Design and Analysis of Cache-Conscious Programs

M.Sc. Thesis

Maz Spork

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
E-mail: halgrim@diku.dk

February 17, 1999

Summary

This thesis is presented in partial fulfillment of the Danish Kandidatgrad i Datalogi (*candidatus scientiarium in Computer Science*) at the University of Copenhagen, and was written during the autumn and winter of 1998 under the supervision of Jyrki Katajainen.

The cache behaviour of programs is studied. It is common knowledge that the cache memory subsystem induces an overhead on the execution time of a program. Most programs, however, are not designed with the details of the memory subsystem in mind, because it requires the introduction of a new layer of hardware constants in the analyses. As the penalty of a cache miss has risen by about a factor two every three years, the cache behaviour of programs has become an important factor. The cache behaviour of a program is closely tied to its memory reference pattern. Memory references that are proximate in time or space to previous references exhibit referential locality and induce a small overhead compared to non-local, or random, memory references.

A rationale is presented for the latency phenomenon and a machine model which allows meticulous analysis of programs, together with a cost model for programs, which includes both instruction cost and memory reference cost under spatial locality. In two case studies, heap construction and mergesort programs, several programs are implemented and their performance under this model is analysed. It is shown analytically that the number of cache misses varies, and, specifically, that programs with superior asymptotical complexity with regard to key comparisons or instruction count, turn out inferior with respect to the cache miss complexity.

The work is highly based on experiments. Almost every program discussed is implemented and the performance is measured on different architectures. In addition, the cache behaviour is measured for most programs using a cache profiler. Through experiments, the constants of the cost model are further assigned concrete values, allowing accurate predictions of the running times of the programs presented.

Last, a methodology for designing programs without the expensive non-local

memory references is presented, based on the results of the experiments, and exemplified through a case study on the problem of finding connected components in directed graphs. The programs are suboptimal compared to known programs with respect to key comparisons or instruction count but involve only sequential access to memory. The methodology will not produce faster programs for current architectures, but if miss penalties continue to grow, it may be useful in a few years.

Key words. Cost models, sequential-access memory, reference patterns, latency, memory hierarchies, locality, performance tuning, meticulous analysis of programs, cache behaviour, profiling.

Acknowledgements. Thanks must go to Finn Schiermer Andersen, J. P. Secher and Martin Lillholm, my fellow students, for their critical proofreading and constructive corrections. Also thanks to Jesper Bojesen for his work on heaps. My supervisor, Jyrki Katajainen, deserves much credit for persistence and for long hours of discussion. Finally, a big thank you to my wife Anja for, among other things, staying sane though the past few months.

Frederiksberg, February 1999

Contents

Chapter 1

Introduction

This thesis is about the behaviour of programs in the presence of latency induced by the top level of the memory hierarchy, called a *cache*. The memory hierarchy is designed to be transparent to the programmer. Few fundamental algorithms are designed with the memory hierarchy in mind, though some perform very well in practice. Worse, almost no new algorithms are designed with awareness of the effects from the memory hierarchy, and some perform poorly in practice. The reason for this is that the memory hierarchy introduces a new layer of parameters which makes latency-conscious programming more complex. In this thesis, I present the phenomenon and a model for predicting the performance of the memory hierarchy, and then propose a methodology for designing latency-conscious programs.

1.1 Algorithms and memory latency

Cache miss penalties have risen by a factor of two every year [3, 8, 10, 31]. The effect of latency times of a factor of 100 or more implies that predictions of execution times using traditional unit-cost models unaware of these effects may give imprecise or even wrong results. The penalty for a page fault in virtual memory is much larger, but has little concern as long as the working set fits in main memory.

Until recently, algorithm designers have neglected the effects of caching, probably because no widely-accepted model incorporating these exists. A cache-conscious design of data structures may lead to a performance gain, but for programs with nonlocal reference patterns and behaviour dependent on input data, the reference pattern is the primary influential factor on the cache performance. Because of the transparency of the memory hierarchy, it is not easy

to include as an abstraction layer for implementation. Caches were introduced to speed up memory accesses based on empirical studies of typical programs' behaviour from a hardware perspective, but caches are not understood well by algorithm designers. Typically, memory access latency is commonly categorized as "random noise" from the system.

The miss rate of a program may cause latency representing a considerable fraction of its execution time. Latency may even dominate. Therefore, good performance requires good memory hierarchy performance in the sense that the miss count for a given program is minimized.

1.2 Overview of the thesis

Unless otherwise stated, I use a system programming language with access to effective addresses of program and data objects such as C [17] or C++ [34] and their abstract environments in examples and analyses. Abstract algorithms are presented in some examples.

This work is about experimental algorithmics, and the methodology is therefore based on experiments. All important theory is experimentally evaluated. All experiments are made by the author; when I refer to other experimental work, it is not in direct comparison. Preprocessing input data to some required data structure is considered a part of the algorithm, that is, the time for reading the input stream to some data structure is measured as part of a program's execution.

The aim is the construction of an analytical model for predicting the behaviour of the memory hierarchy, and attempts are made to discriminate the "random noise" from the execution of programs. This noise is considered to partly, but heavily, depend on the pattern of memory references of a program. With the knowledge of *when* memory references happen in the program and *where* the references are made in the different levels of the memory hierarchy, an analytical method is proposed. Programs with sequential access patterns are studied, and a methodology for designing such programs is presented. Spatial locality is considered in more detail than other kinds of locality.

The thesis may be divided in three main parts.

1. Chapters 2 and 3 present the sources of latency in modern computer architectures, defines the metrics and properties of one such source, the *cache*, and present a program analysis tool for measuring cache behaviour.

2. In Chapter 4, a number of ad-hoc techniques usually employed for tun-

ing program performance with respect to data locality in caches is discussed, followed by the presentation of an analytical model in Chapter 5. Through two case studies in Chapters 6 and 7, the model is employed and evaluated.

3. Chapter 8, presents a methodology for designing programs with sequential reference patterns.

The characteristics of the two reference machines used in the experiments are:

AXP: A 366MHz Digital Alpha AXP 21164 with 32Mb 60ns DRAM, 8kb 4-way set-associative on-chip data cache and 8kb direct-mapped on-chip instruction cache, 256kb 10ns second-level direct-mapped cache, 256Mb swap space, about 10ms worst-case access time to the disk (this machine is used if nothing else is stated).

AMD: A 266MHz Analog Memory Devices K6 with 64Mb DRAM, 32kb two-way set-associative on-chip data cache and 8kb direct-mapped on-chip instruction cache, 256kb 10ns second-level direct-mapped cache, 64Mb swap space, about 10ms worst-case access time to the disk.

In the experiments involving measurements of execution times, the Unix timer with a granularity of 1μs is used, and all experiments are run under the Linux operating system version 2.0 [14] in single-user mode using GCC version 2.7.2.3. Linux' page-out policy uses a modified version of a standard clock algorithm (see, e.g., [29]) that assigns to each page an age, which is adjusted on each reference. The age valuing allows Linux' pager to select pages for page-out on a least frequently used policy.

1.3 Critique

Designing for locality has drawbacks. The abstractions may be more complex because the parameters of the memory hierarchy are introduced in implementations, and the use of such parameters may itself be machine-dependent and lead to less portable programs. This is evident throughout the work.

A second, more general, critical assessment is that designing algorithms with architectural influence is very experimental and is concerned with noisy environments whose performance characteristics are technology-driven and prone to changes in the future.

Two arguments may be raised to justify the interest in such experimental algorithmics: First, the latency does not go away. Accessing data, whether in

a tightly-coupled system or through a network, always implies a transaction, or communication, and will inevitably induce latency. Second, the drive in technology is based not only on architectural trends. As memory subsystems were designed from the characteristics of programs (because it is important to run existing software efficiently), so may programs be designed to utilize the memory as well as possible. Future trends in architecture may very well focus on algorithms and programs which themselves were aware of hardware parameters.

For these reasons, I will attempt to derive the most general results, and not focus on specific implementations or architectures in the theoretical parts of the thesis.

Chapter 2

Memory latency

Latency induced by various parts of the computer architecture causes a constant overhead on the execution of a program. In this chapter, some representatives of memory hierarchies in typical architectures are briefly presented to state where and how the latency originates. While considered “random noise” from a software viewpoint, the latency is a consequence of the instruction stream from a given program. A prerequisite for a quantification of latency is therefore the determination of the metrics of the architecture. The focus is on the different levels of the memory hierarchy because these dominate the latency.

2.1 Sources of latency

Most devices, mechanic as well as electronic, require a grace period before an answer or acknowledgement can be produced and returned to the unit issuing the commands. In a computer system with a central processing unit which serially issues dependent instructions, care must be taken in both software and hardware to eliminate as much idle time as possible.

For slow, loosely-coupled devices (with latency times in the milliseconds or slower), multiprogramming facilities in threaded kernels or time-sharing operating systems offer multiplexing of the devices by monitoring the states of active threads of execution, suspending those waiting for some device to complete a task. These tasks include positioning the heads of a disk, dispatching a packet on a network, and reading a character from a terminal. The combined latency is not reduced by this scheme, but the waiting is allowed to overlap in time, reducing the waiting time for a single thread of execution and thus the average waiting time.

Tighter-coupled devices may have latency times below what make context switches feasible, or may have integral functions that make them impossible. In particular, the instruction feed from memory, and the reading and writing of data items, is a device that is normally inherently serial.

The memory hierarchy consists of several random-access storage levels, with the top level closest to the processor, and level $i+1$ being larger and slower than level i. In each level, memory is divided into disjoint, consecutive *blocks* of uniform size abiding the principle of inclusion: If a block is present in level i, it is also present in level $i+1$. The processor may only access data in blocks at the top level, meaning that *transactions* to and from lower levels are necessary if the working set of a program or set of programs is larger than fits into a particular level. It is these transactions which induce latency because the processor may *stall* in waiting for their completion.

The typical memory hierarchy of a modern computer consists of an on-chip cache ($\approx 2^{13}$ bytes, $\approx$ 1–2 cycles access time), an off-chip *level two* cache ($\approx 2^{18}$ bytes, $\approx$ 4–8 cycles access time), a main memory ($\approx 2^{26}$ bytes, $\approx$ 25–100 cycles access time) and a background disk with gigabyte capacity and access times in the milliseconds.

Latency and "random noise" occurs in other parts of the system than the memory hierarchy:

1. Architectures that issue instructions into a *pipeline* to overlap the different execution stages of the instructions frequently stall for one or more cycles because a value needed by instruction j has not yet been computed by instruction $i < j$, or because a branch instruction drains the pipeline. Such situations are called *hazards* [26] and arise also in multiple-issue architectures (e.g., superscalar processors).

2. A computation may require more registers than physically available, requiring the compiler to *spill* subsets of the working set to memory. Fortunately, the destination of register spilling is typically inside the activation record on the stack, which is usually cached.

3. Time-sharing operating systems has an effect on the execution time. Apart from the wall-clock delays that may be caused by time-sharing, the context switches themselves take a small amount of time and may result in partial drainage of the cache and translation buffers[1], giving a higher miss rate than expected. Context switches in relation to cache performance is studied in [24].

[1] Context switching, or merely the handling of interrupts, also affects *branch prediction* because the instruction stream is broken.

4. Garbage collection produces delays in the execution. Again the cache is effected because this activity involves transfers to and from memory and may pollute the contents of the cache and translation buffer. Garbage collection in relation to cache performance is studied in [23] and [9] presents a performance evaluation of memory allocation with respect to caches.

5. Other devices than the processor may compete for access to memory; contention on the memory bus from network, graphic or other real-time devices through direct-memory access is typical in modern architectures.

In this thesis, the focus will be on latency from data references only, that is, the effects from the parts of the system mentioned in the above list are still considered "random noise". The rationale behind this choice is (1) that the delays from the memory hierarchy are considerably higher than from the other sources, and (2) that the data references follow a pattern which may be predicted by program analysis. Given the hardware constants of a particular architecture, the delays from the data references in a program can be quantified.

2.2 Design patterns

Understanding the behaviour of the architecture, and how architectural parameters influence the efficiency of a particular program, is necessary for designing programs with an optimal utilization of the memory hierarchy. However, the architecture is itself designed partly through experiments on real programs. *Forwarding* in pipelines is e.g., a technique for reducing the number of stall cycles for an instruction consuming a value that has been produced by an earlier instruction in the pipeline, see [26]. The reason is that software is much more expensive than hardware, meaning that architectural refinements which contribute to the faster execution of existing software is subject to intense engineering.

Therefore, the design of software and hardware interacts, and the problems of latency are not solved in one domain only. A cautious suggestion however is that the interaction of, e.g., cache-conscious software may lead to the emergence of memory hierarchies that are able to deliver data from any depth with close to zero latency for some programs.

Chapter 3

Caches

A cache is a place for storing copies of data that has recently been accessed. While the term covers a wide range of techniques and applications, the focus here is the hardware-controlled uniprocessor data cache memory in typical modern computers situated between the memory ports of the processor and the main memory. Caches emerged in the 1960s when the bandwidth requirements to memory rose as a consequence of architectural improvements such as pipelining, vector processing, and faster cycle times. The fact that a cache stores only a fraction of the data present in main memory means that it has a mechanism for selecting positions for and replacing blocks of data. This mechanism is designed to be transparent, i.e., hidden, from the programmer.

3.1 Architectural constants

A cache memory is defined by a small number of static, or architecturally constant, parameters. A cache memory is defined by the *capacity* M (in words), *block size* (or *line size*) B (also in words), and *associativity* a, $1 \leq a \leq b$, where $b = M/B$ is the number of cache blocks. The number of *cache sets* s is $M/(aB)$. M, B and a are all powers of two.

3.1.1 Blocks, tags, sets, and addresses

Each block consists of B consecutively addressed words as well as partial information about the corresponding address v of the block in virtual memory. Given M, B, and a, a block of B words may be placed in the cache in one of a distinct blocks pertaining to the same cache set. The cache set for a block is obtained as a function of v. For a given address v, the *offset* within a given

cache block, v **mod** B, identifies the final position of the block in the cache.

A block situated at virtual address v is placed in a cache set using a hash function such as $(v$ **div** $B)$ **mod** s. As the lower $\log_2 B$ bits of the virtual address v are used as an offset within a cache block, they are not used in set selection. Each cache block is *tagged* with the part of v that cannot be inferred by the set selection formula, i.e., v **div** (Bs).

The selection of a cache set, the offset within a block in that set, and the tag field of the block, are all bit fields of v. Assuming that $\|$ means the concatenation of bit fields, a virtual address $v = v_t \| v_s \| v_f$, where v_t is the tag field, v_s the cache set, and v_f the offset within the block. From the above definitions, the width of v_f in bits, $|v_f|$, is $\log_2 B$ and $|v_s| = \log_2 s$.

3.1.2 Associativity

For caches with $a = 1$ (*direct-mapped caches*), there is only one possible block to choose with a cache set, while caches with $a > 1$ (*associative caches*), there is a choice between a distinct blocks.

Caches with $1 < a < b$ (a-*way set associative caches*) present a possible values from which to choose placement within the cache set. Caches with $a = b$ (*fully associative caches*) have only one possible value for v_s (the block may be placed anywhere in the cache), and thus $v = v_t \| v_f$.

The selection of a block in associative caches is not determined by the cache metrics and block address, rather, it is chosen depending on a dynamic *replacement strategy*. The simplest strategy is the *random choice*, requiring no additional state information at all, while fully time-stamping (with some serial number) a block on each reference facilitates a choice of the *least recently used* block within a set at the expense of a complex selection procedure. A common hybrid of these, *not recently used*, uses a single bit of state information in each block to mark one block in each set most recently used, and makes a random choice between all other blocks in that set, thus approximating least-recently used. Other techniques for approximating least-recently used for small values of a also exist.

Set-associative caches have a lower hit time (time to access data present in the cache) than direct-mapped caches. The selection of a block in a cache set requires an associative look-up in a tag memory inside the cache. This look-up can be done in parallel, making the access time very close to that of a direct-mapped cache. However, this adds significant cost measured in transistor count and die area. Detailed comparisons of the tradeoffs in set associativity and direct mapping is found in [11] and [30]

3.1.3 Hits and misses

A memory reference with address $v = v_t||v_s||v_f$ to the cache is a *hit* if there exists a (valid) block with tag v_t within the cache set v_s. A hit implies presence of data in the cache, requires therefore no external transactions, and is neutral in terms of latency. If no such block exists, the reference is a *miss* and requires one or more memory transactions. Misses occur primarily because the capacity of the cache is smaller than the capacity of the memory at the next level. The *miss rate* r of the run of a program (and its input) is the fraction of the total number of memory references that result in cache misses.

Three types of cache misses can be distinguished, *compulsory* misses; *capacity* misses; and *conflict* (or *interference*) misses [11, 12, 31]. Compulsory misses are the mandatory cache misses made by the program, which is no more than $\frac{1}{B}$'th of the size of input. A subset of these, b misses, are further distinguished as *cold-start* misses to invalid cache blocks. Capacity misses occur when the size of the cache is not sufficiently large to store these elements. Conflict misses arise when two pieces of data map to the same location in the cache. A fully-associative cache will not exhibit conflict misses while a direct-mapped cache is considered more sensitive to conflicts [20]. Fully-associative caches are not typical in production caches due to their high transistor count and reduced efficiency [10].

Generally, capacity misses may be reduced by enlarging the capacity of the cache, while conflict misses may be reduced by increasing associativity. Capacity misses may also be reduced by increasing the block size. These reduction approaches should be viewed in combination as e.g., a larger block size increases the miss penalty but also reduces the number of blocks in a cache and cache set, thus reducing the effect of higher associativity.

In case of a cache miss, the selected block may already contain valid data (either a capacity or conflict miss) and may have been modified since it was read (a *dirty* miss). Dirty cache misses require the original block to be transferred back to the next memory level. Each cache block thus carries a bit of state information (the *dirty bit*) indicating whether the block has been modified since read. It is also necessary to know if a cache block is at all valid, which is also determined by a bit of state information (the *valid bit*).

A cache miss requires at least one transaction and induces an architecturally dependent latency (measured in clock cycles) in the data stream meaning that subsequent operations that need the data must to wait, thereby stalling the execution for a period of time. The choice of replacement strategy has an influence on the performance of the cache in terms the miss rate, but not on the miss penalty.

3.1.4 Other cache properties

A few additional assumptions must be made to rationalize the clean, abstract model of a cache.

It is assumed that both clean and dirty misses cause the same latency because it is common for the memory subsystem to employ a queueing scheme for write transactions, making it possible to perform the read transaction before the write transaction. Caches without a write queue have a higher penalty on dirty misses because the read transaction is deferred until the write transaction has completed. With this assumption, therefore, a cache miss is considered having the cost of one transaction to next-level memory even though two transactions may actually occur (a cache may also employ a *write-through* scheme which means that cache lines are never dirty, and that the cache truly adheres to the principle of inclusion, i.e., that the contents of the cache is a subset of the contents of next-level memory).

It is assumed that the cache is *split* such that separate caches exist for instructions and data. This means that no interference exists between blocks of data and program code. The data and instruction caches are typically metrically different. The assumption allows us to disregard any cache effects from the instruction stream.

3.2 Assessment of cache performance

Caches involves the processor and main memory, today typically dynamic memory using a transistor coupled with a capacitor to store one bit. These memories may lose their information when accessed, and need periodical refreshing. Processor speeds have been growing at a rate of about 50% every year for the past two decades, now several thousand times faster, while dynamic memory speed has grown relatively only a fraction of this, today some 20 times faster.

Caches are designed to reduce latency on memory accesses, using faster and more expensive, static memory sandwiched between the processor and the main memory. As Table 3.1 indicates, the memory access time dropped below the processor cycle time about 1986 for typical workstations (not supercomputers or vector processors).

The performance of caches revolves around the miss rate r (the fraction of all memory references resulting in cache misses), the number of cache misses for a given program and its input, and the miss penalty $\tau^\star$. Intuitively, the performance is inversely proportional to the product of r and $\tau^\star$.

Year	clock speed	DRAM speed
1982	4 MHz	250ns
1986	16 MHz	150ns
1990	50 MHz	100ns
1994	200 MHz	60ns
1998	600 MHz	< 50ns

Table 3.1: Evolution in CPU and memory speed (author's recollection).

Let i be the average number of memory references per instruction (a machine-independent measure), and τ the average time (in cycles) spent executing each instruction, not counting latency. Using a measure for the miss rate r as defined, the expected cost of each instruction becomes $ir\tau^\star + \tau$ clock cycles.

Example 1 (*Latency expectation)*
Assume $\tau = 2$, i.e., two cycles on average are spent executing an instruction (not counting latency), and a penalty of $\tau^\star = 75$ clock cycles on a cache miss. Running a program with a bandwidth requirement of $i = 1.5$ references/instruction and a miss rate of 0.05, the latency induced is $1.5 \times 75 \times 0.05 = 5.625$ cycles/instruction, totalling close to 7.625 cycles/instruction in total adding τ. With a miss rate of 0.02, the total execution average is 4.25 cycles/instruction.

In contrast, a perfect cache ($r = 0$) induces no latency and executes each instruction in $\tau = 2$ cycles, while removing the cache ($r = 1$) implies an average of 114.5 cycles/instruction. This example illustrates the scale of the constant factors and the importance of the miss rate. For most programs, the miss rate is less than 0.1, though some programs have miss rates close to 1, resembling a cache miss on every operation.

3.2.1 Temporal and spatial locality

If the capacity of the cache is significantly smaller than the working set of the program, that is if M is much smaller than the input size, a high proportion of capacity misses will occur, and a significant proportion of the execution time of a program is spent moving data between two (or more) levels of the memory hierarchy. This may result in *thrashing* of the cache, reducing the efficiency of the memory subsystem to that of the next level.

A cache hit implies the *reuse* of an object. Two kinds of reuse are distinguished, temporal reuse and spatial reuse, both properties of typical reference patterns. Temporal reuse occurs when a future reference is made to the same locations

as recent references and spatial reuse suggests that future references are made to locations adjacent or close to previous references.

A common rule of thumb is that a program spends 90% of its execution time in only 10% of its code. This implies that temporal locality must be present in the instruction stream, corresponding to the locality carried by the inner loops. Spatial locality is an obvious property of instruction streams because the instructions in an extended basic block (one entry point, one or more exit points) are contiguous. Therefore, instruction caches perform very well on average.

The choice of the block size is closely tied to the expectations of temporal and spatial locality of programs [32]. A larger block size increases spatial locality (more adjacent memory cells are brought to the cache) but decreases temporal locality (less blocks exist in the cache).

In this strict definition of temporal and spatial locality, no temporal locality exists in a sequential reference pattern and therefore programs with sequential reference patterns benefit from a large block size.

3.2.2 Interference

The calculations in Example 1 were made with a known miss rate. The miss rate depends on the memory reference pattern of the program and the cache metrics. Determining the precise reference pattern is difficult because the address of an object is generally not known at compile time, and therefore the locations to which each object maps in the cache are also unavailable.

If two objects map to the same location, and are referenced in close temporal proximity, they will *interfere* in the cache. A simple example is copying the contents of one array starting at address x to another array starting at address y, one element at at time. If the first element of the two arrays happen to map to the same cache set (if $x/s = y/s$), then *every* element with same index will map to the same cache set. In a direct-mapped cache, a miss will thus occur at every single memory reference because each element with the same index in the two arrays will map, or *alias*, to the same cache set. Generally, to avoid interference, an a-way associativity is required for a overlapping sequential reference streams. If x and y are known, the amount of interference may be predicted using the method presented in [25], for, e.g., numerical computation applications. Interference is reduced with associative caches, but only as much as the associativity permits. Interference will still happen if the number of simultaneous sequential reference trails exceeds the associativity.

Programs with sequential reference patterns are very sensitive to cache interference. In [38], this effect (called *thrashing* in the paper) is studied in

two experiments resembling scientific computing and image processing. The results are that programs with several overlapping sequential reference patterns, or *unit strides*, may cause severe thrashing.

3.2.3 Prefetching

If the location of a future reference can be determined, the reference may be performed in due time to effectively hide the latency. Prefetching [15] will not decrease the bandwidth requirement, which for some machines, e.g., multiprocessors, dominates the latency, but is effective for certain programs, especially those with a reference pattern that can be determined in advance.

3.3 Measuring cache performance

The latency expectation of Example 1 above assumes prior knowledge of the miss rate and the intensity of memory references. These figures cannot be inferred from the program and its input but have to be measured. In this section, a performance-measuring tool is presented that allows precise statistics on miss rates and miss types for a given trial execution of a C++ program. The statistics are useful for evaluating analytical models for predicting cache performance, and generally such tools are widely used for the evaluation of architectural refinements and concrete software implementations alike. Computer architects need such tools to evaluate and analyse the behaviour of their programs on new hardware implementations; software developers use them to identify the critical parts of their programs; even compiler writers find use for such tools to evaluate the effect of some optimizations or to produce input data for profile-directed optimization.

Two separate activities are involved in performance evaluation, namely collection and analysis of data. These activities may be intervowen, presenting a single tool for producing results, or they may exist as separate tools. The information about the behaviour of a running program may be collected in at least three distinct ways, differing in precision, granularity and detail:

1. *Hardware registers.* The processor or device in question may collect statistical data in specially designated registers. Most devices have support for some kind of data collection used for debugging purposes or even to facilitate better performance. The Alpha 21164 [4] has instruction counters and cycle counters that may be used by kernels to report the instruction count and average cycle count per instruction of a specific

process. The advantage of hardware support is granularity and precision.

2. *Kernel registers.* Activities that are directly controlled by the kernel or operating system may be monitored and analysed in the kernel. It is common for operating systems to collect figures on interrupts (different devices), context switches (different causes), disk and network traffic, and on the virtual memory subsystem (see, e.g., the `/proc` file system in [14]). These figures are precise and may be related to specific processes, but the collection procedure is not easy to customize. Another example is the `time` command built into most Unix shell programs, which reports figures on memory and CPU usage.

3. *Program instrumentation.* The program may be compiled with statements inserted for collecting data (sometimes denoted *scaffolding*) at appropriate program points. Such statements may be inserted manually into user programs, e.g., accumulating the number of comparison statements in a program variable, or induced automatically, e.g., by execution profilers or debuggers, allowing details on frequency of execution in distinct parts of the program. This approach is also very common in the testing stages of software development, where parts of the program necessary only for asserting conditions or printing messages are included. Such approaches and the tools that support them are highly portable.

4. *Sampling.* The verbatim program may be run with an interrupt-driven sampler which records, at even periods, the position of the program counter. This is a general approach that requires little or no modification to the source code but produces nothing more than execution frequencies for the different parts of a program.

The interpretation and analysis of the collected material depends solely on the nature of the experiment. If the material consists of accumulated values, it may be ready for interpretation as such. If, on the other hand, the collected material constitutes a *trace*, e.g., a series of data points from the execution of the program, it may require further processing to compute or extract the required statistics.

Program analysis tools have been developed by others for this and other purposes. The simplest tools are designed to measure the (extended) basic block rate [35], that is the number of basic blocks executed, and are useful for constant factor determination. Other tools are based on on-line (using interprocess communication or shared memory) or off-line analysis of generated address traces. The ATOM (Analysis Tool with OM) system [5, 33] is a (commercial) tool-building complex for program analysis which modifies the object modules under programmer control to produce an instrumented version

of the program, thus requiring no modification to the source code, eliminating cumbersome and error-prone editing. Many different analysis tools may be individually built or custom tailored with ATOM, such as I/O- and instruction-counting, cache simulations, and memory allocation monitoring.

In the following section, an off-line cache performance simulator is presented which processes address traces generated by instrumentation for use primarily in C++ programs.

3.4 CPROF — a cache profiling tool

Given a set of constants describing the metrics of a cache, as well as a temporally ordered list of the memory locations of the reads and writes of a trial run of a program, the number and types of cache misses for that program may be computed through a simulation. Measuring cache performance requires precise information about the order and reference location of each memory access as well as the necessary cache metrics. Each memory reference (read or write) is recorded by means of instrumentation, producing an *address trace* corresponding to the memory reference pattern of the program. The access pattern $R = \{r_1, r_2, \ldots, r_n\}$ consists of a series of read/write specifications and corresponding addresses, $r_i = (t, v)$, where $t \in \{r, w\}$ (meaning a read or a write) and v is the address referenced. The address trace is recorded in its entirety, ordered by time of reference, and is subsequently fed to a profiler that performs a simulation of block replacements in a cache given the specification of the cache. The output from the profiler is the hit and miss figures of the execution of the program.

3.4.1 Recording the address trace

The access pattern is recorded by means of calling special functions for each read and write found in the include file `cprof.h` (see Appendix B.1). The function calls are inserted manually by the programmer as follows (where *l* is an *lvalue* and *r* is an *rvalue*):

- `READ(`*l*`)` reads the value of *l*.
- `WRITE(`*l*`,`*r*`)` writes *r* to *l*.

If the symbol `CPROF` is defined, the address trace is written to the file named by the definition of `CPROF`. The program in Example 2 is changed to reflect the memory reads and writes and shown as the program in Example 3.

Example 2 (*A victim program)*

```
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (i = 1; i < n; i++)
    v[i-1] = v[i];
}
```

Example 3 (*Full address trace)*

```
#define CPROF "mytrace.dat"
#include "cprof.h"
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (WRITE(i,1); READ(i) < n; WRITE(i,READ(i)+1))
    WRITE(v[READ(i)-1],READ(v[READ(i)]));
}
```

If `CPROF` is not defined, the function calls impose no overhead on the execution time of the program. If the symbol is defined, the trace file (`mytrace.dat` in Example 3) will contain a reference pattern such as

```
r 0xbffff928
r 0xbffff928
r 0xbffff930
r 0xbffff928
w 0xbffff92c
r 0xbffff928
w 0xbffff928
r 0xbffff928
r 0xbffff928
...
```

It is not necessary, however, to insert `READ` and `WRITE` statements for every value read or written in the program. In particular, data allocated on the stack, such as the control variable `i` in the example, are typically register allocated and can be assumed transparent in cost with respect to the cache. In Example 3, it is the array `v` that is the focal point and the program might be rewritten as shown in Example 4. The choice of which memory accesses to include is entirely up to the user. In later examples, where memory accesses are restricted to pointer dereferencing, the access patterns are similarly restricted to pointers.

Example 4 (*Partial trace)*

```
#define CPROF "mytrace.dat"
#include "cprof.h"
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (int i = 1; i < n; i++)
    WRITE(v[i-1],READ(v[i]));
}
```

3.4.2 Profiling the address trace

The cache is configured by the configuration constants defined in Section 3.1. The capacity and block size is given in words (the word size w, in bits, is a configuration option). Memory is byte-addressed and all references are word-aligned. The cache capacity is $Mw/8$ bytes and it always follows a write-back scheme (dirty misses cost the same as clean misses). The blocks consist of a tag field, a valid bit, and a dirty bit.

The B words of data themselves are not required in the simulation because it is solely the reference patterns, not the data, that are necessary for the block replacements. The cache block contains a time stamp with a serial number of the last reference to the block to facilitate simulation of different replacement policies. The interpretation of addresses, and the hashing to cache blocks, follow the definitions in Section 3.1.1.

For a given reference r_i to address $v = v_t||v_s||v_f$, its corresponding cache set is given by the index v_t. The reference represents a cache hit if and only if there exists a valid cache block in this set with a tag value equal to v_t. Otherwise, the reference is a miss. The hit and miss counts are accumulated by the profiler, and the misses are distributed on cold-start/compulsory, capacity and conflict miss types. The number of miss types are accumulated as follows.

- A cache miss is a cold-start cache miss if the selected cache block is marked invalid (no more than b such misses may occur).
- A cache miss is a compulsory cache miss if it is the first reference to some input block (no more than $\frac{n}{B}$ such misses may occur if n is the size of input).
- A cache miss is a conflict miss if the total number of references to the cache since the last replacement of the selected block is less than B/a

(this is an approximation to real conflict misses, because these are defined properly only in the context of the program).

- A cache miss is a capacity miss in all other cases (in its current version, 0.1, CPROF does not record the compulsory cache misses, instead, they show up as capacity misses).

