

Refining the pure-C cost model

M.Sc. Thesis

Sofus Mortensen

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

Revised March 2001

Summary

This thesis is submitted in partial fulfilment of the requirements of being awarded the Danish *Kandidatgrad* (*candidatus scientiarum*) at the Department of Computer Science, University of Copenhagen (DIKU) and was researched and written during the period July 2000 - March 2001 under the supervision of Jyrki Katajainen.

The pure-C cost model, a simple but realistic model for the performance of programs, was first presented by Katajainen and Träff in 1997. The model is refined by Bojesen, Katajainen and Spork in 1999 in to include the cost of cache misses. In this thesis the model will be further refined to include the cost of *branch mispredictions*. Furthermore it will be discussed how to modify the pure-C cost model to account for the instruction level parallelism of today's *superscalar* processors.

The efficiency of various *mergesort* and *binary search programs* is analysed under the refined model.

It is concluded that branch prediction has a considerable effect on the performance of programs and that the refinement of the pure-C cost model does to some extent improve the accuracy of the model. In spite of this it is concluded that the pure-C model is not sufficient for describing modern computers. The effect of instruction level parallelism need to be taken into account as well.

Acknowledgement

Thanks must go to my advisor Jyrki Katajainen for fruitful discussions and comments. Also I must thank my better half Winnie for putting up with me during the final weeks of the preparation of this text.

Contents

1	Introduction	1
2	Modern computer architecture	3
2.1	Introduction	3
2.2	Pipelining	3
2.2.1	Control hazards	4
2.3	Superscalar processors	6
2.4	Memory organisation	10
2.5	Summary	12
3	The pure-C cost model	13
3.1	Introduction	13
3.2	pure-C	13
3.2.1	pure-C definition	13
3.3	Refining the pure-C cost model	16
3.4	Hierarchical memory model	16
3.5	Branch prediction	17
3.5.1	pure-C with Branch Hints	18
3.5.2	pure-C with conditional assignment	19
3.6	Instruction level parallelism	22
3.7	Function calls	23
3.8	Indirect branching	23
3.9	Calculation of $\lceil \lg x \rceil$	24
3.10	pure-C in STL style	25
3.11	Summary	26
4	Tools	27
4.1	Computer systems	27
4.2	Compiler	27
4.3	Gen<X> and X-Code	29
5	Binary search and range search	31
5.1	Introduction	31
5.2	Binary search	31
5.2.1	The STL implementation	33
5.2.2	Binary search with conditional assignments	36
5.2.3	Straight-line binary search	38
5.2.4	Summary	42

5.3	Range search	42
5.3.1	Range search using STL <code>lower_bound</code>	42
5.3.2	Range search as paired straight-line binary search	44
5.4	Experiments	47
5.4.1	Expectations	48
5.4.2	Results	49
5.5	Summary	52
6	Mergesort	53
6.1	Introduction	53
6.1.1	Stable sorting	54
6.2	Merging	54
6.2.1	Simple merge	55
6.2.2	The 2-way merge of Katajainen/Träff	57
6.2.3	The 3-way merge of Katajainen/Träff	59
6.2.4	The 4-way merge of Katajainen/Träff	62
6.2.5	Back-to-back merge with conditional assignments	64
6.2.6	Paired back-to-back merge with conditional assignments	66
6.2.7	Summary of merge algorithms	68
6.3	Sorting	68
6.3.1	Top-down mergesorting	68
6.3.2	Bottom-up mergesort	75
6.3.3	Natural mergesort	75
6.3.4	Tiled bottom-up mergesort	76
6.4	Experiments	76
6.4.1	Expectations	77
6.4.2	Results	79
6.5	Summary	80
7	Conclusion	83
	Bibliography	84
A	Program source code	87
A.1	Binary Search	87
A.2	Mergesort	97
A.3	<code>limit_data</code>	116
A.4	<code>microtime.h</code>	118

List of Programs

1	Example - with conditional branching	18
2	Example - with conditional assignments	19
3	STL <code>lower_bound</code>	33
4	pure-C implementation of Program 3.	34
5	Optimised <code>lower_bound</code> with conditional assignments.	36
6	pure-C version of Program 5.	37
7	Straight-line binary search	39
8	pure-C implementation of straight-line binary search	40
9	Range search with “cold start”	43
10	Range search with “warm start”	44
11	Paired straight-line binary search	45
12	pure-C implementation of paired straight-line binary search	46
13	Simple merge	55
14	Optimised simple merge	56
15	Katajainen/Träff 2-way merge	58
16	Katajainen/Träff 3-way merge	60
17	Katajainen/Träff 4-way merge	63
18	Back-to-back merge with conditional assignments	65
19	Paired back-to-back merge with conditional assignments	67
20	Top-down mergesort, version 1	69
21	Top-down mergesort, version 2	70
22	Top-down back-to-back mergesort	72

Chapter 1

Introduction

In order to analyse the running time of programs we need a model of computation. The model taught and used in undergraduate algorithm courses is the *random-access machine model* (RAM). In the RAM model instructions are executed in sequence, one by one, and the measure is the number of instructions executed. To abstract from the actual execution time of each instruction, asymptotical or big-oh analysis is performed. Big-oh analysis helps us explaining for example why insertion sort is inferior to mergesort for large data, but fails to advise use in choosing between for instance heapsort and mergesort, since both are $\Theta(n \lg n)$ algorithms. To achieve that we need *meticulous* or little-oh analysis where the constant factors are analysed. In particular we are interested in the constant for the leading term of the function expressing the running time. To employ meticulous analysis we need a concrete machine model.

An example of such a model is the MIX model used by Donald E. Knuth in his book to analyse the running time of programs. Another model is the pure-C cost model, first presented by Katajainen and Träff in [KT97]. The pure-C language is a subset of the C programming language, and the associated cost model, consists simply of counting the number of executed instructions. The pure-C model has the benefit simplicity combined with a language of familiar syntax.

Having a realistic cost model has two advantages. First, it allows us to estimate the running time of existing programs. Second, it gives a valuable insight in what performance is that may help us in developing faster programs.

The topic of this thesis is the refinement of pure-C in order to better estimate the running time of programs on modern microprocessors. The pure-C model has already been refined by Bojesen *et al.* to reflect the cost of cache misses. In this thesis a pure-C cost model refinement for branch prediction is suggested. Furthermore a variant of pure-C is suggested that accounts for the fact that on today's processors can execute multiple instructions in parallel.

The thesis is organised as follows:

- Chapter 2 is a summary of the architectural trends of modern microprocessors. In particular the importance of branch prediction is treated, together with instruction level parallelism of today's microprocessors.
- The pure-C cost model is defined and discussed in Chapter 3. Using the observations made in Chapter 2 the model is refined to account for branch mispredictions.
- In Chapter 3 the tools and computer systems used for experimentation will be described.
- In Chapter 4 the performance of binary search programs is studied. Several algorithms will be studied. It will be shown both in theory and practice that if programmed carefully we are able to execute in parallel two binary searches.
- The performance of Mergesort programs is the topic of Chapter 5. The mergesort algorithms from [KT97] will be re-analysed, and alternative implementations with better branch performance will be presented. Experiments will be performed to back the theoretical findings.

Chapter 2

Modern computer architecture

2.1 Introduction

This chapter is a brief resume of the architectural trends of contemporary micro-processors. The primary sources used are *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson [HP96] and *High Performance Computing* by Dowd and Severance [DS98]. The chapter is outlined as follows:

- Section 2.2 summarises pipelining, for a more complete discussion refer to [HP96, Chapter 3].
- Section 2.3 summarises superscalar processors and post-RISC architecture. Refer to [HP96, Chapter 4] and [DS98, Chapter 2] for details.
- Section 2.4 summarises memory organisation. See [HP96, Chapter 5] and Spork [Spo99], for a more complete discussion.

2.2 Pipelining

Pipelining is a processor implementation technique where the processor starts executing a second instruction before the first has completed. Thus instructions are being executed overlapping.

Patterson [Pat95] has a wonderful analogy of pipelining with *doing laundry*. The analogy is quoted here in full length:

... One key technique is called pipelining. Anyone who has done laundry has intuitively used this tactic. The non-pipelined approach is as follows: place a load of dirty clothes in the washer. When the washer is done, place the wet load in the dryer. When the dryer is finished, fold the clothes. After the clothes are put away, start all

over again. It takes an hour to do one load this way, 20 loads take 20 hours.

The pipelined approach is much quicker. As soon as the first load is in the dryer, the second dirty load goes into the washer, and so on. All the stages operate concurrently. The pipelining paradox is that it takes the same amount of time to clean a single dirty sock by either method. Yet pipelining is faster in that more loads are finished per hour. In fact, assuming that each stage takes the same amount of time, the time saved by pipelining is proportional to the number of stages involved. In our example, pipelined laundry has four stages, so it would be nearly four times faster than non-pipelined laundry. Twenty loads would take roughly five hours.

In the world of microprocessors, the laundry loads are instructions in a program. The steps or *pipeline stages* are a subdivision of the processing of an instruction. The stages of the pipeline involves retrieving or *fetching* instructions from memory, decoding, getting register sources, executing, accessing memory, writing back to registers, and possibly others. The actual number of stages in modern processors vary from 5 to as high as 24. Table 2.1 summarises pipeline lengths of various processors.

Processor	# of stages
DEC Alpha 21164	5
MIPS R4000	8
Pentium Pro	14
Pentium IV	24

Table 2.1: Pipeline lengths of various processors.

Although in theory an n -stage pipelined processor should be n times faster than its non-pipelined equivalent, this does not hold in practice. The reason for this is that there are several events that may delay or *stall* the pipeline. Such events are called *pipeline hazards*.

2.2.1 Control hazards

The most significant kinds of pipeline hazards are the *control hazards* caused by conditional branching. To understand the problem, remember that in a pipelined processor a second instruction is already in process of being executed before the first has completed. Now the problem is the fact that the second instruction is not known until the first instruction (the conditional branch) has at least evaluated the branch condition.

The obvious solution to this problem is not to start executing any instructions after a conditional branch until it is known whether or not the branch is taken. However, this would result in conditional branching that is very expensive on processors with long pipelines.

A solution taken by some older RISC processors to reduce this stall, is to have one or two instructions following the branch, a so-called *branch delay slot*. These are always executed whether or not the branch is taken. However, for very deep pipelines the branch delay slot would require to be very long to get an acceptable low latency for conditional branches.

The technique used in contemporary processors to suppress control hazards, is called *speculative execution*. The idea is to guess or predict, whether or not the branch is taken, and then speculatively execute statements after the branch or at the branch target. When the branch condition is finally evaluated, it will be known whether or not the prediction was correct. If the prediction holds, execution can continue. Should the guess fail, all instructions following the branch that were speculatively processed in the pipeline will have to be rolled back or flushed from the pipeline.

Branch Prediction

Obviously speculative execution is highly dependent on the accuracy of the prediction of branches. Prediction algorithms are typically dynamic using historic behaviour of a branch to predict future behaviour. The simplest dynamic prediction algorithm is the *one-bit algorithm*. A bit table is being indexed by low order bits of the address of the branch, indicating whether or not the branch was last taken. When a branch is met, it is assumed by the predictor to behave the same as last time.

Let us demonstrate the one-bit algorithm by considering the following program that, given an array D of n elements, limits the value of each element to 100.

```
1  for (i=0; i<n; ++i)
2    if (D[i] > 100)
3      D[i] = 100;
```

The program contains two conditional branches one for the loop and one for the `if` statement.

Let us assume $n > 2$. The first iteration of the loop will either be unpredicted if there is no historic record, or *mispredicted* because the last time the branch was met was after the last iteration of the loop from a previous run. The next iterations ($2..n - 1$) the loop branch will always be correctly predicted, because they behave exactly as the previous one. The last iteration will fail. In total the loop branch will be correctly predicted for all but the first and the last iteration, or $n - 2$ times.

Now consider the conditional branch for the `if` statement. The success rate of the predictor algorithm is dependent on the values of the array D . Should they all be less than 100, all branches with the possible exception of the first will be correctly predicted. Maybe they are all less than 100 except for a few, say m , that are scattered out on the array with no two consecutive elements being greater than 100. If this is the case, not only will the m branches corresponding to the m values fail, but also the branch in the immediate following iteration for a total of $2m$ mispredictions.

The above example demonstrates that the one-bit algorithm is not good at handling exceptional branches. Not only is the exceptional branch itself penalised, but also the consecutive appearance of the branch.

The *two-bit algorithm* is designed specifically to counter this problem. The two-bit algorithm uses a bit table as the one-bit algorithm, with each entry having two bits instead of one. The two bits indicate a state in a finite-state machine with 4 states (see table 2.2).

State	Prediction	if taken go to	if not taken go to
0	taken	0	1
1	taken	0	2
2	not taken	1	3
3	not taken	2	3

Table 2.2: State machine used by the two-bit prediction algorithm

If two consecutive occurrences of the same branch take the same direction, the predictor will be in state 0 or 3. Note that a single misbehaving branch occurrence will change the state to 1 or 2, and thus not cause the next occurrence to mispredict the branch.

In the example from before, the loop branch will only fail on the last iteration and not on the first iteration of a subsequent run of the program. If as before the if statement is rarely taken (m times), only one prediction will fail per branch taken.

2.3 Superscalar processors

A *superscalar processor* is a pipelined processor where each stage is capable of processing more than one instruction. Thus not only are the instructions executed overlapping but also in parallel. Superscalar processors are typically capable of executing up to 3-8 instructions per clock cycle. The actual number of instructions per clock cycle is often considerably lower, due to pipelining hazards. In order for instructions to be executed in parallel, they must be *independent*¹.

There are three basic types of dependencies:

1. Data dependencies

Instruction i is *data dependent* on instruction j , if:

- One of instruction i sources depend on the output of instruction j .
- One of instruction i sources is data dependent of the output of instruction j .

