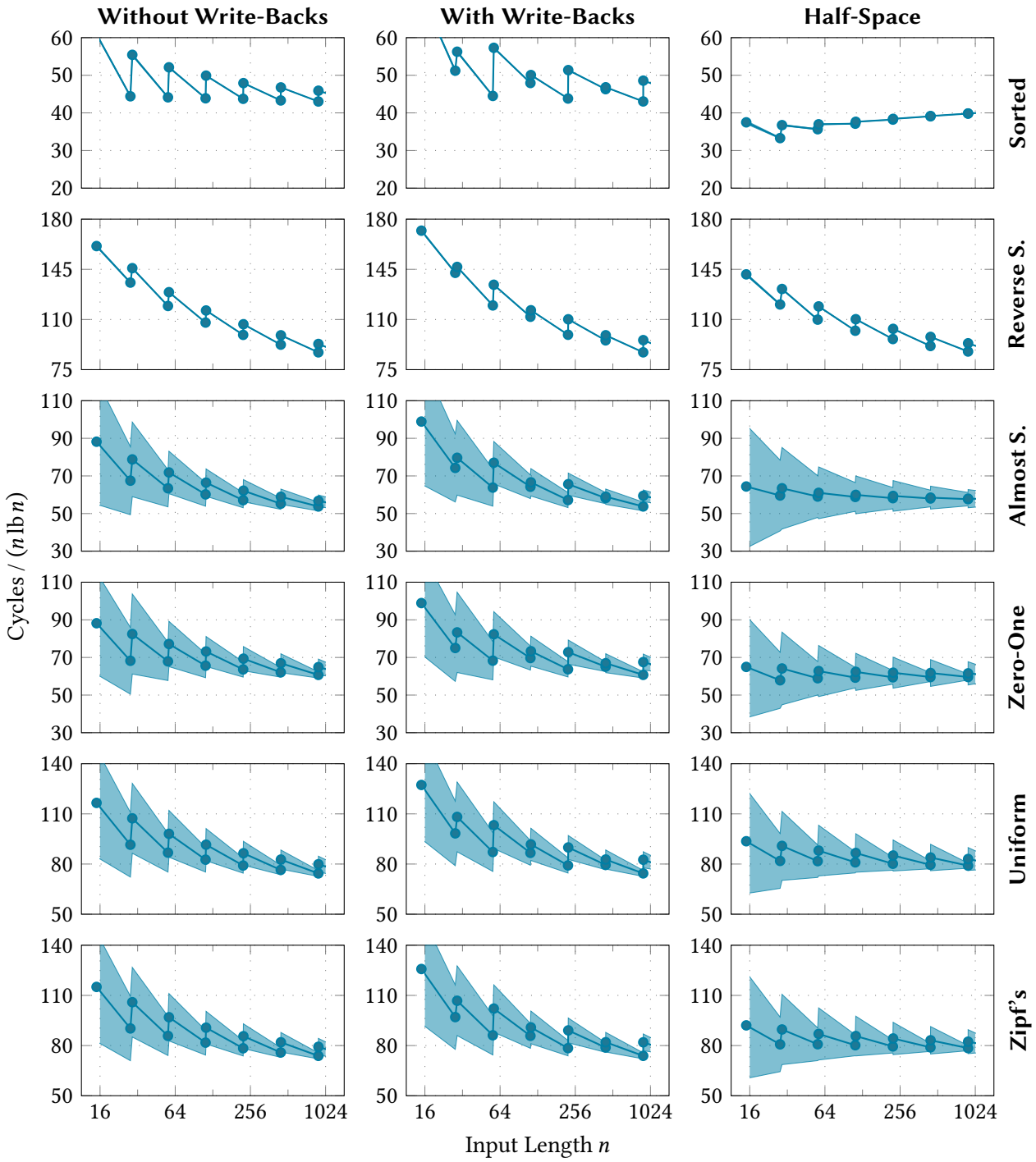


Figure A.16. *An extension to Fig. 3.10. The date size is 64 bits.*

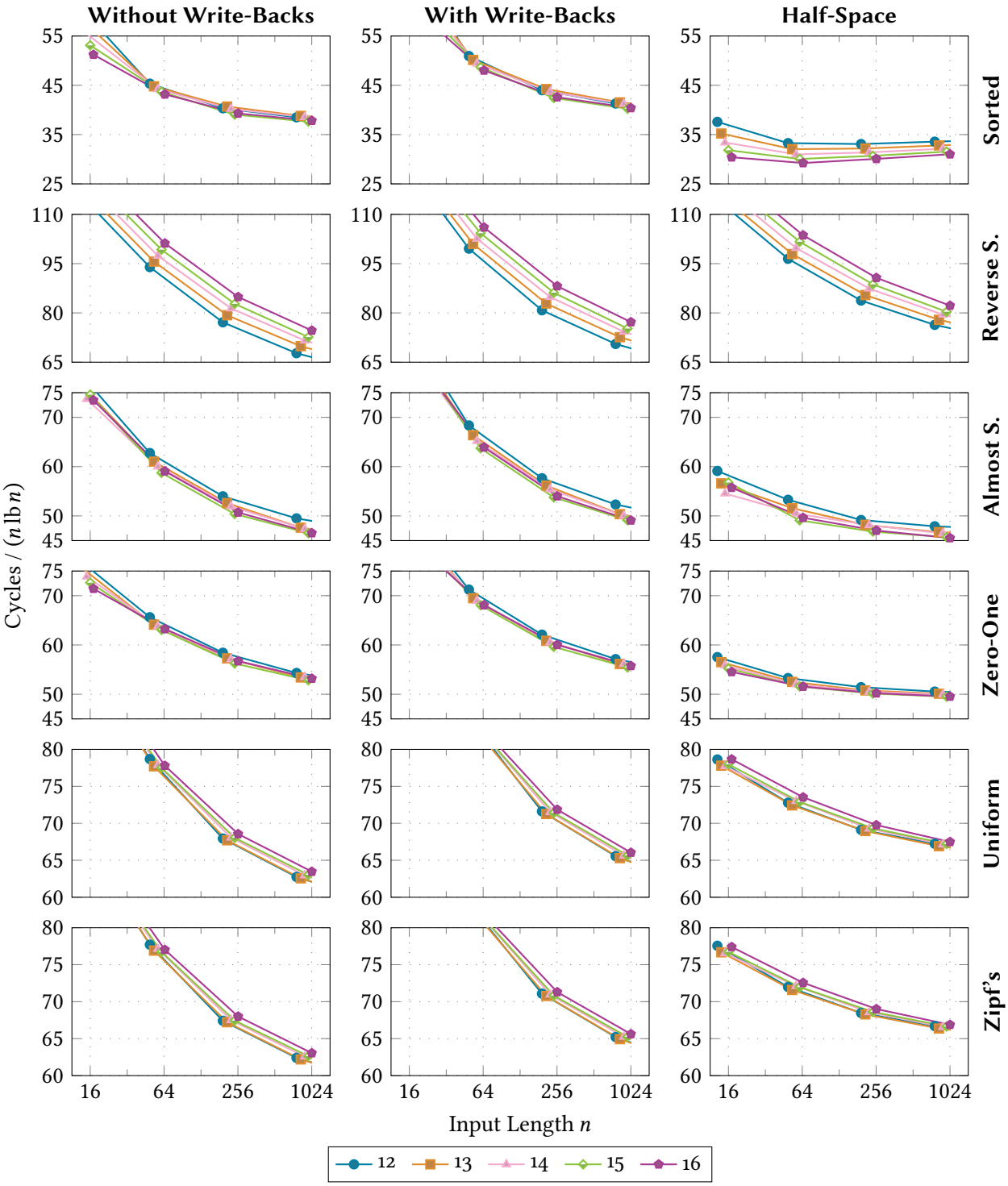


Figure A.17. *MergeSort with different starting run lengths. For starting run length ℓ, the input lengths $n = 4^i \cdot \ell + 1$ with $i = 0, \ldots, 4$ were benchmarked. The date size is 32 bits.*