As the consequence of a cache miss, (1) a cache block is selected, (2) the contents of the block are written in case of a dirty miss, (3) the new block is read, and (4) the fields of the selected block are updated. Only (1) and (4) are necessary for the simulation. Selection of the cache block in its cache set depends on the replacement policy, which is given as a configuration option. The offset ν_f within the cache block is not used.

The profiler may also be instructed to instrument the code to count the number of read and write operations. This is accomplished by defining the symbol `CPROF_COUNT` prior to including the header file, and calling the procedure `cprofstats()` to write the statistics to a desired output stream. Memory reference counting is independent of address-trace recording and the two may be combined in the same simulation. In Appendix A, the command-line arguments, output format, and error messages of the CPROF tool is given.

In order for the address trace to be faithful, it is necessary to insert statements carefully with respect to the data types in question. An assignment statement `a = b`, where `a` and `b` are multi-word structures, will produce a read sequence for all words in `b` followed by a write sequence for all words in `b` (the word size is assumed to be `sizeof(void*)`). This is not an accurate sequence, however, because a block copy is an intermixed sequence of reads and writes. Inserting trace statements for reading or writing large structures will produce inaccurate reference patterns. Unrealisticly high hit/miss ratios may also occur because of superfluous `READ` trace statements placed at program points where the compiler would use a register. This emphasizes the caveats in Section 3.4.1 that care must be taken when choosing candidates for the address traces.

The cache simulation encompasses the execution of a single thread in perfect I/O-conditions. Latency induced by other sources than the cache (see Section 2.1) in the execution of the program will not be captured by the simulation.

While the cache simulator is language independent, the recording of address traces is bound to C++ programs only. It is however a simple matter to write instrumentation code to generate address traces from programs written in other languages.

3.5 Visualization of reference patterns

Each reference r_i may be plotted in the plane at (i, r_i) producing a visual representation of the reference pattern with time (reference index) on the horizontal axis and place (reference value) on the vertical axis. Such graphical plots, although not possible for very large address traces, present the locality properties of the program in an intuitive fashion. Scattered dots mean nonlocal behaviour while proximity in the horizontal axis implies temporal locality and proximity in the vertical axis implies spatial locality.

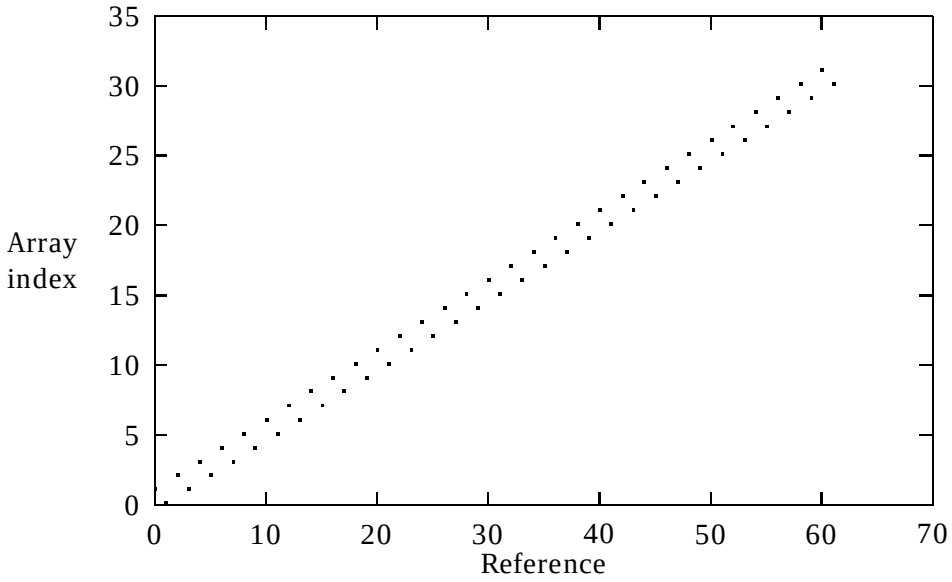

Figure 3.1: Visualization of the reference pattern of Example 4.

The footprint of the program in Example 4 is shown in Figure 3.1.The values on the vertical axis correspond to the array indices of `v` in the program. The tool will relocate all references to begin at address `0` by default; this may be overridden by specifying a scaling factor of `0` which places the actual virtual addresses of the references on the vertical axis. The choice of which data objects to include in the address trace may impact the visualization. If, for example, objects on both the heap and the stack are included, a huge gap may emerge on the plots reflecting the distance of these in memory. Generally, stack allocated objects (`auto` storage class in C++) need not be included in the address traces.

Chapter 4

Program tuning for locality

A number of simple but effective techniques for improving cache performance exist which are easily adopted in programming languages with concrete addressing or in a compiler and/or linker. In this chapter, some such techniques are surveyed which are general and easy to understand. They illustrate the effects of the memory hierarchy from the software side and serve as a reference for understanding the different types of locality issues necessary to further develop a cost and machine model for caches in the next chapter.

4.1 Representation

The following techniques focus on the addresses of data objects with some prior knowledge of how they are referenced. Typically, the structure of the objects themselves reveal the pattern in which they are referenced.

4.1.1 Packing

Few decades ago, memory was a sparse resource, and much attention was given to the choice of representation. The Pascal language thus contains a keyword `packed` that instructs the compiler to attempt to minimize the memory used for arrays [36], and COBOL programmers have made a (dubious) habit of eliminating redundancy by representing years by only two digits. Similarly, the C language [17] offers a `union` data structure in which all members share the same address. Explicit *packing* is simply the technique of reducing the requirement of space through the choice of data types that can hold no more than the maximum value it needs to represent, or by eliminating or sharing parts of a data structure. Today, memory conservation is not a big issue at this

level of implementation, but size still matters for speed.

Programmers not concerned with fine-tuning for space have a tendency to choose representations with ample precision. This is explained by the relatively small number of integral types available in most languages, and by the implementation-dependent precision of these types, which leads to a pessimistic choice of representation. In the same manner, some pre-defined types offered by, e.g., a standard library are inherently pessimistic, as they are designed for genericity.

Packing is straightforward in most cases. One can choose 32-bit floating-point representations instead of 64- or 80-bit ones, and the widths of integers may also be chosen more carefully. On a machine with 64-bit integers (see, eg. [4]), the following definition of a coordinate tuple occupies 16 bytes.

Example 5

```
struct point {
  int x, y;
};
```

With some knowledge of the use of `point`, i.e., the range necessary for each coordinate to span, this requirement can be reduced to a minimum. If it, e.g., needs to address points on an A4 (21×29.7 cm) page of 1 200 dots/inch (472 dots/cm), the maximum coordinate necessary is just above 14 000 and can be represented in only 16 bits, reducing the memory requirement to one fourth, or 4 bytes. Such specializations are easily parameterised:

Example 6

```
template<int i>
struct point {
  int x:i, y:i;
};
```

This far from cuts the execution time by a factor of 4, however. No matter how well-packed a data structure becomes, a complete cache line still needs to be transferred from main memory if the reference results in a miss. Packing therefore primarily has an effect on spatial locality; a data structure being referenced in an uniformly distributed pattern will not benefit much from packing. The promised benefit of packing is therefore related to the block size.

In the following experiment, a graph of 1 000 vertices has been traversed in a breadth-first manner (using a queue) with three distinct approaches: (1) an adjacency linked-list representation with separately allocated memory for each edge, (2) an integer adjacency matrix representation, and (3) a bit-packed adjacency matrix representation. The traversal has been measured

without initialization overhead for graphs of densities varying from 5% to 90%, and the results are shown in Figure 4.1.

The adjacency matrix representations necessitate a complete scan of all n possible edges for each vertex; thus the instruction count of a matrix-represented graph traversal is proportional to n^2. The linked-list representation stores only the edges of the graph, and the traversal is therefore proportional to the number of edges. As the density rises, the spatial locality of the matrix-represented graphs increases, because a higher proportion of the edges will be adjacent in memory.

The linked-list traversal thus shows a steady, linear behaviour in Figure 4.1, while the matrix representations are practically constant. The bit-packed matrix traversal is about 20% faster than the integer counterpart due to added spatial locality in each row of the adjacency matrix. The adjacency list required 24 bits per edge, while the bit-packed adjacency matrix required n^2 bits regardless of the density. The break-even point in this experiment is therefore a density at about $m = n^2/24$, which for a 1 000-vertex graph is about 40 000 edges, or 10%. In [13], bit-packing is reported to reduce the running time even further.

4.1.2 Alignment

A cache block is always aligned in memory at an address divisible by the block size B. Therefore, any data structure not in alignment with the cache width risks one extra transaction from memory due to its placement in two adjacent cache lines. *Alignment* reduces memory-to-cache transactions by fine-tuning spatial locality at a cost of a small memory overhead.

Alignment is achieved by *padding*, adding empty data slots to ensure that each record is located at a block boundary. Assuming that B is the block size, alignment of some object of type `T` to a block boundary is achieved by the following statements:

Example 7

```
char* data = new char [sizeof(T) + B];
int padding = B - data % B;
T* t = (T*) (data + padding);
```

For arrays of objects, alignment of each member of the array is also required, otherwise only the first element of the array is guaranteed to be aligned. Alignment of each member is achieved by padding the data structure itself, ensuring that its size is a multiple of the block size:

Example 8

```
struct aligned_T : T {
  char padding[B - sizeof(T) % B];
};
```

In this way, arrays of `T` may be allocated as `aligned_T` objects, ensuring alignment. Padding is at the expense of memory cells which are allocated but never used. The cost of alignment in terms of a memory overhead becomes $n(m \bmod B) + p \bmod B$, where n is the number of elements allocated, m is the size of each element in bytes, B the cache line size in bytes and p the point of allocation. Thus, the worst case overhead is $(B-1)(n+1)$, although padding may be combined with packing to lower this.

Clearly, for a data structure such as a d-heap, selecting d such that $B = dm$ (for an element size m), and aligning the heap structure on block boundaries will reduce the miss rate and the execution time slightly for heap operations. The reduction has effect also on the capacity of the cache, which will require less blocks to represent the same number of records[1]. Cache performance of aligned d-heaps is studied in [22].

The effect of alignment has been verified in a simple experiment in which a number of randomly chosen elements of an array of words are accessed with and without alignment of the array. For the unaligned array, we must expect $\frac{1}{B}$'th of the references to cross a cache boundary, and thus an improvement in the miss rate of precisely this figure. The experiment shows (see Figure 4.2) that using aligned data structure gains a few percent compared to using an unaligned array. A cache simulation of the address traces from the same experiments (with smaller input sizes) yields a just under 1 cache misses per element for the aligned array and just under 1.125 misses per element for the unaligned array, corresponding to $\frac{1}{B}$ more cache misses per element for $B = 8$ words per cache line.

It should be obvious that, if memory is accessed sequentially rather than in a sequence of random accesses, alignment makes no difference to the miss rate.

Definition 1 An object is B-aligned if B divides its address in memory, that is, an object a is B-aligned if and only if the address of a modulo B is zero.

4.1.3 Interleaving of arrays

Any reference pattern to arrays of objects will exhibit poor spatial locality due to conflict misses if these arrays are separately allocated. Consider the triple

[1] The problem of alignment is also pertinent to distributed caches, in which "false sharing" of a cache block may result in a "ping-pong" effect between caches.

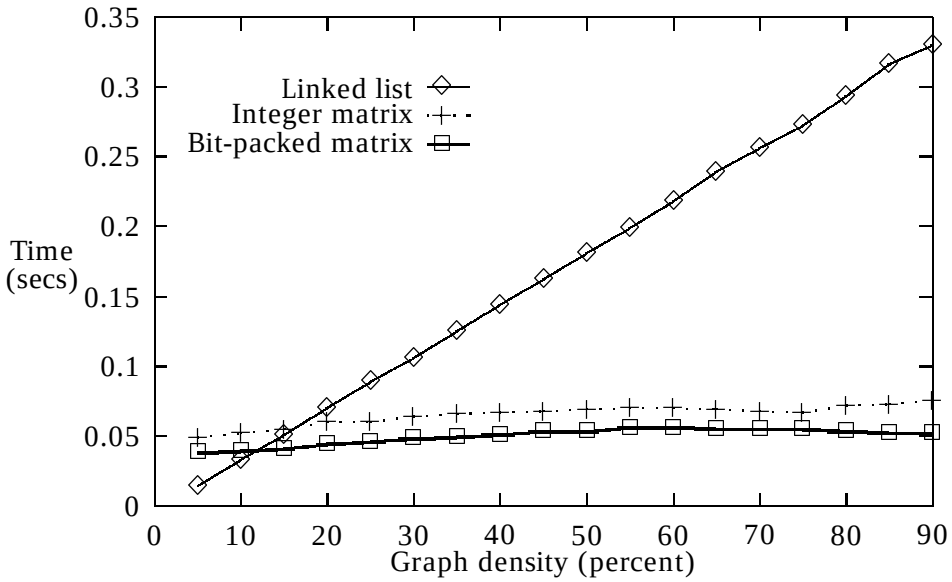

Figure 4.1: Execution times of breadth-first search (1 000-vertex graphs).

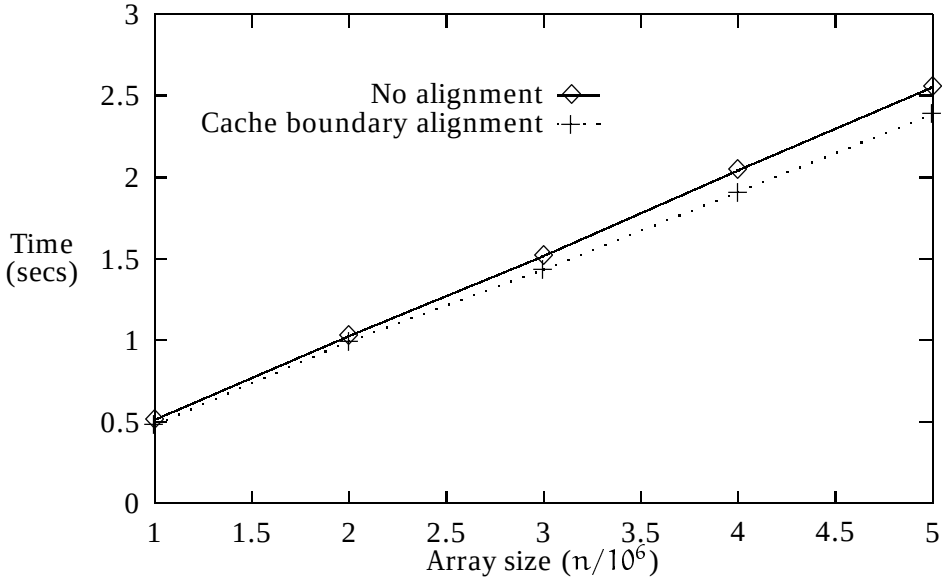

Figure 4.2: Aligned and unaligned random access.

Example 9

```
int a[n], b[n], c[n];
```

designed to be referenced by the same index. Any reference to an element `i` of `a`, `b` and `c` in close temporal proximity will be more vulnerable to a cache miss than if the elements were neighbours in memory. Therefore, the straightforward transformation to

Example 10

```
struct _abc {
  int a, b, c;
} abc[n];
```

will achieve a better hit rate. The three arrays are interleaved into one array such that each element with the same index are proximate elements in the new array. In Figure 4.3, the effect of using interleaved arrays of triples is shown for a randomly distributed reference pattern, that is, each three references to `a[i]`, `b[i]` and `c[i]` is changed to `abc[i].a`, `abc[i].b` and `abc[i].c`. In [38], a larger example more prone to interference (see Section 3.2.2) is sped up by factors 2 to 4 on various architectures.

Depending on the specific indexing into this structure, there may be an overhead in terms of extra instructions to compute the effective addresses of the elements. Conversely, the data structure (interleaved or not) may contain members that do not contribute to referential locality in this manner, e.g., some subordinate field which is not referenced by the same index in close temporal proximity. Such fields would not contribute to added locality by packing, rather, they may lower the effects of packing by separating the fields that were candidates for the interleave.

To avoid this from happening, the members of a structure which are referenced in close temporal proximity should be made proximate in one part of the structure. This technique applies to the complete working set of the program but is easiest to control in conventional data structures. As a rule of thumb, the elements of a structure should be ordered by their reference frequencies.

If the access pattern to the three arrays in the experiment were purely sequential, there will be nothing gained from interleaving except for a reduction of the risk of interference.

4.2 Reference patterns

The following examples focus on the rearrangement of program fragments to achieve better reference locality, requiring some intuition as to the recurrences and loop-carried dependencies of the underlying algorithms.

4.2.1 Loop interchange

Nested loops may exhibit a nonsequential reference pattern in the inner loop, leading to conflict misses in the cache. These are unnecessary and can be eliminated simply by interchanging the outer and inner loop. In the following code fragment, the two-dimensional array `a` is referenced column-wise, leading to poor spatial locality of the reference pattern of the combined two loops.

Example 11

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    a[j][i] += 1;
```

By simply interchanging the two loops, or the indices `i` and `j` inside the inner loop, whichever is easiest, all elements of a cache line are referenced in sequence. For arrays of size 10^5 to 10^6, the effect of interchanging the two loops is shown in Figure 4.4. The graphs clearly show that, subsequent to cache effects setting in at an array size of about 200 000 elements, the column-wise pattern is proportionally slower as the array size increases. As the two programs have the same instruction rate, the speed-up is attributed to cache misses alone. The simulated miss rates in a 1kb direct-mapped cache with 32-byte blocks for an array of 10^5 4-byte integers are given in Table 4.1 and show a miss rate close to 50% for the column-wise nesting, and about 6% for the row-wise nesting. These miss rates are easily explained by the fact that the inner loop contains a read followed by a write to the same address, meaning that the column-wise approach has a cache miss on every iteration, while the row-wise misses every 8th iteration, corresponding to a sequential access pattern with 4-byte integers in 32-byte blocks.

4.2.2 Loop fusion

Code segments which have sequences of identical loop structures accessing the same data structures in the same manner may be *fused* to ensure that every element is used in a single sequence, thus improving the temporal locality of the combined loops. In the following code fragment, each element of `a` and `c`

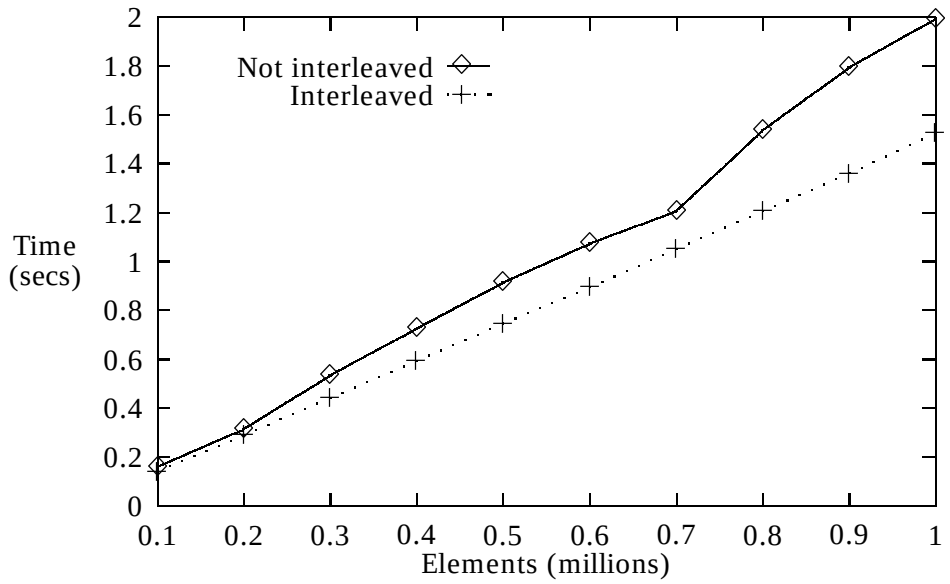

Figure 4.3: Operations in interleaved and uninterleaved arrays.

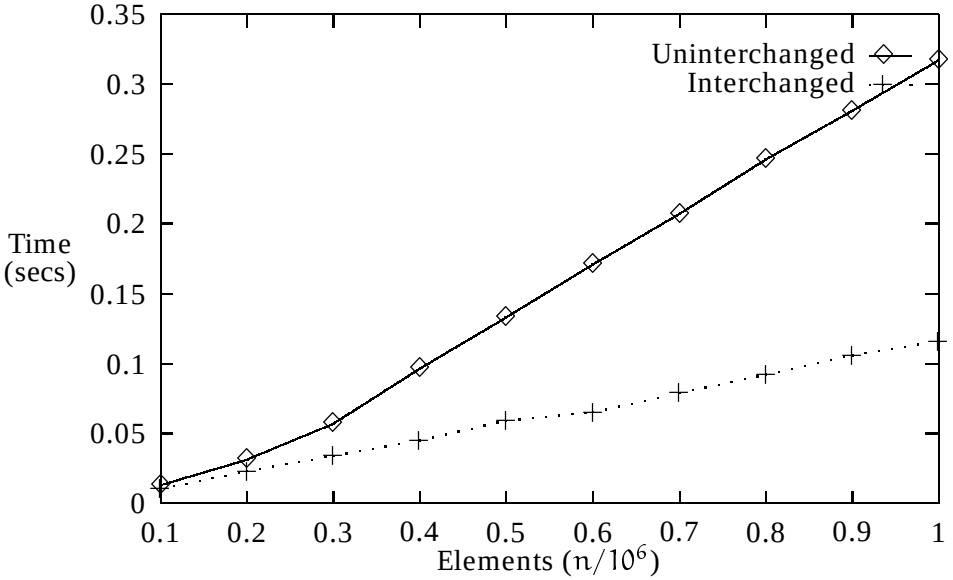

Figure 4.4: Row-wise vs. column-wise scan of a two-dimensional array.

Uninterchanged	Reads	Writes	Total
Hit count:	1343	99856	101199
Miss count:	98513	0	98513
• Cold-start:	128	0	128
• Capacity:	98385	0	98385
• Conflict:	0	0	0
Total:	99856	99856	199712

Interchanged	Reads	Writes	Total
Hit count:	87373	99856	187229
Miss count:	12483	0	12483
• Cold-start:	128	0	128
• Capacity:	12355	0	12355
• Conflict:	0	0	0
Total:	99856	99856	199712

Table 4.1: Simulation of uninterchanged and interchanged loops.

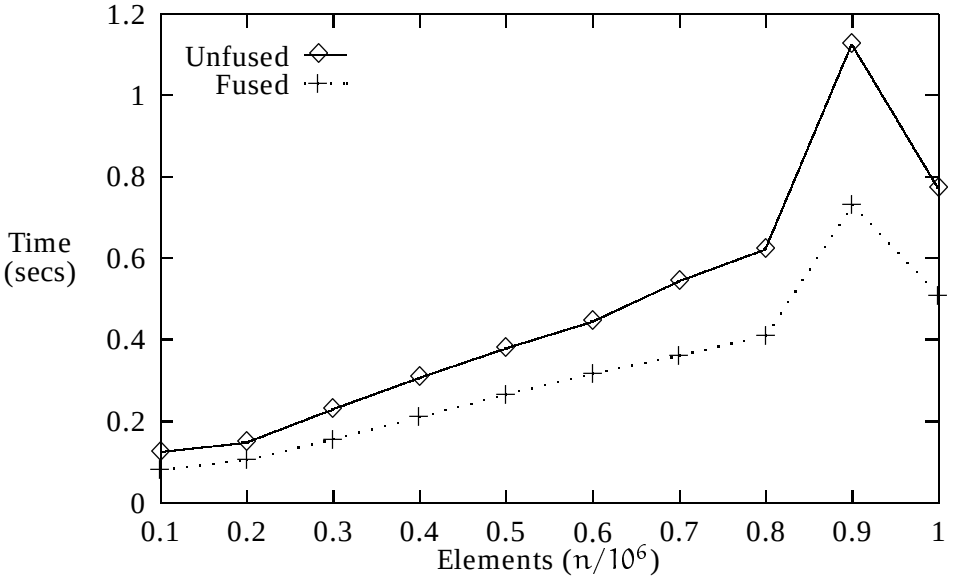

Figure 4.5: Execution of unfused and fused loops.

are read from the cache twice. Fusing the loops such that the bodies of each loop become the body of the fused loop, finishes each access to elements of a and c for each pass of the inner loop.

Example 12

```
for (i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    d[i][j] = a[i][j] + c[i][j];
```

Fusion actually saves more than bandwidth. In this example, n^2 test and jump instruction sequences are eliminated as well. Figure 4.5 shows the effect of loop fusion for the above code fragment for $n^2 = 10^5$ to 10^6 (the anomaly at $n = 0.9$ million is caused by interference).

4.2.3 Blocking

Code segments with reference patterns that access both row and columns in the same inner loop are not candidates for any of the above transformations or improvements. The reason is that any rearrangement of how the data is represented still leads to the same reference patterns, because the accesses are *orthogonal* on the two indices. *Blocking* [20] is a technique used to arrange the access patterns in blocks, not unlike loop interchange. The blocks are not entire rows or columns, but sub-matrices.

The following matrix-multiplication code fragment contains three loops, of which the innermost two contain references to all n^2 elements of c as well as n times the same column of b and one row of a, requiring $2n^3 + n^2$ memory accesses in total.

Example 13

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    for (s = 0, k = 0; k < n; k++)
      s += b[i][k] * c[k][j];
    a[i][j] = s;
  }
```

If the cache capacity is higher than the combined size of the three matrices, all is well, and blocking will not contribute anything. Assuming the cache can hold only $n(n + 1)$ elements (the reference set for the two inner loops), the blocking algorithm may express good temporal locality for the i'th row

of `b`, and the `c` matrix. If this last assumption does not hold, the number of misses is proportional to $2n^3 + n^2$. To understand why this is true, consider the reference pattern for the three matrices (assuming the matrices are organised in row major order):

- The `a` matrix is accessed sequentially from the top-left element to the bottom-right element, requiring each block that `a` occupies to be read exactly once, which is n^2/B blocks.
- An entire row of `b` is accessed sequentially n times before moving on to the next row, requiring n^2/B blocks to be read if a row does not fit in the cache, or n/B blocks otherwise, because of temporal locality, meaning that the `b` matrix is responsible for n^3/B block transfers for large matrices.
- An entire column of `c` is accessed for every element of `a` and this column exhibits neither (1) spatial nor (2) temporal locality because (1) the matrix is organized in row major order, and (2) the complete `c` matrix must be accessed before an element is reused. Therefore, the `c` matrix requires n^3 blocks to be read.

This totals, for large matrices, $(n^3 + n^2)/B + n^3$ blocks to be read. In Figures 4.6 and 4.7, the progress of the multiplication program is depicted as circles indicating memory locations that have been visited, where darker circles indicate older references in each matrix.

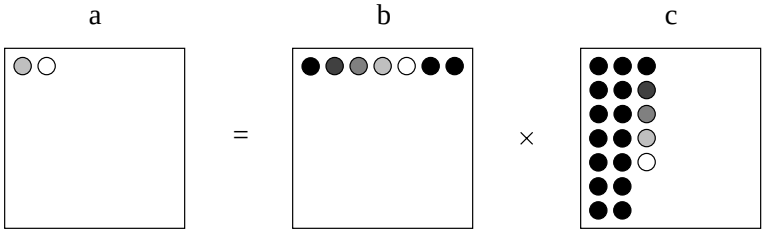

Figure 4.6: References made in early stages of matrix multiplication.

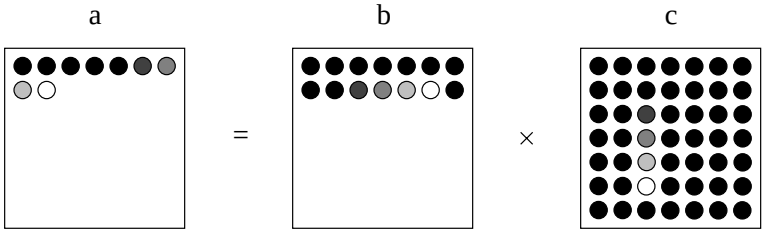

Figure 4.7: References made in later stages of matrix multiplication.

The purpose of *blocking* (or *tiling*) applied to matrix multiplication is to arrange the order of the operations between elements of the `b` and `c` matrices such that the locality of the memory references is improved. The matrices are divided into *blocks* of size $z \times z$, where $z < n$ is the *blocking factor* chosen such that a $z \times z$ submatrix of `c` and z columns of `b` will fit in the cache. The program is changed such that two new inner loops compute in steps of z rather than going from the beginning to the end of `a` and `c`.

Example 14

```
for (jj = 0; jj < n; jj += z)
  for (kk = 0; kk < n; kk += z)
    for (i = 0; i < n; i++)
      for (j = jj; j < min(jj+z,n); j++) {
        s = 0;
        for (k = kk; k < min(kk+z,n); k++)
          s += b[i][k] * c[k][j];
        a[i][j] += + s;
      }
```

The inner body is identical except for the accumulation of product sums to `a` rather than the simple assignment used in the non-blocked program. The blocked program works its way in groups of z columns through the `a` and `b` matrices, and reads each submatrix of size $z \times z$ from the `c` matrix at a time. This approach means that the blocked multiplication needs z outer iterations to multiply the blocks of the `b` matrix with the blocks in the `c` matrix. The blocked program thus has more memory operations than the non-blocked. Each element in `a` is read and written n^3/z times, while each element in `b` and `c` is read n^3 times. The number of memory operations has therefore risen from $2n^3 + n^2$ in the non-blocked program to $2n^3 + 2n^3/z$ in the blocked program, and the instruction count is assumed also to be higher due to the added loops.

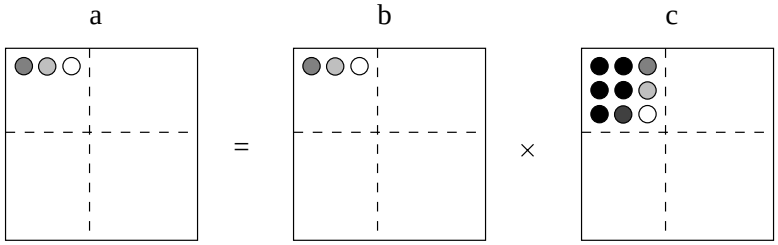

Figure 4.8: Early stages of blocked matrix multiplication.

The locality however has improved by close to a factor B in the `c` matrix because of the temporal locality in the submatrices. Further, for large matrices, spatial locality arise in the accesses to the `b` matrix. In the `a` matrix, spatial locality exists for blocking factors close to B, and temporal locality exists for

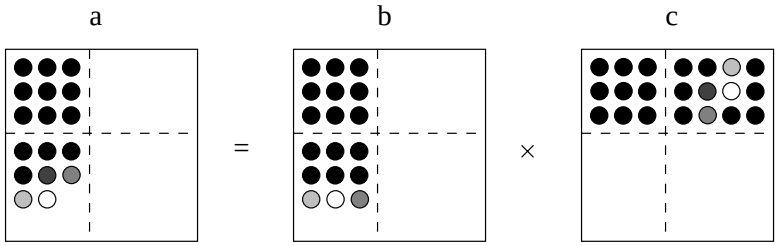

Figure 4.9: Later stages of blocked matrix multiplication.

each z columns if they fit in the cache. In the best case, therefore, the miss rate expectancy is close to $\frac{2}{B}(\frac{n^3}{z} + n^3)$. In Figures 4.8 and 4.9, the progress of the memory references of the blocked matrix multiplication is shown with darker circles indicating older references.

The impact of blocking is highly dependent on the blocking factor, and it is therefore interesting to measure the miss count for different blocking factors. The simulated miss rates for a 1kb 4-way associative cache with 32-byte blocks and a word size of 4 bytes is shown in Figure 4.10. It seems that, in this experiment, a blocking factor of 8 produces fewest misses, except for a matrix of size 32×32, which is caused by interference misses.

For blocking factors as low as 8, the inner loop of the blocked program may be *unfolded* z times with little effort. The loop itself can be avoided, and the repeated references to the `b` matrix may be held in registers through the next-innermost loop. The unfolded, blocked program, which assumes that z divides n, is shown below.