¹For thorough treatise on dependencies and parallelism, see Dowd and Severance [DS98, Chapter 9]

Note that the definition is recursive, and that data dependency is a transitive. Consider the following example written in x86 machine code²:

```

1  add eax, 16      ; eax = 16;
2  mov ebx, eax    ; ebx = eax;
3  xor ecx, ebx    ; ecx = ecx ^ ebx;

```

Instruction two is data dependent on the instruction one, because the source `eax` is the output of instruction one. Instruction three is data dependent on instruction two, because of `ebx`. Since data dependency is transitive, instruction three is also data dependent on instruction one.

2. Name dependencies

Two instructions are said to be *name dependent* if they share the same register or memory resource, but do not have data dependency between them. There are two types of name dependencies, anti-dependencies and output dependencies.

(a) Anti-dependencies

An instruction *A* is said to be anti-dependent of instruction *B*, if *A* write to a target that *B* reads from.

Consider the following example:

```

1  add ebx, eax    ; ebx = ebx + eax;
2  mov eax, 0     ; eax = 0;

```

Instruction two is anti-dependent on instruction one because instruction two writes to the same register, `eax`, that instruction one reads from.

(b) Output dependencies

An output dependency arise when two instructions have the output target register or memory address.

Consider the following example:

```

1  mov eax, ebx   ; eax = ebx;
2  mov eax, 0    ; eax = 0;

```

Instruction two is output dependent on instruction one because instruction two writes to same register, `eax`, as instruction one.

3. Control dependencies

An instruction is *control dependent* on a branch if its execution depends on whether the branch is taken or not. A typical example of control dependency, is that the body of an `if` statement is dependent on the branching condition.

Consider the following example:

```

1          cmp eax, ebx ;
2          jl  label   ; if (eax < ebx) goto label;
3          mov ecx, 1  ; ecx = 1;
4  label:  mov edx, 1  ; edx = 1;

```

²The x86 instruction set is implemented in both the Intel Pentium 2 and AMD Athlon, the processors used for experimentation in this thesis. See Section 4.1.

Instruction three is control dependent on instruction two, since it is only executed if branch (instruction two) is not taken. Also observe that the branch itself (instruction two) is data dependent on instruction one.

Note that due to speculative execution (see page 5), instructions that are control dependent can in fact be executed in parallel.

Deeply pipelined processors may have the execution step itself divided into several pipeline steps. Since the execution step spans over several clock cycles, a dependency may prevent an instruction from entering the execution unit for several clock cycles until the blocking instruction has completed execution.

Dependencies are a great concern for the scheduling of instructions. This is particularly true if the processor is issuing multiple instructions for execution in a pipelined execution unit.

Dynamic scheduling

Dynamic scheduling or *out-of-order execution* is a pipelining technique, whereby the processor attempt to rearrange instructions on-the-fly to obtain a sequence with fewer pipeline stalls. The way that this work is that there is a *re-order buffer* before the execution unit capable of storing several instructions (see Figure 2.1). Each clock cycle the scheduler selects instructions from the buffer that are ready for executions, meaning that they have no dependencies unfinished instruction.

Instructions are executed out-of-order, but care have to be taken to make sure that they are retired in-order. The reason for this is to handle situations where instructions cause exceptions. If an instruction cause an exception, any instruction already executed following the exception causing instruction, must be rolled back. Therefore instruction must be retired in-order. After execution instructions will wait in the retirement buffer until ready for retirement when all previous instructions have completed execution.

The re-order buffer of Intel Pentium 3 can contain 40 instructions³, and the buffer AMD Athlon microprocessor, 72 instructions⁴.

Register renaming

Name dependencies can often be avoided by changing the scheduling of variables. Consider the following example:

```

1   mov eax, in_data[0]    ; eax = in_data[0];
2   add eax, 16           ; eax = eax + 16;
3   mov out_data[0], eax  ; out_data[0] = eax;
4   mov eax, in_data[4]   ; eax = in_data[4];
5   add eax, 16           ; eax = eax + 16;
6   mov out_data[4], eax  ; out_data[4] = eax;
```

³See [Int]

⁴See [Adv99]

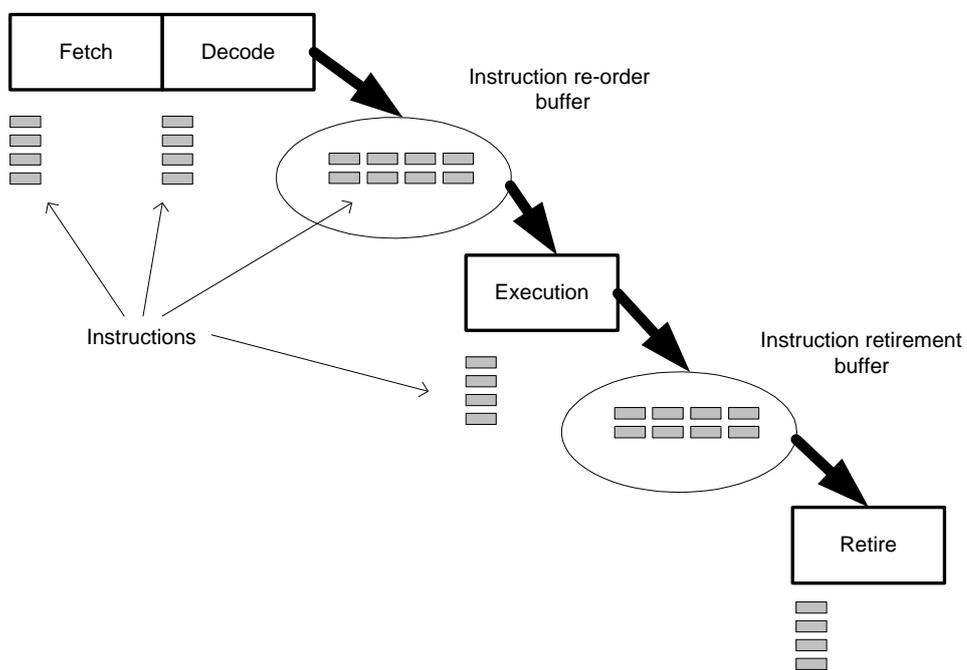


Figure 2.1: Out-of-order execution pipeline

Instruction 3 is data dependent on instruction 2, which in turn is dependent on instruction 1. The same pattern of data dependencies repeat for instructions 4, 5 and 6. Instruction 4 is anti-dependent on instruction 3, and output dependent on instruction 1 and 2. Thus none of the 6 instructions can be executed in parallel.

However the code can quite easily be rewritten to make instructions 1, 2 and 3, independent with instruction 4, 5 and 6 by *renaming* register `x` in instruction 4, 5 and 6:

```

1   mov eax, in_data[0]   ; eax = in_data[0];
2   add eax, 16           ; eax = eax + 16;
3   mov out_data[0], eax  ; out_data[0] = eax;
4   mov ebx, in_data[4]   ; ebx = in_data[4];
5   add ebx, 16           ; ebx = ebx + 16;
6   mov out_data[4], ebx  ; out_data[4] = ebx;

```

Some modern processor architectures are capable of doing register renaming on-the-fly. The processor does this by distinguishing between virtual and physical registers. The virtual registers are the registers that the instructions refer to. Before the instruction reaches the reorder buffer the virtual registers (sources and output) are mapped to physical registers, with intentions of eliminating name dependencies.

The Intel Pentium 3 and the AMD Athlon are both capable of register renaming. For these processors register renaming is very important, since the instruction set only contains 8 registers (being backward compatible to the Intel 80386 processor). The number of physical registers is much higher⁵.

2.4 Memory organisation

The memory of modern computers is organised in several layers. The layer closest to the CPU is fastest, but also the layer with the smallest capacity. Each of the following layers has greater capacity, but also higher latency, that is lower performance. The layer furthest away from the processor, is called the main memory. The layers between the processor and the main memory are called *cache layers*. Each cache layer functions as a local storage for memory data that has been accessed recently. The principle is that a memory cell that just has been accessed have higher probability of being accessed again in near future than others. The memory layers are usually a subset of the higher layers. So if a memory cell is present in a specific cache layer, it is also present in any higher cache layers, as well as the main memory.

The rationale for having caches is primarily to reduce the cost of computer systems. The intention is to provide memory systems with prices like the cheapest layer of memory, but with a performance close to the fastest layer. The basis for this is the *principle of locality* that states that typical programs spends 90% of the time in 10% of the code and data (See [HP96, page 38]).

⁵I have not been able to find out exactly how many physical registers there are. But [DS98, page 393] does state that the physical set of registers is larger than the virtual set of registers.

When the processor issues a memory read, the cache closest to the processor will be searched first. If the memory address is not present in the cache, the cache will attempt to read the data from the next memory level into the cache. While this is happening any instructions data dependent on the memory cell being read will be stalled until the read operation is completed.

A cache has several architectural characteristics:

- **Cache size (M)**. The maximum number of words that the cache can contain.
- **Block size (B)**. The cache is organised in blocks. Whenever data is read or written to a higher memory level, an entire block is being read or written. The block size B is the number of words contained in a single cache block.
- **Associativity**. The cache associativity indicates how many different locations in the cache that a given memory cell can be stored in. In a *fully associative cache*, a memory cell can be placed in any available location in the cache. In a *direct-mapped cache*, every memory cell can only be placed in one specific location in the cache. In an *r-way set associative cache* a given memory cell can only be placed in r different blocks in the cache.
- **Replacement policy**. The replacement policy indicates which block to remove when reading data from memory and no free block is available. If the cache is direct mapped, then there is no choice since the data can only be placed in block. If the cache is set associative or fully associative, then there are several blocks to choose from. The best known replacement policy is *least recently used* (LRU). The policy is to flush the block where most time has passed since the last access.

Although memory may have high latency, high memory bandwidth can still be maintained by reading multiple words at once from memory in parallel. Memory is usually organised so that the reading or writing of cache block can be performed efficiently.

The fact that caches read and write data in blocks, means that when ever you read one word from memory you are actually getting B words. This means that reading n words in sequence will result in n/B cache misses. If we instead read n words, such that no two words lie in the same cache block, we will get n cache misses, or a speed-down of B . Thus it is very important when designing algorithms for systems with hierarchical memory to access memory sequentially.

It is obvious that instructions that are data dependent on a value being loaded from memory will be stalled until the load has completed. Should the load operation result in a cache miss, instructions will be stalled even longer until the cache has been updated. When updating the cache, there is a chance that the block that is going to be flushed has been modified. In that case even longer time will pass, waiting for the block to be written to the next cache layer (or main memory).

For data writes, most processors are organised, so that the write operation does not block the execution of other instructions.

The write itself may be handled differently in the cache depending on the architecture. If the data accessed is already in the cache, then the data element in the cache will be changed, and the block marked as *dirty* (modified). If the data is not cached, then typically the cache block containing the element will be read in from memory (or next cache layer), updated and have the dirty flag set. However some caches have the policy of letting writes pass through the cache (*write-through policy*), if the containing block is not in the cache.

The Pentium 3 processor updates the cache when writing. It is assumed that in most cases data written to memory is going to be used again in near future. However in some cases you know that you are not going to read the data again for a long time. For those situations the Pentium 3 has special instructions that will let us write data to memory without polluting the cache.

The Pentium 3 and the AMD Athlon instruction set includes *cache prefetching instructions*. A cache prefetch is a request to update the cache block containing the specified memory cell. The prefetch instruction does not update registers, and thus it does not incur any dependencies. The purpose of prefetching instructions is to hide the latency of cache misses by having the cache being updated long before it is actually going to be used.

2.5 Summary

In this chapter the most important elements of modern computer architecture has been briefly surveyed. For further discussion I urge the reader to consult [HP96].

I would like repeat three important guidelines from the survey:

- Avoid hard-to-predict branches.
Hard-to-predict are the primary source for branch misses. Branch misses are very expensive, costing up to 15 clock cycles on Pentium Pro architecture.
- Reduce instruction dependencies to take advantage of processor parallelism.
Construct algorithms to take advantage of the fact that multiple instructions can be executed simultaneously.
- Repeated access to memory is faster than arbitrary access.
Sequential access is faster than random access.
This is without doubt the most important guideline, but also the best understood today, because of the work of Spork [Spo99], LaMarca [LaM96], and Bojesen *et al.* [BKS00, Boj00].

Chapter 3

The pure-C cost model

3.1 Introduction

The rationale of pure-C is to provide a realistic cost model for the running time of programs that is reasonable architecture independent, yet sufficiently detailed for accurate analysis. It is also important that the complexity of the model is kept at a manageable level for the analysis of programs to be feasible.

3.2 pure-C

pure-C is a subset of the C programming language (see [KR88]) and thus pure-C programs can be compiled with a standard C compiler. The pure-C cost model is not meant to be an exact model for a specific microprocessor, but a model that will give a good estimate for modern microprocessors in general. The machine model of pure-C is word RAM¹. The pure-C cost model was introduced by Katajainen and Träff [KT97]. pure-C is a glorified assembly language with a C-like syntax. The primitive operations of pure-C are similar in strength to a typical RISC instruction set. The pure-C machine has a limited, but unspecified number of registers, and likewise a limited but unspecified number of memory cells.

3.2.1 pure-C definition

Here follows the pure-C statements as presented by Bojesen *et al.* [BKS00]. Let x be a symbolic name of an arbitrary register (pointer or data), y , z symbolic names for a constant or a register, λ some label and p the name of a pointer register.

1. **Memory-to-register assignments:**
“ $x = *p$ ” (load statements).