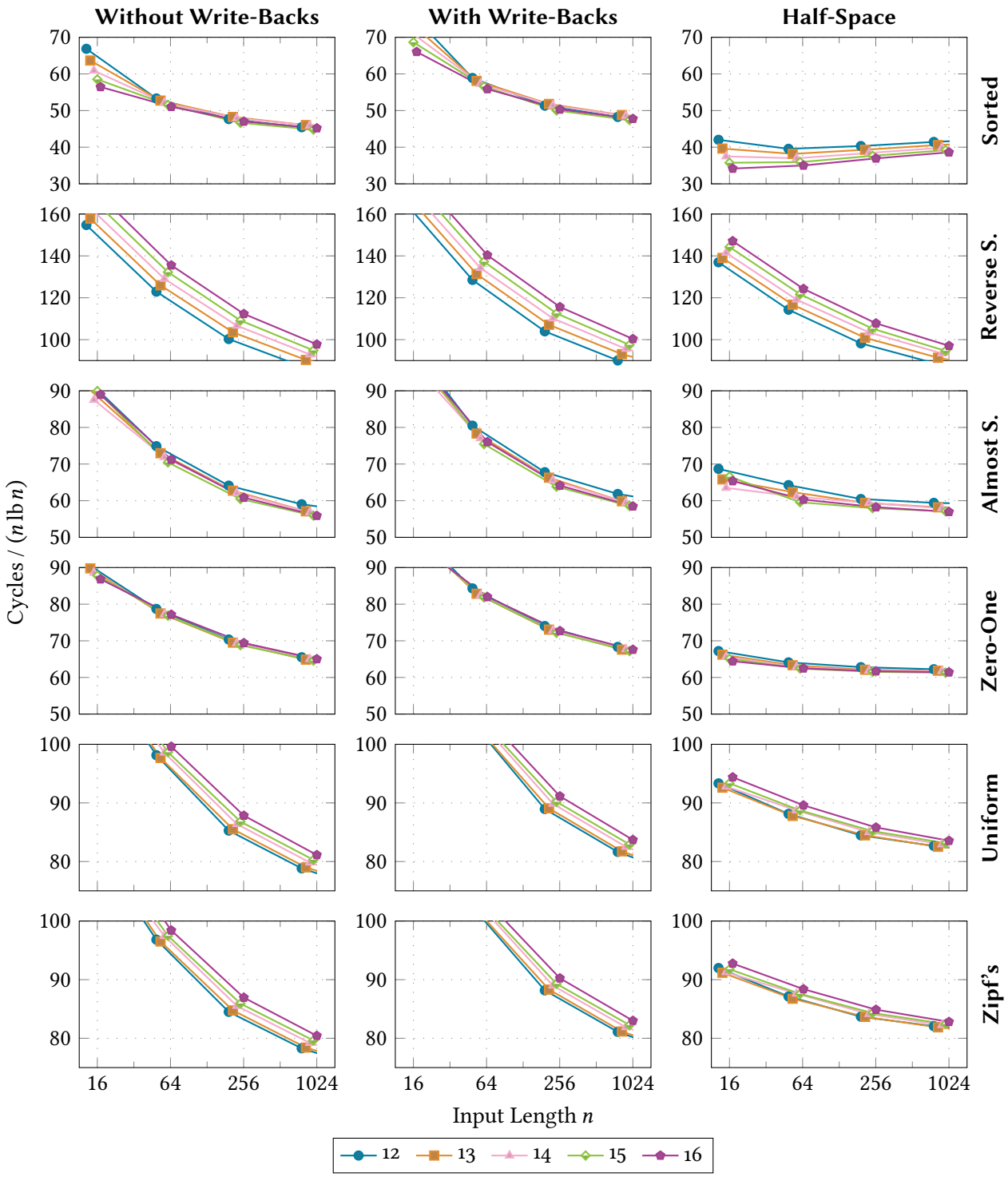


Figure A.18. *MergeSort with different starting run lengths. For starting run length ℓ, the input lengths $n = 4^i \cdot \ell + 1$ with $i = 0, \ldots, 4$ were benchmarked. The date size is 64 bits.*