Example 15

```
for (jj = 0; jj < n; jj += z)
  for (kk = 0; kk < n; kk += z)
    for (i = 0; i < n; i++) {
      p = &b[i][kk];
      s0 = *p++, s1 = *p++, s2 = *p++, s3 = *p++;
      s4 = *p++, s5 = *p++, s6 = *p++, s7 = *p;
      for (j = jj; j < jj + z; j++)
        a[i][j] += s1 * c[kk][j] + s2 * c[kk+1][j]
                 + s2 * c[kk+2][j] + s3 * c[kk+3][j]
                 + s4 * c[kk+4][j] + s5 * c[kk+5][j]
                 + s6 * c[kk+6][j] + s7 * c[kk+7][j];
    }
```

A footprint of the reference pattern of the blocked matrix multiplication is depicted in Figure 4.11 with the `a` matrix residing in the top third, the `b` matrix in the middle third and the `c` matrix in the bottom third of the figure. The

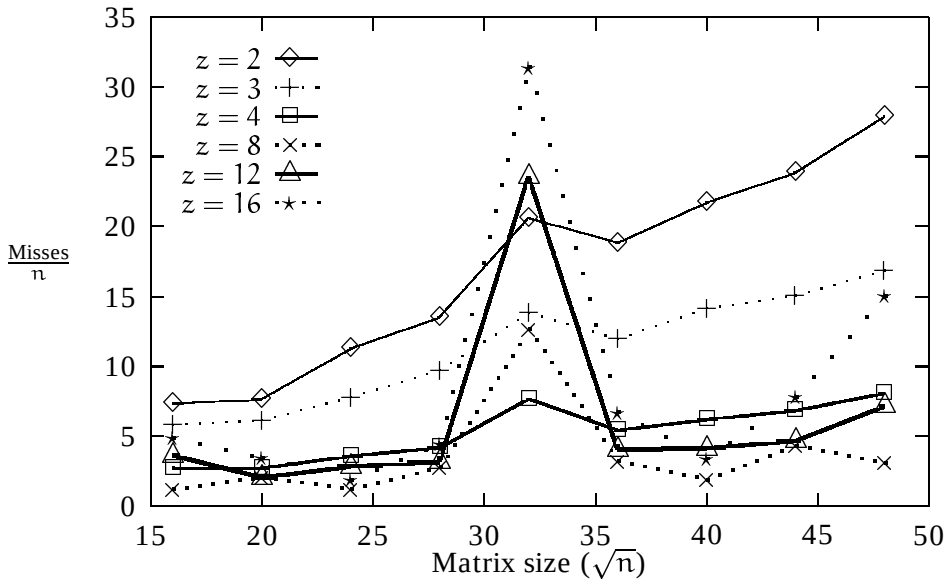

Figure 4.10: Miss counts with different blocking factors.

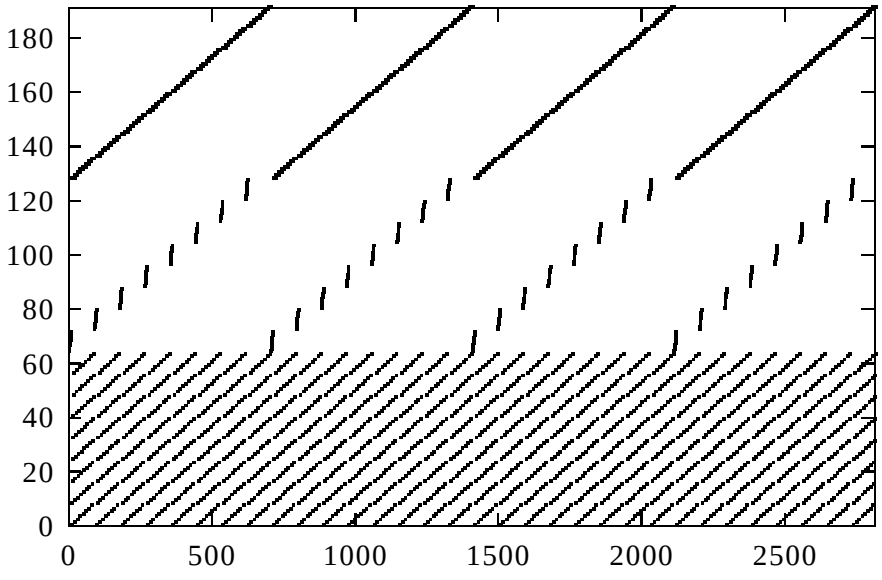

Figure 4.11: Footprint of blocked 8×8 matrix multiplication.

reference pattern clearly shows the spatial locality in the references to `a`, the temporal locality in the references to `b` and the locality of both kinds in the references to `c`.

Blocking should in fact also facilitate better register allocation. If the number of available registers is larger than z^2, an entire sub-matrix may be held solely in registers in the inner loop. The technique shows how both temporal locality (the `c` matrix) and spatial locality (the `b` matrix) is used at the same time, and is aimed primarily at reducing capacity misses. However, since blocking reduces the number of active elements present in the cache at a given point, choosing z^2 less than the capacity of the cache will reduce conflict misses as well. Matrix multiplication is a good example of the latency problem because the arithmetic work performed per memory reference is low.

The effect of blocking is measured on both reference machines for 32-bit floating-point numbers and for matrix sizes of 100×100 up to 300×300 with a blocking factor of 8. On both machines, the blocked programs were faster, though only marginally on the AMD, as shown in Figure 4.12.

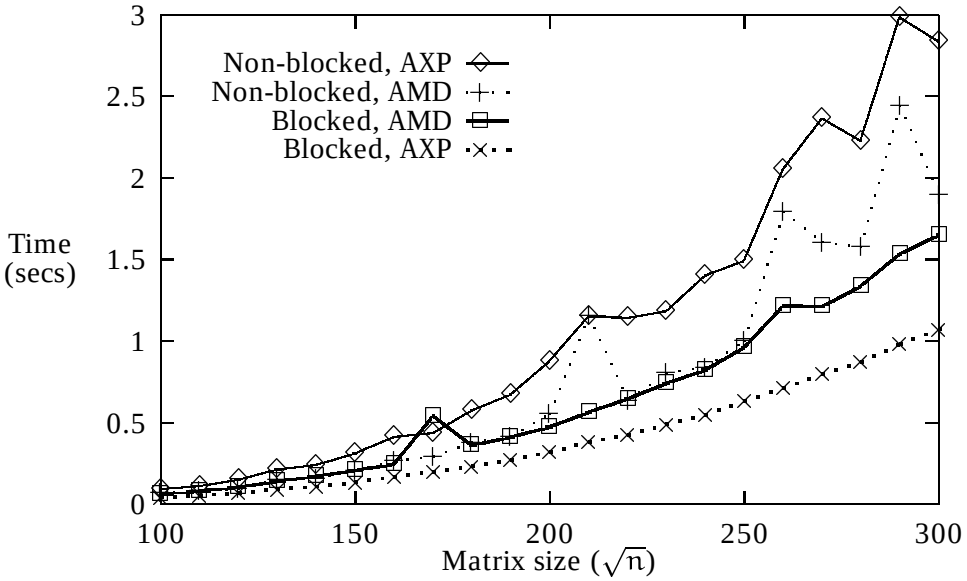

Figure 4.12: Execution time of non-blocked and blocked ($z = 8$) programs.

Also evident in Figure 4.12 is that the execution time of the non-blocked programs seems to depend on more than n. It is noted in [20] that the performance of blocking for matrix multiplication in caches is highly sensitive to the problem size and blocking factor, and that it was not well understood. I suspect two main contributors to the sensitivity are interference and associativity (as discussed in Section 3.2.2). The blocking factors may be chosen at higher values as associativity grows.

4.3 Conclusion

The techniques for improving locality presented in this chapter represent little more than "hints to the programmer," heuristics for rearranging bits and pieces of the program. The experiments show that tuning for locality may improve the running time of programs by a small factor. The reference patterns of the programs were known or anticipated in advance, and each technique suits only one particular type of locality optimization.

Generally, the optimizations increase the instruction rate, and some, interestingly, even increase the number of memory accesses performed. It is therefore a clear conclusion that a *suboptimal* algorithm (in traditional analysis) may be faster than the optimal algorithm due to its data locality properties.

The techniques that resulted in the most dramatic improvements were those which transformed the reference pattern to sequential scan of the memory cells. For programs which already have a sequential access pattern, locality tuning showed little improvement. The concept of sequential access patterns is generalised and developed as an analytical cost model in the next chapter.

Chapter 5

An analytical cost model

It has been shown that the reference pattern of a program influences its running time. In this chapter, an analytical model for cache miss complexity is developed with focus on spatial locality. A prerequisite for such a model is information about *when* a block is accessed and *where* the block is mapped in the cache. The aim of the model is to answer questions such as *what is the speed-up of program A over program B?* or *what is the miss rate of the program?*

5.1 Cost models and machine models

In the analysis of algorithms or programs, it is necessary to choose the kind or kinds of primitive operations around which the analysis revolves, that is, defining a *machine model* that assigns a cost for all primitive operations of the underlying architecture. The running time of a program is the sum of the primitive operations executed on a particular input. Depending on what measure is needed by the analysis, and the detail level, several options exist.

- In *asymptotical analysis*, the primitive operation is typically a key comparison, e.g., the comparison operation in a binary search dominates the complexity and all other operations in the program are considered constant with respect to the number of comparisons.
- In *meticulous analysis*, the goal is to also analyse the constant factors, most importantly those in the leading term of the function that expresses the running time. This requires a machine model that is more detailed because every instruction executed by the program contributes to the constant factor.

- In the analysis of *memory references*, the goal is to analyse the number of memory references made by the program. The memory references may be divided into categories, either as reads or writes, or as random memory accesses or sequential memory accesses. A machine model that is able to express memory references is simple and can be independent of particular architectures.

The challenge in the definition of the machine model is to keep it *abstract* (i.e., relatively independent of any particular architecture) while still maintaining the detail level necessary for the analysis.

For asymptotical analysis, the machine model is about as abstract as can be, requiring no resources or implementation. For meticulous analysis, the machine model must capture the number of instructions executed in a formalism that resembles a typical modern computer, and actual implementations must be carried out to be able to count the number of primitive operations. For memory reference analysis, the machine model can be kept abstract because the memory references can be described by the combination of the abstract algorithm and a concrete definition of the types it uses.

5.2 A sequential-access model

In this section, a model for memory hierarchy performance is presented which focuses solely on spatial locality and whose premise is $B > 1$, i.e., a memory hierarchy with a block size larger than one word. The backbone of the model is a precise account of instruction and memory reference rates of the program. A machine model for instruction cost is presented first, followed by a weighted derivation capturing also the cost of memory references.

5.2.1 Pure C

Meticulous analysis requires a machine model that defines the types and costs of computations and operations. In the following, an adoption of the *Pure C* model from [16] is presented. It is a machine-independent model, though not more so as to resemble the instruction-set architecture of most modern computers.

The assumed *machine model* is based on a *program*, a *memory*, a collection of *registers*, and a processor to execute the program.

- The memory stores N words and allows random access through dereferencing of the registers.

- The register collection consists of a constant but unspecified number of integers (signed or unsigned depending on the context). Each register stores one word. In order for a register to address the entire address space of N words, it follows that the word size is $\lceil \log_2 N \rceil$ bits if the memory is word-addressed. Registers have arbitrary names.
- The program consists of optionally labelled statements in five categories representing a subset of C [17] obtained by removing control structures (`switch`, `for`, `do`, `while`), functions, structures and enumerations, arrays, and all data types other than integers from the language, leaving Pure C with

  1. *Memory-to-register* assignments, that is, read statements "`x = *p;`" and *register-to-memory* assignments, that is write statements "`*p = x;`".
  2. *Register-to-register* assignments, that is, read statements "`x = y;`" and *constant-to-register* assignments "`x = c;`" where `c` is an integer constant.
  3. *Arithmetic expression* assignments, assigning the results of unary operations on registers or constant values as "`x =` $\otimes$`y;`" where $\otimes \in \{-, \tilde{}, !\}$; and the results of binary operations on two registers and/or constant values as "`x = y` $\oplus$ `z;`" where $\oplus \in \{+, -, *, /, \&, |, \hat{}\}$.
  4. *Conditional branches* "`if (x` $\triangleleft$ `y) goto` *label*`;`" in which `x` and `y` are registers or constants and $\triangleleft \in \{<, \leq, ==, \geq, >\}$.
  5. *Unconditional branches* "`goto` *label*`;`".

This formalism is not unlike that used in intermediate stages of compilation, and it is fairly easy to transform a traditional C (or similar) program into a Pure C program because the semantics is the same. While loops in the Pure C programs are normally transformed to avoid the loops ending in an unconditional jump.

Definition 2 (Pure C unit cost) All Pure C statements cost τ each. The number of Pure C operations of a program is a function $f(n)$ of the input size n. The cost of executing a Pure C program as a function of its input size is $f(n)\tau$.

In the example given in Figure 5.1, `bzero()` performs a backwards sequential scan of an array, writing a zero value to all `n` words of an array `a` (similar to the standard C library functions `bzero` or `memset`). The loop of `BSET` becomes the three lines 3–5, and the Pure C cost of `BSET` is $(3n + 3)\tau$

While the Pure C unit-cost model allows machine-independent meticulous analysis of programs, it makes on the one hand some assumptions about the

```
void bset(int* a, int n) {
  int* p = a + n;
  while (p > a)
    *--p = 0;
}
```

```
void BSET(int* a, int n) {
1: int* p = a + n;
2: goto 5;
3: p = p - 1;
4: *p = 0;
5: if (p > a) goto 3;
6: }
```

Figure 5.1: C and Pure C versions of the backwards scan.

capabilities of the underlying hardware, and fails on the other hand to capture some of its properties under those assumptions. The unit-cost assumption is accurate only for machines whose instruction set architecture reflects the model, i.e., register machines of the load/store type capable of handling three registers in arithmetical and logical operations. Some architectures may offer combinations of the Pure C statements as a single operation while others are unable to execute some Pure C statement in a single operation. Conversely, latency from pipeline hazards, spill-code, and branch delays (see Section 2.1) is assumed to be zero.

5.2.2 Weighted Pure C cost

The Pure C unit-cost model is now extended to capture latency expectations for memory accesses by breaking down the memory operations into three distinct categories, and assigning a weight to these operations. Observe that memory is accessed indirectly through registers only, using operations from category 1 in the Pure C model. An *index register*, or pointer, is a register which is used at some point in the program to dereference data in memory using operations from category 1.

Definition 3 When applied to pointer variables, Pure C operations in category 1 (`x = *p; *p = x;`) are henceforth denoted *memory access* operations; operations in category 2 (`p = a;`) are denoted *random pointer assignments*; and operations in category 3 (`p++; p--;`) are denoted *sequential pointer assignments*.

Definition 4 A *sequence* S of operations in a Pure C program is the Pure C operations executed by the Pure C machine, ordered by time of execution. A *subsequence* of S is the Pure C operations of only a specific subset of Pure C executed by the Pure C machine, ordered by time of execution. Two operations in a sequence or subsequence are *adjacent* if they are neighbours in the sequence.

This means that the subsequences can be used to "filter" instructions from the instruction stream, e.g., the subsequence of memory access operations involving a specific index register.

Lemma 1 Any subsequence of adjacent memory access operations involving the same index register, or pointer, will have the Weighted Pure C cost equal to the cost of the first operation in the sequence.

Proof Informally, all memory accesses happen to the same memory location because the index register does not change. Hence, the same block is accessed in all references and only the first may induce a miss. More accurately, the first operation s_0 in the subsequence will bring the block to the top of the memory hierarchy. Because s_1 is a reference to the same block, the access cost will be zero. By transitivity, all references other than s_0 are free, and the complete sequence S will cost the same as s_0. □

Random pointer assignments effectively reposition the pointer to a new, arbitrary address. The block corresponding to this address may or may not be present at the top level of the memory hierarchy.

Definition 5 A random pointer assignment arbitrarily repositions a pointer. A memory access operation has cost $\tau^\star$ if it follows a random pointer operation.

Lemma 2 For any intermixed sequence S consisting of any number of memory access and ρ random pointer assignments, the cost of the sequence is $\rho\tau^\star$.

Proof By Lemma 1, each subsequence of adjacent memory access operations costs the same as the first operation in the sequence. By definition 5, this cost is $\tau^\star$ for random pointer assignments. Therefore, the sequence has cost $\rho\tau^\star$. □

Definition 6 A sequential pointer operation relatively repositions the pointer. A memory access operation that follows a sequential pointer operation costs $\tau^\star$ if the block is not present and 0 otherwise.

Lemma 3 For any subsequence S consisting of any number of memory access operations and σ sequential pointer operations intermixed, the cost of the sequence is $\lceil \frac{\sigma}{B} + 1 \rceil \tau^\star$.

Proof By Lemma 1, each subsequence of adjacent memory access operations costs the same as the first operation in the sequence. By definition 6, this cost amortizes to $\frac{1}{B}\tau^\star$ for a sequential pointer operation. Thus, the cost is $\frac{\sigma}{B}\tau^\star$, which needs to be rounded up (because σ may not divide B), and increased by 1 (because the first memory reference in the sequence may not be at a block boundary). Therefore, the sequence has cost $\lceil \frac{\sigma}{B} + 1 \rceil \tau^\star$. □

These results show that it is possible to express spatial locality in terms of assignments to pointer variables in two distinct categories, absolute (random) and relative (sequential) repositioning.

Definition 7 Let a *memory track* T_i consist of a random pointer assignment followed by σ_i sequential pointer operations. The union of all memory tracks is the sequence of random and sequential pointer operations of the program such that the memory tracks overlap in the sequence.

The fact that the tracks may overlap means that z overlapping tracks may map to distinct blocks. This may be difficult to accomplish and may result in interference misses as discussed in Section 3.2.2. Typically, though, z (equal to the number of pointer variables dereferenced in the same scope), is low.

Consider the following Pure C function `copy()` which copies `n` words from the source array `s` to the destination array `d` with a Pure C unit cost of $(5n+O(1))\tau$.

Example 16

```
void copy(int* d, int* s, int n) {
1: int* t;
2: int* p = s + n;
3: int* q = d + n;
4: goto 9
5: t = *p;
6: *q = t;
7: p--;
8: q--;
9: if (p > s) goto 5;
}
```

In this program fragment the subsequence of random and sequential pointer operations is $[\texttt{p=s+n}, \texttt{q=d+n}, \texttt{p--}_1, \texttt{q--}_1, \texttt{p--}_2, \texttt{q--}_2, \ldots, \texttt{p--}_n, \texttt{q--}_n]$. This corresponds to the two memory tracks

$$\begin{aligned} T_1 &= [\texttt{p=s}, \texttt{p++}_1, \ldots \texttt{p++}_n] \\ T_2 &= [\texttt{q=d}, \texttt{q++}_1, \ldots \texttt{q++}_n] \end{aligned}$$

Lemma 4 The cost of the memory track T_i is $\lceil \frac{\sigma_i}{B} + 1 \rceil \tau^\star$.

Proof The cost is $\tau^\star$ for the first operation and $\frac{\sigma_i}{B}$ for the σ_i following. If σ_i divides B, the first operation will have zero cost because the complete sequence amortizes to $\frac{\sigma_i}{B}$. Otherwise, one extra block must be read, so, in general, the cost is $\lceil \frac{\sigma_i}{B} + 1 \rceil \tau^\star$. □

This cost is an upper bound. If no memory access exists between two pointer operations in a sequence, the first pointer operation will cost B or $\frac{1}{B}$ in the

model but nothing in reality. More likely, the two tracks may interfere and exhibit temporal locality, which is not captured by the model. In addition, a memory track may be a continuation of another, previous, memory track, which can be difficult to discover in the analysis of a program. The footprints of reference patterns are useful to identify the memory tracks.

Theorem 1 The cost of the memory references of a program is $\sum_{i=1}^{\rho} \lceil \frac{\sigma_i}{B} + 1 \rceil \tau^\star$ where ρ is the number of memory tracks (equal to the number of random pointer assignments), σ_i is the number of sequential pointer operations in memory track i and B is the block size.

Proof The cost of the memory operations in the memory track T_i is $\lceil \frac{\sigma_i}{B} \rceil \tau^\star$ (Lemma 4). The cost of all disjoint memory tracks is the sum of their costs, which is $\sum_{i=1}^{\rho} \lceil \frac{\sigma_i}{B} + 1 \rceil \tau^\star$. □

We now have a cost model for memory operations that describes spatial reference locality and requires analysis of a (partial) implementation of an algorithm. The cost model does not take temporal locality into account. This means that the cost of the memory operations may be lower in practice. In the following, the cost model is used in an example.

5.2.3 Example: Repeated touching

Consider the problem of touching (writing some value) to each of n elements of an array m times. The two algorithms shown in Figure 5.2 solve this problem in two distinct ways, ONESWEEP by touching each element m times in succession, and MSWEEP by touching each element once each over m sweeps of the array. The algorithms have the same asymptotical Pure C unit-cost complexity $\Theta(mn)$.

More exact figures are easily extracted from the Pure C versions of the programs (Figure 5.3) by counting the instructions. The Pure C cost of `ONESWEEP` is $(3mn + 5n + 3)\tau = (3mn + 5n)\tau + O(1)$, while the Pure C cost of `MSWEEP` is $(3nm + 5m + 4)\tau = (3mn + 5m)\tau + O(1)$. While the leading term is the same for the Pure C cost of the two programs, `ONESWEEP` is slightly slower for small values of m/n.

The memory-access costs are substantially different for the two programs. `ONESWEEP` has a single memory track of size n while `MSWEEP` has m memory tracks of size n. Therefore, the memory-access cost of `ONESWEEP` is $\lceil \frac{n}{B} + 1 \rceil \tau^\star$ and the memory-access cost of `MSWEEP` is $m \lceil \frac{n}{B} + 1 \rceil \tau^\star$. To sum up, the weighted Pure C costs of the two programs are

```
ONESWEEP([a_1, ..., a_n], m) :
1:  i = 0
2:  while (i < n) do
3:    j = 0
4:    while (j < m) do
5:          a_i = 0
6:          j = j + 1
7:    end while
8:    i = i + 1
9:  end while
```

```
MSWEEP([a_1, dots, a_n], m) :
1:  j = 0
2:  while (j < m) do
3:    i = 0
4:    while (i < n) do
5:      a_i = 0
6:      i = i + 1
7:    end while
8:    j = j + 1
9:  end while
```

Figure 5.2: Two ways of m touches to each of the n elements in a.

$$\begin{aligned} \text{cost}(\texttt{ONESWEEP}) &= (3mn + 5n)\tau + \lceil \tfrac{n}{B} + 1 \rceil \tau^\star + O(1) \\ \text{cost}(\texttt{MSWEEP}) &= (3nm + 5m)\tau + m\lceil \tfrac{n}{B} + 1 \rceil \tau^\star + O(1) \end{aligned}$$

The miss rates of the two programs were obtained by simulating the cache behaviour of their access patterns for a 4kb direct-mapped cache with a block size of 256 bits = 32 bytes for $m = 10$ and $n = 0.1 \times 10^4$ to 1×10^4 in steps of 10^3. In Figure 5.4, the results are shown as the number of misses per memory reference, where the number of memory references for each experiment is mn. The simulation clearly shows that, when cache effects set in at $n > 10^3$ (corresponding to a working set of 4kb for a word size of 4 bytes per array element), `MSWEEP` degrades considerably while `ONESWEEP` has a constant number of misses. For $4n > M$ (the working set larger than the capacity of the cache), `MSWEEP` falls to about 0.125 cache misses per memory reference.

The Pure C implementations in Figure 5.3 were were run on both reference machines for higher values of n and m, namely $m = 10^3$ and $n = 0.1 \times 10^6$ to 1.6×10^6. This range corresponds to a working set of 0.4Mb to 4.8Mb. The results are shown in Figure 5.5. On both reference machines, `ONESWEEP` performed about 5.9 times better than `MSWEEP`.

The block size of both reference machines is 256 bits which means that 8 array elements of 32 bits each will fit in a cache block. Referring to the weighted Pure C costs of the two programs and the experiments on the AMD reference machine for $n = 1.6 \times 10^6$, $m = 10^3$, and selecting $B = 8$, the Pure C unit cost τ and memory reference cost $\tau^\star$ can be calculated from

$$\begin{cases} 4.808 \times 10^9 \tau + 2 \times 10^5 \tau^\star &= 14.51 \text{ seconds} \\ 4.800 \times 10^9 \tau + 2 \times 10^8 \tau^\star &= 84.64 \text{ seconds} \end{cases}$$

```
void ONESWEEP(int* a, int n, int m) {
 1: int* p = a;
 2: int* q = a + n;
 3: goto 10
 4: int j = m;
 5: goto 8;
 6: *p = 0;
 7: j = j - 1;
 8: if (j > 0) goto 6;
 9: p = p + 1;
10: if (p < q) goto 4;
11: }
```

```
void MSWEEP(int* a, int n, int m) {
 1: int j = m;
 2: int* q = a + n;
 3: goto 10;
 4: int* p = a;
 5: goto 8;
 6: *p = 0;
 7: p = p + 1;
 8: if (p < q) goto 6;
 9: j = j - 1;
10: if (j > 0) goto 4;
11: }
```

Figure 5.3: Pure C versions of the repeated-touching programs.

which gives $\tau = 17.5ns$ and $\tau^\star = 175ns$. On the AXP reference machine, the corresponding figures are $\tau = 11ns$ and $\tau^\star = 110ns$.

Theorem 2 The cost of a cache miss is 10 Pure C units.

Proof The above experiment concludes that $\tau^\star/\tau = 10$, that is, the cost of a cache miss is 10 Pure C units. □

5.3 Capturing temporal locality

The memory-access cost model does not capture all spatial locality of a reference pattern. However, each memory track may be considered a *continuation* of a previous memory track. This means that the cost of the continuing memory track is $\tau^\star$ less because it can be assumed that the random pointer assignment in the head of the track is free.

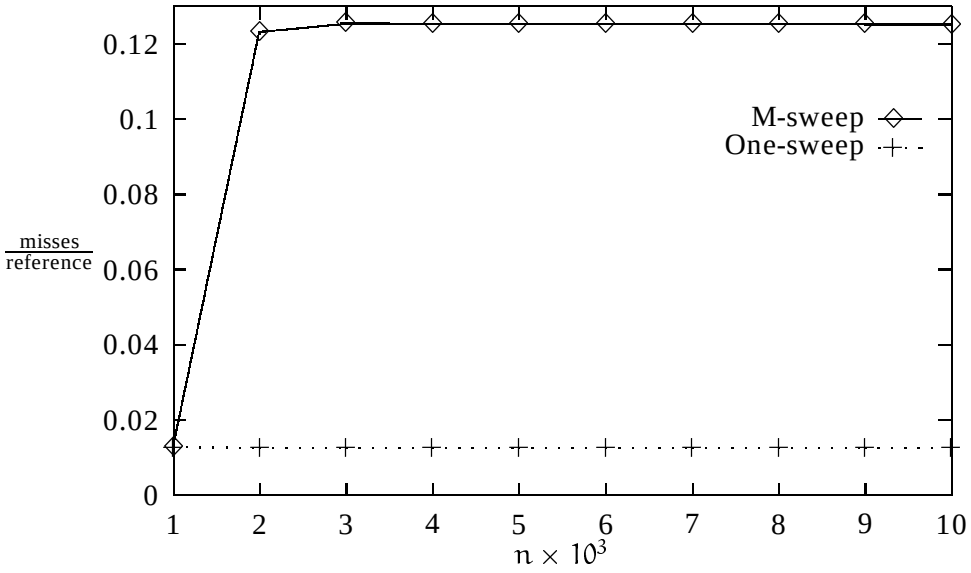

Figure 5.4: Simulated cache ($B = 8$) performance for the array sweep.

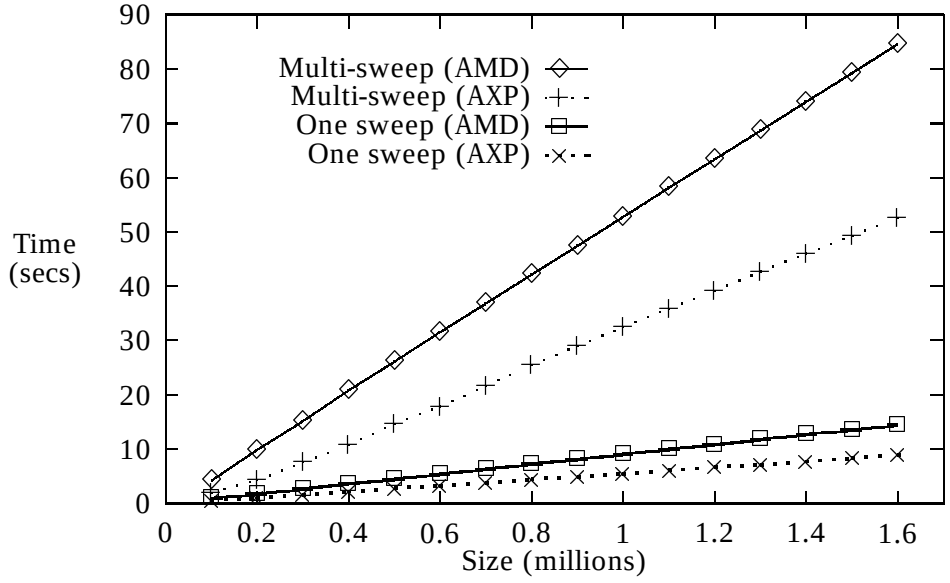

Figure 5.5: Array sweep ($m = 1000$).

Worse, the model does not capture temporal locality at all. Temporal locality is difficult to predict for a reference pattern for which some properties is not known in advance, especially that a subset of the references are known to be made to memory locations that are known to be touched by previous references. Some programs have a reference pattern that is heavily dependent on the input data, and for these programs, an assessment of temporal locality is difficult. Other programs have a strict reference pattern only partially dependent of the input data.

5.4 Conclusion

A machine model for meticulous (*little-oh*) analysis has been presented together with a cost model for instructions and memory references. The machine model restricts memory operations to be performed indirectly through index registers, or pointer variables, and captures only spatial locality. The result is a measure of the number of cache misses and Pure C instructions as a function of the size of the input.

It has further been shown empirically that $\tau^\star \approx 10\tau$, that is, the cost of a cache miss is approximately 10 Pure C units.

Chapter 6

Cache behaviour of heap construction

In the previous two chapters, some ad-hoc techniques for improving cache behaviour of programs and a cost model for the cache miss complexity has been discussed. In this chapter, a study of *heap construction* programs is presented and their cache behaviour is studied in part using the weighted Pure C model.

6.1 Implicit heaps

An *implicit binary heap* is a binary tree of n nodes represented by an array $[a_1, \ldots, a_n]$. The tree is complete except from possibly the bottom level, which is filled from left to right. The tree has depth $\lceil \log_2(n+1) \rceil$.

The nodes are stored in the array such that node i is placed at index i in the array. For all nodes i, the left and right children *left*(i) and *right*(i) of node i are located at indices $2i$ and $2i+1$ in the array, respectively, and the parent node *parent*(i) of a node $i > 1$ is located at array index $\lfloor i/2 \rfloor$.

Each node consists of a key element and possibly some associated data. The key of node i is denoted k_i. The tree is a (minimum) heap if it satisfies the *heap property* which says that for all nodes i ($i > 1$), $k_{\mathit{parent}(i)} \leq k_i$. It follows from the heap property that the root element has the minimum key, and by enforcing the heap property on insertions and removals of elements in the heap, it may be used in a sort or as a priority queue.

Two critical heap operations are defined, (1) SIFTUP$([a_1, \ldots, a_n], i)$, which rearranges the path in the heap represented by the array $[a_1, \ldots, a_n]$ from node i to the nearest ancestor j with a key less than a_i such that the heap

property is ensured for all nodes in the path (see Algorithm 1), and (2) SIFTDOWN$([a_1, \ldots, a_n], i)$ which rearranges the subtree under node i such that all elements satisfy the heap property (see Algorithm 2). SIFTUP is typically used when adding elements to a heap, while SIFTDOWN is used when deleting elements from the heap.