¹For a definition of the RAM machine model see [Jon97]

2. **Register-to-memory assignments:**
“*p = x” (store statements).
3. **Register-to-register assignments:**
“x = y”.
4. **Unary arithmetic assignments:**
“x = \ominus y”, where $\ominus \in \{-, \sim\}$.
5. **Binary arithmetic assignments:**
“x = y \oplus z”, where $\oplus \in \{+, -, *, /, \&, |, \wedge, \ll, \gg\}$.
6. **Conditional branches:**
“if (y \triangleleft z) goto λ ”, where $\triangleleft \in \{<, <=, ==, !=, >=, >\}$.
7. **Unconditional branches:**
“goto λ ”.

pure-C has no flow control statements such as `while` and `for`. These, however, can easily be translated into pure-C constructs.

Definition 1 (pure-C unit cost). n pure-C statements takes time $n\tau$ to execute.

Thus according to the model each pure-C statement executes in time τ . The total running time of a pure-C program executing n pure-C statements is $n \cdot \tau$.

Types

The newer pure-C definition by Bojesen *et al.* [BKS00] has no direct mention of allowed type beyond registers and pointers. Although it is implied by the fact that the machine model is word RAM, that registers and pointers are words, it is not explicit whether integer register should be interpreted as signed integers (`int`) or unsigned integers (`unsigned`). The original pure-C definition from Katajainen and Träff [KT97] differ slightly from the newer. The original explicitly defines that the content of registers are signed integers (`int`), and pointer registers as pointers to signed integers (`int*`). The same types are used in the later pure-C definition from Spork [Spo99]. For the context of this thesis the types of the original definition will be used.

Complexity of statements

It can be discussed whether it is fair to use uniform cost for all binary arithmetic operations. While multiplication on many machines is just as fast as addition, division is often more expensive. The pure-C model could be refined for realism by adding a penalty cost for very complex operations like division.

As commented by Bojesen *et al.* [BKS00], the pure-C conditional branch however seems more complex than the other statements, combining comparison arithmetics with branching. Many real machine instruction sets only allow conditional branching on the status of certain flags or by simple comparisons of a register to 0. The pure-C conditional branch is complex, possibly requiring two instructions on real machines.

Addressing modes

The pure-C load/store statements reflect *pointer addressing* mode (i.e. $x = *p$), whereas most modern processors support additionally at least *displacement addressing* (i.e. $x = d[i]$;). Displacement addressing can be expressed in pure-C using 2 statements:

```
1  p = d + i;
2  x = *p;
```

Often array addressing in programs can be transformed to pointer addressing at minimal extra cost in program performance. The idea is that *sequential memory access* can be rewritten as pointer incrementing instead of index incrementing. Consider the following C program.

```
1  for (i = 0; i < n; ++i)
2    d[i] = 0;
```

First let us translate the program to an extended pure-C accepting array notation:

```
1      p = d;
2      i = 0;
3      goto test;
4  loop: p[i] = 0;
5      i = i + 1;
6  test: if (i < n) goto loop;
```

Obviously the program has an operations count of $3n + 3$.

Now consider the following equivalent pure-C program where the array notation is transformed to pointer notation.

```
1      p = d;
2      end = d + n;
3      goto test;
4  loop: *p = 0;
5      p = p + 1;
6  test: if (p < end) goto loop;
```

The pure-C operations count is $3n + 3$, and the extra cost of avoiding array notation is 1 regardless of n . Hence the addition of array notation in pure-C would for programs that access memory sequentially, be nothing but a variation of syntax. It can be argued that index notation should be added to pure-C since many processors support displacement indexing. But as long as we can rewrite our programs to pointer notations, I see no need.

3.3 Refining the pure-C cost model

On modern microprocessors the actual execution time of each instruction varies depending on the context of execution. Factors such as pipelining of instructions, availability of execution resources, caching of data, etc, can have effect on the running time of programs as already seen in Chapter 2. Observations made in Chapter 2 will be used to refine the pure-C cost model to give more realistic running-time predictions.

- The work by Spork [Spo99] and Bojesen *et al.* [BKS00] will be briefly summarised in Section 3.4. They extend pure-C with a model for the effects of a cached memory system.
- In Section 3.5, pure-C will be extended with the costs of branch mispredictions.
- In section 3.6, refinements of pure-C to take into account the peculiarities of today's highly parallel *superscalar* microprocessors, will be discussed.

3.4 Hierarchical memory model

Spork [Spo99] investigates the effect of cache on programs, or more specifically the cost of arbitrary memory access versus sequential access. Spork refines pure-C by adding penalties for cache misses, and develops a simple model for determining these by classifying memory access as sequential or random. The primary parameter in this model is the *cache block size*, B . In Spork's model n sequential memory accesses over n elements costs $n\tau_*/B$, whereas n random memory accesses costs $n\tau_*$.

Bojesen *et al.* [BKS00] refines this model by also considering the cache size, M . They analyse the number of cache misses, under the assumption of a fully-associative cache, with a least-recently-used (LRU) replacement policy. Furthermore Bojesen *et al.* defines the model to analyse multiple levels of cache.

Following is the definition as taken from [Boj00], assuming a memory hierarchy with l levels of cache:

Definition 2 (Cache cost model). *A load or store operation has an extra penalty τ_i , if it incurs a miss at level i of the memory hierarchy, $i \in \{1, 2, 3, \dots, l\}$.*

All analyses in this thesis will be assuming systems with only one cache layer unless specified otherwise. Furthermore it shall be assumed that the cache is fully associative with LRU replacement policy.

Theorem 1. *A scan over n elements stored sequentially in memory results in no more than $1 + \lceil (n-1)/B \rceil$ cache misses.*

Proof. n elements stored in sequence cover at most $1 + \lceil (n-1)/B \rceil$ cache blocks. The elements are being accessed only once in sequence, thus none of the cache blocks will read more than once. \square

Bojesen discusses in [Boj00, Chapter 4] how to analyse cache misses under the more realistic assumption of k -way set-associative and direct mapped caches. Such analyses are interesting from the point of view of learning how real-life caches can affect program performance, but is inappropriate for the pure-C cost mode for the reason of being too complicated.

Problems with the pure-C cache cost model

On many modern machines as described in Chapter 2 a store operation does not actually incur extra pipeline stalls on cache misses. This is so because the following instructions are allowed to continue, while the write operation waits for the cache to be updated.

This makes it tempting to remove the extra penalty for write misses in the pure-C model. However, be aware that a write cache miss may actually cause two cache block transfers. One for updating the cache block when the miss occurs, and another if at a later point the cache block is flushed and thus forced to be written back to memory. Should the forced write back occur during a read operation, then the any instructions dependent on the read operation will have to wait longer.

So if removing the penalty for the write miss, would require us to invent a model the extra latency incurred by modified cache blocks. In favour of simplicity, I choose to stick to the model as defined by Bojesen *et al.* [BKS00].

3.5 Branch prediction

As described in Section 2.2.1 a branch misprediction by the branch predictor causes pipeline stalls. I propose to extend the pure-C cost model with a penalty cost for branch mispredictions:

Definition 3 (Branch misprediction cost). *A branch instruction has an extra cost of τ_b if the branch is mispredicted.*

In order to analyse the number of branch mispredictions of a program we must simulate the prediction algorithm. Should we choose to use a real-life prediction algorithm, such as the two-bit algorithm, it is not hard to imagine that analysis would become quite complicated. Instead we will take another approach on analysing branch mispredictions by assuming a naive prediction algorithm that flips a coin to guess whether a branch is taken or not.

Theorem 2. *n pure-C conditional branch instructions results in average in $n/2$ mispredictions.*

Proof. The probability of an executed conditional branch resulting in a mispredict is 0.5. The prediction of branch is not affected by previous branch behaviour, thus the n predictions are independent, and the expected number of mispredictions is $n/2$. \square

There are many branches that are predictable. A typical example of this is the conditional branch in a simple C for loop of n iterations. We will expect the branch to be taken $n - 1$ times out of n . As described in the Section 2.2.1 these branches can be predicted by the two-bit algorithm with only a single branch miss for the last iteration. The naive prediction algorithm assumed in pure-C will result in unrealistically many branch misses, since every branch has 50% of being mispredicted.

3.5.1 pure-C with Branch Hints

Certain RISC architectures allow conditional branches to be accompanied with a *branching hint*, indicating whether or not the branch can be expected to be taken. The branch hints can be used by the programmer (or compiler), should he know that the branch is very likely to be taken (or not taken).

I propose to extend pure-C with branch hints as follows. A branch hint is expressed in pure-C by adding a C comment after the branch statement saying either `hint: branch taken` or `hint: branch not taken`.

Let us see this in practice with an example. The following C program limits the value of all n elements to 100.

```

1 void limit_data(unsigned long* begin, unsigned long* end)
2 {
3     for (i=0; i<n; ++i)
4         if (D[i] > 100) D[i] = 100;
5 }
```

Following is a pure-C implementation of the same program:

Program 1: Example - with conditional branching

```

1 void limit_data(unsigned long* begin, unsigned long* end)
2 {
3     unsigned long v;
4     goto test;
5 loop:
6     v = *begin;
7     if (v < 100) goto skip_if;
8     *begin = 100;
9 skip_if:
10    begin = begin + 1;
11 test:
12    if (begin < end) goto loop; /* hint: branch taken */
13 }
```

Let us now analyse the performance of this program. Let m denote the number of elements greater than 100. Assume $n > 0$. First let us count the number of pure-C statements executed. Each iteration is starting at line 5, and takes 4 or 5 statements depending on whether the current element is greater than 100.

Thus, in the loop the number of statements executed in total is $5m + 4(n - m) = 4n + m$. Before entering the loop 4 statements (1-4) are processed. The pure-C operations count is $4n + m + 4$.

In each iteration either line 4 or 9 is executed, the branch hints holds for all but the last iteration. The conditional branch of line 6 which is executed n times, cannot be predicted and has no hint, and the average number of misprediction is $n/2$ times in total.

Summing up, this results has a worst-case expected cost of $(4n + m + 5)\tau + (n/2 + 1)\tau_b$

3.5.2 pure-C with conditional assignment

With branch mispredict being expensive, some microprocessors have adopted special predicated instructions that sometimes can be used as a replacement of conditional branching. The Pentium Pro family of microprocessors has conditional register-to-register moves, where the move is committed depending on a flag.

I propose that pure-C is extended with conditional register-to-register assignments. Let x be a symbolic name of an arbitrary register (pointer or data) and y , z , s , t symbolic names for a constant or a register.

8. Conditional register-to-register assignment

“ $x = y \triangleleft z ? s : t$ ”, where $\triangleleft \in \{<, <=, ==, !=, >=, >\}$.

According to Definition 1 the conditional assignment statement (8) takes the same time to execute in the basic cost model as any other pure-C statements. Furthermore, it cannot cause branch misprediction, since there is no branching.

Now let us consider the same example as before, by replacing the hard-to-predict conditional branch with a conditional assignment.

Program 2: Example - with conditional assignments

```

1 void limit_data(unsigned long* begin, unsigned long* end)
2 {
3     unsigned long v;
4     goto test;
5 loop:
6     v = *begin;
7     v = v > 100 ? 100 : v;
8     *begin = v;
9     begin = begin + 1;
10 test:
11     if (begin < end) goto loop; /* hint: branch taken */
12 }
```

Let us now analyse the program. Every iteration is starting at line 5, and takes 5 statements. Before entering the loop 4 statements are executed. Total number

of pure-C statements is $4 + 5n$. Assuming $n > 0$ the hint of line 4 holds, and the branch of line 9 is predicted correctly every time but the last time. Total running time is $(5n + 4)\tau + \tau_b$.

Let us now assume that $m = n/2$ and then calculate how large τ_b must be before Program 2 is faster than Program 1.

$$\begin{aligned} (4n + \frac{n}{2} + 4)\tau + (\frac{n}{2} + 1)\tau_b &> (5n + 4)\tau + \tau_b \\ &\Downarrow \\ \frac{n\tau_b}{2} &> \frac{n\tau}{2} \\ &\Downarrow \\ \tau_b &> \tau \end{aligned}$$

So according to the model, Program 2 is faster if $\tau_b > \tau$.

I have measured the performance of both program on an Athlon 1000 MHz and a Pentium 450 MHz machine. Figure 3.5.2 and 3.5.2 shows execution time in clock cycles per element as function of the distribution of elements. The distribution, let us call it p , denotes probability that a given element is greater than 100. The array size used is 1024 element. For each distribution the programs have been run several times and the median of the results is displayed as the running time².

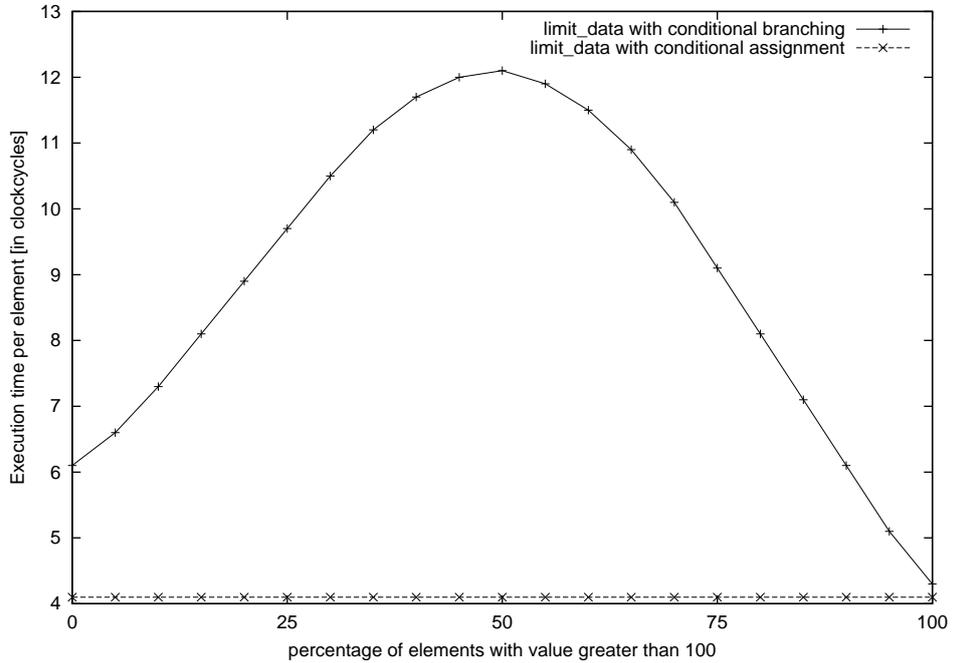


Figure 3.1: Branch prediction benchmark on 1000 MHz Athlon

²See Section 4.2 for a discussion of why median and not average is being used.