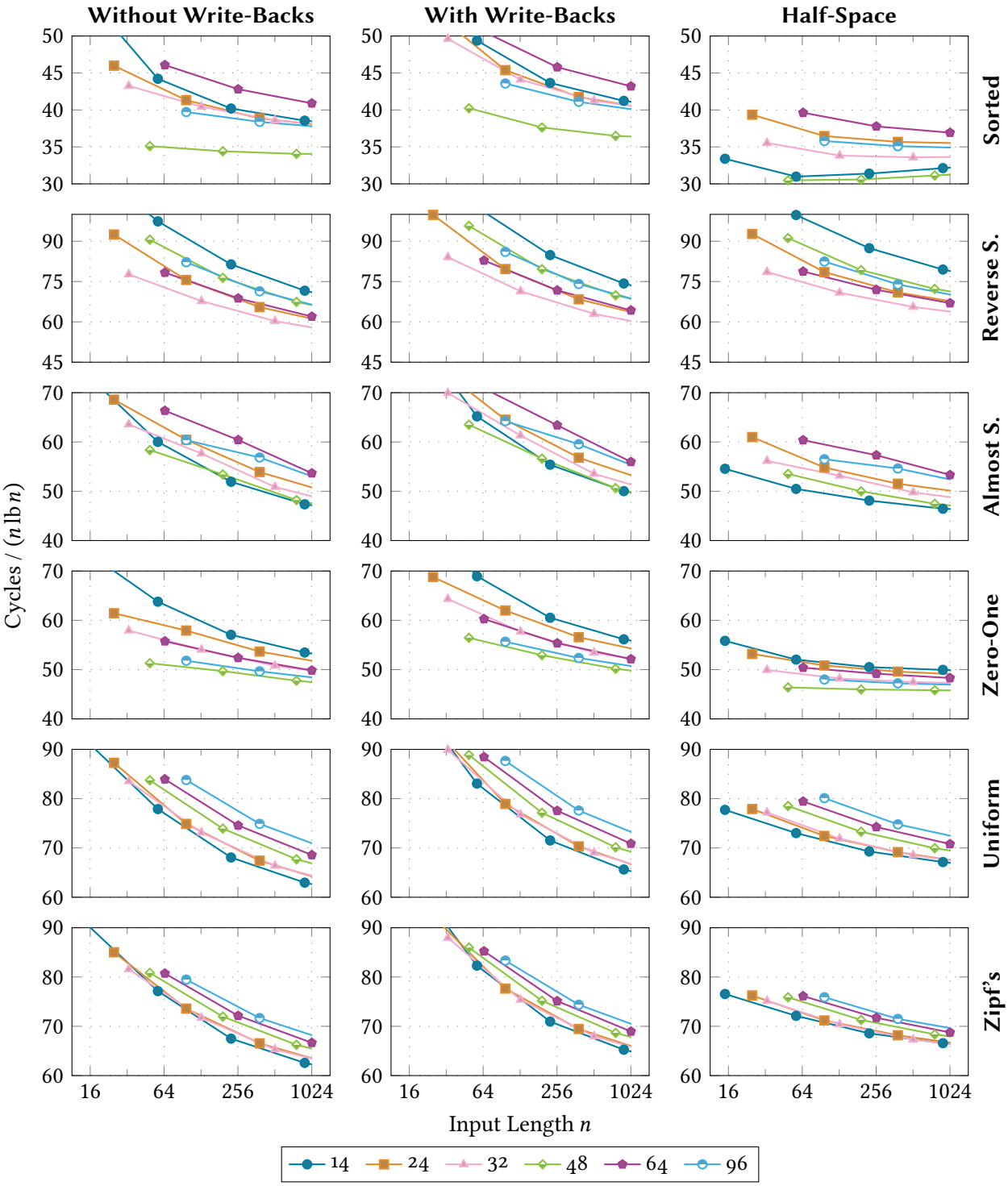


Figure A.19. *MergeSort with different starting run lengths. For starting run length ℓ, the input lengths $n = 4^i \cdot \ell + 1$ with $i = 0, \ldots, 3$ were benchmarked. The date size is 32 bits.*