Algorithm 1 (*Sift-up)*

SIFTUP$([a_1, \ldots, a_n], i)$:
1: $j = parent(i)$
2: $t = a_i$
3: **while** $(j \geq 1$ **and** $a_j > t)$ **do**
4: $\quad a_i = a_j$
5: $\quad i = j$
6: $\quad j = parent(i)$
7: **end while**
8: $a_i = t$

Algorithm 2 (*Sift-down)*

SIFTDOWN$([a_1, \ldots, a_n], i)$:
1: $t = a_i$
2: $j = left(i)$
3: **while** $(j < n)$ **do**
4: $\quad$ **if** $(a_j > a_{j+1})$ **then**
5: $\quad\quad j = right(i)$
6: $\quad$ **if** $(t > a_j)$ **then**
7: $\quad\quad a_i = a_j$
8: $\quad\quad i = j$
9: $\quad\quad j = left(i)$
10: $\quad$ **else**
11: $\quad\quad i = j$
12: $\quad\quad j = n + 1$
13: $\quad$ **end if**
14: **end while**
15: **if** $(j = n$ **and** $t > a_j)$ **then**
16: $\quad a_i = a_j$
17: $\quad a_j = t$
18: **else**
19: $\quad a_i = t$
20: **end if**

In the algorithms above, the comparison between two elements of the array is assumed to compare the key components of the two elements, which means that either $a_i = k_i$ or the comparison operator $>$ is overloaded for the data structure represented in the array to compare the keys of the structures. In the following, the *depth* of a node is defined as the number of edges between the node and the root of the heap, and the *height* of a node is defined as the number of edges between the node and the bottom level of the heap, assuming that the bottom level of the heap is complete (i.e., the heap contains $2^j - 1$ nodes for some integer j).

Property 1 SIFTUP requires 2 memory accesses per iteration (one read and one write). In the worst case (for a path from node i to the root), SIFTUP requires $2\lceil \log_2 i \rceil$ memory references.

Property 2 SIFTUP requires 6 Pure C units per iteration.

A Pure C implementation is given in Appendix B.4.1.

Property 3 SIFTDOWN accesses both the left and the right child for each node it visits (two reads and one write). In the worst case (for a path from node i to a leaf node), SIFTUP requires $3(\lceil \log_2 n \rceil - \lceil \log_2 i \rceil + 1) - 1$ memory references.

Property 4 SIFTDOWN requires 9 Pure C units per iteration.

A Pure C implementation is given in Appendix B.4.2.

6.2 Heap construction

The construction of an implicit heap from a complete set of elements is now studied. Given a binary tree represented as an array, the tree must be arranged such that all nodes satisfy the heap property. Four solutions to this problem are presented in the following sections, namely

1. construction of a heap by repeatedly adding elements from the top to the bottom using SIFTUP [37].
2. construction of a heap by repeatedly heapifying subtrees from the bottom to the top using SIFTDOWN [7].
3. a variant of the bottom-up method which performs the operations in a different order [1].
4. construction of a heap by *selection* [27].

6.2.1 Top-down repeated insertion

As stated, SIFTUP is the critical heap operation for insertion, and the text-book solution to the heap construction problem also happens to be the repeated application of SIFTUP for each consecutive node $1, \ldots, n$ in the heap (see Algorithm 3). This is the first phase of Williams' *heapsort* [37].

Algorithm 3 (*Williams)*

WILLIAMS$([a_1, \ldots, a_n])$:
1: $i = 2$
2: **while** $(i <= n)$ **do**
3: SIFTUP$([a_1, \ldots, a_n], i)$
4: $i = i + 1$
5: **end while**

Lemma 5 Heap construction based on repeated SIFTUP requires $\Theta(n \log_2 n)$ key comparisons, $(6n \log_2 n) + o(n)$ Pure C operations and $2n \log_2 n$ memory access operations.

Proof The worst that can happen is that every SIFTUP must travel to the root node of the heap, which is equal to the sum of the depths of each node to be added. That is, the i'th SIFTUP must iterate $\lceil \log_2 i \rceil$ times, and the complete heap construction will require $\sum_{i=1}^{n} \lceil \log_2 i \rceil < n \log_2 n$ key comparisons. An example of an input sequence that reaches this bound is an descendingly sorted array which has $k_i > k_{i+1}$. Every iteration of Williams' algorithm will add an element to the heap which has the hitherto lowest seen key value, and therefore must travel to the root.

As the Pure C cost of the loop in SIFTUP is 6τ (Property 2, the Pure C cost of the implementation of Williams' algorithm (see Appendix B.4 becomes $(n \log_2 n + o(n))\tau$, where the $o(n)$ term stems from the few instructions in the outer loop. Similarly, each iteration of SIFTUP requires one read and one write operation, yielding a maximum memory access count of $2n \log_2 n$. □

6.2.2 Bottom-up subtree heapification

In [7], Floyd presented a heap construction algorithm based on bottom-up application of SIFTDOWN. The tree is reheapified from node $\lfloor \frac{n}{2} \rfloor$ up to node 1 in $\lfloor \frac{n}{2} \rfloor$ applications of SIFTDOWN.

Algorithm 4 (*Floyd)*

FLOYD($[a_1, \ldots, a_n]$) :
1: $i = parent(n)$
2: **while** $(i > 0)$ **do**
3: SIFTDOWN($[a_1, \ldots, a_n], i$)
4: $i = i - 1$
5: **end while**

Lemma 6 Heap construction based on repeated SIFTDOWN requires $\Theta(n)$ key comparisons, $9n + o(n)$ Pure C operations and $3n$ memory access operations.

Proof The worst that can happen is that every SIFTDOWN must travel to the bottom of the tree, which is equal to the sum of the heights of each node to be added. That is, the j'th SIFTUP (which processes node $i = \lceil n/2 \rceil - j$) must iterate $\lceil \log_2 n \rceil - \lceil \log_2 i \rceil$ times, and the complete heap construction will require $\sum_{i=1}^{\lfloor n/2 \rfloor} \lceil \log_2 n \rceil - \lceil \log_2 i \rceil < n$ key comparisons. An example of an input sequence that reaches this bound is, as in Lemma 5 a descendingly sorted array.

As the Pure C cost of the loop in SIFTDOWN is 9τ, the Pure C cost of the implementation of Floyd's algorithm (see Appendix B.4) is $(9n+o(n))\tau$, where the $o(n)$ term stems from the few instruction in the outer loop. Similarly, each iteration of SIFTDOWN will read the values of two nodes (the children of node a_i) and write one value to node a_i. The complete heap construction thus requires no more than $3n$ memory accesses.

6.2.3 Post-order bottom-up subtree heapification

Floyd's algorithm may be rewritten so that the *order* of the SIFTDOWN operations follow a recursive post-order rather than a linear scan, shown in Algorithm 5. The number of SIFTDOWN operations stays the same so the worst-case number of key comparisons and memory operations is identical to that of Algorithm 4.

The order of the SIFTDOWN operations performed in the post-order descent is such that when SIFTDOWN is invoked on node i, then the SIFTDOWN for the two subtrees of i have been invoked. Algorithms 4 and 5 are equivalent because both perform SIFTDOWN for all non-leaf nodes in a sequence where the children have been processed before the parent.

Algorithm 5 (*Post-order Floyd)*

```
POSTORDERFLOYD([a_1, ..., a_n], i) :
1:  if (right(i) < parent(n)) then
2:     POSTORDERFLOYD([a_1, ..., a_n], right(i))
3:  if (left(i) < parent(n)) then
4:     POSTORDERFLOYD([a_1, ..., a_n], left(i))
5:  SIFTDOWN([a_1, ..., a_n], i);
```

A non-recursive version of Algorithm 5 is now given, which eliminates the need for stack space and removes the function call overhead.

Algorithm 6 (*Non-recursive post-order Floyd)*

```
NONRECURSIVEPOSTORDER([a_1, ..., a_n]) :
1:  ℓ = log_2 n
2:  i = 2^ℓ − 1
3:  while (i >= 2^(ℓ−1)) do
4:     SIFTDOWN(i)
5:     p = 2
6:     while (i mod p = 0) do
7:        SIFTDOWN(i div p)
8:        p = p × 2
9:     end while
10:    i = i − 1
11: end while
```

The program scans level $\lfloor \log_2 n \rfloor$ of the heap (assuming the bottom level is complete) from right to left. For each node added, the nested loop follows a path towards the root, calling SIFTDOWN for each node visited, until a node is found which is not a left child. In the Pure C implementation in Appendix B.4, a node satisfying that the next p ancestors are left children will have the lower p bits equal to zero.

Lemma 7 The non-recursive post-order version of Floyd's method requires $\Theta(n)$ key comparisons, $9n + o(n)$ Pure C operations and $3n$ memory access operations.

Proof The number of key comparisons is equivalent with that given in the proof in Lemma 6. The main loop of the non-recursive bottom-up implementation costs slightly more than the basic loop in the program based on Floyd's method, yet it will still be denoted $o(n)$ for all iterations. In Pure C terms, therefore, the cost is still $(9n + o(n))\tau$. □

6.2.4 Median partitioning heap construction

In [27], a heap construction approach using a repeated partitioning of the set of nodes is proposed for heaps on external storage. A simplification of this for traditional binary heaps is now presented. The method does not strictly implicitly build a heap because some extra space is required.

The strategy is to employ *median partitioning* of the array to successively determine each level of the heap from the bottom up. First, the leaf level is determined, then the next higher level, and so on. In order to extract the records of level ℓ, a *selection* problem for all remaining elements must be solved which determines the lowest key value belonging to level ℓ. In this selection, the heap elements on levels higher than ℓ are excluded.

Because level ℓ contains half the elements of the elements in levels 1 through ℓ, the nodes 1 through $2^\ell - 1$ must be partitioned around its median, that is, the selection must arrange the array elements $[a_1, \ldots, a_{2^\ell-1}]$ such that $\forall i, 1 \leq i < 2^{\ell-1}$ and $\forall j, 2^{\ell-1} \leq j < 2^\ell$, $a_i \leq a_j$. In Algorithm 7, a selection algorithm is given based on the method used in *Quicksort* [18]. It partitions the array $[a_1, \ldots, a_n]$ around k, such that the k smallest values are placed in $[a_1, \ldots, a_k]$. Applying this algorithm with $k = \lceil (n+1)/2 \rceil$ yields an array partitioned around the median of the array elements. The pivot element assigned in line 2 is assumed to be a randomly chosen element in $[a_l, \ldots, a_r]$.

Algorithm 7 (*Select)*

```
SELECT([a_1, ..., a_n], l, k, r) :
1:  if (r − l = 1) then
2:    return l
3:  else
4:    p = PIVOT(l, r)
5:    k' = PARTITION([a_1, ..., a_n], l, p, r)
6:    if (k < k') then
7:      return SELECT([a_1, ..., a_n], l, k, k')
8:    elseif (k > k') then
9:      return SELECT([a_1, ..., a_n], k' + 1, k, r)
10:   else
11:     return k
12:   end if
13: end if
```

By applying SELECT for levels $\ell \in \{\lceil \log_2 n \rceil \ldots, 2\}$ in succession, that is, applying SELECT$([a_1, \ldots, a_n], 1, 2^{\ell-1}, 2^\ell)$ for $\ell \in \{\lceil \log_2 n \rceil \ldots, 2\}$, this clearly produces a legitimate heap. Partitioning an array of size n requires, on average,

a linear number of key comparisons because each partition, on average, splits the array in half; [28] states that the number of key comparisons is $4n$ on average (for randomly permuted input). SELECT is applied for each level in the tree, $\lceil \log_2 n \rceil$ invocations, each with half the nodes than the previous invocation. The number of key comparisons for the complete heap construction using median partitioning therefore becomes $8n + o(n)$ on average.

The number of memory operations is equal to the number of key comparisons, that is, the entire partitioning requires $8n + o(n)$ memory operations on average.

The inner loops of PARTITION (lines 6 to 8 and 9 to 11 of Algorithm 8) account for about 4 Pure C operations each (see the Pure C implementation in Appendix B.4.3). This implies that each key comparison translates to a Pure C cost of 4τ, and the number of Pure C operations for a partition becomes $16n + o(n)$. The complete selection thus has $32n + o(n)$ Pure C operations.

Algorithm 8 (*Partition)*

PARTITION$([a_1, \ldots, a_n], l, p, r)$:
1: $t = a_p$
2: SWAP(a_l, a_p)
3: $i = l$
4: $j = r$
5: **repeat**
6: **repeat**
7: $i = i + 1$
8: **until** $(i \geq r$ **or** $a_i > t)$
9: **repeat**
10: $j = j - 1$
11: **until** $(t \geq a_j)$
12: **if** $(i < j)$ **then**
13: SWAP(a_i, a_j);
14: **else**
15: SWAP(a_p, a_j);
16: **return** j;
17: **end if**
18: **forever**

6.3 Reference locality properties

The reference locality properties of the four heap construction programs are now examined. To make the analyses simpler, it is assumed that the heaps are aligned.

Lemma 8 The left and right child of any element i will be within the same block regardless of how the heap is aligned in memory.

Proof The children are positioned at array indices $2i$ and $2i+1$. For any i, $\lfloor \frac{2i}{B} \rfloor = \lfloor \frac{2i+1}{B} \rfloor$ which means they belong to the same block. □

Lemma 9 In a B-aligned heap (a heap whose first element is at a memory location divisible by B) of n elements, the first element (the element with lowest index) in each level $\ell, \ell = 0, 1, 2, \ldots$ will be B-aligned for all $\log_2 B < \ell \leq \lceil \log_2 n \rceil$ if B is a power of 2.

Proof By Definition 1, element 1 in the heap is B-aligned. The first element of level ℓ is at position $2^{\ell-1}$ in the implicit heap. If B is a power of 2, then $z = \log_2 B$ is an integer. Elements at positions $2^{\ell-1}$, $\ell > z$, will therefore by the definition also be B-aligned because $2^{\ell-1}$ **mod** $2^z = 0$. □

This also implies that the elements in level $z < \ell \leq \lceil \log_2 n \rceil$ occupy exactly $2^{\ell-z-1}$ blocks.

6.3.1 Top-down repeated insertion

Williams' algorithm works its way down the tree in a breadth-first fashion, invoking SIFTUP, which works its way back up the tree each time. For every i'th invocation of SIFTUP, the parent nodes of nodes i and $i+1$ will be the same in half of the time; the grandparents of elements i and $i+1$ will be the same in 3/4th of the time, and so on. On average, therefore, we must expect the nodes in the path from node i to the root node to be present in the cache because the adding of elements to the heap proceeds from left to right through each level of the tree from the bottom up.

Theorem 3 Assuming that the heap is B-aligned, and that the cache can hold $\Omega(\log_2 n)$ blocks, the number of cache misses in Williams' algorithm is $\frac{2n}{B}$.

Proof In the worst case, adding the elements in the ℓ'th level of the heap involves accessing every block in every level from ℓ to the top, which is equal to $\frac{2^\ell}{B}$ blocks. Because the reference pattern is from left to right in each level

in the tree, each block is read only once under the assumption that the cache may hold $\Omega(\log_2 n)$ blocks (and that the replacement strategy of the cache is such that the last block accessed in each level is replaced as the last block of that level). For all levels in the heap, the combined number of blocks accessed becomes the sum of the block references, which is equal to $\sum_{i=1}^{\lceil \log_2 n \rceil} \frac{2^\ell}{B} < \frac{2}{B} n$. □

The number of blocks accessed is in fact slightly lower than $\frac{2}{B} n$ because the top $\log_2 B$ levels of the heap fits completely into one block which never, under the assumption that the cache may hold $\log_2 n$ blocks, will be replaced.

If the assumption that the heap is B-aligned does not hold, the additions of heap elements in level ℓ may result in 1 extra block being read. Therefore, an unaligned heap will have at least $\log_2^2 n$ more misses with this program than an aligned heap.

6.3.2 Bottom-up subtree heapification

Floyd's algorithm works its way up the tree, invoking SIFTDOWN for each subtree. In contrast to Williams' algorithm, temporal locality is generally absent because two adjacent invocations of SIFTDOWN will process two disjoint subtrees.

The i'th step in Floyd's program requires, in the worst case, that a path from heap element a_i to the bottom of the heap be traversed. The length of this path is $\lceil \log_2 n \rceil - \ell + 1$, where ℓ is the depth of heap element a_i, and for each step through the path, the program examines two adjacent elements. This totals $3n$ memory operations of which n are known to be spatially local due to Lemma 8. The spatial locality of the memory references for step i is now considered. We shall need the following lemma.

Lemma 10 The paths from heap element i to the bottom of the tree and heap element $i + 1$ to the bottom of the heap share at most $\log_2 B$ cache blocks.

Proof In the best case, element i is on a cache boundary. This means that elements i and $i+1$ share the same cache block in level ℓ, where ℓ is the depth of element i. As the paths to the bottom of the heap are disjoint, the distance between the elements in each successive level under ℓ will double with each level. If the distance between the elements in some level under ℓ is not smaller than B, the elements do not share the cache block in that level. The distance is exceeds B in level $\ell + \log_2 B$ because it doubles with each level. Therefore, the two paths share $\log_2 B$ cache blocks in the best case. □

This implies that the disjoint paths from all elements in some level ℓ to the

bottom of the tree must read every cache block in levels $\ell, \ldots, \ell + \log_2 B - 1$, and the same number of cache blocks as there are elements in level ℓ for each level $\ell + \log_2 B, \ldots, \lceil \log_2 n \rceil$. In Figure 6.1, the paths are shown in a fragment of a heap for a block size of 4. Two levels down from the top level in the heap fragment, no two paths share blocks.

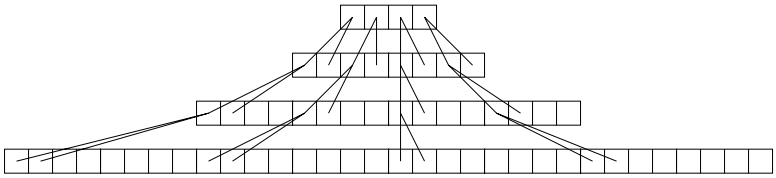

Figure 6.1: A trivial example of the paths traversed by Floyd's algorithm.

Floyd's algorithm works by scanning each level $\lfloor \log_2 n \rfloor, \ldots, 1$ from right to left. The spatial locality of the i'th step in Floyd's algorithm therefore depends on which blocks it may share with the $B - 1$ previous steps. This is presented for all steps in the lemma below.

Theorem 4 Assuming that the heap is B-aligned, and that the cache can hold $O(\frac{n}{B})$ blocks, the number of cache misses in Floyd's algorithm is bound by $\frac{n(\log_2 B + 2)}{B}$.

Proof We consider the bottom $\log_2 B$ levels and the top $z = \log_2 n - \log_2 B$ levels separately. The number of elements in the bottom $\log_2 B$ levels of the heap is $n - 2^z \leq n(1 - \frac{1}{B})$ elements, and the number of elements in the top z levels of the heap is $\frac{n}{B}$.

For the operations in level ℓ, $\ell \geq z$ (the bottom $\log_2 B$ levels), every block in level $\ell, \ldots, \lceil \log_2 n \rceil$ must be read in the worst case (Lemma 10). In level $h = \lceil \log_2 n \rceil - \ell + 1$, there are $n/2^h$ elements and $n/(2^h B)$ cache blocks. Thus, the number of blocks necessary to read is bound by

$$\begin{aligned}
\sum_{h=1}^{\log_2 B} \sum_{i=1}^{h} \frac{n}{2^i B} &= \Big(\sum_{h=1}^{\log_2 B} 1 - \frac{1}{2^h} \Big) \frac{n}{B} \\
&= \Big(\log_2 B - \sum_{h=1}^{\log_2 B} \frac{1}{2^h} \Big) \frac{n}{B} \\
&\leq \Big(\log_2 B + \frac{1}{B} - 1 \Big) \frac{n}{B}
\end{aligned}$$

For the operations in level ℓ, $\ell < z$ (the top $\lceil \log_2 n \rceil - \log_2 B - 1$ levels), every block in levels $\ell, \ldots, \ell + \log_2 B - 1$ must be read in the worst case (Lemma 10),

and $2^{\ell-1}$ blocks in each level $\ell+\log_2 B, \ldots, \lceil \log_2 n \rceil$ must be read. The number of cache blocks in levels $\ell, \ldots, \log_2 B - 1$ combined is $\sum_{i=0}^{\log_2 B-1} \frac{2^{\ell+i}}{B} = 2^{\ell}$, and the number of cache blocks in levels $\ell + \log_2 B, \ldots, \lceil \log_2 n \rceil$ that is necessary to read is $2^{\ell-1}(\lceil \log_2 n \rceil - \log_2 B - \ell)$. Thus, the number of blocks that are necessary to read is bound by

$$\begin{aligned}
\sum_{\ell=1}^{z-1} \left(2^{\ell} + 2^{\ell-1}(z-\ell)\right) &= \sum_{\ell=1}^{z-1} \left(2^{\ell-1}(z+2)\right) + \sum_{\ell=1}^{z-1} \left(2^{\ell-1}\ell\right) \\
&\leq 2^{z-1}(z+2) - 2^{z-1}(z-1) \\
&= 2^{z-1} \times 3 \\
&= \frac{3n}{B}
\end{aligned}$$

To sum up, the bottom $\log_2 B$ levels of the heap results, in the worst case, in $\frac{n(\log_2 B+1/B-1)}{B}$ cache misses, while the top $\lceil \log_2 n \rceil - \log_2 B$ levels of the heap results in $\frac{3n}{B}$ misses. Combined, the total number of cache misses for Floyd's algorithm is bound by $\frac{n(\log_2 B+2)}{B}$. □

This proves that Floyd's algorithm, while asymptotically superior to Williams' algorithm with respect to the number of key comparisons, has a larger number of cache misses.

6.3.3 Post-order bottom-up subtree heapification

The variant of Floyd's algorithm which orders the SIFTUP operations as a post-order traversal of the tree is now considered. The ordering of the operations means that the left and right subtree of a node i have been processed just before processing node i.

Theorem 5 Assuming that the heap is B-aligned and that the cache can hold $O(B)$ blocks, the number of cache misses in post-order bottom-up heap construction is $\frac{n}{B}$.

Proof As in the proof for Theorem 4, the bottom $\log_2 B$ levels are considered separately. The same number of blocks will be read as for the regular Floyd algorithm, but here, under the assumption that the cache can hold $2^{\log_2 B} = B$ blocks, the two subtrees will be present in the cache for the operations in the bottom $\log_2 B$ levels of the heap. The number of cache misses in the bottom $\log_2 B$ levels is therefore bound by

$$\sum_{i=1}^{\log_2 B} \frac{2^{\lceil \log_2 n \rceil - i}}{B} < \frac{n}{B}$$

The top $\lceil \log_2 n \rceil - \log_2 B$ levels contain $\frac{n}{B}$ elements. The proof from Theorem 4 states that an upper bound for cache misses for these elements is $\frac{n}{B}$ cache misses. The post-order does at least as well as the traditional Floyd's algorithm, and therefore the number of cache misses for the post-order method is bound by $\frac{2n}{B}$ in total. □

Changing the order of the SIFTDOWN operations thus produces a better upper bound.

6.3.4 Median partitioning heap construction

The partitioning accesses memory in a sequential manner. It was shown in Section 6.2.4 that the average-case number of memory references for the complete selection was $8n + o(n)$. Because the partitioning is strictly sequential, the number of cache misses is directly related to the number of memory references, and it is therefore possible to divide the memory reference count by B, yielding an estimate for the cache misses for the median partitioning of $\frac{8n}{B} + o(n)$. The sequential reference pattern leads to a requirement of the cache capacity of $O(1)$ blocks only.

The upper bound is in fact conservative because some temporal locality exists between two consecutive calls to the partition procedure, especially in the later stages of the heap construction.

6.4 Referential footprints

The referential footprints of the four programs for a heap of 256 elements, for randomly permuted input data, is given in Figures 6.3 to 6.6. These footprints are now discussed.

For the Floyd-based heap construction program, the reference pattern (Figure 6.3) shows lines emerging from the top. The first two lines from the left represent the subtree element i to heapify (line 1 in Algorithm 2) and the two children $\mathit{left}(i) = 2i$ and $\mathit{right}(i) = 2i + 1$ of this element (line 4 in the algorithm), explaining the lower gradient of the second line. About 300 references into the pattern, the next line, representing heap level $i + 2$, emerges at the top, and each line introduced along the temporal edge represents a new, lower heap level visited for element i. Towards the end, i.e., the top levels of

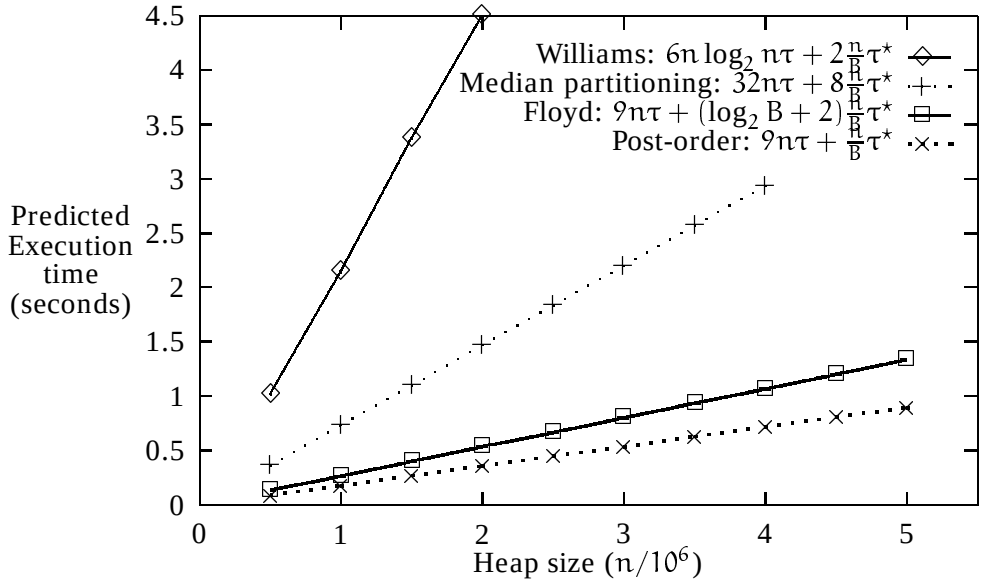

Figure 6.2: Predicted worst-case performance ($B = 8, \tau = 17\text{ns}, \tau^\star = 10\tau$).

the tree, the distance between the dots become greater, reflecting both a poor temporal and spatial locality because two subtrees traversed in succession will have no elements in common.

The post-order heap construction shows a different picture in the footprint in Figure 6.4. Each slowly falling pattern from the left represents SIFTDOWN operations in each level of the tree. As the two subtrees of every node are processed before the node itself, these patterns are not straight but exhibit a waveform representing the path to the bottom level for each SIFTDOWN operation. It is evident from the footprint picture that these waves represent temporal locality because, in contrast to the footprint of the Floyd-based program, the distance between two dots on the horizontal axis is, on average, small.

The locality of Williams' heap construction approach is evident from Figure 6.5. The solid, steepest line corresponds to line 1 in algorithm 1, which is the reading of each new element i to be added to the heap. The next, second steepest line plots the references to *parent*(i), which is at position $i/2$ for binary heaps. The third shows references to the grandparent, and so on. The lower lines are not contiguous because SIFTUP rarely reaches very far up the tree on average (on random input) before it terminates. When it does reach far up the tree, it has the effect in the graph of disturbing the steepest line by stretching it on the temporal axis. There are $\lceil \log_2 n \rceil$ lines in the graph, corresponding to the depth of the heap.

The heap construction program based on median partitioning requires reading

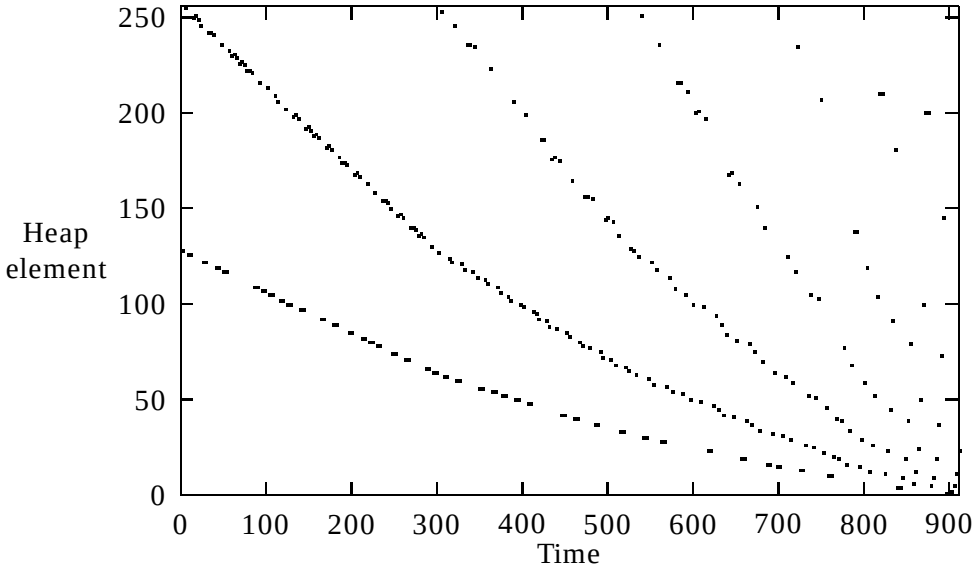

Figure 6.3: Footprint of Floyd's method ($n = 256$).

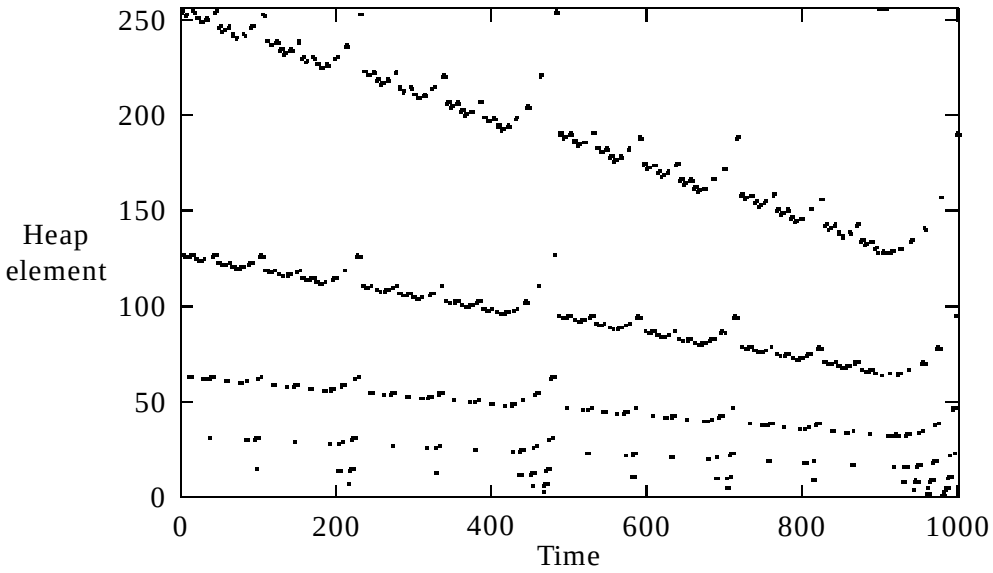

Figure 6.4: Footprint of the post-order Floyd ($n = 256$).