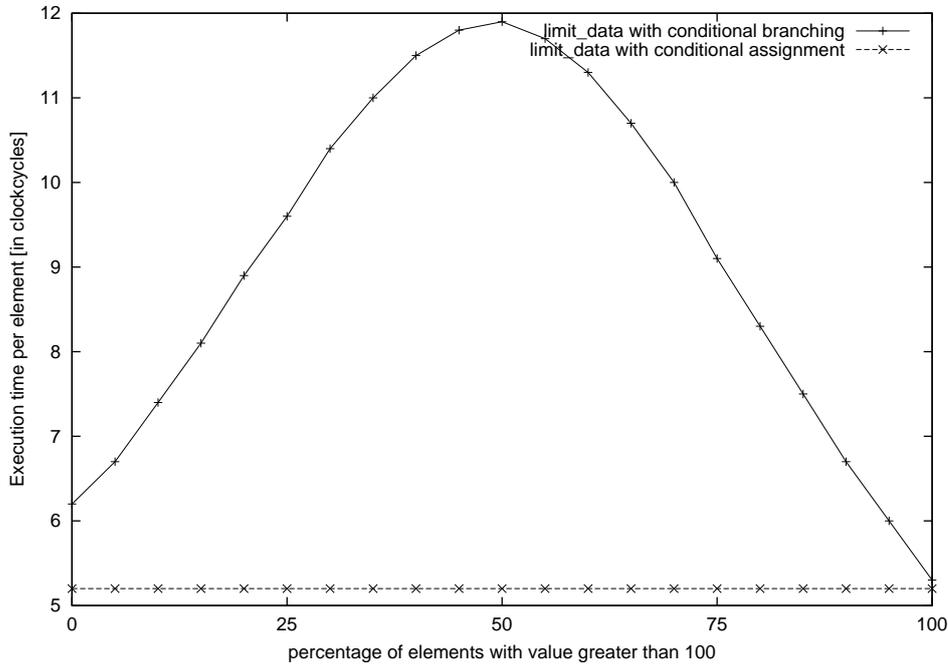


Figure 3.2: Branch prediction benchmark on 450 MHz Pentium 2

From the graph it can be seen that Programme 2 is faster for all values of p . It is interesting to see how the execution time for Programme 1 rises as p gets closer to 50%. This is because of the fact that when p is close to either 0% or 100%, the branch of line 7 in Programme 1, is more likely to behave as previous iterations.

Quite surprisingly the experiment shows that Programme 1, runs faster when $p = 100\%$ than when $p = 0\%$. This contradicts the theoretical pure-C cost. I think that the reason for this is fact that conditional branches (when correctly predicted) on some architectures take a little longer when the branch is taken compared to when the branch is not taken (and correctly predicted).

From the experiment we can make a rough estimate of both τ and τ_b for the 1000 MHz Athlon. From the graph we can read out at $p = 50\%$ that Programme 2 runs in approximately 4 clock cycles per element, and that Programme 1 runs in approximately 9.5 clock cycles (ccs) per element. Since $p = 50\%$, m has an average value of $n/2$, and the pure-C cost per element for Programme 1 is $4.5\tau + \tau_b/2$. The per element cost for Programme 2 is 5τ .

$$\begin{aligned}
 4.5\tau + \frac{\tau_b}{2} = 12.1 \text{ ccs} & \quad \wedge \quad 5\tau = 4.1 \text{ ccs} \\
 \Downarrow & \\
 \tau = 0.8 \text{ ccs} & \quad \wedge \quad \tau_b = 16.8 \text{ ccs}
 \end{aligned}$$

Since the machine is running at 1000 MHz, 1 ccs is equivalent to 1 nanosecond. The fact that the estimated τ is less than 1 clock cycle indicates that the Athlon

is capable of executing more than instruction per clock cycle.

Another way of estimating τ_b is to compare the cost of Program 1 when $p = 100\%$ with $p = 50\%$, and $p = 100\%$ with $p = 0\%$. This approach gives estimates of $\tau = 0.9 \wedge \tau_b = 16.5$ and $\tau = 1.5 \wedge \tau_b = 10.5$ for the 1000 MHz Athlon. The result when $p = 0\%$ might be tainted, because of the comment made above about branches being more expensive when taken.

Table 3.1 summarises the estimated values of τ and τ_b . As a rule of thumb for the two processors, τ_b is approximately 15τ .

System	τ	τ_b
AMD 1000 MHz Athlon	0.8 ns	16.8 ns
Intel 450 MHz Pentium 2	2.3 ns	32.0 ns

Table 3.1: Summary of branch misprediction benchmark

3.6 Instruction level parallelism

The pure-C cost model as defined assumes that all instructions are executed in sequence. This follows directly from Definition 1. The pure-C cache model assumes that no pure-C statements can be executed while a cache miss is being handled. This is however not entirely realistic. Modern superscalar microprocessors are capable of executing several instructions in each clock cycle as described in Section 2.3.

I will not extend the pure-C cost model to accommodate for instruction level parallelism, but instead suggest how an alternative model could be defined. The ILP pure-C model, as it could be called, should have a basic constant p denoting the level of parallelism, or the number of instructions that potentially can be executed simultaneously.

The basic idea is to allow up to p independent pure-C operations to be executed simultaneously:

Definition 4 (ILP pure-C cost model). *The cost of executing p independent pure-C instructions is τ . By independent is implied that none of the instructions have mutual name, anti-, output or control dependencies (Refer to Section 2.3).*

Observe that since the model does not allow control dependencies, there is no speculative parallel execution (although there is still speculative execution in the sense that instructions after the branch will be in the pipeline simultaneously with the branch, see Section 2.2.1).

When writing ILP pure-C programs, I suggest that independent instructions are written on the same line separated by semi colons, so it is easier to get a feeling of how parallelisable the program is. For the sake of simple analysis it is assumed that there is no dynamic scheduling or automatic register renaming. It is up to the pure-C programmer to schedule statements and registers for parallel execution.

The pure-C cache cost model can be reused for ILP pure-C. We allow multiple independent load or store operations to be executed simultaneously, but for each that incurs a cache miss a penalty of τ_* will be added to the total cost. The cache model is not realistic since cache misses will result in delays for any following instructions, dependent or not of the miss incurring statement. In other words, the model assumes that nothing can happen simultaneously with a cache miss. The pure-C cost model extension for branch misprediction can be reused as well in the ILP pure-C cost model. Every branch misprediction penalises the total cost with τ_b .

3.7 Function calls

pure-C has no notion of function calls. For non-recursive functions, function calls can be avoided by inlining the body of the function.

A function call is often implemented by saving local register variables to the stack, setting up the function arguments, pushing the return address on the stack and then jumping to the function entry point. Most instruction set provides a CALL and RET instruction for function call and return that handles the pushing of the return address.

I suggest that the pure-C syntax can be relaxed to allow function calls. If the call is non-recursive, the function can be inlined and there is no extra cost. If the call is recursive, then there will be a constant, but unspecified, extra cost.

3.8 Indirect branching

Most machine instruction sets including *indirect branch instructions*. An indirect branch is a branch where the target address is specified by a register variable. Indirect branches occur in C programs when calling functions through a function pointer. Another example is virtual method calls in C++. Indirect branches may also be used by the compiler to optimise switch statements. For instance, consider the following *switch* statement:

```
1  switch (i) {
2      case 0:
3          // statements
4      case 1:
5          // statements
6
7      // etc
8
9      case 100:
10         // statements
11     default:
12         // statements
13 }
```

If pure-C is extended with indirect branching, we can translate the above `switch` statement to the following pure-C code. It is assumed that `jump_table` is a table with the memory addresses of the labels `case_0`, `case_1`, etc.

```

1   if (i < 0) goto default;
2   if (i > 100) goto default;
3
4   goto jump_table[i]; // this is 3 Pure C operations.
5
6   case_0:
7     // statements
8   case_1:
9     // statements
10
11  // etc
12
13  case_100:
14    // statements
15  default:
16    // statements

```

Unfortunately the indirect `goto` statement is not part of the C programming language, and thus pure-C programs that use it would no longer be compilable with a C compiler. Without indirect branching the `switch` statement could be translated into a series conditional branches using a binary search like scheme, so that $\lceil \lg n \rceil^3$ branches were required before the right label was found (where n is the number of “cases” in the `switch` statement).

3.9 Calculation of $\lceil \lg x \rceil$

For some of the programs presented herein, we will need efficient calculation of $\lceil \lg x \rceil$, where x is an unsigned machine word. The following program calculates $\lceil \lg x \rceil$, where the machine word size is 32 bits:

```

1   // calculates floor( log2( n ) )
2   inline unsigned long log2(unsigned long n)
3   {
4       long rv = 0;
5       if (n & 0xffff0000) { rv += 16; n >>= 16; }
6       if (n & 0xff00) { rv += 8; n >>= 8; }
7       if (n & 0xf0) { rv += 4; n >>= 4; }
8       if (n & 0xc) { rv += 2; n >>= 2; }
9       if (n & 0x2) { rv += 1; n >>= 1; }
10      return rv + log_table[n];
11  }

```

The program can be optimised slightly by substituting line 7, 8, and 9 with a table lookup. The program can easily be extended to work for 64 bit machine

³Following the notation of [CLR94] $\lg n$ means $\log_2 n$.

simply by adding another line, and it is easy to prove that the pure-C cost is bounded by $O(\lg W)$, where W is the word size in bits. We will not calculate an accurate bound for $\log 2$, since running time will not affect the constant of the significant term of the running time of the programs presented herein.

There is another way of calculating $\lceil \lg x \rceil$ that works by converting the value to floating point, and read out the exponent from the floating point value. Unfortunately this method cannot be implemented in C without making assumption about how both floating point values and integer values are stored in memory⁴. Also, since pure-C does not have any notation for floating point numbers, we cannot implement it in pure-C either.

3.10 pure-C in STL style

In this thesis all programs, including pure-C programs, will be written to be generic in the style of STL⁵ whenever possible. This means that where ever possible functions will be written generic, substituting types with template types, pointer types with iterators (see [Jos99, pp. 251]). The following example demonstrates a generic version of Program 2.

```

1  template<class ForwardIterator, class LessThanComparable>
2  void limit_data(ForwardIterator begin,
3                 ForwardIterator end,
4                 const LessThanComparable& limit)
5  {
6      std::iterator_traits<ForwardIterator>
7          ::value_type v;
8
9      goto test;
10 loop:
11     v = *begin;
12     v = v < limit ? v : limit;
13     *begin = v;
14     begin = begin + 1;
15 test:
16     if (begin < end) goto loop; /* hint: branch taken */
17 }
```

When analysing such pure-C program it is assumed that all types are word size, and that all iterators are a pointer type.

⁴For instance, Intel machines store integers in little-endian, while Motorola processors in big-endian.

⁵Standard Template Library. See [Jos99]

3.11 Summary

In this chapter a realistic cost model for programs has been presented. The model counts operations similar in strength to RISC instructions, cache misses and branch prediction misses.

I have contributed to the pure-C cost model by adding penalties for branch mispredictions. Furthermore I have suggested how to make a variation of the pure-C cost model that would reflect the instruction level parallelism of today processors.

Critique

The main problem with the pure-C model is the assumption that everything is being executed in sequence, not allowing the next operation to be carried out, until the previous has finished. The ILP pure-C model is an attempt to cater for this, but does not solve the problem entirely, since it does not allow cache misses to be dealt with simultaneously with anything else.

Chapter 4

Tools

In this chapter the tools and libraries used for experimentation will be discussed.

4.1 Computer systems

I have conducted the experiments on two different computer systems both running Windows 2000, SP1. The first machine is an Intel Pentium 2 450 MHz and the other an AMD Athlon running 1000 MHz. Both machines implement the same instruction set and hence the programs for experimentation can run on both without re-compilation. Table 4.1 summarises the two systems.

Name	Clock freq.	L1 cache	L2 cache
AMD Athlon	1000 MHz	128 Kb ¹	256 Kb
Intel Pentium 2	450 MHz	32 Kb ²	512 Kb

Table 4.1: Computer systems

4.2 Compiler

I have examined two compilers for suitability for experimentation:

- Microsoft Visual C++ 6.0, abbreviated MSVC.
- Intel C++ Compiler 4.5, 5.0, abbreviated ICL.

MSVC has a very attractive IDE (Integrated Development Environment) with build-in debugger, but it turned out that the compiler back-end is not capable of using the `CMOV` instructions for conditional assignments. Furthermore MSVC lacks partial specialisation which is needed for various parts of STL, such as `iterator_traits`.

ICL is capable of generating CMOV instructions, and has partial specialisation. Furthermore it acts as a replacement for MSVC that plugs into the Microsoft IDE (Microsoft Visual Studio), therefore ICL was chosen. At the time when I was performing the experiments, I was using ICL version 4.5. To my surprise the STL algorithms `lower_bound` was very slow compared to equivalent code without the use of templates. When performing the same experiments using MSVC the STL `lower_bound` was much faster, and it suggested that ICL might have problems with optimising code heavily based on templates (a so-called *abstraction overhead*). When the beta version of ICL 5.0 was released early 2001, the problem disappeared³, and ICL now generates code for `lower_bound` that is more efficient than the code generated by MSVC.