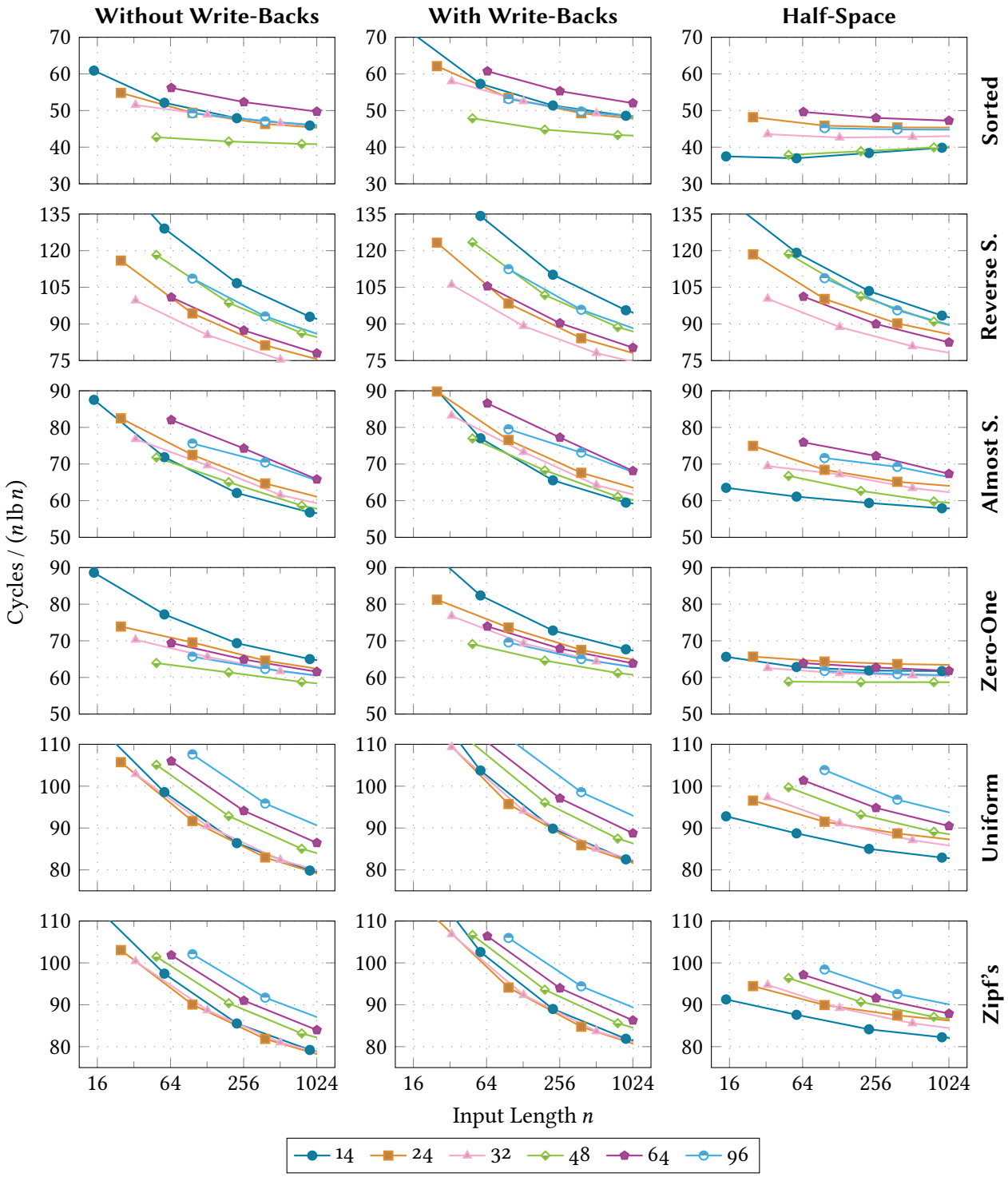


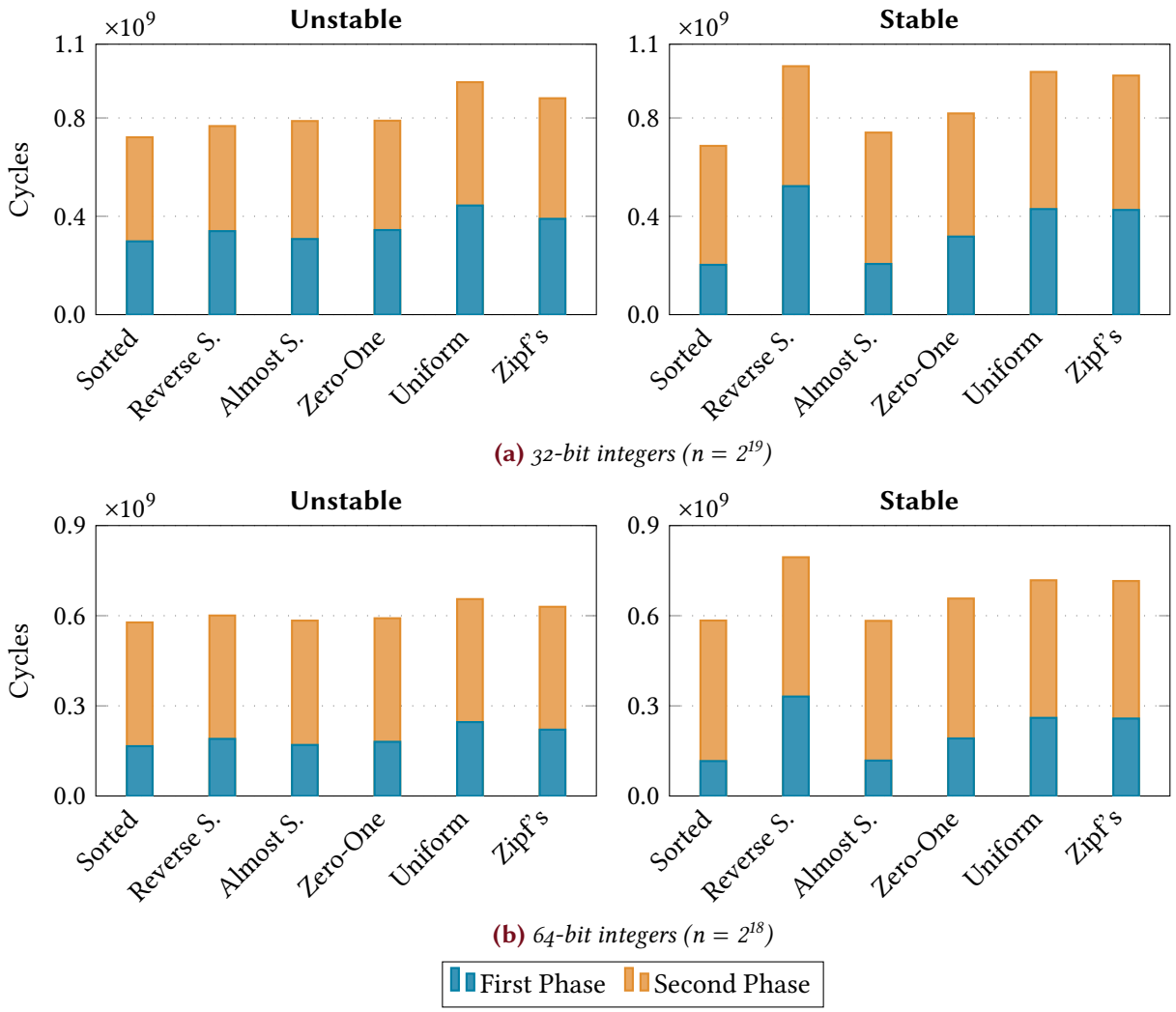
Figure A.20. *MergeSort with different starting run lengths. For starting run length ℓ, the input lengths $n = 4^i \cdot \ell + 1$ with $i = 0, \ldots, 3$ were benchmarked. The date size is 64 bits.*

Appendix B.

Further Measurements on Sorting in the MRAM

This appendix contains a comprehensive collection of measurements expanding the content of Chapter 4. Every measurement was repeated ten times with the sorting algorithms in their default configuration unless explicitly noted otherwise. By default, CACHE_SIZE is set to 1024, SEQREAD_CACHE_SIZE is set to 512, and WRAM QuickSort is used during the formation of starting runs.