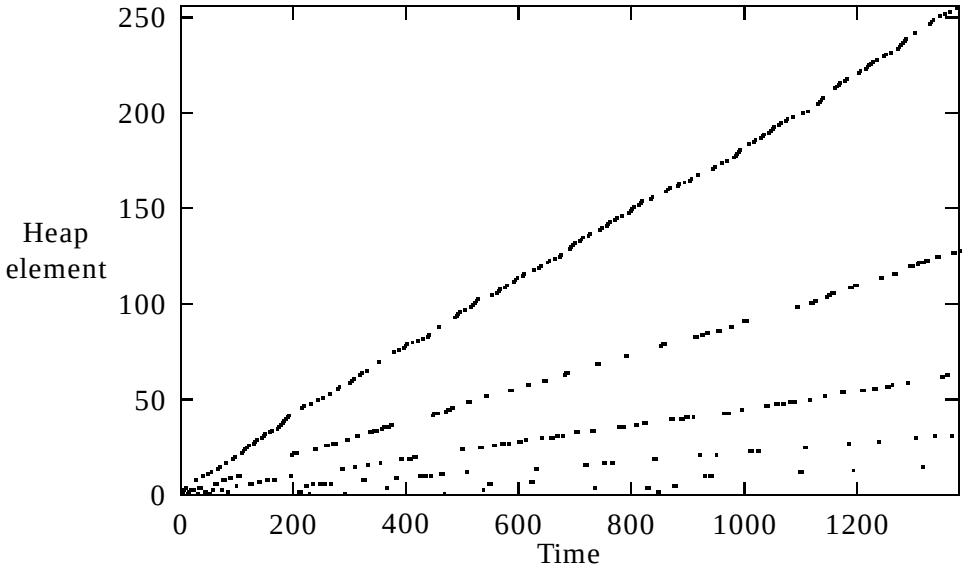

Figure 6.5: Footprint of Williams' method ($n = 256$).

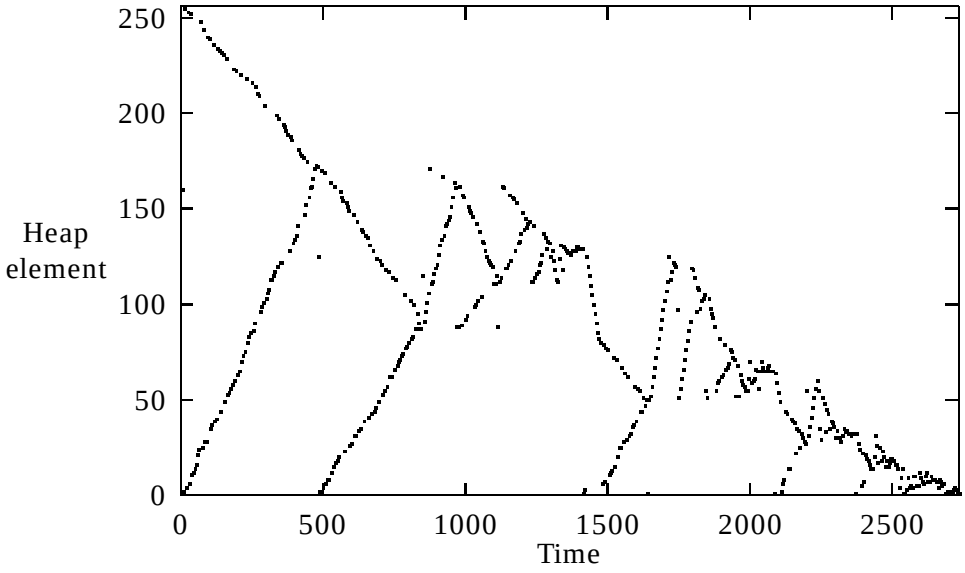

Figure 6.6: Footprint of the median partitioning ($n = 256$).

the pivot element, which is assumed to be an arbitrary element in the array $[a_1, \ldots, a_n]$. Apart from this operation, the algorithm has a sequential reference pattern: The array is read from a_1 through a_k (the i index) and from a_n down to a_k (the j index). Each consecutive invocation of SELECT continues from the same indices that the previous invocation left and the spatial locality property of the this heap construction method is therefore high. This is supported by the referential footprint picture of the program (Figure 6.6).

Program	Reads	Writes	Total	Predicted
Williams	1 802	1 802	3 604	3 584
Floyd	750	375	1 125	1 024
Median	1 651	591	2 242	2 048

Table 6.1: Reference counts for 256 elements.

In Table 6.1, the actual reference counts for the three programs are given for constructing heaps of 256 elements with descendingly sorted input data. The figures support the predictions from Section 6.2 that Williams' algorithm accesses $2(n \log_2 n$ elements, Floyd's accesses $3n$ elements and the median-partitioning algorithm accesses $8n$ elements (as the median-partitioning reference count varies as a result of the randomized pivot selection, the read and write figures in Table 6.1 is averaged over several executions). Throughout the assessments of miss counts and reference counts in this section, the figures have been a few percent lower than expected, which is due to the locality of the top few levels of the heap that fit completely into a single cache block.

6.5 Heap construction experiments

The four heap construction algorithms have been implemented (see Appendix B.4) and the running times have been measured on the AXP reference machine for heaps of size 500 000 to 5 000 000 elements of 4 bytes each, and the cache behaviour in a 32kb direct-mapped cache has been simulated for heaps of size 5 000 to 50 000, again with 4 bytes per element. The cache blocks in both the AXP reference machine and the simulation is 32 bytes, which means that a cache block will hold 8 elements. The experiments have been performed with sorted as well as permuted input data. On the AXP reference machine, an array of 5 million word-sized integers is more than will fit in the internal memory. The results for the experiments with large heaps will therefore also represent some effects from paging.

The predicted worst-case performance for the four programs is shown in Figure 6.2 with the assumption that 8 elements fit in a cache block, that $\tau = 17$ns, and that $\tau^*/\tau = 10$.

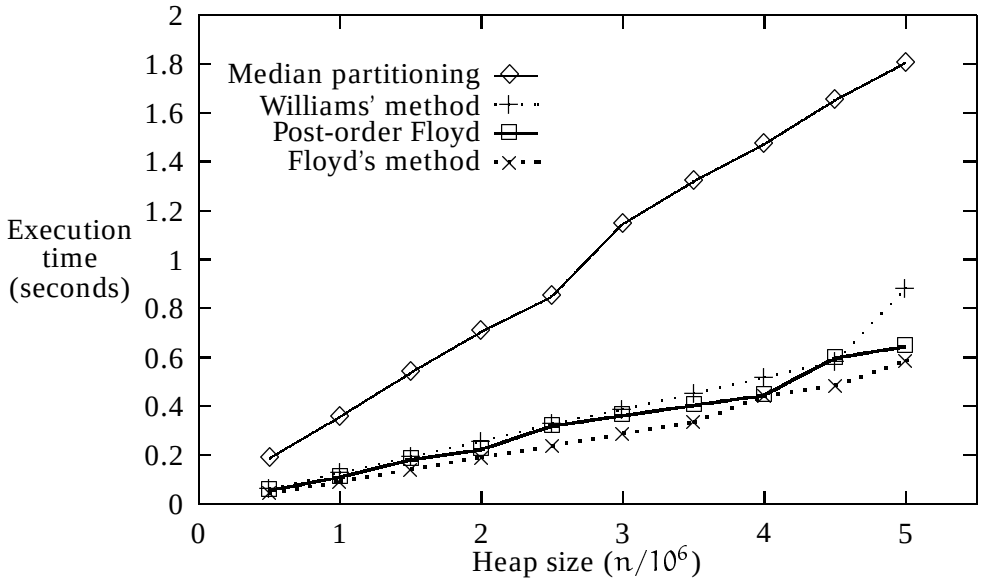

Figure 6.7: Heap construction (ascending input).

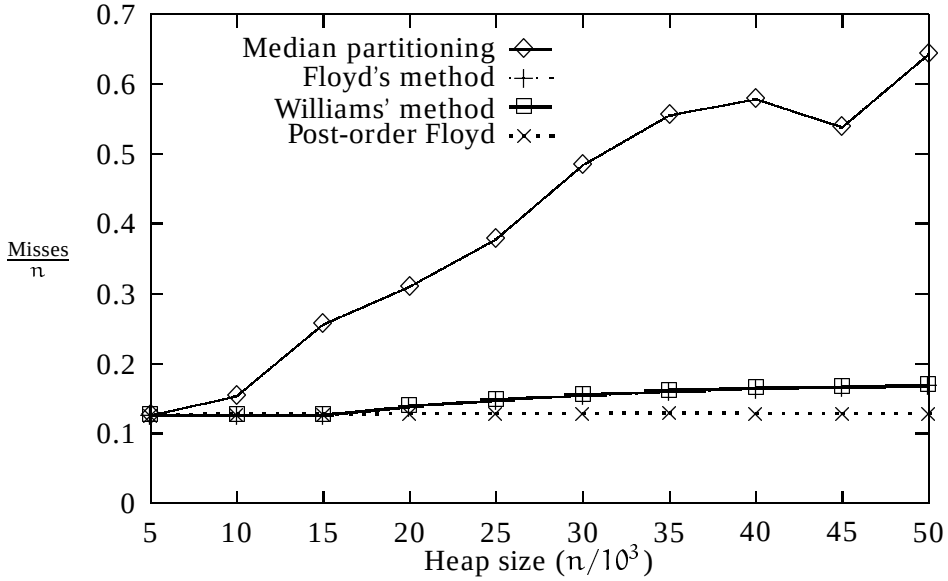

Figure 6.8: Miss counts (32kb direct-mapped cache, ascending input).

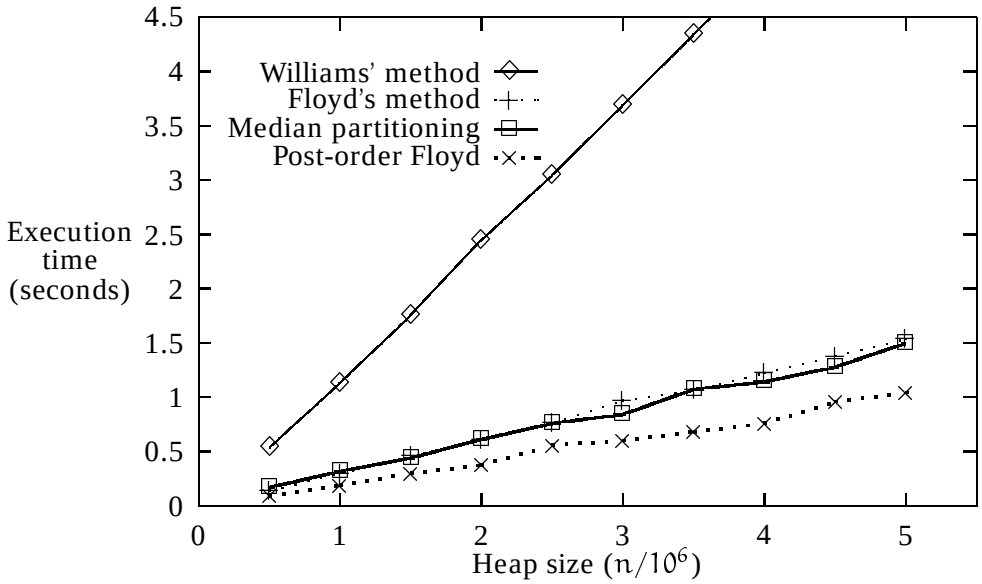

Figure 6.9: Heap construction (descending input).

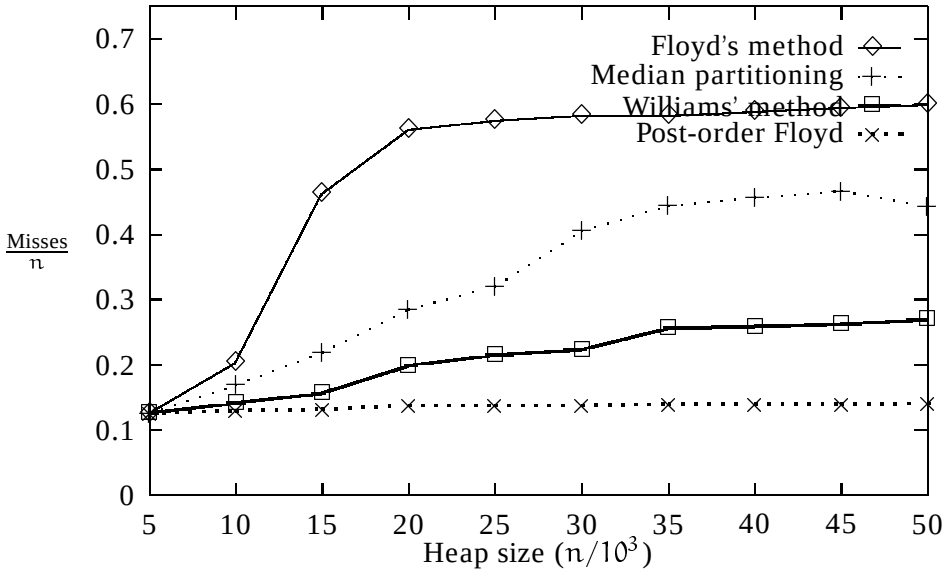

Figure 6.10: Miss counts (32kb direct-mapped cache, descending input).

In Figures 6.7 and 6.8, the input array to the heap construction programs consisted of ascendingly sorted values, in Figures 6.9 and 6.10, the input was sorted on decreasing keys, and Figures 6.11 and 6.12 shows the results from randomly permuted input data[1].

The experiment with the input array sorted in ascending order, that is, an array already satisfying the heap property, all but the median-partitioning program run in approximately the same time. The three other programs are based on the heap operations SIFTUP and SIFTDOWN and perform no actual interchanges of elements in the heap, and access only elements in adjacent levels of the heap. The cache simulation for this experiment shows that the programs based on Williams' and Floyd's algorithms have a close-to-constant number of cache misses per element, rising from just over 0.125 (which is the ideal $\frac{n}{B}$ for the cache metrics and type sizes) to about 0.17 for the largest input array. The program based on a post-order traversal of the heap has constant number of cache misses. The instruction count of the post-order heap construction is higher than the traditional Floyd-based program, which means that, in this experiment, Floyd's algorithm is the fastest. The median-partitioning program has a worst-case cache performance in a situation where the input array is already sorted, and the miss rate seems to rise steadily to about 0.7 misses per element. The prediction for a cache that can hold 8 elements in a block is 1 cache miss per element, though, as discussed in Section 6.3.4, some temporal locality has not been accounted for.

For descendingly sorted input data, no single element in the heap satisfies the heap property. This is the worst-case input for the programs based on Williams' and Floyd's algorithms. In this experiment, the program based on Williams' algorithm needs to travel to the root for every invocation of SIFTUP, and this program clearly performs much worse than any of the other three. The median-partitioning and the program based on Floyd's traditional algorithm are nearly equally fast, while the program based on post-order traversal is about 50% faster. The cache simulation shows that the post-order traversal again has close to a constant number of cache misses per element over the entire range, corresponding to a near-perfect utilization of the cache. Williams' algorithm has, due to its high degree of temporal locality, about twice the number of misses, about 0.25 cache misses per element, which is exactly as predicted by Theorem 3, namely $\frac{2}{B}$ misses per element, for a cache block that will hold 8 elements. Although the miss count for Williams' method is lower than that for Floyd's and the median-partitioning, the number of instructions necessary for the full traversal to the root for each element added to the heap is does not make Williams' program faster. The miss rate for the Floyd-based program evens out at about 0.6 misses per element, which is very close to the

[1]The permutation is performed by interchanging element $a_i, i \in \{1, \ldots, n\}$ by element a_j, where j is randomly chosen in $\{1, \ldots, i\}$.

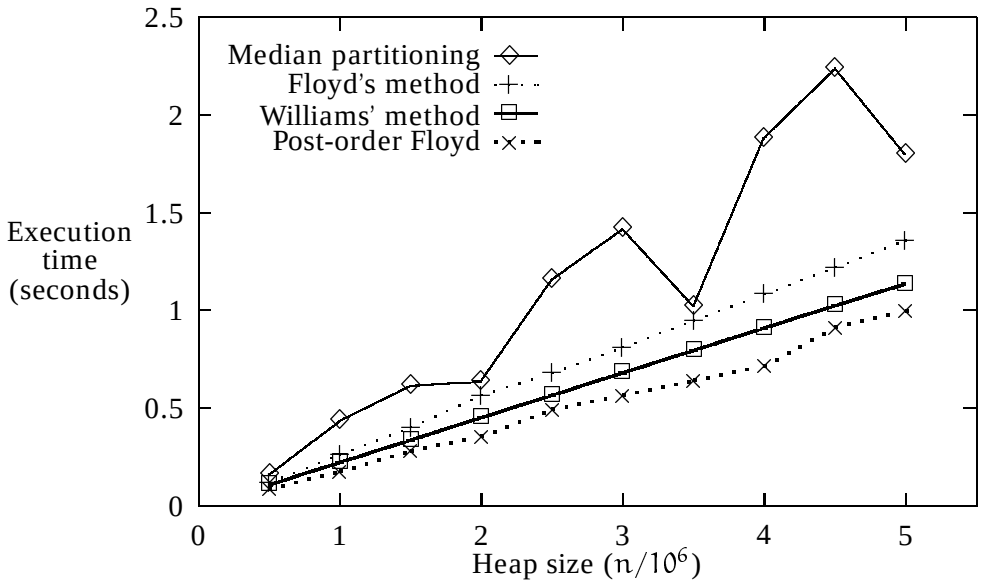

Figure 6.11: Heap construction (permuted input).

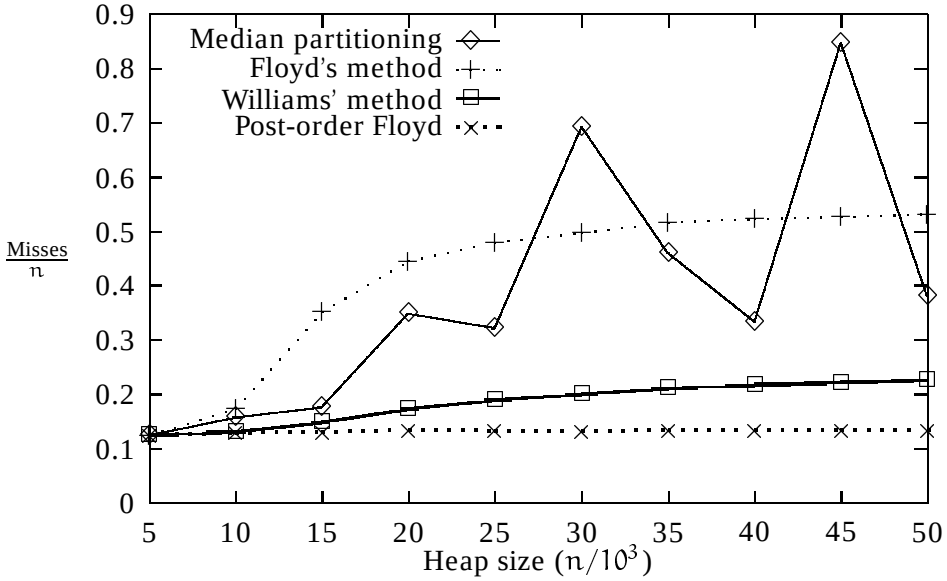

Figure 6.12: Miss counts (32kb direct-mapped cache, permuted input).

predicted $\frac{\log_2 B+2}{B}$ predicted by Theorem 4 for a cache block that will hold 8 elements.

The experiments with randomly permuted input (see Figures 6.11 and 6.12) show that the post-order Floyd-based program performs best, and that the program based on Williams' algorithm performs better than the one based on Floyd's. The raggedness of the median-partitioning, which is slowest, is apparently caused by the randomization in selecting the pivot. Comparing the miss rates for the program based on Floyd's method in Figures 6.10 and 6.12 shows little difference in the miss rates for worst-case and average-case input, which is evidence that this program, on average, must travel to the bottom of the heap for every SIFTDOWN operation performed. The program based on Williams' method has slightly fewer misses in the average case than the worst case but shows a dramatic improvement in execution time, which is evidence that this program, on average, travels only two to three levels of the heap for each SIFTUP operation performed.

For all experiments performed, the post-order implementation of Floyd's algorithm has a constant $\frac{n}{B}$ cache misses, or close to the ideal 0.125 misses per element, and is the fastest of the heap construction programs for any input.

6.6 Conclusion

Four heap construction methods have been analysed and their performance have been measured:

- Post-order bottom-up heap construction is fastest on worst-case as well as average input and has a miss count close to the ideal $\frac{1}{B}$ per element regardless of the input, lower than for any of the other. The post-order heap construction requires $O(B)$ cache blocks.
- Floyd's bottom-up heap construction is fastest on best-case input (an array that already satisfies the heap condition) and has a miss count of $\frac{n \log_2 B+2}{B}$ for a cache with $O(\frac{n}{B})$ blocks.
- Williams' top-down heap construction is faster than Floyd's on average-case input and has a miss count of $\frac{2n}{B}$ for a cache with $\Omega(\log_2 B)$ blocks.
- Median partitioning heap construction is slowest for average-case and worst-case input (which in this case is an input array already satisfying the heap condition) and has a miss count of $\frac{8n}{B}$ for a cache of $O(1)$ blocks.

In Table 6.2, the analyses for the four programs are summarised. In the columns representing the number of Pure C operations and the number of cache misses estimated, the figures are assumed to have an additional $O(n)$ operations.

Program	Worst-case asymptotical	Instruction count	Blocks required	Miss count
Williams	$\Theta(n \log_2 n)$	$6n \log_2 n\tau$	$\Omega(\log_2 n)$	$2\frac{n}{B}\tau^\star$
Floyd	$\Theta(n)$	$9n\tau$	$O(\frac{n}{B})$	$\frac{n(\log_2 B+2)}{B}\tau^\star$
Post-order	$\Theta(n)$	$9n\tau$	$O(B)$	$\frac{n}{B}\tau^\star$
Median	$O(n)$	$32n\tau$	$O(1)$	$8\frac{n}{B}\tau^\star$

Table 6.2: Summary of heap construction programs.

It has been shown that traditional analysis of programs based on the complexity in terms of the number of some key operation, e.g., a comparison, is inaccurate if the cache effects are disregarded. This supports the claims made in the introductory chapter that algorithms are not designed with locality issues in mind, and that there is some disorder between theory and practice.

Cache effects have been shown to be a dominating source of "random noise," and to be predictable. Spatial as well as temporal locality has in this chapter been simple to predict accurately, because the reference patterns of most of the programs was given. This is not true for all programs. Further, it has been shown that there is a limit to the strategy of optimizing reference locality at the expense of instruction cost, which suggests that any method for analysis of locality must include instruction counts as well as miss counts.

A final comment relates to the interaction of algorithms in the cache. Many algorithms, such as heaps or set unions, are used as underlying, abstract data types by some other algorithm, effectively intermixing the reference patterns of the two programs. Some interference must be expected in real applications, and the results presented until now are therefore approximate. The block requirement of $O(1)$ in the median partitioning method, however, will not suffer from such interaction, and may, in spite of its inferior performance, have it uses.

Chapter 7

Cache behaviour of mergesort

The memory reference behaviour of sequential *mergesort* is studied. Mergesort is chosen because it is the method of choice for external-memory sorting due to its sequential access behaviour. The sequential access behaviour is analysed according to the memory-access cost model of Chapter 5.

A textbook [18] mergesort algorithm is chosen as the base algorithm with a few traditional optimizations. The standard iterative mergesort performs $\lceil \log_2 n \rceil$ passes, where the i'th pass merges adjacent, sorted subarrays of size 2^{i-1} into sorted subarrays of size 2^i. Special cases exist for the last subarrays to be merged in each pass, and for the last pass itself. Mergesort requires an auxiliary array to store the merged subarrays in each pass, and has thus a space requirement of $2nz + O(1)$, where z is the record size.

7.1 Merging sorted subarrays

The critical procedure in mergesort is the merge itself, which merges sorted input arrays to produce a sorted output array. In the following, merging of subarrays with various degrees is discussed.

7.1.1 two-way merge

If $[a_1, \ldots, a_n]$ denotes the input array and $[b_1, \ldots, b_n]$ denotes the output array, then the merging of the two sorted input subarrays $[a_l, \ldots, a_{m-1}]$ and $[a_m, \ldots, a_{r-1}]$ is performed by scanning from left to right, moving one element at a time to the output array, producing the sorted subarray $[b_l, \ldots, b_{r-1}]$. The merge is presented as Algorithm 9 and shown in a Pure C implementation in

Figure 7.1. The first loop merges the input arrays until one is exhausted, and then one of the two last loops copies the remaining elements to the output array. The complexity of MERGE is trivially $\Theta(n)$, where n is the size of the output array.

The Pure C implementation employs a few simple optimizations. First, the elements of the input arrays are read ahead, which means that each iteration begins with the variables u and v being the two current elements to compare. This restricts the number of memory references such that each element in the input arrays are read exactly once, and each element in the output array is written exactly once. Second, the outer while loop is split in two parts to eliminate the if-else structure. Finally, the implementation uses pointers instead of array indices.

Algorithm 9 (*Merge)*

MERGE$([a_l, \ldots, a_{m-1}], [a_m, \ldots, a_{r-1}], [b_l, \ldots, b_{r-1}])$:

```
1:  i = l
2:  j = m
3:  k = l
4:  while (i < m and j < r) do
5:    if (a_i < a_j) then
6:      b_k = a_i
7:      i = i + 1
8:    else
9:      b_k = a_j
10:     j = j + 1
11:   end if
12:   k = k + 1
13: end while
14: while (i < m) do
15:   b_k = a_i
16:   i = i + 1
17:   k = k + 1
18: end while
19: while (j < r) do
20:   b_k = b_j
21:   j = j + 1
22:   k = k + 1
23: end while
```

Property 5 The Pure C cost of a merge is less than $(7s + O(1))\tau$, where s is the size of the output array.

```
MERGE2(int* up, int* vp, int* dp, int k) {
          1: int* lu = up + k;
          2: int* lv = vp + k;
          3: u = *up;
          4: v = *vp;
          5: if (u > v) goto 11;

 6: *r++ = u;                11: *r++ = v;
 7: up++;                    12: vp++;
 8: if (p == lp) goto 17;    13: if (q == lq) goto 22;
 9: u = *up;                 14: v = *vp;
10: if (u <= v) goto 6;      15: goto 5;

16: v = *vp;                 21: u = *up;
17: *r++ = v;                22: *r++ = u;
18: vp++;                    23: up++;
19: if (q < lq) goto 16;     24: if (p < lp) goto 21;
20: return;                  25: return; }
```

Figure 7.1: Pure C implementation of a two-way MERGE.

The Pure C implementation merges the two sorted input arrays pointed to by up and vp, each of length k into the sorted output array pointed to by dp of size $s = 2k$. The dominating part of the program is the loop in line 5–10 (11-16), which carries 6 (7) Pure C instructions. This loop is iterated s' times, where $s/2 \leq s' < s$ times, corresponding to one complete input array and zero or more elements of the other input array having been copied to the output array. The trailing loop (lines 16–19 or 21–24) copies the element left in the non-exhausted array to the output array and carries 4 Pure C instructions. This loop is iterated $s - s'$ times. The bound on s' means that the merging loop may be iterated $s-1$ times, each of at least 7 instructions. Therefore, the Pure C cost of a merge is $(7s + O(1))\tau$. The implementation and analysis of merge is a slight simplification to the general case where the sizes of the input arrays may not be equal.

The Pure C merge may be improved further under the assumption that the subarray with the smallest tail is known, in which case the bounds checking is necessary only in that array. This saves little in this implementation, but in an m-way merge, it may save a significant number of instructions.

Property 6 MERGE has a memory access cost of $\lceil \frac{2s-3}{B} + 3 \rceil \tau^\star$, where $s = (r-l)$, that is, the size of the resulting sorted subarray.

Proof In weighted Pure C terms, MERGE contains 3 memory tracks, namely,

$T_1 = [a_l, \ldots, a_{m-1}]$, $T_2 = [a_m, \ldots, a_{r-1}]$, and $T_3 = [b_l, \ldots, b_{r-1}]$. By Definition 7, we have $\rho = 3$, $\sigma_1 = m - l - 1$, $\sigma_2 = r - m - 1$, and $\sigma_3 = r - l - 1$. By Theorem 1, the memory access cost of the memory tracks in MERGE is $\sum_{i=1}^{\rho} \lceil \frac{\sigma_i}{B} + 1 \rceil \tau^\star = \lceil \frac{2(r-l)-3}{B} + 3 \rceil \tau^\star$. □

7.1.2 Three-way and four-way merge

A two-way merge is a special case of an m-way merge which, generally, merges m sorted input arrays of size k into a sorted output array of size km. For relatively small values of m, the current elements of the input arrays may be held in registers and used in the same fashion as a two-way merge, this time only comparing more elements at a time. For larger values of m, a data structure such as a selection tree or a heap may be necessary to keep the current elements sorted.

In Figure 7.2, a three-way merge is presented that spends less than $7 + O(1)$ Pure C instructions for each element in the output array. This merge receives three pointers to input arrays, `up`, `vp` and `xp`, and places the sorted elements in the output array initially pointed to by `dp`. It is further assumed in the program that the input array pointed to by `xp` has the smallest tail of the three, eliminating the need for bounds checking in the two other input arrays. The three-way merge thus finishes when all elements of `xp` have been merged and copied to the output array. As the two non-exhausted input arrays still contain elements to be merged, the three-way merge falls back on a two-way merge for these elements.

Similarly, a four-way merge may be constructed to merge input arrays in groups of four. Again, assuming that the input array with the smallest tail is known, bounds checking is only necessary for this array, and the four-way merge falls back on the three-way merge for the remaining elements in the three non-exhausted arrays. A four-way merge is listed in Appendix B.5 with a Pure C cost of less than $(7s + O(1))\tau$ where s is the size of the output array.

A general m-way merge may be realised using a heap of size m, where each heap element contains a pointer to a position in an input array, and the comparison between two elements in the heap involves comparing the keys in the input arrays to which they refer. Each iteration of merge uses the root element of the heap to obtain the smallest of the m input array elements, then increments the pointer in the root element of the heap, and performs a SIFT-DOWN(1) operation. With the addition of a mechanism for marking exhausted input array pointers, this constitutes m-way merge in $O(s \log_2 m)$ time, where s is the size of the output array.

```
MERGE3(int* up, int* vp, int* xp, int* dp, int k) {
           1: int* upl = up + k;
           2: int* vpl = vp + k;
           3: int* xpl = xp + k;
           4: int u = *up, v = *vp, x = *xp;
           5: if (u <= v) goto 10;
           6: goto 18;

 7: *dp++ = u;                  15: *dp++ = v;
 8: u = *++up;                  16: v = *++vp;
 9: if (v < u) goto 18;         17: if (u <= v) goto 10;
10: if (u <= x) goto 7;         18: if (v <= x) goto 15;
11: *dp++ = x;                  19: *dp++ = x;
12: if (++xp == xpl) goto 23;   20: if (++xp == xpl) goto 22;
13: x = *xp;                    21: x = *xp;
14: goto 10;                    22: goto 18;

           22: MERGE(up,vp,dp,upl,vpl); }
```

Figure 7.2: Pure C implementation of a three-way MERGE.

7.2 Bottom-up mergesort

The merge procedures presented above are now applied in a two-way and a four-way mergesort, and the Pure C cost of each program is calculated.

7.2.1 Two-way mergesort

In Algorithm 10, MERGE is applied for adjacent subarrays of size $k = 2^{i-1}$ in lines 4–7 in the i'th iteration of the outer loop. If n is not a power of 2, a special case exists for the remaining n **mod** $2k$ elements. If n **mod** $2k \geq k$, MERGE is applied for two subarrays of size k and n **mod** k (line 9), otherwise there is only one (already sorted) subarray of size n **mod** k, which is simply copied from a to b (lines 11–14). This is all done for iterations $i = 1, 2, \ldots, \lceil \log_2 n \rceil$ (for $k = 1, 2, 4, 8, \ldots$).

Each pass copies the entire contents of the input array a to the output array b. To avoid unnecessary copying of the elements back to a to prepare for the next pass, the two arrays are "interchanged" in line 17. This does not physically interchange any of the elements in the two arrays, rather, the $a \leftrightarrow b$ simply implies swapping the identity of the two arrays. The consequence is that the two arrays alternate in being the source and the destination array in each

pass of MERGESORT. The user of MERGESORT must be aware of this property of MERGESORT because the placement of the sorted output depends on n. Assertions of which of the two is the new auxiliary array may be necessary in the user's code[1]. The cost of MERGESORT is trivially $\Theta(n \log_2 n)$.