The following switches have been used when compiling with ICL:

```
/G6 /ML /W3 /GX /O2 /YX /FD -Qip -Qrestrict -QxiM /c
```

The optimiser of ICL was doing its job too good, because it is translating simple `if` statements into equivalent conditional assignments. This is unfortunate since I wanted to examine the effect of using conditional assignments, vs. conditional branches. By removing the `-QxiM` switch, however, the usage of conditional assignment could be suppressed. Removing the `-QxiM` causes ICL to generate code for the older Pentium processor, instead of the Pentium 2. The Pentium 2 is backwards compatible with the Pentium, but the Pentium does not have conditional assignments.

STL

For all implementation I have used the Standard C++ Library implementation of STLport 4.0. MSVC has its own Std. C++ Lib. implementation, but personal experience has showed it to be hopelessly flawed, and far from ISO compliant.

Timing

Pentium Pro compatible microprocessors have a special 64 bit register that count the number of clock cycles since the machine was started. This register can be read with a machine instruction called RDTSC. For high-precision performance measurements I have developed a set of C++ functions (`microtime.h`) that uses RDTSC to measure elapsed number of clock cycles. Appendix A.4 lists the source for `microtime.h`.

Measuring

The actual running time of a program may vary from time to time. For instance a quicksort program does not only depend on the size on the input, but also on the distribution of elements in the input. Furthermore the state of the machine

³Quite interestingly Intel acquired Kuck & Associates (KAI) during 2000. KAI is the producer of the KAI C++ compiler, which is known for being very good at optimising template based code. It can only be speculated whether KAI compiler technology is being used in ICL 5.0.

may affect the running time, because of cache and branch predictor buffers. Therefore it is normal practice to report the average of several runs.

I have found that taking the average of several runs is not quite good enough. The operating systems, Windows 2000, on which I am testing on is multi-tasked and has many system processes running simultaneously with the test program. This means that there may be test runs which are interrupted by thread switches to system processes, and consequently the measurement will report both the time used by test program and the interrupting system process. Even with special timers that only measure the CPU time spend in the test program, the problem persist. The reason of this is the fact that the interruptions may cause both program code and data from the test program to be flushed in the cache. Furthermore it may affect the branch prediction buffers. If the test runs are short, and thread switches scarce, only a few test runs will be affected.

My own experiments have shown that when taking the average, we need a very high number of runs in order to get stable results. However if we use the median, far fewer runs are needed to get the same result. I assume this is because when we take average, “tainted” runs may affect the average, whereas when taking median those runs have less effect. Although the median is a stable measure, it will not always give the same result as taking the average. If the distribution of execution times is skewed, the median may vary considerably from the average.

4.3 Gen<X> and X-Code

Gen<X>⁴ is a code generation tool from DevelopMentor designed to help reduce the coding overhead of repetitive task. The way it works it to generate *parameterised code templates* for standard functions, classes, files, etc. The templates can then be used to generate actual code when the parameters have been specified. A typical example could be a template for setting up a standard development environment, or creating C++ files with a standard header stating author, date, purpose, etc.

The heart of Gen<X> is the X-Code engine. X-Code is the language used for writing code templates in. It has the syntax as ASP⁵, and the X-Code engine itself is command-line tool for processing X-Code translating code templates into code⁶.

X-Code templates for C++ code looks very much like C++ code. Consider the following example:

```
1 // First program by <%= author %>
2
3 #include <iostream>
4
```

⁴<http://www.develop.com/genx>

⁵Active Server Pages - a template language used for web site HTML code generation generation in Microsoft web servers..

⁶Technically speaking the X-Code engine is not the command-line tool, but a dynamic library that the command-line tool calls into.

```
5 void main()
6 {
7   <% for (var i = 1; i <= 5; ++i) } %>
8     std::cout << "<%= i %>\n";
9   <% } %>
10 }
11
```

Which produces the following C++ code, when author is set to “Sofus Mortensen”:

```
1 // First program by Sofus Mortensen
2
3 #include <iostream>
4
5 void main()
6 {
7   std::cout << "1";
8   std::cout << "2";
9   std::cout << "3";
10  std::cout << "4";
11  std::cout << "5";
12 }
```

A block marked by `<%= expr %>` is substituted by the value of what `expr` evaluates to. A block marked by `<%` and `%>` encapsulates a scripting block containing JavaScript code that can be used to control the logic of how the X-Code engine processes the script.

Chapter 5

Binary search and range search

5.1 Introduction

This chapter is inspired by an article on binary search by Jon Bentley [Ben00a] and the code tuning column from his book *Programming Pearls* [Ben00b, Ben86]

5.2 Binary search

The *searching problem* consists of finding a specific element in a sequence. Formally given a sequence of n elements $A = \langle a_1, a_2, \dots, a_n \rangle$ taken from a set with total ordering, and an element v , the problem is to find the index i such that $a_i = v$ or special value \top if v is not in A .

Assuming that the sequence A is sorted or that we are allowed to sort it beforehand, the searching problem can be solved efficiently by *binary search*. Binary search is a recursive algorithm whereby in each step the midpoint of the remaining elements are compared with v . If the midpoint is equal to v the algorithm ends, otherwise we recurse in the half in which v possibly lies. It is easy to prove that the asymptotical running time of binary search is $O(\lg n)$.

The searching problem with preprocessing can be solved efficiently using hashing in expected constant time per query. See [MP95, Section 8.4 and 8.5] and [CLR94, Chapter 12]. Binary search can also be used to solve the *lower-bound searching problem* which consists of finding the index of the first element greater than or equal to the sought element in a sorted sequence.

Definition 5 (The lower-bound searching problem). *Given a sequence of n non-decreasing elements $A = (a_1, a_2, \dots, a_n)$ and an element v , the problem is to find the index i such that a_i is the first element in the sequence A for which $a_i \geq v$, or $n + 1$ if no such element exists.*

The Standard C++ Library comes with several variants of binary search. One of them `lower_bound`, see [Jos99, page 413], solves the lower-bound searching problem. `lower_bound` has the following interface:

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator begin,
                           ForwardIterator end,
                           const T& val);
```

The arguments `begin` and `end`, denotes a sequence of elements `[begin, end)`, where `begin` is the position of the first element, and `end` the position of element following the last element in the sequence. The argument `val` is the element sought. `lower_bound` returns an the position to the first element that has a value greater than or equal to the sought element, or the `end` position if no such element exists.

We will be construction pure-C implementations of `lower_bound` with the same interface as STL `lower_bound`. For the analysis, it shall be assumed that:

- The element type is a word.
- A `ForwardIterator` is pointer to a word.
- Elements are stored sequentially in memory.
- That type `T` is identical to the element type.

5.2.1 The STL implementation

We will start by considering stripped version of the `lower_bound` found implementation of STLport. The `lower_bound` from STLport differs from the version presented here (Program 3) because it is designed to work with compilers that does not have *partial specialisation*¹, which is required in order for `iterator_traits`² to work with pointers.

Program 3: STL `lower_bound`

```

1  template<class ForwardIterator, class T>
2  ForwardIterator lower_bound(ForwardIterator begin,
3                             ForwardIterator end,
4                             const T& val)
5  {
6      std::iterator_traits<ForwardIterator>::difference_type half;
7      ForwardIterator mid;
8
9      size_t len = std::distance(begin, end);
10
11     while (len > 0)
12     {
13         half = len >> 1;
14         mid = begin;
15         std::advance( mid, half );
16         if (*mid < val)
17         {
18             begin = mid;
19             ++begin;
20             len = len - half - 1;
21         }
22         else
23             len = half;
24     }
25     return begin;
26 }
```

Theorem 3. *Program 3 solves the lower-bound searching problem.*

Proof. The proof is by induction on the number of elements, `len`. The body of the `while`-loop is repeated until `len` is zero. At the first iteration, the range of valid return values is `[begin,end]`, where `end` marks that `val` is greater than all elements in the range.

`len = 0`: If `len` is zero, the sequence is empty and there is only one valid return value, namely the “end” element. Since `len` is zero, the returned value in variable `begin` is the “end” element.

`len > 0`: Assume that the range of valid output values is the pointers from `begin` and up to, including, `begin + len`, where `begin + len` is the “end” element. The range of output values gets split up into two subranges; `[begin, begin`

¹See [Str95, pp. 342]

²See [Jos99, pp. 285]

+ half] and [begin + half + 1, begin + len], where half = $\lfloor \text{len}/2 \rfloor$. The sought element (val) is compared against the element at position begin + half. If val is greater than begin[half], then val is greater than all elements in the left subrange, and thus the solution must be found in the right subrange. If val is less than or equal to begin[half], then the solution is either begin[half] or another element in the left subrange.

□

In order to employ the pure-C cost model in analysing the running time of Program 3 we need to consider the pure-C translation of the program.

Program 3 is carefully programmed to work with *forward iterators*, as well as *random iterators*³. The pure-C implementation however will assume random iterators. The pure-C implementation is listed in Program 4.

Program 4: pure-C implementation of Program 3.

```

1  template<class RandomIterator, class T>
2  RandomIterator lower_bound(RandomIterator begin,
3                           RandomIterator end,
4                           const T& val)
5  {
6      RandomIterator mid;
7      std::iterator_traits<RandomIterator>
8          ::value_type mid_value;
9      ptrdiff_t len = end - begin;
10     ptrdiff_t half;
11     goto loop_start;
12     recurse_right_half:
13         begin = mid;
14         begin = begin + 1;
15         len = len - half;
16         len = len - 1;
17     loop_start:
18         if (len == 0) goto exit_loop; /* hint: not taken */
19         half = len >> 1;
20         mid = begin + half;
21         mid_value = *mid;
22         if (mid_value < val) goto recurse_right_half;
23         len = half;
24         goto loop_start;
25     exit_loop:
26         return begin;
27     }
28

```

Before analysing the running time, we need a lemma:

Lemma 1. *The body of the while loop of Program 4 is executed no more than $1 + \lceil \lg n \rceil$ times for $n > 0$, where n is the number of elements in the range.*

³See [Jos99, pp. 254-255]

Proof. Let $T(n)$ denote the number of times the body of the `while` loop is executed. If $n = 0$ the body is not executed, and thus $T(0) = 0$. If $n > 0$, $T(n)$ is bounded according to the solution of the following recurrence.

$$\begin{aligned} T(n) &\leq 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\leq 2 + \lfloor \lg \left\lfloor \frac{n}{2} \right\rfloor \rfloor \\ &= 2 + \lfloor \lg \frac{n}{2} \rfloor \\ &= 2 + \lfloor \lg n - 1 \rfloor \\ &= 1 + \lfloor \lg n \rfloor. \end{aligned}$$

The third step uses the formula $\lfloor \lg \lfloor x \rfloor \rfloor = \lfloor \lg x \rfloor$ which follows from the fact that:

$$\lg x = \text{integer} \implies x = \text{integer}$$

and an observation by Graham *et al.* [GKP94, equation 3.10] stating that when $f(x)$ is integer $\implies x$ is integer, then:

$$\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor \text{ and } \lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$$

□

Property 1. *The pure-C operations count of Program 4 is bounded above by $9\lfloor \lg n \rfloor + O(1)$.*

Proof. According to Lemma 1 the loop is repeated $1 + \lfloor \lg n \rfloor$ times. Each iteration costs no more than 9, which is what an iteration costs when the algorithm recurses in the right half. The prologue and epilogue is no more than a constant number of operations. Hence the upper bound of $9\lfloor \lg n \rfloor + O(1)$. □

Property 2. *The average number of branch mispredictions for Program 4 is bounded above by $(2 + \lfloor \lg n \rfloor)/2$.*

Proof. The branch of line 18 will be correctly predicted for all iterations but the last. The branch of line 22 is according to Lemma 1 executed $1 + \lfloor \lg n \rfloor$ times. The branch has no hint and is hence resulting in $(1 + \lfloor \lg n \rfloor)/2$ mispredictions on an average according to Theorem 2. □

Property 3. *The number of cache misses incurred by execution of Program 4 is bounded above by $\lfloor \lg n \rfloor - \lfloor \lg B \rfloor + O(1)$.*

Proof. For data sets larger than the cache size, every memory access can be assumed to be a cache miss. However, when the remaining range has been reduced to no greater than B words (assuming $n > B$), no more than a constant number of cache misses can occur. According to Lemma 1, there are $1 + \lfloor \lg n \rfloor$ memory accesses, and hence the number of cache misses is upper bounded by $\lfloor \lg n \rfloor - \lfloor \lg B \rfloor + O(1)$. □

5.2.2 Binary search with conditional assignments

The implementation presented here is based on the binary search implementation found in [Ben00b, pp. 94]. I have modified it to have same interface and functionality as `lower_bound`, although this implementation only supports random iterators. Furthermore I have modified the algorithm to use conditional assignment, instead of conditional branching.

This implementation improves on 3 by simplifying the work done in the inner loop. The trick to take the first iteration of the loop out into the prologue code. The range of elements is then divided in two possibly overlapping sequences with $2^{\lfloor \lg n \rfloor}$ elements in each. The middle element will be compared with `val`, and the sequence where `val` possibly lies in is selected. The situation now is that the number of remaining candidate positions will always be a power of two. This means in each iteration the number of remaining elements will halved exactly, independent of the comparison of the middle value with `val`. Furthermore we can avoid updating the `end` variable. The `end` variable is not needed because, it equals the number of remaining elements plus `begin`. Hence in each iteration, we will half the number of remaining elements, and if the middle element is smaller than `val`, `begin` will be assigned the position after the middle element.

Program 5: Optimised `lower_bound` with conditional assignments.