B.1. Sequential MergeSort



(a) *32-bit integers ($n = 2^{19}$)*

(b) *64-bit integers ($n = 2^{18}$)*

Figure B.1. *An extension to Fig. 4.4 with both a stable and an unstable full-space MergeSort.*

Tasklets	CACHE	4 × SEQREAD_CACHE		
		512	1024	2048
11	256	745	721	708
	512	750	712	696
	1024	723	724	692
12	256	782	759	749
	512	787	750	735
	1024	763	762	732
16	256	994	969	957
	512	1002	964	945
	1024	971	981	945

(a) *32-bit integers ($n = 2^{19}$ per tasklet)*

Tasklets	CACHE	4 × SEQREAD_CACHE		
		512	1024	2048
11	256	544	513	498
	512	533	499	480
	1024	512	498	473
12	256	582	546	529
	512	569	530	510
	1024	544	527	502
16	256	770	714	683
	512	752	695	662
	1024	707	692	655

(b) *64-bit integers ($n = 2^{18}$ per tasklet)*

Table B.1. *An extension to Table 4.1. The input size per tasklet is fixed to* 2 MiB.

Acronyms

API	application programming interface	9
CPU	central processing unit	4
DDR4	Double Data Rate 4	4
DIMM	Dual In-Line Memory Module	4
DMA	direct memory access	5
DPU	DRAM processing unit	4
DRAM	Dynamic RAM	1
GPU	graphics processing unit	4
IRAM	Instruction RAM	6
MRAM	Main RAM	5
PIM	in-memory processing	1
PnM	near-memory processing	1
PuM	processing using memory	1
RAM	random-access memory	4
RISC	reduced instruction set computer	7
SDRAM	Synchronous DRAM	4
WRAM	Working RAM	5

Bibliography

[1] Michael Axtmann et al. *Engineering In-place (Shared-memory) Sorting Algorithms*. 3rd Feb. 2021. arXiv: 2009.13569v2 [cs.DC].

[2] Timo Bingmann, Andreas Eberle and Peter Sanders. 'Engineering Parallel String Sorting'. In: *Algorithmica* 77.2 (18th Sept. 2015). Ed. by Mohammad Taghi Hajiaghayi, pp. 235–285. ISSN: 1432-0541. DOI: 10.1007/s00453-015-0071-1.

[3] Avrim Blum and Manuel Blum. *Lecture 3. Probabilistic Analysis and Randomized Quicksort*. Carnegie Mellon University. 6th Sept. 2011. URL: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0906.pdf (visited on 26/07/2024).

[4] Avrim Blum and Manuel Blum. *Lecture 5. Comparison-based Lower Bounds for Sorting*. Carnegie Mellon University. 13th Sept. 2011. URL: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf (visited on 26/07/2024).

[5] Manuel Blum et al. 'Time bounds for selection'. In: *Journal of Computer and System Sciences* 7.4 (Aug. 1973). Ed. by Arnold Leonard Rosenberg, pp. 448–461. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(73)80033-9.

[6] Amirali Boroumand et al. 'Google Workloads for Consumer Devices. Mitigating Data Movement Bottlenecks'. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (24th–28th Mar. 2018). Ed. by Matthew Fluet. ASPLOS '18. Williamsburg: Association for Computing Machinery, 19th Mar. 2018, pp. 316–331. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173177.

[7] Sitian Chen et al. *UpDLRM. Accelerating Personalized Recommendation using Real-World PIM Architecture*. 20th June 2024. arXiv: 2406.13941 [cs.IR].

[8] Marcin Ciura. 'Best Increments for the Average Case of Shellsort'. In: *Fundamentals of Computation Theory*. Ed. by Rūsiņš Freivalds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2nd Aug. 2001, pp. 106–117. ISBN: 978-3-540-44669-9. DOI: 10.1007/3-540-44669-9_12.

[9] Michael Codish et al. 'Optimizing Sorting Algorithms by Using Sorting Networks'. In: *Formal Aspects of Computing* 29 (3 1st May 2017). Ed. by Moreno Falaschi and Augusto Sampaio, pp. 559–579. DOI: 10.1007/s00165-016-0401-3.

[10] Thomas H. Cormen et al. *Algorithmen. Eine Einführung*. German. Trans. English by Paul Molitor. 4th ed. Munich: Oldenbourg Wissenschaftsverlag, 17th Oct. 2013. 1319 pp. ISBN: 978-3-486-74861-1.

[11] Fabrice Devaux. 'The true Processing In Memory accelerator'. In: *Hot Chips 31 Symposium*. HC31 (Stanford University, 18th–20th Aug. 2019). New York City: Institute of Electrical and Electronics Engineers, 19th Aug. 2019. ISBN: 978-1-7281-2089-8. DOI: 10.1109/HOTCHIPS.2019.8875680.

[12] Stefan Edelkamp and Armin Weiß. *BlockQuicksort: How Branch Mispredictions don't affect Quicksort*. 2016. arXiv: 1604.06697v2 [cs.DS].