Algorithm 10 (*Mergesort)*

MERGESORT$([a_1, \ldots, a_n], [b_1, \ldots, b_n])$:
1: $k = 1$
2: **while** $(k < n)$ **do**
3: $j = 1$
4: **while** $(j < n - 2k)$ **do**
5: MERGE$([a_j, \ldots, a_{j+k}], [a_{j+k}, \ldots, a_{j+2k}], [b_j, \ldots, b_{j+2k}])$
6: $j = j + 2k$
7: **end while**
8: **if** $(j + k < n)$ **then**
9: MERGE$([a_j, \ldots, a_{j+k}], [a_{j+k}, \ldots, a_n], [b_j, \ldots, b_n])$
10: **else**
11: **while** $(j < n)$ **do**
12: $b_j = a_j$
13: $j = j + 1$
14: **end while**
15: **end if**
16: $k = 2k$
17: $a \leftrightarrow b$
18: **end while**
19: **return** $[a_1, \ldots, a_n]$

Property 7 The Pure C cost of mergesort is $(7n \log_2 n + O(n))\tau$.

The Pure C merge spends $7s + O(1)$ instructions in each merge. Pass i of mergesort merges $n/2^{i-1}$ subarrays each of size 2^{i-1} into $n/2^i$ subarrays of size $s = 2^i$. The number of Pure C instructions executed in pass i of mergesort is therefore $7n + o(n)$, and, for all $\lceil \log_2 n \rceil$ passes, this sums up to $7n \log_2 n + O(n)$ Pure C units.

[1]This may be avoided by several means. If the number of iterations performed is odd, a final copy may be done to ensure that the sorted elements are placed in the original output array. Alternatively, the *first* pass may instead be performed in-place if it is determined in advance that an odd number of iterations is needed.

7.2.2 Four-way mergesort

The only difference between the two-way mergesort presented and a four-way mergesort is that four input subarrays have to be passed to the merge procedure, and that the size of each output subarray s is now 4^i. This means that the size of the subarrays quadruple through each iteration and, consequently, that a four-way mergesort will need $\lceil \log_4 n \rceil$ iterations.

A four-way bottom-up mergesort inspired by [16] is presented as Algorithm 11 with the special case merge for input sizes different than some power of 4 omitted. The four-way mergesort is based on a four-way merge which receives four input arrays of size k and returns an output array of size $s = 4k$, which is listed as a Pure C program in Appendix B.5.

Algorithm 11 (*Four-way mergesort)*

MERGESORT4$([a_1, \ldots, a_n], [b_1, \ldots, b_n])$:
1: $k = 1$
2: **while** $(k < n)$ **do**
3: $j = 1$
4: **while** $(j < n - 4k)$ **do**
5: MERGE4$([a_j, \ldots, a_{j+k}]$,
6: $[a_{j+k}, \ldots, a_{j+2k}], [a_{j+2k}, \ldots, a_{j+3k}]$,
7: $[a_{k+3k}, \ldots, a_{j+4k}], [b_j, \ldots, b_{j+4k}])$
8: $j = j + 4k$
9: **end while**
10: $k = 4k$
11: $a \leftrightarrow b$
12: **end while**
13: **return** $[a_1, \ldots, a_n]$

Property 8 The Pure C cost of four-way mergesort is $(3.5n \log_2 n + O(n))\tau$.

The Pure C four-way merge spends, in the worst case, $7s + O(1)$ instructions in each merge. Pass i of the four-way mergesort merges $n/4^{i-1}$ subarrays each of size 4^{i-1} into $n/4^i$ subarrays of size $s = 4^i$. The number of Pure C instructions executed in pass i of mergesort is therefore $7n + o(n)$, and, for all $\lceil \log_4 n \rceil$ passes, this sums up to $7n \log_4 n + O(n) = 3.5n \log_2 n + O(n)$ Pure C units, that is, half the time of the two-way mergesort.

7.3 Reference locality properties

The memory-access cost of the two-way and four-way mergesort is now considered, and compared to *tiled* mergesort, which performs the sort in two distinct stages for increased cache efficiency.

7.3.1 Two-way mergesort

The two-way bottom-up mergesort is considered in two phases. First, the memory access cost of MERGE is considered without regard to spatial locality between consecutive applications. Second, the memory tracks in MERGE are extracted and combined to become memory tracks in MERGESORT with the assumption that the arrays are aligned on cache block boundaries. Ideally, due to property 6, each pass should cost $(\frac{2n}{B} + o(1))\tau^\star$, and the complete mergesort cost $(\frac{2n\log_2 n}{B} + o(n))\tau^\star$.

Lemma 11 The $i'th$ pass of MERGESORT for an array of size n has a combined memory access cost in MERGE of $\frac{n}{B}(2 + \frac{3(B-1)}{2^i})\tau^\star$.

Proof The i'th pass of MERGESORT merges subarray pairs of size 2^{i-1} each into subarrays of size $s = 2^i$. MERGE is thus called $\frac{n}{s}$ times in the i'th pass. Each MERGE has, by Lemma 6 $\lceil\frac{2s-3}{B} + 3\rceil$ memory references. The number of cache misses in the i'th pass is therefore

$$\Big(\frac{2s-3}{B} + 3\Big)\Big(\frac{n}{s}\Big) = \frac{n}{B}\Big(2 + \frac{3(B-1)}{s}\Big).$$

□

The term $3(B-1)/2^i$ may be large for small i. For all $\lceil\log_2 n\rceil$ passes in MERGESORT, the number of cache misses becomes

$$\begin{aligned}\sum_{i=1}^{\lceil\log_2 n\rceil}\Big(\frac{n}{B}(2 + \frac{3(B-1)}{2^i})\Big) &\leq \frac{n}{B}\Big(2\lceil\log_2 n\rceil + 3(B-1)\Big)\\ &= \Big(\frac{n(2\lceil\log_2 n\rceil - 3)}{B} + 3n\Big)\\ &= \frac{2n\log_2 n}{B} + o(n)\end{aligned}$$

The memory tracks of MERGE are now considered as memory tracks in MERGESORT instead. Observe that, from the point of view of MERGESORT, the output array is accessed in one single memory track because each application

of MERGE continues from the point in the array where the previous MERGE ended. Furthermore, the memory track for the left subarray to be merged in the a array is a continuation of the memory track for the right subarray of the previous iteration. This is trivially seen in lines 5–6 of Algorithm 10. First, the arrays are assumed to be aligned on cache boundaries:

Lemma 12 If the arrays are B-aligned, any subarray at position $2^{i-1}j$ will, for any integer j, be B-aligned if $i > \log_2 B$ and B is a power of 2.

Proof By Definition 1, the first element in the array is B-aligned. If B is a power of 2, then $z = \log_2 B$ is an integer. Subarrays at position $2^{i-1}j$, $i > z$, will therefore, by definition also be B-aligned because $2^{i-1}j \ \mathbf{mod}\ 2^z = 0$ for any j. □

This implies that a subarray of size 2^{i-1} where $i > \log_2 B$ exactly occupy an integral number of cache blocks.

Lemma 13 Assuming the arrays are aligned on cache block boundaries, the i'th pass of mergesort has a memory cost of $\frac{n}{B}(2 + \frac{3(B-1)}{n})\tau^\star$.

Proof The memory operations of the merges are divided into the operations in the three subarrays.

For the *output array* it is observed that each consecutive application of MERGE continues the memory track of the previous application and, therefore, all memory references to the output array constitute a single memory track of size n. The memory operations in a *left subarray* of the input array (i.e., the subarrays $[a_1, \ldots, a_{m-1}]$) continues the memory track of the right subarray of the previous application and, therefore, all memory references to the left subarrays of the input array constitute a single memory track of size $n/2$. The memory references to the *right subarrays* in the input array are considered in two distinct situations for iteration i of MERGESORT:

$i \leq \log_2 B$: The two subarrays to be merged are within the same cache block. The continuations of memory tracks for two invocations on MERGE will be within the same block, and for all subarrays smaller than $\log_2 B$, each block will be accessed only once.

$i > \log_2 B$: The two subarrays to be merged occupy an integral number of cache blocks (Lemma 12). This means that each block will be accessed only once.

The memory references to the right subarrays in the input array for the i'th iteration of MERGESORT in a B-aligned array therefore costs $(\frac{(n/2)-1}{B} + 1)\tau^\star$.

Combined with the memory tracks for the output array and the left subarray of the input array, the number of cache misses in the i'th pass of mergesort becomes

$$\begin{aligned} & \left(\frac{n/2-1}{B}+1\right)+\left(\frac{n/2-1}{B}+1\right)+\left(\frac{n-1}{B}+1\right) \\ = & \left(\frac{2n-3}{B}-3\right) \\ = & \frac{n}{B}\left(2+\frac{3(B-1)}{n}\right). \end{aligned}$$

□

For all $\lceil \log_2 n \rceil$ passes in MERGESORT, the number of cache misses is $\log_2 n$ times the cost of pass i, or

$$\begin{aligned} \sum_{i=1}^{\lceil \log_2 n \rceil} \frac{n}{B}\left(2+\frac{3(B-1)}{n}\right) &= \frac{n}{B}\left(2\log_2 n+\frac{3(B-1)\log_2 n}{n}\right) \\ &= \frac{2n\log_2 n}{B}+O(\log_2 n) \end{aligned}$$

7.3.2 Four-way mergesort

The four-way mergesort performs the same operations as the two-way mergesort, only in groups of four input arrays. By the same arguments as those used for the two-way mergesort, it is easy to see that, assuming cache block alignment, the number of memory operations performed by the four-way mergesort is precisely half of what the two-way mergesort uses. As the four-way mergesort performs half the number of iterations, and as each block of input and output is read exactly once each for each pass, the number of cache misses is half that of the two-way mergesort.

More precisely, the i'th pass of the four-way mergesort requires, in the worst case, $\frac{2n-5}{B}+5$ cache misses, giving the complete four-way mergesort a cache miss count of

$$\begin{aligned} \sum_{i=1}^{\lceil \log_4 n \rceil} \frac{n}{B}\left(2+\frac{5(B-1)}{n}\right) &= \frac{n}{B}\left(2\log_4 n+\frac{5(B-1)\log_4 n}{n}\right) \\ &= \frac{n\log_2 n}{B}+O(\log_4 n). \end{aligned}$$

In plain terms, the memory cost of a four-way mergesort is halved compared to a two-way mergesort.

7.3.3 Tiled four-way mergesort

The number of cache misses of two- and four-way mergesort is strictly bound to the number of scans made over the arrays. In *tiled* mergesort, the sort is performed in two stages. First, n/z subarrays of size $z < n$ are completely sorted, then the n/z sorted subarrays are merged as usual in $\log_4(n/z)$ iterations. The approach is similar to the blocked programs presented in Section 4.2.3. The tiled mergesort is shown as Algorithm 12 (which assumes that z is a power of 4) and listed as Pure C in Appendix B.5.

Algorithm 12 (*Tiled mergesort)*

TILEDMERGESORT$([a_1, \ldots, a_n], [b_1, \ldots, b_n], z)$:
1: **if** $(n \leq z)$**return** MERGESORT4$([a_1, \ldots, a_n], [b_1, \ldots, b_n])$
2: $i = 1$;
3: **while** $(i < n)$ **do**
4: MERGESORT4$([a_i, \ldots, a_{i+z}], [b_i, \ldots, b_{i+z}])$
5: $i = i + z$
6: **end while**
7: $k = z$
8: **while** $(k < n)$ **do**
9: $j = 1$
10: **while** $(j < n - 4k)$ **do**
11: MERGE4$([a_j, \ldots, a_{j+k}]$,
12: $[a_{j+k}, \ldots, a_{j+2k}], [a_{j+2k}, \ldots, a_{j+3k}]$,
13: $[a_{k+3k}, \ldots, a_{j+4k}], [b_j, \ldots, b_{j+4k}])$
14: $j = j + 4k$
15: **end while**
16: $k = 4k$
17: $a \leftrightarrow b$
18: **end while**
19: **return** $[a_1, \ldots, a_n]$

Lemma 14 The number of cache misses in tiled four-way mergesort is bound by $\frac{n(\log_2 n - \log_2 z + 2)}{B} + O(\log_4 n)$, where z is the tiling factor.

Proof The first stage of tiled mergesort performs n/z four-way mergesorts. Each mergesort sorts z elements from an input to an output array. These $2z$

elements fit in the cache and, consequently, it is necessary to read $2z/B$ blocks in each mergesort. This translates to a miss count of $2n/B$ for the first stage of tiled mergesort.

The second stage performs n/z iterations as a usual four-way mergesort. The number of misses in the second stage is therefore

$$\begin{aligned}\frac{2n\log_4\left(\frac{n}{z}\right)}{B}+O(\log_4 n) &= \frac{2n(\log_4 n-\log_4 z)}{B}+O(\log_4 n)\\ &= \frac{n(\log_2 n-\log_2 z)}{B}+O(\log_4 n)\end{aligned}$$

To sum up, the number of misses in the complete tiled mergesort is

$$\frac{n(\log_2 n-\log_2 z+2)}{B}+O(\log_4 n)$$

□

For large blocking factors, this is clearly an improvement over a four-way mergesort. The Pure C unit cost is necessary to predict the performance of tiled mergesort and is given below.

Lemma 15 The Pure C cost of tiled mergesort is $(7n\log_2 n+O(n))\tau$.

Proof The first stage of tiled mergesort performs n/z four-way merges, each sorting z elements with a cost of $3.5z\log_2 z+O(z)$ Pure C units. For all n/z applications of mergesort, the Pure C cost of the first stage of tiled mergesort sums up to $3.5n\log_2 z+O(n)$ Pure C units.

The second stage of tiled mergesort performs $\lceil\log_4(n/z)\rceil$ iterations as a usual mergesort, each iteration carrying a Pure C cost of $7n+o(n)$. Combined, therefore, the number of Pure C instructions needed for all iterations in the second stage is

$$\left(7n+o(n)\right)\log_4\left(\frac{n}{z}\right)=3.5n(\log_2 n-\log_2 z)+O(n)$$

To sum up, the number of Pure C instructions necessary for both stages of tiled mergesort is $3.5n\log_2 n+O(n)$. □

7.3.4 Tiled m-way mergesort

The second stage of tiled mergesort is proposed as a single n/z-way merge in [21].

Lemma 16 The second stage of n/z-way tiled mergesort induces $2n/B$ cache misses.

Proof As there is only one single merge, each element in the input array is read exactly once, and each element in the output array is written exactly once. Assuming B divides n/z and that the arrays are B-aligned, this corresponds to two memory tracks of size n. Consequently, the number of cache misses for the second stage is $2n/B$. □

Using an n/z-way merge for the second stage of mergesort therefore restricts the number of cache misses for the complete mergesort to $4n/B$.

Lemma 17 The Pure C cost of an n/z-way tiled mergesort is $(9n\log_2 n - 5.5n\log_2 z + O(n))\tau$.

Proof By Lemma 15, the number of Pure C units executed in the first stage of tiled mergesort is $3.5n\log_2 z + O(n)$. In an n/z-way merge using a heap of n/z elements, the merge requires n SIFTDOWN-operations of $9\log_2(n/z)$ Pure C operations each (see Section 6.1), which gives the second stage of the mergesort a Pure C cost of $9n\log_2(n/z) + O(n)$ Pure C operations. Summing up, the number of Pure C operations in both stages of an n/z-way mergesort combined is $9n\log_2 n - 5.5n\log_2 z + O(n)$. □

Even though the number of cache misses in the tiled mergesort with a single n/z-way merge as the second stage is only $4n$, the Pure C cost has risen considerably. It therefore clearly depends on the $\tau^\star/\tau$ ratio if a n/z-way merge is faster than a traditional four-way merge of the n/z blocks. For $\tau^\star/\tau = 10$, the weighted cost of tiled mergesort the n/z-way merge is about 66 percent higher than for the mergesort based on a four-way merge.

Some restrictions apply. If z is chosen as $M/4$ to make room in the cache for both input tiles, output tiles and the elements of the heap, then the heap structure needs about $8n/M$ words of cache memory. If the spatial locality of the merge is valid for all memory tracks, then one block must be present for all $4n/M$ memory tracks (not counting the output track), which corresponds to $4nB/M$ words of cache memory. To sum up the cache must hold at least $(4nB + 8n)/M$ words, and therefore, for e.g., $B = 8$,

$$M \geq \frac{4nB + 8n}{M} \Leftrightarrow \frac{M^2}{40} \geq n$$

For small caches, this restricts the feasibility of the n/z-way merge. A cache of 8kb ($M = 2\,048$ words) will at best be able to sort about 100 000 elements, while a cache of 64kb ($M = 16\,384$ words) will sort no more than 6.7 mil-

lion without inducing extra cache misses. Added to this is some measure of interference misses in the n/z-way merge.

The n/z-way merge has not been implemented.

7.4 Experiments and referential footprints

The two-way, four-way and tiled mergesort programs are compared in tests and held against the predicted results. The predicted running times are depicted in Figure 7.3 for two-way, four-way and tiled, four-way mergesort with a tile size of $4\,096$. The predicted execution time of the tiled mergesort with the second stage as an $m = n/z$-way merge for $z = 2\,048$ is also shown. The array sizes span from 2^{16} to 2^{22} word-sized elements.

The measured running times on the AXP reference machine is shown in Figure 7.4 and are a few percent higher than the estimates. The tiled, four-way mergesort with a tile size of $4\,096$ elements is fastest, though by a small margin compared to tile sizes of $1\,024$ and $16\,384$. The gap from the execution times of the mergesort programs based on the four-way merge to the two-way merge is a result of the halving the of cache miss count.

The referential footprints of the two-way, four-way and tiled, four-way mergesort programs is shown in Figures 7.5 to 7.7 for a sort of 64 elements and a tile size of 16. The original input array is in the lowest 64 elements, and the original output array is represented as the highest 64 elements of all the figures. In all footprints, the sequential reference pattern to the output array is clearly seen. The two-way and four-way merge programs differ by the number of memory tracks visible in the footprints of the input array, while the tiled, four-way mergesort footprint differs from the traditional four-way mergesort by the first $2z\frac{n}{z}\log_4(\frac{n}{z}) = 256$ references which corresponds to $n/z = 4$ four-way mergesorts of $z = 16$ elements. The footprints will be graphically less nice if n is not a power of four.

In Figure 7.8, the number of cache misses per element is shown for a 4kb 4-way set-associative cache both as the predicted miss count for the three programs and the actual measured miss count from the profiler.

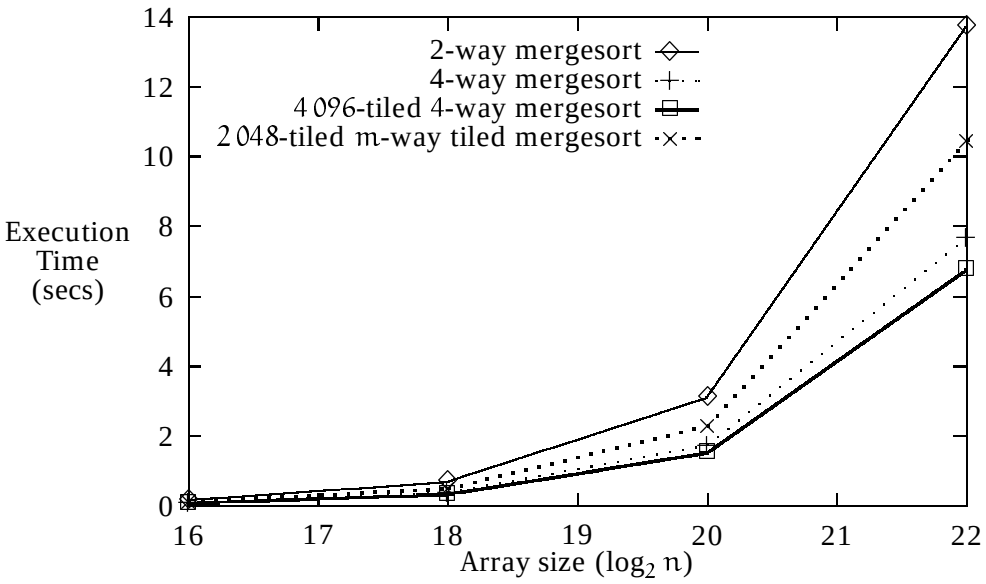

Figure 7.3: Predicted execution times of the mergesort programs.

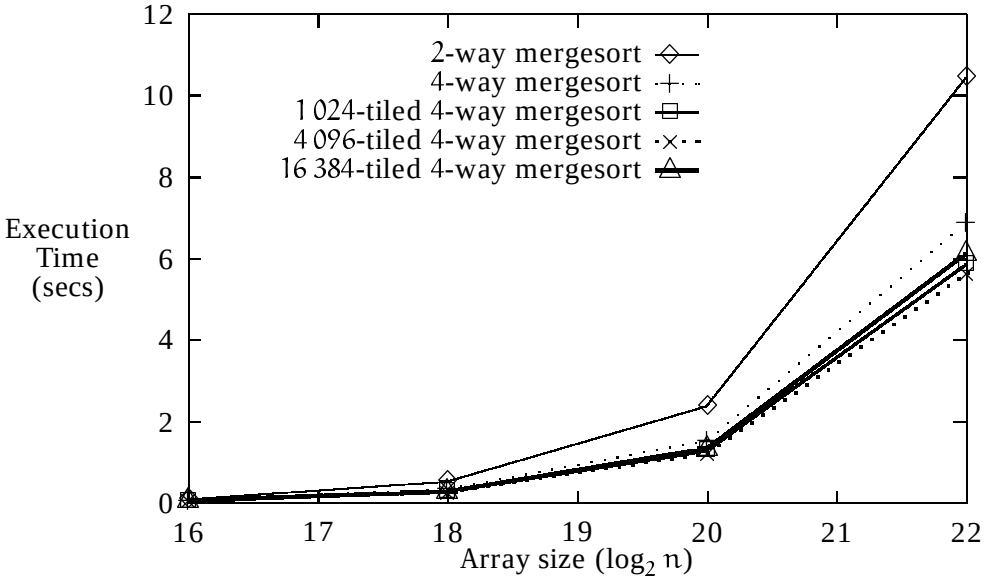

Figure 7.4: Actual execution times of the mergesort programs.

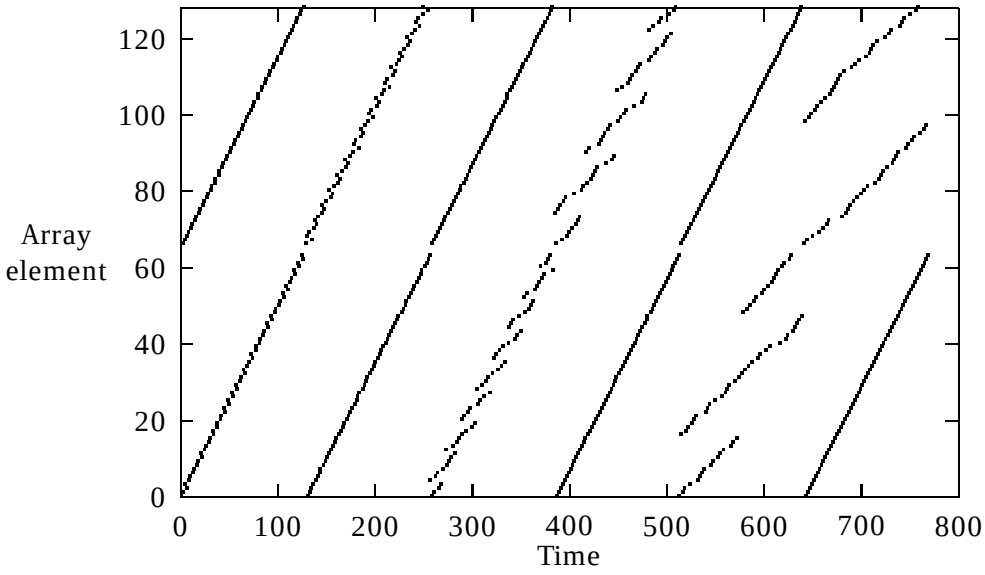

Figure 7.5: Two-way mergesort referential footprint ($n = 64$).

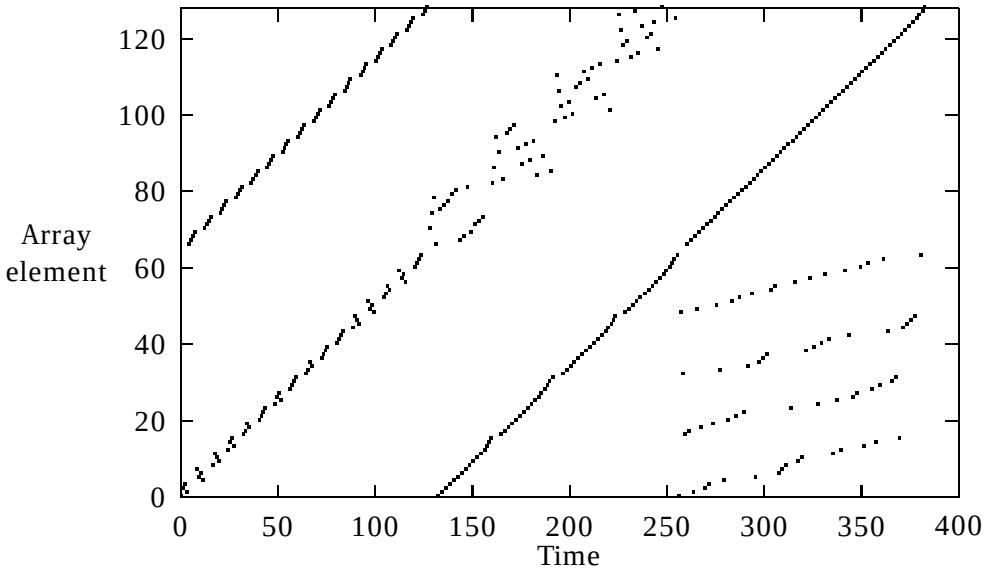

Figure 7.6: Four-way mergesort referential footprint ($n = 64$).

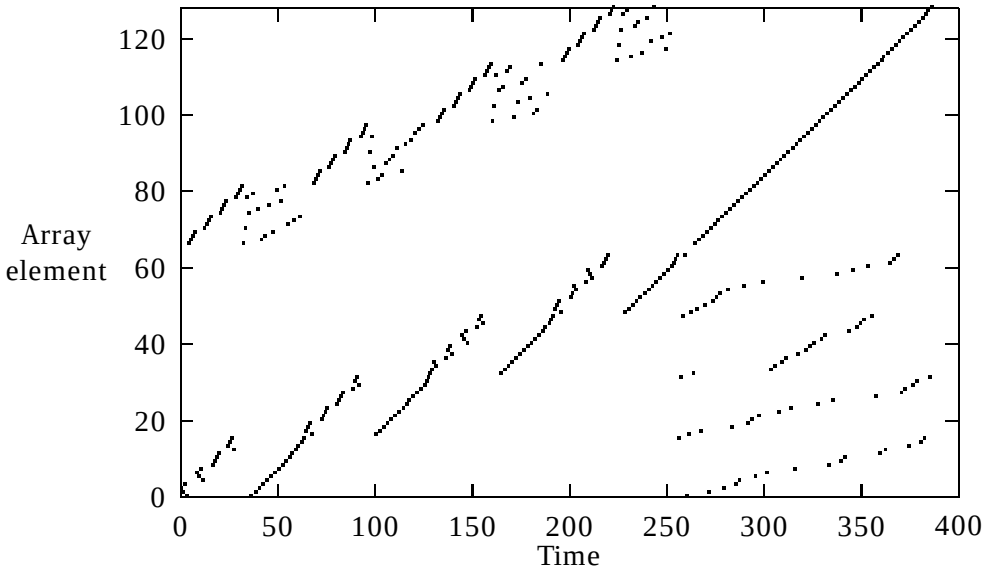

Figure 7.7: Tiled four-way mergesort referential footprint ($n = 64$).

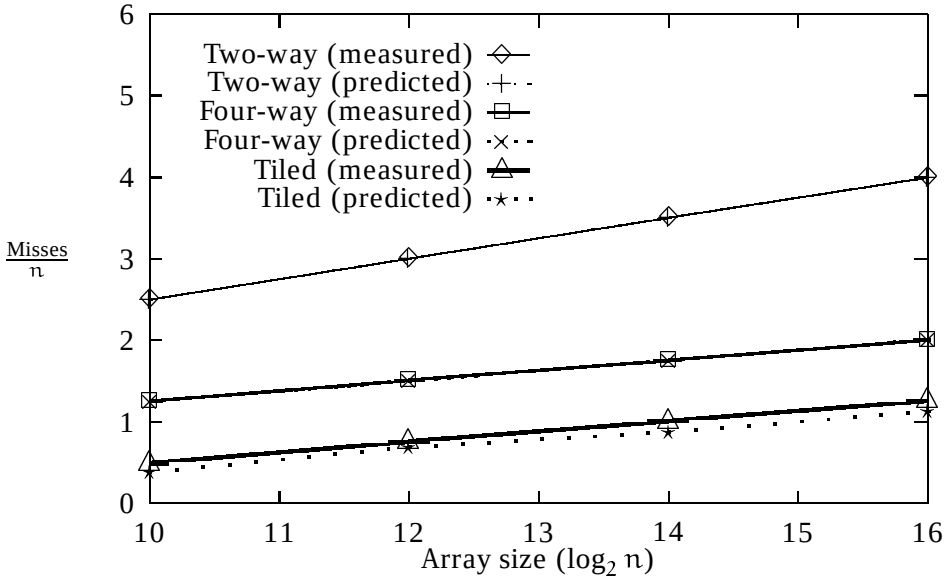

Figure 7.8: Miss rates of mergesort (4-way 4kb cache).

7.5 Conclusion

The cache behaviour of mergesort programs has been studied. Mergesort based on an m-way merge is clearly superior for higher values of m because fewer memory references and thus fewer cache misses in the sequential reference patterns occur. However, due to the added instruction cost for housekeeping m heads of input arrays to the merge in e.g., a heap, the feasibility of a general m-way merge depends on the miss penalty. Further, it has been shown that blocking is effective in mergesort, as the tiled mergesort was superior to a four-way mergesort. In theory, an eight-way mergesort would be even faster than a four-way mergesort, under the assumption that the actual merge procedure, consisting of both an eight-, four-, three-, and two-way merge, will fit in the instruction cache.

The weighted cost model presented in Chapter 5 has been proven accurate for predicting the performance of mergesort. The miss count analyses are nearly identical to the measured miss counts for all mergesort programs, while the actual running times divert by a few percent.

Mergesort is particularly sensitive to interference misses as discussed in Section 3.2.2 because length of the subarrays merged as well as the number of cache sets are both powers of 2. In an m-way merge, there are $m + 1$ actual memory tracks, which means that, to avoid interference, an associativity of at least $m + 1$ is required. In practice, set-associativity of more than 8 is rare, and therefore m-way merges with high values of m are considered prone to interference.

Chapter 8

A sequential-access graph algorithm

The following case study focuses on the *connected components* problem and proposes a solution with a constant number of random pointer operations, that is, an algorithm with only sequential access patterns. The methodology is to seek a solution which mimics the cache behaviour of mergesort such that, for an input size of n elements, $\log_2 n$ main passes are performed, but within each pass, no loop-carried dependencies exist. This means that, for each pass, the computations may be ordered such that they are performed for elements that are placed consecutively in memory.

8.1 Connected components

Given a directed graph $G = (V, E)$, $n = |V|$ and $m = |E|$, compute for each $v \in V$ a representative of its connected component in G. The representatives may be arbitrary but must be unique. The input is E represented as an unordered sequence of integer tuples.

8.2 A traditional solution

The traditional solution to this problem is based on a series of recursive depth-first searches. Let $E' = \{(v, w), (w, v) \mid (v, w) \in E\}$ such that $G' = (V, E')$ is an undirected version of G. Further associate to each $v \in V$ a component representative c_v initialised with the value `nil`. Each c_v is marked by the algorithm with the representative of the connected component to which it

belongs. DFSCONNCOMP performs a depth-first traversal for each $v \in V$ that is previously unvisited. An unvisited vertex has $c_v = \texttt{nil}$.

Algorithm 13 (*DfsConnComp)*

DFSCONNCOMP$(G = (V, E))$:
1: **for each** $v \in V$ **do**
2: DFSMARK(v, v)

Algorithm 14 (*DfsMark)*

DFSMARK(v, c) :
1: **if** $c_v \neq$ nil **then**
2: $c_v = c$
3: **for each** $\{w \mid (v, w) \in E'\}$ **do**
4: DFSMARK(w, c)
5: **end if**

The traversal visits every vertex in the component to which v belongs, marking each with the component representative.