```

1  template<typename RandomIterator, typename T>
2  RandomIterator lower_bound(RandomIterator begin,
3                             RandomIterator end,
4                             const T& val)
5  {
6      std::iterator_traits<RandomIterator>
7          ::difference_type n = end - begin;
8
9      if (n == 0) return end;
10
11     ptrdiff_t i = (1 << log2(n)) - 1;
12     begin = begin[i] < val ? begin + (n - i) : begin;
13
14     while (i > 0) {
15         i = i >> 1;
16         begin = begin[i] < val ? begin + i + 1 : begin;
17     }
18     return begin;
19 }
20
```

For the analysis of Program 5, we shall consider the following pure-C translation:

Program 6: pure-C version of Program 5.

```

1  template<class RandomIterator, class T>
2  RandomIterator lower_bound(RandomIterator begin,
3                             RandomIterator end,
4                             const T& val)
5  {
6      ptrdiff_t n = end - begin;
7
8      if (n == 0) goto end; /* hint: not taken */
9
10     ptrdiff_t i = (1 << log2(n)) - 1;
11                     // O(log sizeof(ptrdiff_t))
12
13     RandomIterator br, mid;
14     std::iterator_traits<RandomIterator>
15         ::value_type mid_e;
16
17     mid = begin + i;
18     mid_e = *mid;
19     br = begin + n;
20     br = br - i;
21     begin = mid_e < val ? br : begin;
22
23     loop:
24         i = i >> 1;
25         mid = begin + i;
26         mid_e = *mid;
27         br = begin + i;
28         br = br + 1;
29         begin = mid_e < val ? br : begin;
30         if (i > 0) goto loop; /* hint: branch taken */
31     ennd:
32         return begin;
33 }
34

```

Property 4. *The pure-C operations count of Program 6 is bounded above by $7\lfloor \lg n \rfloor + O(1) + O(\lg W)$, W is the word size in bits.*

Proof. The loop is repeated $\lfloor \lg n \rfloor$ times. Each iteration executes 7, and the prologue and epilogue code is $O(1) + O(\lg W)$ operations. \square

Property 5. *The number of branch mispredictions for Program 6 is 1*

Proof. There is only one conditional branch (line 30), and that branch will only fail for the last iteration. \square

Property 6. *The number of cache misses incurred by the execution of Program 6 is bounded above by $\lfloor \lg n \rfloor - \lfloor \lg B \rfloor + O(1)$.*

Proof. Same as for Property 3. □

5.2.3 Straight-line binary search

The loop of Program 5 is repeated no more than $\lfloor \lg n_{\max} \rfloor = W$ times, where n_{\max} is the highest possible value for n . It is assumed that pointers are stored in machine words, and thus no more than 2^W data elements can be indexed with a pointer. This implies that the loop is repeated no more than W times, if all elements in range are stored in memory. Note that this might not always be the case since it is conceivable to have iterators for elements stored in external memory, element stored in a database, etc. We will ignore such iterators by assuming that $1 + \lfloor \lg n \rfloor \leq W$.

The idea of *straight-line binary search* is to fully unroll the loop of Program 5. This is feasible because $1 + \lfloor \lg n \rfloor \leq W$, and thus the number of code lines required for this is quite limited. The implementation is called straight-line binary search because the loop has been transformed to code with no branching at all, where the execute can be said to run straight through.

The X-Code⁴ template for the implementation can be found in Program 7.

⁴See Section 4.3

Program 7: Straight-line binary search

```

1  <% var wordsize = 32 %>
2
3  template<class RandomIterator, class T>
4  RandomIterator lower_bound(RandomIterator begin,
5                             RandomIterator end,
6                             const T& val)
7  {
8      std::iterator_traits<RandomIterator>::
9          difference_type n, logn, i;
10     n = end - begin;
11     if (n == 0) return end;
12     logn = log2(n);
13     i = (1 << logn) - 1;
14     begin = begin[i] < val ? begin + (n - i) : begin;
15
16     switch (logn) {
17     <%
18     for (var x = wordsize - 1; x > 0; --x) {
19     %>
20     case <%= x %>:
21         begin = begin[<%= (1 << x-1) - 1 %>] < val
22             ? begin + <%= 1 << x-1 %> : begin;
23     <%
24     }
25     %>
26     default:
27         break;
28     }
29     return begin;
30 }
31

```

According to Jon Bentley's *Programming Pearl* [Ben00b, pp. 95] straight-line binary search has been a known trick among programmers since the early 1960's. The version listed in [Ben00b] works only a fixed n of 1000, it does not use conditional assignments (the compiler might though), and it is not compatible with STL `lower_bound`. Jyrki Katajainen presented a version compatible with STL during his performance engineering course. The version presented here is constructed from scratch based on the binary search algorithms in [Ben00b] and the STL implementation.

For the analysis we shall consider the pure-C translation of Program 7 which can be found as Program 8.

Program 8: pure-C implementation of straight-line binary search

```

1  <% var wordsize = 32 %>
2
3  template<class RandomIterator, class T>
4  RandomIterator lower_bound(RandomIterator begin,
5                             RandomIterator end,
6                             const T& val)
7  {
8      std::iterator_traits<RandomIterator>::
9          difference_type n, logn, i;
10     n = end - begin;
11     if (n == 0) goto exit;
12     logn = log2(n);
13     i = 1 << logn;
14     i = i - 1;
15
16     RandomIterator br, mid;
17     std::iterator_traits<RandomIterator>
18         ::value_type mid_e;
19
20     mid = begin + i;
21     mid_e = *mid;
22     br = begin + n;
23     br = br - i;
24     begin = mid_e < val ? br : begin;
25
26     switch (logn) {
27     <%
28         for (var x = wordsize - 1; x > 0; --x) {
29         %>
30     case <%= x %>:
31         mid = begin + <%= (1 << x-1) - 1 %>;
32         mid_e = *mid;
33         br = begin + <%= (1 << x-1) %>;
34         begin = mid_e < val ? br : begin;
35     <%
36     }
37     %>
38     default:
39         break;
40     }
41     return begin;
42 }
```

Property 7. *The pure-C implementation of Program 7 has a pure-C operation count of $4\lceil \lg n \rceil + O(\lg W)$, where W is the word size in bits.*

Proof. In the prologue code `log2` is being called resulting in a contribution of $O(\lg W)$ pure-C operations (see Section 3.9). Besides `log2`, only a constant number of pure-C statements are executed in the prologue code. In the body of the

switch statement, `begin` is being updated $\lfloor \lg n \rfloor$ times requiring in total $4\lfloor \lg n \rfloor$ pure-C operations. Hence the total operations count of $4\lfloor \lg n \rfloor + O(\lg W)$. \square

Property 8. *The number of branch mispredictions for Program 7 is 1.*

Proof. Only one branch is required, the indirect branch of line 26. \square

Property 9. *The number of cache misses incurred by the execution of Program 7 is bounded above by $\lfloor \lg n \rfloor - \lfloor \lg B \rfloor + O(1)$.*

Proof. Same as for Property 3. \square

The straight-line binary search program appears to be very efficient with only 4 pure-C operations per $\lfloor \lg n \rfloor$. However, the ILP pure-C cost model suggested in Section 3.6 reveals that the program might not be using the capability of a modern processor to the fullest. Consider this fragment from the fully unrolled loop:

```

1     ...
2     begin = mid_e < val ? br : begin;
3
4     mid = begin + 511;
5     mid_e = *mid;
6     br = begin + 512;
7     begin = mid_e < val ? br : begin;
8
9     mid = begin + 255;
10    ...

```

The pure-C operation at line 4 is data dependent on the previous operation at line 2. The operations at line 5 and 6 are dependent on the operation at line 4, but has no mutual dependencies. The operation at line 7 is dependent on both 5 and 6. Thus according to the ILP pure-C model, only the instructions at line 5 and 6 can be executed in parallel.

Property 10. *Program 8 has an ILP pure-C cost of $3\tau\lfloor \lg n \rfloor + \lfloor \lg n/B \rfloor\tau_* + \tau_b + O(W)$, assuming that the level of parallelism p is at least 2.*

Proof. The contribution from cache misses and branch mispredictions is the same under the ILP pure-C cost model, as under the pure-C cost model. From the loop fragment it can be seen that in each iteration only 2 operations out of 4 can be executed in parallel, hence if $p \geq 2$ the basic ILP pure-C cost is $3\tau\lfloor \lg n \rfloor$. \square

If $p = 1$ the ILP pure-C cost of Program 8 is exactly the same as under the regular pure-C cost model. When $p \geq 2$ the basic cost per iteration is reduced from 4 to 3, a reduction of 25%, which is not impressive considering that the machine is capable of executing p times as many instructions in the same time.

5.2.4 Summary

Table 5.1 summarises the 6 merge algorithms presented. Comparing the STL version with the straight-line version, we see that the number of pure-C operations has been halved, and that the number of branch mispredictions have been reduced from $1 + \lfloor \lg n \rfloor / 2$ to 1.

Name	Basic / τ	Branch / τ_b	Cache / τ_*
STL	$9 \lfloor \lg n \rfloor + O(1)$	$1 + \lfloor \lg n \rfloor / 2$	$3 + \lfloor \lg n \rfloor - \lfloor \lg B \rfloor$
Cond. assignments	$7 \lfloor \lg n \rfloor + O(\lg W)$	1	$3 + \lfloor \lg n \rfloor - \lfloor \lg B \rfloor$
Straight-line	$4 \lfloor \lg n \rfloor + O(\lg W)$	1	$3 + \lfloor \lg n \rfloor - \lfloor \lg B \rfloor$

Table 5.1: Summary of merge algorithms

5.3 Range search

Definition 6 (The range-searching problem). *Given a sequence of non-decreasing elements $A = (a_1, \dots, a_n)$, and two elements u, v , $u < v$ to find indices i, j that solves the searching problem for respectively u and v on the sequence A .*

The interpretation of the range-searching problem is to find the subrange of elements that lies in the set $[u, v)$.

There is no function in STL for solving the range-search problem, although it is easy to do solve it using two calls to `lower_bound`. However, if an algorithm for solving the range-search problem were to be added, I would expect it to have the following interface:

```
template<class ForwardIterator, class T>
std::pair<ForwardIterator, ForwardIterator>
    range_search(ForwardIterator begin,
                ForwardIterator end,
                const T& lower,
                const T& upper);
```

`range_search` takes a sequence $[begin, end)$ and two values, `lower` and `upper` as arguments. It returns a subrange specified by two iterators containing exactly the elements of $[begin, end)$ that lies in $[lower, upper)$.

5.3.1 Range search using STL `lower_bound`

It is obvious from Definition 6 that the problem can be solved using two `lower_bound` invocations. In this section I will present two ways of doing so. In [Ben00a] Bentley discusses the cache performance of both methods. Bentley refers to the two variants as *paired binary searches* with “warm start” and with “cold start”.

“Cold start”

The “cold start” version (see Program 9) is nothing but two binary searches. First `lower_bound` is used to first find the position i corresponding to u , and then j corresponding to v .

Program 9: Range search with “cold start”

```

1  template<class ForwardIterator, class T>
2  std::pair<ForwardIterator, ForwardIterator>
3      range_search(ForwardIterator begin,
4                  ForwardIterator end,
5                  const T& lower,
6                  const T& upper)
7  {
8      return std::make_pair(
9          std::lower_bound(begin, end, lower),
10         std::lower_bound(begin, end, upper)
11     );
12 }
13

```

Since Program 9 is nothing but two executions of `lower_bound` the pure-C cost of its pure-C translation is upper bounded by the twice the cost of Program 4. Hence the following properties (assuming that Program 3 is used for `lower_bound`):

Property 11. *The pure-C operations count of Program 9 is bounded above by $18\lfloor \lg n \rfloor + O(1)$.*

Property 12. *The average number of branch mispredictions for Program 9 is bounded above by $\lfloor \lg n \rfloor + O(1)$.*

Property 13. *The number of cache misses incurred by the execution of Program 9 is bounded above by $\lfloor \lg n \rfloor - \lfloor \lg B \rfloor + O(1)$.*

“Warm start”

In the range search with “warm start” the fact that $u < v$ is exploited to reduce the required number of comparisons. The implementation (see Program 10) works by first finding the position i corresponding to u using `lower_bound`, and then search for the index corresponding to v among the elements (a_i, \dots, a_n) instead of (a_1, \dots, a_n) .

Program 10: Range search with “warm start”

```

1  template<class ForwardIterator, class T>
2  std::pair<ForwardIterator, ForwardIterator>
3      range_search(ForwardIterator begin,
4                  ForwardIterator end,
5                  const T& lower,
6                  const T& upper)
7  {
8      ForwardIterator it = std::lower_bound(begin, end, lower);
9      return std::make_pair(
10         it,
11         std::lower_bound(it, end, upper)
12     );
13 }
14

```

The upper bound for pure-C operations, branch mispredictions, and cache are obviously also upper bounds for Program 10. Furthermore in the case where i is 1, is obvious that second call to `lower_bound` takes exactly as long time as for Program 9, and consequently Program 10 can be at least as slow as Program 9. However often the pure-C operations count of Program 10 will be lower. If for instance i is approximately $n/2$, one comparison will be saved since the second binary search will only need to search half as many elements.

Bentley describes in [Ben00a] the fact that the cache behaviour of the “cold start” version is often better than of the “warm start” version. The reason for this is the fact that in the “cold start” version (Program 9) the two calls to `lower_bound` have higher probability of accessing the same elements of the array than Program 10. The first comparison, for instance is always the same, whereas for 10 the first comparison will only be the same for the extreme case where $i = 1$.

I find Bentleys use of “warm start” and “cold start” very confusing, since warm and cold start is often used in connection with cache, where a cold start is a start where no relevant data is cached, and a warm start is a start with relevant data preloaded in the cache. It is particularly confusing, since Bentleys “cold start” has better cache performance than the “warm start”.

5.3.2 Range search as paired straight-line binary search

Program 9 was preferred over 10 because of the better cache performance. Program 9, however, has another advantage can be made use of. The two calls to `lower_bound` has no mutual data dependencies, and consequently there might be an opportunity for parallel execution on a modern superscalar processor.

In order for the two `lower_bounds` to be executed in parallel, we need to *interleave* two binary searches together, so that every two following instructions in the loop are mutually independent. The STL implementation uses branching which causes control dependencies (see 2.3), and therefore infeasible. Instead we will use the straight-line binary search of Program 7, since it has no branching at all.

Performing both `lower_bound` calculations at once, furthermore has the advantage that only one call to `log2` is required. Likewise the indirect branch implementing the switch statement is only executed once.

The *paired straight-line binary search* is listed in Program 11

Program 11: Paired straight-line binary search