[13] Hannu Erkiö. 'The Worst Case Permutation for Median-of-Three Quicksort'. In: *The Computer Journal* 27.3 (Jan. 1984), pp. 276–277. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.3.276.

[14] Robert W Floyd. 'Algorithm 245. Treesort 3'. In: *Communications of the ACM* 7 (12 1st Dec. 1964). Ed. by Calvin Carl Gotlieb, p. 701. DOI: 10.1145/355588.365103.

[15] Lukas Geis. *Random Number Generation in the Pim-Architecture*. Research Project Report. Version 8c11f1f. Goethe University Frankfurt, 2024. 13 pp. URL: https://github.com/lukasgeis/upmem-rng/blob/main/report/report.pdf (visited on 19/05/2024).

[16] Christina Giannoula et al. 'Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures'. In: *SIGMETRICS Perform. Eval. Rev.* 50.1 (2nd June 2022), pp. 33–34. ISSN: 0163-5999. DOI: 10.1145/3547353.3522661.

[17] Kailash Gogineni et al. 'SwiftRL. Towards Efficient Reinforcement Learning on Real Processing-In-Memory Systems'. In: *International Symposium on Performance Analysis of Systems and Software* (5th–7th May 2024). ISPASS '24. Indianapolis: Institute of Electrical and Electronics Engineers, 16th June 2024, pp. 217–229. ISBN: 979-8-3503-7638-8. DOI: 10.1109/ISPASS61541.2024.00029.

[18] Juan Gómez-Luna et al. 'Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System'. In: *IEEE Access* 10 (10th May 2022), pp. 52565–52608. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3174101.

[19] John Leroy Hennessy and David Andrew Patterson. *Computer Architecture. A Quantitative Approach*. With a forew. by Luiz André Barroso. 5th ed. San Francisco: Morgan Kaufmann Publishers Inc., 29th Sept. 2011. 880 pp. ISBN: 978-0-12-383872-8.

[20] Charles Antony Richard Hoare. 'Quicksort'. In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10.

[21] Bongjoon Hyun et al. 'Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology'. In: *International Symposium on High-Performance Computer Architecture* (2nd–6th Mar. 2024). HPCA '24. Edinburgh: Institute of Electrical and Electronics Engineers, 2nd Apr. 2024, pp. 263–279. DOI: 10.1109/HPCA57654.2024.00029.

[22] Kanela Kaligosi and Peter Sanders. 'How Branch Mispredictions Affect Quicksort'. In: *Algorithms. 14th Annual European Symposium* (11th–13th Sept. 2012). Ed. by Yossi Azar and Thomas Erlebach. ESA '06. Zurich: Springer Berlin Heidelberg, pp. 780–791. ISBN: 978-3-540-38875-3. DOI: 10.1007/11841036.

[23] Jyrki Katajainen and Jesper Larsson Träff. 'A meticulous analysis of mergesort programs'. In: *Algorithms and Complexity*. Ed. by Giancarlo Bongiovanni, Daniel Pierre Bovet and Giuseppe Di Battista. Berlin, Heidelberg: Springer Berlin Heidelberg, 12th Mar. 1997, pp. 217–228. ISBN: 978-3-540-68323-0. DOI: 10.1007/3-540-62592-5_74.

[24] Liu Ke et al. 'RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing'. In: *47th Annual International Symposium on Computer Architecture* (30th May–3rd June 2020). Ed. by Lisa O'Conner. ISCA '20. Valencia: Institute of Electrical and Electronics Engineers, 13th July 2020, pp. 790–803. DOI: 10.1109/ISCA45697.2020.00070.

[25] Gokcen Kestor et al. 'Quantifying the energy cost of data movement in scientific applications'. In: *International Symposium on Workload Characterization* (22nd–24th Sept. 2013). IISWC '13. Portland: Institute of Electrical and Electronics Engineers, 9th Jan. 2014, pp. 56–65. ISBN: 978-1-4799-0555-3. DOI: 10.1109/IISWC.2013.6704670.

[26] Hans Werner Lang. *Algorithmen in Java*. German. Ed. by Margit Roth. 2nd ed. München: Oldenbourg Wissenschaftsverlag, 16th Dec. 2009. 384 pp. ISBN: 978-3-486-59340-2. DOI: 10.1524/9783486593402.

[27] Ying Wai Lee. *Empirically Improved Tokuda Gap Sequence in Shellsort*. 21st Dec. 2021. arXiv: 2112.11112v1 [cs.DS].

[28] Can Li et al. 'Analogue signal and image processing with large memristor crossbars'. In: *Nature Electronics* 1 (4th Dec. 2017), pp. 52–59. ISSN: 2520-1131. DOI: 10.1038/s41928-017-0002-z.

[29] m69 "snarky and unwelcoming" [sic!]. *Fastest way to sort 10 numbers? (numbers are 32 bit)*. Stack Overflow. 24th Aug. 2015. URL: https://stackoverflow.com/a/32173153 (visited on 11/08/2024).