Proposition 1 DFSCONNCOMP is linear.

Each vertex is visited exactly once each in the loop in DFSCONNCOMP, while DFSMARK will behave differently depending on whether the vertex v has been visited. If v has been visited, it may access all adjacent nodes to v. If v has not been visited, it accesses nothing. As DFSMARK will perform a recursive call for unvisited vertices only, it will access each vertex in the graph exactly once. Combined, the number of accesses to the graph is $2n$, and these are all random pointer assignments, yielding a memory access cost of $2Bn$[1]. □

8.3 A parallel solution

A parallel solution is now proposed for solving the connected-components problem. The rationale for giving a parallel solution is that each processor

[1]The algorithm requires that the input is arranged such that the depth-first traversal may find an arbitrary vertex in constant time. This requires a permutation of the input stream E into, e.g., an adjacency list, which involves $n + m$ random pointer operations. Furthermore, for the depth-first algorithm to be able to reach all vertices from any invocation of *DfsMark*, it is necessary to transform G into an undirected variant by mirroring all edges: If the edge (w, v) is represented for each edge (v, w), effectively doubling the preprocessing time. Accounting for preprocessing also, the gross memory access cost thus becomes $2B(2n + m)$.

executes one step independent of all other processors. The parallel program may be converted to a sequential program by executing a step for each processor in sequence, and, to also achieve a sequential reference pattern, the input data is sorted first.

A parallel algorithm for solving the connected-component problem for acyclic graphs is given below. In each component, the component representative is assumed to be the vertex with the smallest value. The object of the algorithm is to associate each vertex in a component with its component representative; this is accomplished through two parallel steps, *parent assignment* and *pointer jumping*.

Proposition 2 A directed graph $G = (V, E)$ is transformed into an acyclic, directed graph (dag) $G' = (V, E')$ by reversing all edges (v, w), $w > v$ such that each $e = (v, w)$ in E' has the property $v < w$. This transformation can be performed in linear time, and is done in the process of reading the input E.

The mapping of a vertex x to its component representative is denoted $p(x)$ (*parent* of x), and is the vertex y in the graph for which an edge (y, x) exists with the smallest value of y, that is $p(x) = \textbf{min}\ \{y \mid (y, x) \in E'\}$. A parallel algorithm for assigning $p(x)$ for all $x \in E'$ working only on edges is given in algorithm 15[2]. It is assumed that, initially, $p(x) = x$.

Algorithm 15 (*Parallel Parent Assignment)*

PPA$(e = (v, w))$:
1: **if** $(v < p(w))$ **then**
2: $\quad p(w) = v$

A key observation is that G' consists of m vertices w with no endpoint in E' (no edge $(v, x) \in E'$), and $\lceil m \rceil$ vertices v with no starting point in E' (no edge $(y, v) \in E'$). This means that a number of paths exist through the vertices in G'. The goal of the next algorithm is to perform a *pointer jump* for all edges such that each edge (v, w) is changed to $(p(v), w)$. Denote an edge with the property $p(w) = v$ a *parent edge*.

This approach is known from algorithms on trees, e.g., path decomposition in set union algorithms, and will reassign every vertex to the component representative in $\log_2 h$ steps, where h is the height of the tree. This is easily proven, as each pointer jump will halve the height of the tree. In Figure 8.3, such a pointer jump operation for a small tree is depicted.

[2]The algorithm assumes that the two references to $p(x)$ are atomic which is a restriction to the PRAM model, realised by synchronization.

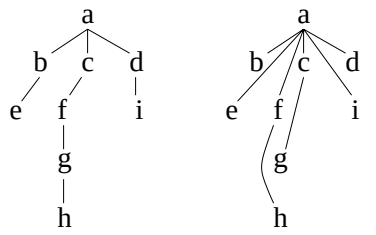

Figure 8.1: A pointer jump in a tree.

Consider the tree(s) induced by the parent assignments of algorithm 15, i.e., $T' = (E'', V)$ where $E'' = \{(v, w) \in E' \mid p(w) = v\}$. Performing pointer jumping in T' will make $p(x)$ point to each root in T'. This will not touch any non-parent edges. If z edges $\{(v_1, w), \ldots, (v_z, w)\}$ exist in E', z iterations will be required before all those edges are included in T' and selected for pointer jumping. Therefore, these edges are changed from (v, w) to $(p(w), v)$, changing the direction of the edge. This operation maintains connectivity by transitivity. As w is connected to $p(w)$ and v is connected to w, is is possible to connect $p(w)$ to v. *This can happen only once for any edge.* The dag pointer jump is shown for a small subgraph in Figure 8.3, and the parallel pointer jump algorithm listed in algorithm 16. It is easy to see that the algorithm maintains the invariant $v < w$ for all edges in E'.

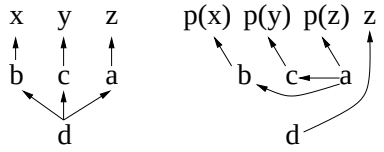

Figure 8.2: A pointer jump in the dag.

Algorithm 16 (*Parallel Pointer Jump)*

PPJ($e = (v, w)$) :
1: **if** ($p(w) < v$) **then**
2: $e = (p(w), v)$
3: **else**
4: $e = (p(v), w)$
5: **end if**

Lemma 18 $\lceil \log_2 n \rceil + 1$ alternating application PPA and PPJ will result in $p(x)$ being the component representative of vertex x.

Proof The pointer jumping in the tree induced by the parent edges will be decomposed completely in $\lceil \log_2 n \rceil$ steps. Any other edge will be reversed

at some point during the decomposition of the parent edges, thus becoming a parent edge itself. This delays the pointer jump for such an edge by 1, yielding a worst-case decomposition time of $\lceil \log_2 n \rceil + 1$. □

8.4 A sequential-access solution

The transformation of algorithms 15 and 16 to sequential counterparts is now investigated.

8.4.1 Preliminaries

Let $G'' = (E'', V)$, where $E'' = E \cup \{(v, v) \mid v \in V\}$, that is G'' is a directed graph with reflexive edges. Further let $<$ be a partial order on E'' such that $e = (v_e, w_e) < f = (v_f, w_f)$ iff $v_e < v_f$ or $(v_e = v_f$ and $w_e < w_f)$, and $>$ be a partial order on E'' such that $e = (v_e, w_e) > f = (v_f, w_f)$ iff $w_e < w_f$ or $(w_e = w_e$ and $v_e < v_f)$ corresponding to sorting the edges in E'' on v, w and w, v, respectively.

Sorting the edges on $<$ partitions the edge set E'' into n adjacent subsets $T_1, \ldots, T_n$, where subset $T_i = \{(v_i, v_i), (v_i, x), (v_i, y) \ldots\}$. Each subset T_i thus shares the same vertex v_i and the first edge of the subset is the reflexive edge (T_i is a tree of depth 1 with v as the root).

Sorting the edges on $>$ partitions the edge set E'' into n adjacent subsets $R_1, \ldots, R_n$, where subset $R_i = \{\ldots, (x, w_i), (y, v_i), (v_i, v_i)\}$. Each subset R_i thus shares the same vertex w_i and the last edge of the subset is the reflexive edge (R_i is a tree of depth 1 with w as the root, with reverse edge directions).

The sequential-access algorithm works on edges rather than vertices. This is unambiguous because the reflexive edges (v, v) are representatives of the vertices inside the edge set. In the following, therefore, the parent of an edge $p(e = (v, w))$ is the component representative of both v and w.

8.4.2 Parent assignment

Assume that E'' is a poset on $>$. The parent $p(e)$, $e \in R_i$ is v, where (v, w) is the first edge in R_i. This vertex will have the smallest number in R_i. This is shown in the program fragment below, where `min` is the first edge in each R_i, changing value only if a new subset is reached (a new value w is met in the sequence).

```
void spa (Edge* E, int m) {
  Edge min (nil, nil);
  for (int i = 0; i < m; i++) {
    if (min.w != E[i].w)
      min = E[i];
    E[i].p = min.v;
  }
}
```

8.4.3 Pointer jumping

Assume that parent assignments have been done for all edges in E'' and that E'' is now a poset on $<$. The parallel pointer jump is transformed to a sequential algorithm by performing each step in sequence. The first edge f_i of any subset T_i will be the reflexive edge, and $p(f_i)$ is the parent of the entire tree. Therefore, the sequential pointer jumping can use $p(f_i)$ as a replacement for $p(v)$ in the parallel pointer jumping. The program does not reassign the reflexive edges, as they are needed through all applications of the parent assignment and pointer jumping to pass the value corresponding to $p(v)$ in the parallel pointer algorithm.

```
void spj (Edge* E, int m) {
  Edge min (nil, nil);
  for (int i = 0; i < m; i++) {
    if (min.v != E[i].v)
      min = E[i];
    else if (E[i].p < E[i].v)
      E[i].w = E[i].v, E[i].v = E[i].p;
    else
      E[i].v = min.p;
  }
}
```

8.4.4 Miss count of the sequential-access program

Theorem 6 The sequential-access connected-components program requires $n' \log_2 n(\log_2 n + 2)$ sequential pointer operations, where $n' = m + n$, and thus $(n' \log_2 n(\log_2 n + 2))/B$ cache misses.

Proof As shown in Lemma 18, $\lceil \log_2 n \rceil + 1$ applications of the two steps are necessary to complete the decomposition. Each step in the sequential algorithm involves two passes and two sorts over all edges in E'' (where $|E''| = n' = m + n$).

The two sequential passes combined cost $2n/B$ misses because they are sequential and constitute two single memory tracks. The memory cost of the parent assignments and pointer jumps for the entire program, therefore, is $\frac{2n \log_2 n'}{B}\tau^\star$.

The cost of sorting is assumed to be the 4-way mergesort of Section 7.2.2, which costs $\frac{n' \log_2 n'}{B}\tau^\star$, where z is the blocking factor, for one sort of the n' elements. For all $\log_2 n'$ passes, the total memory cost of sorting becomes $\frac{n' \log_2^2 n'}{B}$. Combined, the cost of decomposing and sorting in the connected-components sequential-access program is $\frac{n' \log_2 n' (\log_2 n' + 2)}{B}$. The $O(\log_2 n')$ random pointer assignments are disregarded. □

8.5 Conclusion

For a cache with a miss penalty of about 10 Pure C instructions, this miss count is much too high for the sequential-access approach to be feasible. For today's cache parameters, therefore, the added cost of sorting the input $\log_2 n$ times is too expensive. If the block size becomes larger, or if the miss penalty on a sequential access operation becomes smaller, the technique may find its uses.

Some possibilities for improvement also exist. Through the sequential-access connected-components algorithm, $m - n$ duplicates will emerge in E''. These edges may be removed from E'' as they show up. In [2], interleaved clean-up steps of redundant edges in minimum spanning trees is investigated, and this is reminiscent of duplicate deletion in the connected component problem, though the upper bound will not change. Further, tiled multi-way mergesort may be applied to lower the dominating $\log^2$ term.

Chapter 9

Conclusion

The main contributions of this work is

- An analytical model for predicting the execution time of programs including their memory reference cost.
- A program analysis tool for measuring cache miss counts of real programs.
- A study of heap construction programs with different performance characteristics, and the proposition of a heap construction algorithm with superior performance,
- A study of mergesort programs with different performance characteristics,
- A methodology for designing programs with only a constant number of random-access operations.

In the presentation of the analytical model, the costs of a (Pure C) instruction and the cost a cache miss were derived through experiments and used in the subsequent chapters for predicting the running times of the programs presented. These predictions turned out very close to the measured results.

The footprints of memory accesses from the programs are interesting because they indicate reference locality in time and space on the two axes. They are interesting because they depict the actual reference patterns graphically, sometimes revealing less obvious properties of locality.

The weighted Pure C cost model requires implementation, which makes the analyses under this model much more cumbersome than traditional asymptotical analysis. It may, however, be argued that only the part of a program

which contributes to the leading term of the complexity is required for implementation and analysis.

In the analyses of heap construction programs, it was shown that a program with optimal complexity required a higher number of cache misses than its suboptimal competitor. The extra number of cache misses depends on both the input size and the cache block size and the result suggests that other algorithms of this kind, which are known to be optimal, may perform worse than expected due to the cache behaviour.

Through the study of mergesort programs, it was shown that sequential-access patterns are easy to work with analytically and that blocking is an effective technique for sorting as well as for numerical methods. The limits to reducing the number of passes, and thus the miss count, was also shown as a consequence of the ratio between instruction cost and memory access cost.

Designing programs with only sequential access patterns, as presented in the last chapter, turned out not to be an improvement over known solutions. However, as with the mergesort programs, it may be a matter of time before the cost of a cache miss in terms of instructions grows to a level that makes the methodology feasible. In any case, the methodology scales and is clearly sound for the latency times induced by a network transaction or a page fault.

Programs with only sequential access references impose different requirements on the cache metrics than a general program. The execution time of a sequential-access program scales linearly with the block size. In typical production caches, however, the block size is chosen from other performance factors than spatial locality. A sequential-access program will further pollute the entire contents of a cache in its stride, which is unnecessary because their cache block requirement is $O(1)$ and because they have no temporal locality. An optimal cache configuration for a sequential-access algorithm has a large block size, few blocks, and full associativity, e.g., $a = 8$ and $B = M/8$, corresponding to a low number of simultaneously overlapping memory tracks.

The real promise is that the referential properties of sequential-access algorithms may give directions to designs of memory hierarchies, as hinted in Section 2.2. Memory operations in modern architectures include specially augmented instructions for e.g., *prefetching* blocks in sequence. Assuming a load or store instruction carries information that it is sequential- or random-access, the memory hierarchy will be able to initiate the transaction for block $i+1$ as soon as the transaction for block i has finished, placing each block in a read buffer as they arrive. This would effectively remove the amortised $\frac{\tau^*}{B}$ latency for sequential-access operations if the bandwidth is not exceeded. In [19] and [6], *stream buffers* are introduced to initiate transactions to consecutive blocks, called *unit strides*. The experiments in [6] show that, on the SPEC92 benchmarks, improvements of up to 40% were achieved in simulations.

9.1 Directions and further work

Other algorithms with non-local reference patterns exist for which a sequential-access algorithm may be found using the methodology presented in this thesis. These include set union algorithms and other graph algorithms. As the sequential-access algorithms depend heavily on architectural constants, it is important to simulate the programs on experimental architectures with e.g., stream buffers.

The cache profiler presented also requires more features to be generally useful. It is based on source-level instrumentation, which requires recompilation and cumbersome annotation of memory operations. By changing the operation to link time, stating which program variables should be monitored, the profiler would be more flexible in use. In addition, it must be able to record source statements (line numbers) responsible for each reference, attempt to track interference misses, and measure the instruction rate of the program too.

Bibliography

[1] J. BOJESEN, Heap implementations and variations, Technical report, DIKU (Oct. 1998).

[2] D. CHERITON AND R. E. TARJAN, Finding minimum spanning trees, *SIAM Journal on Computing* **5** (1976), 310–313.

[3] D. W. CLARK, Cache Performance in the VAX-11/780, *ACM Transactions on Computer Systems* **1**, 1 (1983), 24–37.

[4] R. L. S. (ED.), Alpha Architecture Reference Manual, 2nd rev., Technical report, Digital Equipment Corporation (1994).

[5] A. EUSTACE AND A. SRIVASTAVA, ATOM: A Flexible Interface for Building High Performance Program Analysis Tools, Technical Report WRL Research Report 94/2, Digital Equipment Corporation Western Research Laboratory (1994).

[6] K. I. FARKAS, N. P. JOUPPI, AND P. CHOW, How useful are non-blocking loads, stream buffers and speculative execution in multiple-issue processors?, in *Proceedings of the First International Symposium on High-Performance Computer Architecture*, IEEE Computer Society TCCA (Jan. 22–25, 1995), 78–89.

[7] R. W. FLOYD, Algorithm 113: Treesort, *Communications of the ACM* **5** (1962), 434.

[8] J. D. GEE, M. D. HILL, D. N. PNEVMATIKATOS, AND A. J. SMITH, Cache Performance of the SPEC92 Benchmark Suite, *IEEE Micro* **13**, 4 (Aug. 1993), 17–27.

[9] D. GRUNWALD, B. ZORN, AND R. HENDERSON, Improving the Cache Locality of Memory Allocation, in *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices **28** (1993), 177–186.

[10] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture — A Quantitative Approach*, Morgan Kaufmann (1996).

[11] M. D. HILL, A case for direct-mapped caches, *Computer* **21**, 12 (1988), 25–40.

[12] M. D. HILL AND A. J. SMITH, Evaluating associativity in CPU caches, *IEEE Transactions on Computers* **38**, 12 (Dec. 1989), 1612–1630.

[13] J. JOHN R. BLACK, C. U. MARTEL, AND H. QI, Graph and hashing algorithms for modern architectures: Design and performance, in *Proceedings of the 1998 Workshop on Algorithmic Engineering*, Saarbrücken, Germany (August 1988), 37–48.

[14] M. K. JOHNSON, *Linux Kernel Hacker's Guide*, Addison Wesley (1996).

[15] N. P. JOUPPI, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in *Proceedings of the 17th Annual Symposium on Computer Architecture*, Seattle, Washington (May 1990), 364–373.

[16] J. KATAJAINEN AND J. L. TRÄFF, A meticulous analysis of mergesort programs, in *Proceedings of the 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science 1203*, Springer-Verlag (1997), 217–228.

[17] B. KERNIGHAN AND D. RICHIE, *The C Programming Language, 2nd edition*, Prentice Hall (1988).

[18] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley (1973).

[19] B. KUMAR, A model of spatial locality and its application to cache design, Stanford University (1979).

[20] M. S. LAM, E. E. ROTHBERG, AND M. E. WOLF, The cache performance and optimization of blocked algorithms, *ACM SIGPLAN Notices* **26** (1991), 63–74.

[21] A. LAMARCA AND R. E. LADNER, The influence of caches on the performance of sorting, in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, Louisiana (1997), 370–379.

[22] A. G. LAMARCA, *Caches and Algorithms*, Ph.D. Thesis, University of Washington (1996).

[23] A. M. G. MAYNARD, C. M. DONNELLY, AND B. R. OLSZEWSKI, Contrasting characteristics and cache performance of technical and multi-user commercial workloads, *ACM SIGPLAN Notices* **29** (1994), 145–145.

[24] J. C. MOGUL AND A. BORG, The effect of context switches on cache performance, in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara (1991), 75–84.

[25] C. F. O. TEMAM AND W. JALBY, Cache interference phenomena, in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1994), 261–271.

[26] D. A. PATTERSON AND J. L. HENNESSY, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann (1994).

[27] J. K. R. FADEL,K. V. JACOBSEN AND J. TEUHOLA, Heaps and heapsort on secondary storage, *to appear* (1998), .

[28] R. SEDGEWICK, *Algorithms*, Addison Wesley (1988).

[29] A. SILBERSCHATZ AND P. B. GALVIN, *Operating System Concepts, 5th edition*, Addison Wesley (1988).

[30] A. J. SMITH, A comparative study of set-associative memory mapping algorithms and their use for cache and main memory, *IEEE Transactions on Software Engineering* **4** (1978), 121–130.

[31] A. J. SMITH, Cache memories, *ACM Computing Surveys* **14** (1982), 473–530.

[32] A. J. SMITH, Line (Block) Size Choice for CPU Cache Memories, *IEEE Transactions on Computers* **C-36**, 9 (Sept. 1987), 1063–1075.

[33] A. SRIVASTAVA AND A. EUSTACE, ATOM: A system for building customized program analysis tools, in *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)* (June 1994), 196–196.

[34] B. STROUSTRUP, *The C++ Programming Language, 3rd edition*, Addidon-Wesley (1995).

[35] D. W. WALL, Systems for late code modification, Technical Report WRL Research Report 92/3, Digital Equipment Corporation Western Research Laboratories (may 1982).

[36] J. WELSH AND J. ELDER, *Introduction to Pascal, 3rd edition*, Prentice Hall (1988).

[37] J. W. J. WILLIAMS, HEAPSORT, *Communications of the ACM* **7** (1964), 347–348.

[38] P. C. WU AND K. C. HUANG, Case Studies on Cache Performance and Optimization of Programs with Unit Strides, *Software — Practice and Experience* **27**, 2 (Feb. 1997), 167.

Appendix A

CPROF usage

The CPROF tool is available, including source code, under the GNU Public License and may be obtained from the author by e-mail or from the Worldwide Web at `http://www.diku.dk/students/halgrim/cprof`. The source code for the profiler is omitted in this print, though the include file for recording the address traces is included. Below, the usage of the profiler is described.

A.1 Invocation of the profiler

The reference pattern is fed to the profiler CPROF which will compute statistics for a given cache configuration. CPROF accepts the following optional command-line arguments:

`-a`a	set associativity to a,
`-b`B	set block size to B words,
`-f`*file*	set the input file to *file*,
`-h`	help,
`-l`	produce LaTeX output,
`-M`M	set cache capacity to M words,
`-p`p	set replacement policy (`rnd`, `nru` or `lru`),
`-v`	verbose output (all memory references),
`-w`w	set word size to w bits.

If associativity is set to 1, the cache will be direct-mapped; if it is set to M/B (or 0, for convenience), it will be fully associative. All other values produces an a-way set-associative cache.

A.2 Output format

The profiler reports the total number of references; the hit and miss count; the types of misses (compulsory, capacity and conflict), all for both read and write accesses. In addition, it gives the hit/miss ratio and the miss rate and recapitulates the metrics for reference. The profiler's output for the simulation of the program in Example 4 with $N=2^{15}$ for a 16kb 4-way associative cache with NRU replacement is shown below.

Example 17

```
$ cprof -a4 -pnru -m4096 -fmytrace.dat
Cache profile: mytrace.dat
--------------------------
Cache metrics:
- capacity:         4096 words (16 kb)
- associativity:    4-way
- bits per word:    32
- bytes per block:  32
- words per block:  8
- cache sets:       128
- replacement:      nru

Statistics:
- references:    65534 (  32767 reads,  32767 writes )
- hit count:     61437 (  28670 reads,  32767 writes )
- miss count:
  - cold-start     512 (    512 reads,      0 writes )
  - capacity      3585 (   3585 reads,      0 writes )
  - conflict         0 (      0 reads,      0 writes )
  - total         4097 (   4097 reads,      0 writes )
- hit/miss ratio: 14.9956
- miss rate:      6.25172%
```

The corresponding LaTeX (tabular environment) output as follows.

A.3 GNU PLOT output

The address trace used for the cache profiling may be converted to the GNU PLOT input format by a supplementary tool *gpconvert* (part of the CPROF package). GNU PLOT is a cross-platform, command-driven interactive function plotting program available under the GNU Public License by FTP from

mytrace.dat	Reads	Writes	Total
Hit count:	28670	32767	61437
Miss count:	4097	0	4097
• Cold-start:	512	0	512
• Capacity:	3585	0	3585
• Conflict:	0	0	0
Total:	32767	32767	65534

Table A.1: An example of formatted output from CPROF.

`prep.ai.mit.edu`. GNU PLOT may be invoked to produce the final graphical representation. The *gpconvert* tool optionally accepts a scaling factor to produce a horizontal axis corresponding to, e.g., array indices.

Error conditions

The profiler will complain if the associativity is too high, that is if $aB > M$, if the word size is not a multiple of 8, on bad command line arguments or missing files, and on errors in the input stream. The profiler will compile in standard GCC environments and uses the *getopt* library routines for parsing the command line.

Appendix B

Source code

The tools and programs presented in this thesis is available in source code under the GNU Public License from the Worldwide Web[1], or via e-mail from the author.

B.1 Cache profiling tools

Only the `cprof.h` header file is included in this print.

```
// cprof.h - address trace scaffolding
//
// CPROF must be defined as the (double-quoted)
// file name of your address trace, or undefined
// if you don't want a trace.

#if !defined(_CPROF_H)
#define _CPROF_H

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>

#if defined(CPROF)
typedef char* GENERIC_POINTER;
static GENERIC_POINTER _pointer;
static const WORDSIZE = sizeof(_pointer);
```

[1] URL: `http://www.diku.dk/students/halgrim/src`.

```
ofstream reftrace(CPROF, ios::out);
#endif
#if defined(CPROF_COUNT)
#define _PLR(i)((i!=1)?"s":"")
static unsigned long _rc = 0, _wc = 0;
static void cprofstats (ostream& os = cout) {
  os <<"Memory references:\n"
     <<setw(8)<<setfill(' ')<<_rc<<" read"<<_PLR(_rc)<<endl
     <<setw(8)<<setfill(' ')<<_wc<<" write"<<_PLR(_wc)<<endl
     <<setw(8)<<setfill(' ')<<(_rc+_wc)<<" total"<<endl;
}
#endif

template<class T>
inline T& READ(T& p) {
#if defined(CPROF)
  GENERIC_POINTER vp = GENERIC_POINTER(&p);
  for (unsigned i = 0; i < sizeof(T); i+=WORDSIZE) {
    unsigned long ref = (unsigned long) (vp+i);
    reftrace << "r " << hex << ref << endl;
  }
#endif
#if defined(CPROF_COUNT)
  _rc++;
#endif
  return p;
}
template<class T>
inline T& WRITE(T& p, T v) {
#if defined(CPROF)
  GENERIC_POINTER vp = GENERIC_POINTER(&p);
  for (unsigned i = 0; i < sizeof(T); i+=WORDSIZE) {
    unsigned long ref = (unsigned long) (vp+i);
    reftrace << "w " << hex << ref << endl;
  }
#endif
#if defined(CPROF_COUNT)
  _wc++;
#endif
  return p = v;
}

#endif
```

B.2 Program tuning for locality

B.2.1 Packing

```
// packing.cc

#include <iostream.h>
#include <assert.h>

// queue for breadth-first searches etc.
// holds only references; no overflow check

template<class T>
struct Queue {
  T** rep;
  unsigned size, tail, head;
  bool isFull;
  void inc (unsigned& z) {
    z = (z + 1) % size;
  }
  Queue (int s) {
    rep = new T* [size = s];
    head = tail = 0;
    isFull = false;
  }
  ~Queue () { delete rep; }
  bool Push (T* t) {
    if (!isFull) {
      rep[tail] = t;
      inc (tail);
      if (head == tail)
        isFull = true;
      return 1;
    }
    cerr << "Queue: overflow\n";
    return 0;
  }
  T* Pop () {
    if (head != tail || isFull) {
      T* t = rep [head];
      inc (head);
      isFull = false;
      return t;
```

```
    }
    cerr << "Queue: underflow\n";
    return 0;
  }
  bool isEmpty () {
    return head == tail && !isFull;
  }
};

int n, m;

struct EdgeLink {
  int w;
  EdgeLink* next;
  EdgeLink (int w0) : w (w0), next (0) { }
  ~EdgeLink () {
    if (next) delete next;
  }
};

struct Vertex {
  bool isVisited;
  EdgeLink *head, *tail;
  Vertex () {
    head = tail = 0;
    isVisited = false;
  }
  ~Vertex () {
   if (head) delete head;
  }
  void AddEdgeTo (int w) {
    if (tail)
      tail->next = new EdgeLink (w),
        tail = tail->next;
    else head = tail = new EdgeLink (w);
  }
  void Mark () {
    isVisited = true;
  }
  bool isMarked () {
    return isVisited;
  }
};
```

```
Vertex* AdjListGraph;

void AdjListPrep () {
  for (int i = 0; i < n; i++)
    AdjListGraph [i].isVisited = false;
}

void AdjListDepthFirst (int v) {
  if (!AdjListGraph[v].isMarked ()) {
    AdjListGraph[v].Mark ();
    for (EdgeLink* nbrs = AdjListGraph[v].head;
           nbrs; nbrs = nbrs->next)
      AdjListDepthFirst (nbrs->w);
  }
}

void AdjListBreadthFirst (int v) {
  Queue<Vertex> S (n);
  S.Push (&AdjListGraph[v]);
  AdjListGraph [v].Mark ();
  int visited = 0;
  while (!S.isEmpty ()) {
    Vertex& z = *S.Pop ();
    visited++;
    for (EdgeLink* nbrs = z.head; nbrs; nbrs = nbrs->next)
      if (!AdjListGraph [nbrs->w].isMarked ()) {
        AdjListGraph[nbrs->w].Mark ();
        S.Push (&AdjListGraph [nbrs->w]);
      }
  }
  assert (visited == n);
}

void AdjListRepresentation () {
  int v, w;
  AdjListGraph = new Vertex [n];
  for (int i = 0; i < m; i++) {
    cin >> v >> w;
    AdjListGraph[v].AddEdgeTo (w);
    AdjListGraph[w].AddEdgeTo (v);
  }
}

// integer matrix representation
```

```
struct AdjRow {
  int* EdgeList;
  bool isVisited;
  AdjRow () {
    isVisited = false;
    EdgeList = new int [n];
  }
  ~AdjRow () {
    delete EdgeList;
  }
  void AddEdgeTo (int w) {
    EdgeList [w] = 1;
  }
  void Mark () {
    isVisited = true;
  }
  bool isMarked () {
    return isVisited;
  }
};

AdjRow* IntMatrixGraph;

void IntMatrixPrep () {
  for (int i = 0; i < n; i++)
    IntMatrixGraph [i].isVisited = 0;
}

void IntMatrixDepthFirst (int v) {
  if (!IntMatrixGraph[v].isMarked ()) {
    IntMatrixGraph[v].Mark ();
    for (int i = 0; i < n; i++)
      if (IntMatrixGraph[v].EdgeList [i] == 1)
        IntMatrixDepthFirst (i);
  }
}

void IntMatrixBreadthFirst (int v) {
  Queue<AdjRow> S (n);
  S.Push (&IntMatrixGraph[v]);
  IntMatrixGraph[v].Mark ();
  int visited = 0;
  while (!S.isEmpty ()) {
```

```
    AdjRow& z = *S.Pop ();
    visited++;
    for (int i = 0; i < n; i++)
      if (z.EdgeList [i] == 1)
        if (!IntMatrixGraph [i].isMarked ()) {
          IntMatrixGraph [i].Mark ();
          S.Push (&IntMatrixGraph[i]);
        }
  }
  assert (visited == n);
}

void IntMatrixRepresentation () {
  int v, w;
  IntMatrixGraph = new AdjRow [n];
  for (int i = 0; i < m; i++) {
    cin >> v >> w;
    IntMatrixGraph[v].AddEdgeTo (w);
    IntMatrixGraph[w].AddEdgeTo (v);
  }
}

// bit-packed matrix representation

typedef unsigned long bitstring;
const B_WIDTH = sizeof (bitstring) * 8;
const B_MASK = B_WIDTH - 1;
const B_SHIFT = 5; // md.

struct PackedAdjRow {
  bitstring* EdgeList;
  PackedAdjRow () {
    EdgeList = new bitstring [(n>>B_SHIFT)+1];
  }
  ~PackedAdjRow () {
    delete EdgeList;
  }
  void AddEdgeTo (int w) {
    EdgeList [w>>B_SHIFT] |= (1<<(w&B_MASK));
  }
};

PackedAdjRow* BitMatrixGraph;
bitstring* Visited;
```

```
inline void Mark (int v) {
  Visited [v>>B_SHIFT] |= (1<<(v&B_MASK));
}

inline bool isMarked (int v) {
  return Visited [v>>B_SHIFT] & (1<<(v&B_MASK));
}

void BitMatrixPrep () {
  for (int i = 0; i < (n>>B_SHIFT)+1; i++)
    Visited [i] = 0;
}

void BitMatrixDepthFirst (int v) {
  if (!isMarked (v)) {
    Mark (v);
    for (int i = 0; i < n; i++)
      if (BitMatrixGraph[v].EdgeList[i>>B_SHIFT]
              & (1<<(i&B_MASK)))
        BitMatrixDepthFirst (i);
  }
}

void BitMatrixBreadthFirst (int v) {
  Queue<PackedAdjRow> S (n);
  S.Push (&BitMatrixGraph[v]);
  Mark (v);
  int visited = 0;
  while (!S.isEmpty ()) {
    PackedAdjRow& z = *S.Pop ();
    visited++;
    bitstring b, mask;
    for (int i = 0; i < n; i++, mask <<= 1) {
      if (!(i&B_MASK))
        b = z.EdgeList [i>>B_SHIFT], mask = 1;
      if ((b & mask) && !isMarked (i)) {
        Mark (i);
        S.Push (&BitMatrixGraph[i]);
      }
    }
  }
  assert (visited == n);
}
```

```
void BitMatrixRepresentation () {
  int v, w;
  BitMatrixGraph = new PackedAdjRow [n];
  Visited = new bitstring [(n>>B_SHIFT)+1];
  for (int i = 0; i < m; i++) {
    cin >> v >> w;
    BitMatrixGraph[v].AddEdgeTo (w);
    BitMatrixGraph[w].AddEdgeTo (v);
  }
  cerr << " (bit string width is " << B_WIDTH << " bits) ";
}

#include <time.h>
#include <stdlib.h>

main (int argc, char** argv) {
  bool err = false;
  unsigned choice;
  if (argc == 2) {
    choice = atoi (argv [1]);
    if (choice > 2) err = true;
  } else err = true;

  if (err) {
    cerr << "Say " << (*argv) << " 0|1|2, where\n"
         << "  0 = adjacency list, "
            "1 = int matrix, 2 = bitpacked matrix.\n";
    exit (10);
  }

  cerr << "Packing experiment:\nReading..." << flush;

  cin >> n >> m;

  int t, t0, T = 0, L = 10;  // L is the loop bound

  switch (choice) {
  case 0:
    AdjListRepresentation ();
    cerr << "done.\nSearching..." << flush;
    for (int i = 0; i < L; i++) {
      AdjListPrep ();
      t0 = clock ();
```

```
      AdjListBreadthFirst (0);
      t = clock ();
      T += (t - t0);
    }
    cerr << "done.\nAdjacency list representation: " << flush;
    break;
  case 1:
    IntMatrixRepresentation ();
    cerr << "done.\nSearching..." << flush;
    for (int i = 0; i < L; i++) {
      IntMatrixPrep ();
      t0 = clock ();
      IntMatrixBreadthFirst (0);
      t = clock ();
      T += (t - t0);
    }
    cerr << "done.\nInteger matrix representation: " << flush;
    break;
  case 2:
    BitMatrixRepresentation ();
    cerr << "done.\nSearching..." << flush;
    for (int i = 0; i < L; i++) {
      BitMatrixPrep ();
      t0 = clock ();
      BitMatrixBreadthFirst (0);
      t = clock ();
      T += (t - t0);
    }
    cerr << "done.\n"
         "Bit-packed matrix representation: " << flush;
    break;
  }
  cerr << (T/double((CLOCKS_PER_SEC*L)))
       << " seconds." << endl;

  cout << (T/double((CLOCKS_PER_SEC*L))) << endl;

}
```