```

1  template<class RandomIterator, class T>
2  std::pair<RandomIterator, RandomIterator>
3      range_search(RandomIterator begin1,
4                  RandomIterator end,
5                  const T& v1,
6                  const T& v2)
7  {
8      RandomIterator begin2 = begin1;
9
10     std::iterator_traits<RandomIterator>::
11         difference_type n, logn, i;
12     n = end - begin1;
13     logn = log2(n);
14     i = (1 << logn) - 1;
15
16     begin1 = begin1[i] < v1 ? begin1 + (n - i) : begin1;
17     begin2 = begin2[i] < v2 ? begin2 + (n - i) : begin2;
18
19     switch (logn) {
20     <%
21     for (var x = wordsize; x > 0; --x) {
22     %>
23     case <%= x %>:
24         begin1 = begin1[<%= 1 << x-1 %> - 1] < v1
25             ? begin1 + <%= 1 << x-1 %> : begin1;
26         begin2 = begin2[<%= 1 << x-1 %> - 1] < v2
27             ? begin2 + <%= 1 << x-1 %> : begin2;
28     <%
29     }
30     %>
31     default:
32         break;
33     }
34     return std::make_pair(begin1, begin2);
35 }
```

For the analysis we shall consider a pure-C translation of Program 11:

Program 12: pure-C implementation of paired straight-line binary search

```

1  <% var wordsize = 32 %>
2
3  template<class RandomIterator, class T>
4  RandomIterator lower_bound(RandomIterator begin,
5                             RandomIterator end,
6                             const T& v1,
7                             const T& v2)
8  {
9      RandomIterator begin2 = begin1;
10
11     std::iterator_traits<RandomIterator>::
12         difference_type n, logn, i;
13     n = end - begin;
14     if (n == 0) goto exit;
15     logn = log2(n);
16     i = 1 << logn;
17     i = i - 1;
18
19     RandomIterator br1, mid1, br2, mid2;
20     std::iterator_traits<RandomIterator>
21         ::value_type mid_e1, mid_e2;
22
23     mid1 = begin1 + i;           mid2 = begin2 + i;
24     mid_e1 = *mid1;             mid_e2 = *mid2;
25     br1 = br1 + n;             br2 = begin2 + n;
26     br1 = br1 - i;             br2 = br2 - i;
27
28     begin1 = mid_e1<v1 ? br1 : begin1; begin2 = mid_e2<v2 ? br2 : begin2;
29
30     goto jump_table[logn];
31 <%
32 for (var x = wordsize - 1; x > 0; --x) {
33 %>
34 label_<%= x %>:
35     mid1 = begin1 + <%= (1<<x-1)-1 %>; mid2 = begin2 + <%= (1<<x-1)-1 %>;
36     mid_e1 = *mid1;           mid_e2 = *mid2;
37     br1 = begin1 + <%= (1<<x-1) %>;   br2 = begin2 + <%= (1<<x-1) %>;
38     begin1 = mid_e1<v1 ? br1 : begin1; begin2 = mid_e2<v2 ? br2 : begin1;
39 <%
40 }
41 %>
42 exit:
43     return begin;
44 }
```

Property 14. *The pure-C operations count of Program 12 is bounded above by $8\lceil \lg n \rceil + O(\lg W)$.*

Proof. The prologue code executed $O(\lg W)$ pure-C statements. In the body of

the switch, `begin1` and `begin2` is being updated $\lfloor \lg n \rfloor$ times requiring in total $8\lfloor \lg n \rfloor$ pure-C operations. \square

Property 15. *The average number of branch mispredictions for Program 12 is bounded above by 1.*

Proof. Only one branch is required, the indirect branch of line 30. \square

Property 16. *The number of cache misses incurred by the execution of Program 12 is bounded above by $2(\lfloor \lg n \rfloor - \lfloor \lg B \rfloor) + O(1)$.*

Proof. Same as argument as for Property 3, except every iteration two memory accesses are performed. \square

Property 17. *The ILP pure-C cost of Program 12 is bounded above by $4\tau\lfloor \lg n \rfloor + 2\lfloor \lg n/B \rfloor\tau_* + \tau_b + O(W)$, assuming that the level of parallelism p is at least 2.*

Proof. The contribution from branch mispredictions and cache misses follows from Property 15 and 16. Property 12 gives an upper bound for the basic contributions of $8\lfloor \lg n \rfloor + O(\lg W)$. However since the 8 pure-C operations of each iteration of the loop, are really 4 pairs of two mutually independent pure-C operations, they can be executed simultaneously if $p \geq 2$. Hence the total cost of $4\tau\lfloor \lg n \rfloor + 2\lfloor \lg n/B \rfloor\tau_* + \tau_b + O(W)$. \square

5.4 Experiments

I have benchmarked binary searching and range searching for the following programs:

- Binary search (`lower_bound`):
 - The STL version (Program 3).
 - The optimised binary search with conditional assignments (Program 5).
 - The straight-line binary search (Program 7).
- Range search (`range_search`):
 - Range search using STL `lower_bound` with “cold start” (Program 9) and “warm start” (Program 10).
 - Range search using two straight-line binary searches with “cold start” (See `range_search_not_paired` in namespace `straight_line` in Appendix A.1).
 - Range search using paired straight-line binary search (Program 11).

When benchmarking for a specific size of n several runs are being made. For each run a sequence of n random elements (using `rand` from `<cstdlib>`) is generated and sorted. Then for each program several searches for random elements is being performed. The median of these searches is found is for each run. Finally the average of medians is calculated for each program and used as measure for the performance of the program. Before each test run, care is taken to touch all elements in sequence, so all programs start with the same cache state.

Data sizes from $2^4 - 1$ and up to $2^{19} - 1$ are being benchmarked. The complete program for testing the binary search programs described herein can be found in Appendix A.1.

The Intel Compiler was clever enough to use conditional assignments even for the STL version. As described in Section 4.2, a special build suppressing the generation of conditional moves had to be made to get the intended program. Since the special build is generating code for the older Pentium processor instead of Athlon and Pentium 2, it is conceivable that the instruction scheduling is not ideal for the Athlon and Pentium 2. For this reason I am presented result both for the special Pentium build, as well as the normal build, even though the latter uses conditional assignments.

5.4.1 Expectations

It was demonstrated with the `limit_data` on page 22 that the cost of branch mispredictions can be severe. It was estimated that for the Pentium 2 and AMD Athlon τ_b was in the vicinity of 15τ . Table 5.2 summarises the estimated running time for `lower_bound` ignoring cache misses (that is assumes that n is small) and under the assumption that $\tau_b = 15\tau$. Hence we expect for small n where

Name	Cost / τ
STL	$16.5 \lfloor \lg n \rfloor + O(1)$
Optimised binary search with cond. assignments	$7 \lfloor \lg n \rfloor + O(\lg W)$
Straight-line	$4 \lfloor \lg n \rfloor + O(\lg W)$

Table 5.2: Summary of merge algorithms for small n , assuming $\tau_b = 15\tau$

all elements fit in the cache, straight-line binary search to perform better than the optimised binary search (Program 5.2), which in turn should perform better than the STL version. According to the figures in Table 5.2 there should be a speed up from the STL version to the straight-line version of approximately a factor 4. However I do not expect quite to obtain that high a speed-up, since the constant term of the running time is not insignificant, when the highest term is only $\lg n$.

For the `range_search` it is expected that differences between “warm start” and “cold start” will behave as explained by Jon Bentley. I expect to see an advantage for “warm start” with small arrays, and an advantage for “cold start” with long arrays. Furthermore I hope that Paired straight-line binary searches will run

faster than two straight-line binary searches possibly up to a factor two (see Property 10).

5.4.2 Results

The results for `lower_bound` can be seen in Figure 5.1 and 5.2. The graphs display running time in clock cycles per $\lg n$ as function of n . Figure 5.2 shows the results for the 450 MHz Pentium 2, and Figure 5.1 the results for the 1000 MHz Athlon. Note that the running time is displayed on a logarithmic scale.

On both graphs it is very obvious when the cache limits are hit. The cache limits for the two level of cache are marked on the graph with L1 and L2.

For both processors the optimised `lower_bound` with conditional assignments is clearly faster than the STL version, by up to a factor 2 when the array searched in fits in the cache. This follows the expectations. However, the straight-line binary search is unexpectedly running on par with the optimised binary search with no more than 2% mutual deviation. According to Table 5.2 it was expected to be approximately twice as fast. At first I thought that this could be because the compiler was automatically unrolling the loop of Program 5, inspecting the assembler output however revealed that this is not the case.

The graphs also compare the STL version with `std::lower_bound`, where the STL version is compiled to produce Pentium code without the use of conditional assignments and the `std::lower_bound` compiled allowing conditional assignments. It have verified that the code for `std::lower_bound` does indeed use conditional assignments by inspecting the assembler output. As might be expected `std::lower_bound` is considerable faster than the STL version on the Athlon, for arrays that fit in the cache. On the Pentium 2 they are both running approximately at the same rate for array that fit in the cache. This is a surprise, since I would have expected the version using conditional assignments to be faster. For large array the STL version without conditional assignment is faster on both the Athlon and the Pentium 2.

The results for `range_search` can be seen in Figure 5.3 and Figure 5.4. The most significant observation from the graph is the superiority of the Paired straight-line program. When the array fits in cache the paired straight-line program is approximately twice as fast as running two straight-line binary search programs in sequence. Even though roughly the same amount of work is being done, the paired straight-line manage to perform the two binary searches in parallel, effectively achieving a speed-up of a factor 2. This is an astonishing result.

We see that for small array sizes, range search based on `std::lower_bound` compiled for Pentium Pro, is slightly faster for “warm starts” than “cold starts” on AMD Athlon. This confirms the observation by Bentley in [Ben00a]. For the Pentium 2 however the difference is insignificant, and for large arrays for both the Pentium 2 and the Athlon the difference is hardly noticeable.

The cache limits are also very apparent in the results for `range_search`, as was the case for `lower_bound`.

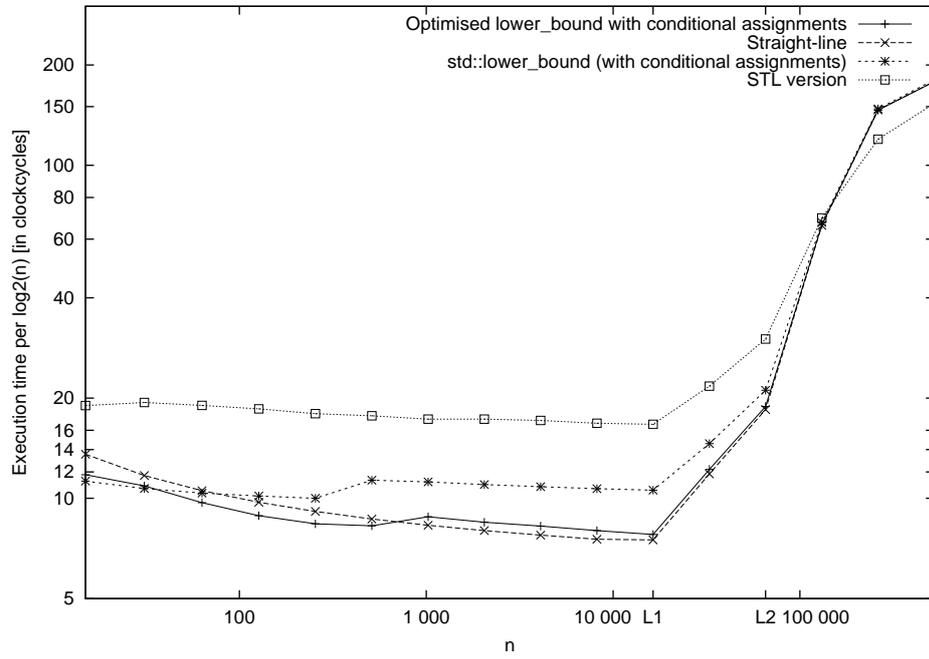


Figure 5.1: lower_bound on a 1000 MHz Athlon

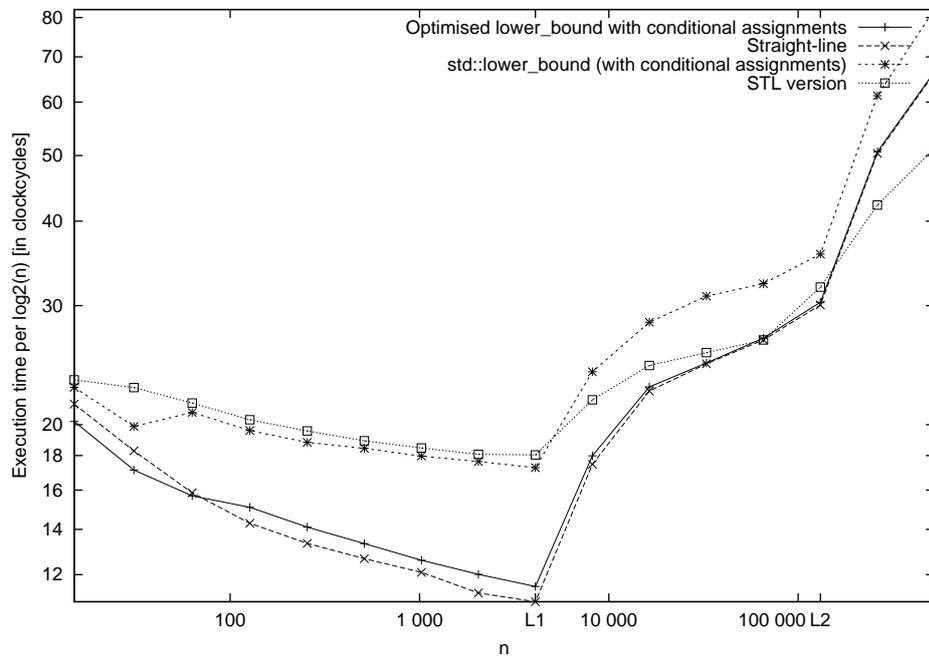


Figure 5.2: lower_bound on a 450 MHz Pentium 2

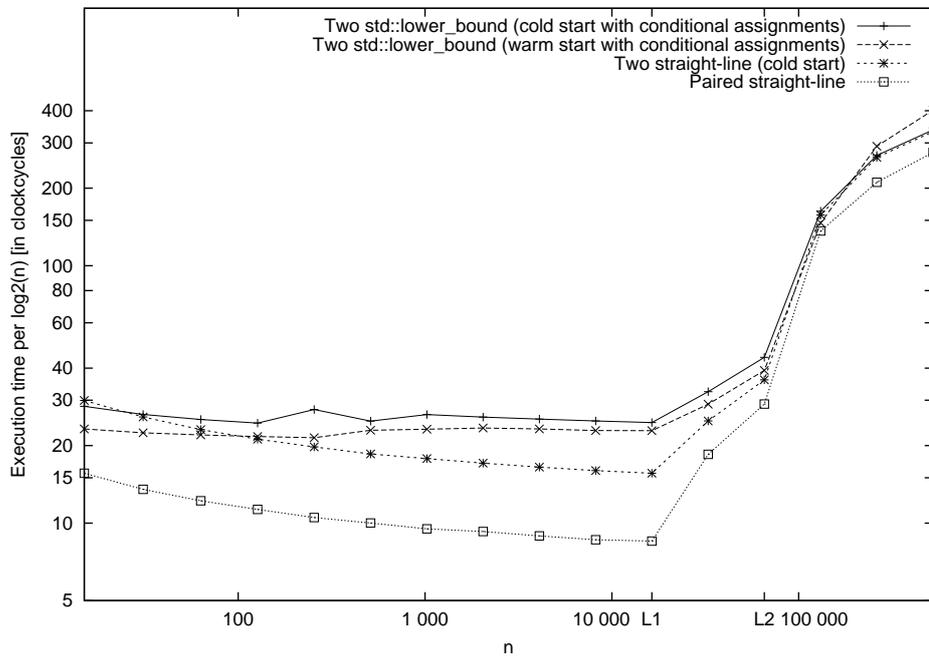


Figure 5.3: range_search on a 1000 MHz Athlon

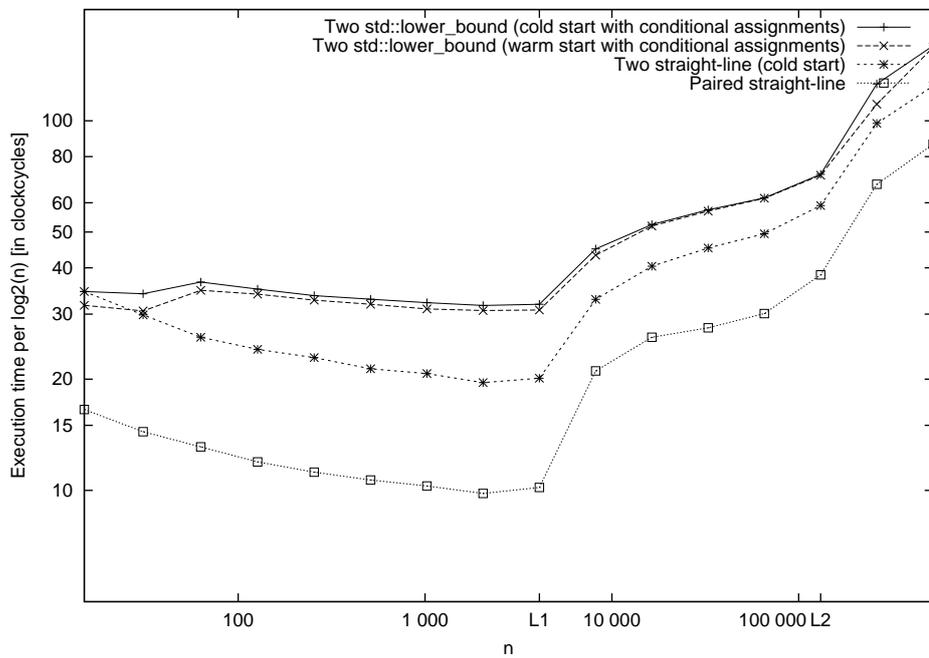


Figure 5.4: range_search on a 450 MHz Pentium 2

5.5 Summary

The performance of binary search and range search has been analysed under the pure-C cost model.

For binary search the refined pure-C cost model predicted a speed-up of two for short arrays, when comparing STL version (without conditional moves) to the optimised version. It has been verified that this speed-up holds in practice. The basic pure-C cost model predicts only a speed-up of 30%, and hence it can be said that the refined model is more accurate.

For range search, the suggested ILP pure-C cost model predicts that the paired straight-line program is twice as fast as two invocations of straight-line binary search. Surprisingly this turned out to be true in practice as well! Experimentation shows that we can perform two binary searches at the cost of one. This extraordinary result demonstrates that instruction level parallelism is an important aspect of the efficiency of programs.

Chapter 6

Mergesort

6.1 Introduction

Given a sequence of n elements, the *sorting problem* consists of permuting the sequence to non-descending order. Sorting is one of the oldest and most well-studied problems in the history of computer science. For an in-depth treatise on sorting, we refer to Volume 3, *Sorting and Searching of The Art of Computer Programming* by Donald Knuth [Knu98].

The topic of this chapter is *mergesort*. Mergesort is the collective term for sorting algorithms that are based on the principle of merging sorted subarrays into one sorted array. The fundamental element of mergesorting is the merge algorithm. A *two-way merge* algorithm takes as input two sorted sequences, and combines the two into a single sorted sequence. *d-way merge* combines d sorted sequences into one.

The idea in mergesort is to consider the initial unsorted array of n elements, as n sorted subarrays of length 1. The algorithm then proceeds by merging subarrays until only one array remains.

Knuth concludes in Volume 3 of *The Art of Computer Programming* [Knu98] that quicksort is the fastest algorithm for internal sorting, superior to, among other algorithms, mergesort. However, he only considered two-way mergesorting and not mergesorting relying on multi-way merging. Furthermore the model used for analysis, the MIX model, does not take in consideration the cache effects.

Mergesort is studied carefully by Katajainen and Träff [KT97] where little-oh analysis under the basic pure-C model is performed. Katajainen and Träff demonstrate that 4-way mergesorting is faster than both 2-way mergesort and surprisingly quicksort as well, both in theory and practice. The mergesort algorithms presented in [KT97] will be re-analysed and benchmarked in this chapter, with the purpose of consolidating their results.

In [Spo99] Maz Spork analyses the cache performance of mergesort algorithms, more specifically the implementations from [KT97]. It is shown that multi-way mergesorting have better cache behaviour 2-way mergesorting.

Mergesort is an $O(n \lg n)$ algorithm, and thus an optimal comparison-based sorting algorithm. There exists however $o(n \lg n)$ sorting algorithms. These are not solely based on comparison of elements, but rely also on *range reduction schemes* and *packed sorting*. Range reduction is a technique where linear time is spent reducing the problem of sorting n words of size b bits, to sorting n words of size $b/2$ bits. Packed sorting is a variant of Batcher's sort¹, where parallelism is obtained by packing multiple elements of a small size, into one machine word.

The popular article of Nilsson [Nil00] is a good introduction to such an algorithm. These modern sorting algorithms are the fastest known today from an analytical viewpoint, and Nilsson furthermore claims in [Nil96] that they perform well in practice. Pedersen [Ped99], however, shows that these algorithms have very poor cache performance, and because of this, little practical significance.

6.1.1 Stable sorting

Definition 7. *A sorting algorithm is said to be stable, if the order of equal elements is preserved.*

When merging r subarrays $\{A_1, A_2, \dots, A_r\}$ into one, we look upon the arrays as being ordered in the sense that element x from subarray A_i is prior to any element from subarray A_j if $j > i$, in addition to any element in A_i that lies after x . With this concept of order, we can define stable merging analogous to stable sorting.

Definition 8. *A merge algorithm is said to be stable if the order of equal elements - in the sense defined above - is preserved.*

As will be shown later, a stable mergesort algorithm is easily attainable if the underlying merging algorithms are stable.

6.2 Merging

In this section, the following merge implementations will be discussed:

- simple merge,
- Katajainen/Träff 2-way merge,
- Katajainen/Träff 3-way merge,
- Katajainen/Träff 4-way merge,
- back-to-back merge with conditional assignments,
- paired back-to-back merge with conditional assignments.

When analysing merge algorithms, n specifies the total number of elements in all subarrays. Furthermore it is assumed for an d -way merge algorithm that the d input arrays are of size $\lfloor n/d \rfloor$ or $\lceil n/d \rceil$.

¹See [Knu98, pp. 230].

6.2.1 Simple merge

The first merge implementation to be considered is a naïve, unoptimised implementation. The implementation can be found in Program 13. The basic idea is to pick the subarray with the smallest element and move that element to the destination array (d). This is repeated until one of the two subarrays has been exhausted for elements. Finally, any remaining elements in either of the subarrays are moved to the destination. Obviously only one of the two subarrays can have remaining elements.

Program 13: Simple merge