[30] Hermann Maurer and Hans-Werner Six. *Datenstrukturen und Programmierverfahren*. German. Ed. by Heinrich Görtler. Leitfäden der angewandten Mathematik und Mechanik 25. Teubner Studienbücher Informatik. Stuttgart: B. G. Teubner, 1974. 222 pp. ISBN: 3-519-02328-8.

[31] David R. Musser. 'Introspective Sorting and Selection Algorithms'. In: *Software: Practice and Experience* 27 (8 8th Jan. 1999), pp. 983–993. DOI: 10.1002/(SICI)1097-024X(199708)27%3A8<983%3A%3AAID-SPE117>3.0.CO%3B2-%23.

[32] Onur Mutlu. *Memory-Centric Computing*. 13th Sept. 2023. arXiv: 2305.20000 [cs.AR].

[33] Joel Nider et al. 'A Case Study of Processing-in-Memory in off-the-Shelf Systems'. In: *USENIX Annual Technical Conference* (14th–16th July 2021). USENIX ATC '21. USENIX Association, July 2021, pp. 117–130. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/nider.

[34] Biagio Peccerillo et al. 'A survey on hardware accelerators. Taxonomy, trends, challenges, and perspectives'. In: *Journal of Systems Architecture* 129 (24th May 2022), p. 102561. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2022.102561.

[35] Orson Raphael Lennard Peters. *Pattern-defeating Quicksort*. 9th June 2021. arXiv: 2106.05123 [cs.DS].

[36] Tim Peters. *Timsort*. 2002. URL: https://bugs.python.org/file4451/timsort.txt (visited on 10/10/2024).

[37] Paul R. *Fastest sort of fixed length 6 int array*. Stack Overflow. 7th May 2010. URL: https://stackoverflow.com/a/2786959 (visited on 11/08/2024).

[38] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures. The Basic Toolbox*. 1st ed. Cham: Springer International Publishing AG, 11th Sept. 2019. 509 pp. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0.

[39] Donald Lewis Shell. 'A high-speed sorting procedure'. In: *Communications of the ACM* 2 (7 1st July 1959). Ed. by Alan J. Perlis, pp. 30–32. DOI: 10.1145/368370.368387.

[40] Gagandeep Singh et al. 'NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling'. In: *30th International Conference on Field-Programmable Logic and Applications* (31st Aug.–4th Sept. 2020). FPL '20. Gothenburg: Institute of Electrical and Electronics Engineers, 13th Oct. 2020, pp. 9–17. ISBN: 978-1-7281-9902-3. DOI: 10.1109/FPL50879.2020.00014.

[41] Malte Skarupke. *I Wrote a Faster Sorting Algorithm*. 27th Dec. 2016. URL: https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/ (visited on 10/10/2024).

[42] Oscar Skean, Richard Ehrenborg and Jerzy W. Jaromczyk. *Optimization Perspectives on Shellsort*. 1st Jan. 2023. arXiv: 2301.00316v1 [cs.DS].

[43] UPMEM. 'UPMEM Processing In-Memory. Ultra-efficient acceleration for data-intensive applications'. PIM Technology Paper. Grenoble, Sept. 2021.

[44] *UPMEM DPU SDK Documentation*. Version 2024.1.0. UPMEM. Grenoble, 2024. URL: https://sdk.upmem.com/2024.1.0.

[45] Ingo Wegener. 'BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)'. In: *Theoretical Computer Science* 118 (1 13th Sept. 1993), pp. 81–98. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90364-Y.

[46] Sebastian Wild and Markus E. Nebel. 'Average Case Analysis of Java 7's Dual Pivot Quicksort'. In: *Algorithms. 20th Annual European Symposium* (10th–12th Sept. 2012). Ed. by Leah Epstein and Paolo Ferragina. ESA '12. Ljubljana: Springer Berlin Heidelberg, pp. 825–836. ISBN: 978-3-642-33090-2. DOI: 10.1007/978-3-642-33090-2_71.

[47] John William Joseph Williams. 'Algorithm 232. Heapsort'. In: *Communications of the ACM* 7 (6 1st June 1964). Ed. by George Elmer Forsythe, pp. 347–348. DOI: 10.1145/512274.512284.

[48] Niklaus Wirth. *Algorithmen und Datenstrukturen*. German. Ed. by Heinrich Görtler. Leitfäden der angewandten Mathematik und Mechanik 32. Teubner Studienbücher Informatik. Stuttgart: B. G. Teubner, 1975. 376 pp. ISBN: 3-519-02330-X.

Erklärung zur Abschlussarbeit

gemäß § 34, Abs. 16 der Ordnung für den Masterstudiengang Informatik vom 17. Juni 2019

Hiermit erkläre ich

(Nachname, Vorname)

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass die von mir eingereichten schriftlichen gebundenen Versionen meiner Masterarbeit mit der eingereichten elektronischen Version meiner Masterarbeit übereinstimmen.

Frankfurt am Main, den

Unterschrift der/des Studierenden