B.2.2 Alignment

```
// alignment.cc

#include <iostream.h>
#include <time.h>
#include <stdlib.h>
#include <assert.h>
#include "../myrand.h"

//#define CPROF "align.dat"
//#include "../cprof.h"

const B = 32;

typedef int Element;

int rnd (Element* a, int n) {
  Element foo;
  volatile Element* acc = &foo;
  const d = 2;
  for (int i = 0; i < n/d; i++) {
    int j = myRand (n/d) * d;
    *acc += a [j];
    *acc += a [j+1];
  }
  return foo;
}

main (int argc, char** argv) {
  int n = atoi (argv [1]);

  char* data = new char [sizeof(int) * n + B];

#ifdef ALIGNMENT
  int padding = B - (unsigned long) data % B;
#else
  int padding = B - (unsigned long) data % B + (B-4);
#endif

  int* a = (int*) ((char*) data + (int) padding);

#ifdef VERBOSE
```

```
  cout << "&a = " << hex << ((unsigned long) a) << dec << endl;
  cout << "(&a mod B) = " << ((unsigned long) a % B) << endl;
  cout << "sizeof(Element) = " << sizeof(Element) << endl;
#endif

  int t, t0, T = 0, L = 2;
  for (int i = 0; i < L; i++) {
    t0 = clock ();
    rnd (a, n);
    t = clock ();
    T += (t - t0);
  }

  cout << n << " " << (T/double((CLOCKS_PER_SEC*L))) << endl;

}
```

B.2.3 Interleaving

```
// interleaving.cc

#include <iostream.h>
#include <time.h>
#include <stdlib.h>
#include <assert.h>
#include "../myrand.h"

int* a;
int* b;
int* c;

struct Abc {
  int a, b, c;
} *abc;

void main (int argc, char** argv) {
  n = atoi (argv [1]);

  a = new int [n];
  b = new int [n];
  c = new int [n];

  abc = new Abc [n];
```

```
  int t, t0, T1 = 0, T2 = 0, L = 10;

  for (int i = 0; i < L; i++) {
    t0 = clock ();

    for (int j = 0; j < n; j++) {
      int k = myRand (n);
      a[k] = b[k] + c[k];
    }

    t = clock ();
    T1 += (t - t0);

    t0 = clock ();

    for (int j = 0; j < n; j++) {
      int k = myRand (n);
      abc[k].a = abc[k].b + abc[k].c;
    }

    t = clock ();
    T2 += (t - t0);
  }

  cout << n << endl;
  cout << "Uninterleaved: "
       << (T1/double((CLOCKS_PER_SEC*L))) << endl
       << "Interleaved: "
       << (T2/double((CLOCKS_PER_SEC*L))) << endl;
}
```

B.2.4 Fusion

```
// fusion.cc

#include <iostream.h>
#include <time.h>
#include <stdlib.h>
#include <assert.h>

void main (int argc, char** argv) {
  int n = atoi (argv [1]);
```

```
  double* A = new double [n*n];
  double* B = new double [n*n];
  double* C = new double [n*n];

  int i, j,  t, t0, T1 = 0, T2 = 0, L = 10;

  for (int z = 0; z < L; z++) {
    for (i = 0; i < n*n; i++)
      B[i]=C[i]=1;

    t0 = clock ();
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        A[i*n+j] = 1/B[i*n+j] * C[i*n+j];
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        B[i*n+j] = A[i*n+j] + C[i*n+j];
    t = clock ();
    T1 += (t - t0);

    t0 = clock ();
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        A[i*n+j] = 1/B[i*n+j] * C[i*n+j],
          B[i*n+j] = A[i*n+j] + C[i*n+j];

    t = clock ();
    T2 += (t - t0);
  }

  cout << n << " "
       << (T1/double((CLOCKS_PER_SEC*L))) << "  "
       << n << " "
       << (T2/double((CLOCKS_PER_SEC*L))) << "  "
       << endl;
}
```

B.2.5 Blocking

```
// blocking

#include <iostream.h>
```

```
#include <stdlib.h>
//#define CPROF_COUNT
//#define CPROF "b.dat"
#include "../cprof.h"

inline int min (int x, int y) {
  return x > y ? y : x;
}

main (int argc, char** argv) {
  int i, j, k, s, jj, kk;

  int n = atoi(argv[1]);
  int z = atoi(argv[2]);

  int a[n][n],b[n][n],c[n][n];

  for (jj = 0; jj < n; jj += z)
    for (kk = 0; kk < n; kk += z)
      for (i = 0; i < n; i++)
for (j = jj; j < min (jj+z,n); j++) {
  s = 0;
  for (k = kk; k < min(kk+z,n); k++)
    s += READ(b[i][k]) * READ(c[k][j]);
  WRITE(a[i][j], READ(a[i][j]) + s);
}

#if defined(CPROF_COUNT)
  cprofstats ();
#endif
}
```

B.2.6 Improved blocking

```
// blocking, hard-coded with a blocking factor of 8

#include <iostream.h>
#include <stdlib.h>
//#define CPROF_COUNT
#define CPROF "sb.dat"
#include "../cprof.h"
#include <time.h>
```

```
inline int min (int x, int y) {
  return x > y ? y : x;
}

main (int argc, char** argv) {
  int i, j, k, jj, kk;
  int n = atoi(argv[1]);
  int z = 8;

  register float s0, s1, s2, s3, s4, s5, s6, s7, *p;
  float a[n][n],b[n][n],c[n][n];

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i][j] = 0, b[i][j] = 0, c[i][j] = 0;

  clock_t t, t0, T = 0;
  int L = 4;
  for (int l = 0; l < L; l++) {
    t0 = clock ();

  for (jj = 0; jj < n; jj += z)
    for (kk = 0; kk < n; kk += z)
      for (i = 0; i < n; i++) {
        p = &b[i][kk];
        s0 = READ(*p++);
        s1 = READ(*p++);
        s2 = READ(*p++);
        s3 = READ(*p++);
        s4 = READ(*p++);
        s5 = READ(*p++);
        s6 = READ(*p++);
        s7 = READ(*p++);
        for (j = jj; j < min (jj+z,n); j++) {
          WRITE(a[i][j], READ(a[i][j]) +
                s0 *  READ(c[kk+0][j])
                + s1 * READ(c[kk+1][j])
                + s2 * READ(c[kk+2][j])
                + s3 * READ(c[kk+3][j])
                + s4 * READ(c[kk+4][j])
                + s5 * READ(c[kk+5][j])
                + s6 * READ(c[kk+6][j])
                + s7 * READ(c[kk+7][j]));
```

```
        }
      }
    t = clock();
    T += (t - t0);
  }

  cout << n << " " << (T/double(CLOCKS_PER_SEC*L))<< endl;

#if defined(CPROF_COUNT)
  cprofstats ();
#endif
}
```

B.3 Repeated touching

B.3.1 One-sweep

```
void OneSweep (int* a, int n, int m) {
  int j, x;
  volatile int* p = a;
  int* q = a + n;
  goto l4;
l1:
  j = 0;
  goto l3;
l2:
  WRITE(*p,0);
  j++;
l3:
  if (j < m) goto l2;
  p++;
l4:
  if (p < q) goto l1;
}
```

B.3.2 Multi-sweep

```
void MSweep (int* a, int n, int m) {
  int j = 0, x;
  volatile int* p;
  int* q = a + n;
```

```
    goto l4;
  l1:
    p = a;
    goto l3;
  l2:
    WRITE(*p,0);
    p++;
  l3:
    if (p < q) goto l2;
    j++;
  l4:
    if (j < m) goto l1;
  }
```

B.4 Heap construction

B.4.1 Sift-up

```
void PureCSiftUp (Element* a, unsigned i, unsigned n) {
  Element t = READ(a[i]);
  Element t2, t3;
  unsigned j = i;
  goto PureCSiftUp1;
PureCSiftUp0:
  WRITE(a[j],t2);
  j = parent(j);
PureCSiftUp1:
  if (j == root) goto PureCSiftUp2;
  t3 = parent(j);
  t2 = READ(a[t3]);
  if (t2 > t) goto PureCSiftUp0;
PureCSiftUp2:
  WRITE(a[j],t);
}
```

B.4.2 Sift-down

```
void PureCSiftDown (Element* a, unsigned n, unsigned i) {
  Element t = READ(a[i]);
  Element t2, t3;
  unsigned j = child(i);
```

```
  goto PureCSiftDown2;
PureCSiftDown0:
  t2 = READ(a[j]);
  t3 = READ(a[j+1]);
  if (t2 <= t3) goto PureCSiftDown1;
  j++;
  t2 = t3;
PureCSiftDown1:
  if (t <= t2) goto PureCSiftDown3;
  WRITE(a[i],t2);
  i = j;
  j = child(i);
PureCSiftDown2:
  if (j < n) goto PureCSiftDown0;
PureCSiftDown3:
  if (j < n) goto PureCSiftDown4;
  t2 = READ(a[n]);
  if (t > t2) goto PureCSiftDown4;
  j = parent(n);
  WRITE(a[j],t2);
  WRITE(a[n],t);
  goto PureCSiftDown5;
PureCSiftDown4:
  WRITE(a[i],t);
PureCSiftDown5:
  ;
}
```

B.4.3 Median partitioning

```
int* PureCPartition(int* p, int* q, int* r) {
  int x, y, z, *i, *j;
  x = *q;
  z = *p;
  *q = z;
  *p = x;
  i = p;
  j = r;
Partition1:
  i++;
  if (i >= r) goto Partition2;
  z = *i;
  if (z < x) goto Partition1;
```

```
  Partition2:
    j--;
    z = *j;
    if (z > x) goto Partition2;
    if (i > j) goto Partition3;
    z = *i;
    y = *j;
    *j = z;
    *i = y;
    goto Partition1;
  Partition3:
    z = *p;
    y = *j;
    *j = z;
    *p = y;
    return j;
  }
```

B.4.4 Bottom-up non-recursive Floyd

```
  void NonRecursivePostOrder (int n) {
    int i, j, k, z;
    i = (1<<(log_2(n))) - 1;
    j = i>>1;
    k = n>>1;
    while (i < j) {
      if (i <= k)
        SiftDown(i);
      for (z=i; !(z&1); z>>=1)
        SiftDown(z>>1);
    }
  }
```

B.4.5 Heap experiment code

```
// heap.cc

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
```

```
//#define CPROF_COUNT yes
//#define CPROF "heaptrace.dat"
#include "../cprof.h"

typedef int Element;

#define child(p) (p<<1)
#define parent(p) (p>>1)
#define root (1)

template<class T>
inline void Swap(T& x, T& y) {
  T z = READ(x);
  WRITE(x, READ(y));
  WRITE(y, z);
}

struct Heap {
  enum ConsMethod { williams, floyd, median,
                    postorder, nonrec, experiment };

  Element* heap;
  unsigned n;

  Heap (Element* array, int size, ConsMethod mthd)
    : n (size), heap (array) {
    switch (mthd) {
    case williams:    WilliamsConstruct();    break;
    case floyd:       FloydConstruct();       break;
    case median:      MedianConstruct();      break;
    case postorder:   PostOrderConstruct (1); break;
    case nonrec:      NonRecPostOrder ();     break;
    case experiment:  PO2(1);                 break;
    }
  }

  void PureCSiftUp (unsigned i) {
    Element* a = heap - 1;
    Element t = READ(a[i]);
    Element t2, t3;
    unsigned j = i;
    goto PureCSiftUp1;
  PureCSiftUp0:
```

```
    WRITE(a[j],t2);
    j = parent(j);
  PureCSiftUp1:
    if (j == root) goto PureCSiftUp2;
    t3 = parent(j);
    t2 = READ(a[t3]);
    if (t2 > t) goto PureCSiftUp0;
  PureCSiftUp2:
    WRITE(a[j],t);
  }

  void SiftUp (int i) {
    Element* a = heap - 1, t = READ(a[i]), t2;
    int j;
    for (j = i;
         j > root && (t2 = READ(a[parent(j)])) > t;
         j = parent(j))
      WRITE(a[j], t2);
    WRITE(a[j], t);
  }

  void PureCSiftDown (unsigned i) {
    Element* a = heap - 1;
    Element t = READ(a[i]);
    Element t2, t3;
    unsigned j = child(i);
    goto PureCSiftDown1;
  PureCSiftDown0:
    t2 = READ(a[j]);
    t3 = READ(a[j+1]);
    if (t2 > t3) j++, t2 = t3;
    if (t <= t2) goto PureCSiftDown2;
    WRITE(a[i],t2);
    i = j;
    j = child(i);
  PureCSiftDown1:
    if (j < n) goto PureCSiftDown0;
  PureCSiftDown2:
    if (j < n) goto PureCSiftDown3;
    t2 = READ(a[n]);
    if (t > t2) goto PureCSiftDown3;
    j = parent(n);
    WRITE(a[j],t2);
    WRITE(a[n],t);
```

```
  goto PureCSiftDown4;
PureCSiftDown3:
  WRITE(a[i],t);
PureCSiftDown4:
  ;
}

void SiftDown (int i) {
  Element* a = heap - 1, t = READ(a[i]), t2, t3;
  unsigned j;
  for (j = child (i); j < n; i = j, j = child(i)) {
    t2 = READ(a[j]), t3 = READ(a[j+1]);
    if (t2 > t3) j++, t2 = t3;
    if (t > t2) WRITE(a[i],t2);
    else break;
  }
  if (j == n && t > READ(a[n])) {
    WRITE(a[parent(n)],READ(a[n]));
    WRITE(a[n],t);
  } else WRITE(a[i], t);
}

void PostOrderConstruct (unsigned i) {
  if (2*i+1 <= n/2) PostOrderConstruct (2*i+1);
  if (2*i <= n/2) PostOrderConstruct (2*i);
  SiftDown (i);
}

void PO2(unsigned i) {
  PureCSiftUp(i);
  if (2*i<=n) PO2(2*i);
  if (2*i+1<=n) PO2(2*i+1);
}

unsigned log2 (double i) {
  return (i<2)?0:1+log2(i/2+0.5);
}

void NonRecPostOrder () {
  int i, j, k, z;
  for (i = 1; i < n; i*=2);
  i = i / 2 - 1, j = i / 2, k = n / 2;
  for (; i > j; i--) {
    if (i <= k)
```

```
      SiftDown(i);
    for (z = i; !(z&1); z/= 2)
      SiftDown(z/2);
  }
}

void WilliamsConstruct () {
  for (unsigned i = 1; i <= n; i++)
    PureCSiftUp(i);
}

void FloydConstruct () {
  for (unsigned i = n/2; i > 0; i--)
    PureCSiftDown(i);
}

unsigned Random (unsigned r) {
  return unsigned ((double(r) * rand ()) / RAND_MAX);
}

Element* PureCPartition(Element* p, Element* q, Element* r) {
  Element x, y, z, *i, *j;
  x = READ(*q);
  z = READ(*p);
  WRITE(*q, z);
  WRITE(*p, x);
  i = p;
  j = r;
Partition1:
  i++;
  if (i >= r) goto Partition2;
  z = READ(*i);
  if (z < x) goto Partition1;
Partition2:
  j--;
  y = READ(*j);
  if (y > x) goto Partition2;
  if (i >= j) goto Partition3;
  WRITE(*j, z);
  WRITE(*i, y);
  goto Partition1;
Partition3:
  WRITE(*j, x);
  WRITE(*p, y);
```

```
  return j;
}

Element* Partition(Element* p, Element* q, Element* r) {
  Element t = READ(*q);
  Swap(*p, *q);
  Element* i = p;
  Element* j = r;
  for (;;) {
    do i++; while (i < r && READ(*i) < t);
    do j--; while (t < READ(*j));
    if (i < j)
      Swap(*i, *j);
    else {
      Swap(*p, *j);
      return j;
    }
  }
}

Element* Select(Element* p, Element* q, Element* r) {
  if (r - p == 1) return p;
  Element* p2 = p + Random (r-p-1);
  Element* q2 = PureCPartition (p, p2, r);
  if (q < q2)
    return Select(p, q, q2);
  else if (q == q2)
    return q;
  else return Select(q2 + 1, q, r);
}

void MedianConstruct () {
  for (unsigned nn = n; nn > 1; nn /= 2)
    Select (heap, heap + nn/2, heap + nn);
}

bool Verify () {
  Element* a = heap - 1;
  bool violated = false;
  for (int i = n - 1; i > 1; i--)
    if (a[i/2] > a[i]) {
      cerr << ":" << i;
       violated = true;
    }
```

```
    return !violated;
  }

  void Permute (Element* a, unsigned n, char z) {
    if (z == 'a')
      for (unsigned i = 0; i < n; i++)
        a[i] = i+1;
    else
      for (unsigned i = 0; i < n; i++)
        a[i] = n - i;

    if (z == 'p')
      for (unsigned i = 1; i < n; i++) {
        unsigned j = unsigned ((double(i) * rand ()) /
                               RAND_MAX);
        Element t = a[i];
        a[i] = a[j], a [j] = t;
      }
  }
};

main (int argc, char** argv) {
  if (argc < 3) {
    cerr << "usage: " << argv[0] << " "
         << "<elements> <method> <data>"
         << endl;
    cerr << "       where <method> \\in {williams, floyd, "
      "median, postorder, nonrec}" << endl
         << "         and <data> \\in "
         << "{ascending, descending, permuted}"
         << endl;
    exit (99);
  }

  int N = atoi(argv[1]);
  Element* a = new Element [N];
  char choice = *argv[2], z;

  Heap* heap;

  Heap::ConsMethod method;

  if (argc > 3)
    z = *argv[3];
```

```
  else
    z = 'p';

  switch (choice) {
  case 'w': method = Heap::williams; break;
  case 'f': method = Heap::floyd; break;
  case 'm': method = Heap::median; break;
  case 'p': method = Heap::postorder; break;
  case 'n': method = Heap::nonrec; break;
  case 'e': method = Heap::experiment; break;
  default:
    cerr << argv[0]<< ": bad construction flag (w|f|m|p|n)"
         << endl;
    exit (99);
  }

  clock_t t, t0, T = 0;
  int L = 1;
  for (int l = 0; l < L; l++) {
    heap->Permute (a, N, z);
    t0 = clock ();
    heap = new Heap (a, N, method);
    t = clock();
    T += (t - t0);
  }

  cout << N << " " << (T/double(CLOCKS_PER_SEC*L))<< endl;

#if defined(CPROF_COUNT)
  cprofstats ();
#endif

  if (!heap->Verify ())
    cerr << "Heap construction failed." << endl;

  delete a;
  delete heap;

}
```

B.5 Sorting

B.5.1 Two-way merge

```
// two-way merge
// merges the sorted subarrays up, vp
// of size (vp_last - vp)
// where the last element of vp is smallest

template<class E>
inline void Merge2
(E* up, E* vp, E* dp, E* up_last, E* vp_last) {
  //  E u = READ(*up), v = READ(*vp);
  E u = *up, v = *vp;
  goto test;
 out_u:
  WRITE(*dp++,u);
  u = READ(*++up);
 test:
  if (u < v) goto out_u;
 out_v:
  WRITE(*dp++,v);
  if (++vp == vp_last) goto mexit;
  v = READ(*vp);
  goto test;
 mexit:
  WRITE(*dp++,u);
  if (++up == up_last) goto mexit2;
  u = READ(*up);
  goto mexit;
 mexit2: ;
}
```

B.5.2 Three-way merge

```
// three-way merge
// merges the sorted subarrays up, vp, xp
// of size (xp_last - xp)
// where the last element of xp is smallest

template<class E>
inline void Merge3
```

```
(E* up, E* vp, E* xp, E* dp,
 E* up_last, E* vp_last, E* xp_last) {
  //  E u = READ(*up), v = READ(*vp), x = READ(*xp);
  E u = *up, v = *vp, x = *xp;
  if (u <= v) goto test_ux;
  goto test_vx;
 out_u:
  WRITE(*dp++,u);
  u = READ(*++up);
  if (v < u) goto test_vx;
 test_ux:
  if (u <= x) goto out_u;
 out_xu:
  WRITE(*dp++,x);
  if (++xp == xp_last) goto mexit;
  x = READ(*xp);
  goto test_ux;
 out_v:
  WRITE(*dp++,v);
  v = READ(*++vp);
  if (u <= v) goto test_ux;
 test_vx:
  if (v <= x) goto out_v;
 out_xv:
  WRITE(*dp++,x);
  if (++xp == xp_last) goto mexit;
  x = READ(*xp);
  goto test_vx;
 mexit:
  //assert(up < up_last);
  //assert(vp < vp_last);
  //assert(xp == xp_last);
  if (*(vp_last-1) < *(up_last-1))
    Merge2(up,vp,dp,up_last,vp_last);
  else
    Merge2(vp,up,dp,vp_last,up_last);
}
```

B.5.3 Four-way merge

```
// four-way merge
// merges the sorted subarrays up, vp, xp, yp
// of size (yp_last - yp)
// where the last element of yp is smallest

template<class E>
inline void Merge4
(E* up, E* vp, E* xp, E* yp, E* dp,
 E* up_last, E* vp_last, E* xp_last, E* yp_last) {
  E u = READ(*up), v = READ(*vp), x = READ(*xp), y = READ(*yp);
  if (u <= v && x <= y) goto test_ux;
  if (u <= v && x > y) goto test_uy;
  if (u > v && x <= y) goto test_vx;
  goto test_vy;
 out_ux:
  WRITE(*dp++,u);
  u = READ(*++up);
  if (v < u) goto test_vx;
 test_ux:
  if (u <= x) goto out_ux;
 out_xu:
  WRITE(*dp++,x);
  x = READ(*++xp);
  if (x <= y) goto test_ux;
  if (y < u) goto out_yu;
 out_uy:
  WRITE(*dp++,u);
  u = READ(*++up);
  if (v < u) goto test_vy;
 test_uy:
  if (u <= y) goto out_uy;
 out_yu:
  WRITE(*dp++,y);
  if (++yp == yp_last) goto mexit;
  y = READ(*yp);
  if (y < x) goto test_uy;
  goto test_ux;
 out_vx:
  WRITE(*dp++,v);
  v = READ(*++vp);
  if (u <= v) goto test_ux;
```

```
 test_vx:
  if (v <= x) goto out_vx;
 out_xv:
  WRITE(*dp++,x);
  x = READ(*++xp);
  if (x <= y) goto test_vx;
  if (y < v) goto out_yv;
 out_vy:
  WRITE(*dp++,v);
  v = READ(*++vp);
  if (u <= v) goto test_uy;
 test_vy:
  if (v <= y) goto out_vy;
 out_yv:
  WRITE(*dp++,y);
  if (++yp == yp_last) goto mexit;
  y = READ(*yp);
  if (y < x) goto test_vy;
  goto test_vx;
 mexit:
  //assert (up < up_last);
  //assert (vp < vp_last);
  //assert (xp < xp_last);
  //assert (yp == yp_last);

  // determine subarray with smallest tail
  u = *(up_last-1);
  v = *(vp_last-1);
  x = *(xp_last-1);
  if (u < x) {
    swap (up, xp); swap (up_last, xp_last); swap(u,x);
  }
  if (v < x) { swap (vp, xp); swap (vp_last, xp_last); }
  Merge3(up,vp,xp,dp,up_last,vp_last,xp_last);
}
```

B.5.4 Mergesort programs

```
// 4-way mergesort
// this currently only handles arrays of sizes
// which are powers of 4...

template<class E>
```

```
E* MergeSort4(E* a, E* b, int n) {
  E *up, *vp, *xp, *yp, *dp, u, v, x, y, *t;
  E *tup, *tvp, *txp, *typ;

  for (int l = 1; l <= (n>>2); l <<= 2) {
    for (int i = 0; i < n; i += l<<2) {
      up = a+i, vp = up+l, xp = vp+l, yp = xp+l;

      // determine subarray with smallest tail
      u = *(up+l-1);
      v = *(vp+l-1);
      x = *(xp+l-1);
      y = *(yp+l-1);

      tup = up, tvp = vp, txp = xp, typ = yp;

      if (x < y) { swap (txp, typ); swap (x,y); }
      if (v < y) { swap (tvp, typ); swap (v,y); }
      if (u < y) { swap (tup, typ); }
      Merge4(tup,tvp,txp,typ,b+i,tup+l,tvp+l,txp+l,typ+l);
    }
    t = a, a = b, b = t;
  }
  return a;
}

// 2-way mergesort
// this currently only handles arrays of sizes
// which are powers of 2...

template<class E>
E* MergeSort2(E* a, E* b, int n) {
  E *up, *vp, *dp, u, v, *t;
  E *tup, *tvp;

  for (int l = 1; l <= (n>>1); l <<= 1) {
    for (int i = 0; i < n; i += l<<1) {
      up = a+i, vp = up+l;

      // determine subarray with smallest tail
      u = *(up+l-1);
      v = *(vp+l-1);
      tup = up, tvp = vp;
      if (u < v) { swap (tup, tvp); }
```

```
      Merge2(tup,tvp,b+i,tup+l,tvp+l);
    }
    t = a, a = b, b = t;
  }
  return a;
}


// tiled mergesort
// assumes that tilesize divides n

template<class E>
E* TiledMergeSort(E* a, E* b, int n, int tilesize) {
  E *up, *vp, *xp, *yp, *dp, u, v, x, y, *t;
  E *tup, *tvp, *txp, *typ, *d;

  if (n > tilesize) {
    d = MergeSort4(a,b,tilesize);
    for (int i = tilesize; i < n; i += tilesize)
      MergeSort4(a+i, b+i, tilesize);

    if (d != a)
      t = a, a = b, b = t;

    for (int l = tilesize; l <= (n>>2); l <<= 2) {
      for (int i = 0; i < n; i += l<<2) {
        up = a+i, vp = up+l, xp = vp+l, yp = xp+l;

        // determine subarray with smallest tail
        u = READ(*(up+l-1));
        v = READ(*(vp+l-1));
        x = READ(*(xp+l-1));
        y = READ(*(yp+l-1));

        tup = up, tvp = vp, txp = xp, typ = yp;

        if (x < y) { swap (txp, typ); swap (x,y); }
        if (v < y) { swap (tvp, typ); swap (v,y); }
        if (u < y) { swap (tup, typ); }
        Merge4(tup,tvp,txp,typ,b+i,tup+l,tvp+l,txp+l,typ+l);
      }
      t = a, a = b, b = t;
    }
    return a;
```

```
  } else return MergeSort4(a,b,n);
}
```

B.6 Connected components

```
// concomp.cc
// sequential-access connected components

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <time.h>
#define CPROF "seq.dat"
#include "../cprof.h"
#include "mergesort.cc"

const nil = -1;

template<class T>
inline void swap (T& a, T& b) {
  T t = a;
  a = b, b = t;
}

struct Edge {
  int v, w, p;
  Edge (int v0, int w0) : v(v0), w(w0), p(0) { }
  Edge () : v(nil), w(nil), p(nil) { }
  operator<(Edge& e) {
    return ((v < e.v) || (v == e.v && w < e.w));
  }
  operator==(Edge& e) {
    return ((p == e.p) && (v == e.v) && (w == e.w));
  }
};

struct RevEdge {
  int w, v, p;
  operator<(RevEdge& e) {
    return ((v < e.v) || (v == e.v && w < e.w));
  }
  operator==(RevEdge& e) {
```

```
    return ((p == e.p) && (v == e.v) && (w == e.w));
  }
};

Edge* aux;

inline Edge* vsort (Edge*& a, int& n) {
  MergeSort(a, aux, n);
  return a;
}

inline Edge* wsort (Edge*& a, int& n) {
  RevEdge* ra = (RevEdge*) a;
  RevEdge* raux = (RevEdge*) aux;
  MergeSort (ra, raux, n);
  a = (Edge*) ra, aux = (Edge*) raux;
  return a;
};

// pre: E is sorted on w, v

void spa (Edge* e, int n) {
  int mw = nil, mv = nil, w;

  for (Edge* np = e + n; e < np; e++) {
    w = READ(e->w);
    if (mw != w)
      mw = w, mv = READ(e->v);
    WRITE(e->p, mv);
  }
}

// pre: edges sorted on v,w

void spj (Edge* e, int n) {
  int mv = nil, mp = nil, v, p;

  for (Edge* np = e + n; e < np; e++) {
    v = READ(e->v);
    if (mv != v) {
      mv = v;
      mp = READ(e->p);
    }
    else {
```

```
      p = READ(e->p);
      if (p < v) {
        WRITE(e->w, v);
        WRITE(e->v, p);
      }
      else {
        WRITE(e->v, mp);
      }
    }
  }
}

main () {
  int i, j, vnum, wnum, v, w, stage = 0;
  cin >> vnum >> wnum;

  // assumptions:
  // - vertices are numbered from 0 to n-1
  // (used for creating reflexive edges)

  Edge* edges = new Edge [wnum + vnum];
  aux = new Edge [wnum + vnum];

  for (i = 0; i < wnum; i++) {
    cin >> v >> w;
    if (v > w) swap (v, w);        // effect: acyclic graph
    edges [i].v = v;
    edges [i].w = w;
  }

  for (j = 0; j < vnum; j++, i++)
    edges [i].v = edges [i].w = j;

  wnum += vnum;

  for (int z = wnum; z; z >>= 1) {
    stage++;
    wsort (edges, wnum);
    spa (edges, wnum);
    vsort (edges, wnum);
    spj (edges, wnum);
  }

  delete edges;
```

```
  delete aux;
}
```