```

1  template<class ForwardIterator, class OutputIterator>
2  OutputIterator merge(ForwardIterator b1,
3                      ForwardIterator e1,
4                      ForwardIterator b2,
5                      ForwardIterator e2,
6                      OutputIterator dest)
7  {
8      while (b1 < e1 && b2 < e2)
9      {
10         if (*b1 <= *b2)
11         {
12             *dest = *b1;
13             b1++;
14         }
15         else
16         {
17             *dest = *b2;
18             b2++;
19         }
20         dest++;
21     }
22
23     while (b1 < e1)
24     {
25         *dest = *b1;
26         dest++; b1++;
27     }
28
29     while (b2 < e2)
30     {
31         *dest++ = *b2++;
32         dest++; b1++;
33     }
34
35     return dest;
36 }
37

```

Property 18. *Simple merge is stable.*

The simple merge algorithm as presented is inefficient because the `while` loop

checks at each iteration whether both subarrays are non-empty. But since only one array pointer was updated during the last iteration, only that subarray could possibly have been emptied, and thus it is sufficient only to check the end condition for that subarray. This optimisation has been used in the pure-C implementation of simple merge that can be found as Program 14.

Program 14: Optimised simple merge

```

1  template<class ForwardIterator, class OutputIterator>
2  OutputIterator merge(ForwardIterator b1,
3                      ForwardIterator e1,
4                      ForwardIterator b2,
5                      ForwardIterator e2,
6                      OutputIterator dest)
7  {
8      std::iterator_traits<ForwardIterator>::value_type v1, v2
9      v1 = *b1;
10     v2 = *b2;
11     goto test;
12 out_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
13         if (b1 >= e1) goto exit_1; /* hint: not taken */
14 test:  if (v1 <= v2) goto out_1;
15         b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
16         if (b2 < e2) goto test; /* hint: taken */
17         goto exit_2;
18 loop_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
19 exit_2: if (b2 < e2) goto loop_2; /* hint: taken */
20         goto exit;
21 loop_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
22 exit_1: if (b1 < e1) goto loop_1; /* hint: taken */
23 exit:  ;
24         return dest;
25 }
26

```

Property 19. *The pure-C operation count of Program 14 is bounded above by $6n + O(1)$.*

Proof. The inner loop starts at label test. There are two routes through the loop depending on whether $v1 \leq v2$. On both routes 6 pure-C operations are executed.

Any elements that are not moved in the inner loop will be moved either by the loop starting at label exit_1 or the loop at exit_2. Both loops costs 5 operations per iteration.

Each element will be moved either in the inner loop, or in one of the two final loops, for maximal cost of 6 pure-C operations per element. In addition to this, there are 3 operations for the first four lines, and possibly 3 extra if no elements are moved in the final loop. \square

Property 20. *The average number of branch misses incurred when executing Program 14 is less than or equal to $n/2 + O(1)$.*

Proof. There are three conditional branches in Program 14.

- Line 13 and 16: If one of these branches mispredicts, it is because all elements have been moved from either $[b1, e1)$ or $[b2, e2)$. This will only happen once.
- Line 9 : Prior to moving each element to `dest`, it is decided whether to move the element referred by `b1` or `b2` to the destination. This conditional branch has no hint, and since n decisions are made, the contribution from this branch is $n/2$ on an average (according to Theorem 2).
- Line 14 and 17: Only one of line 14 and 17 will be reached. The branch prediction will only fail after the last element has been moved.

□

Property 21. *The number of cache misses incurred when executing Program 14 is bounded above by $2n/B + O(1)$.*

Proof. Follows from Theorem 1 and the fact that merge reads/writes $2n$ elements sequentially. □

6.2.2 The 2-way merge of Katajainen/Träff

It is hard to imagine that 2-way merging can be done any faster than 6 pure-C operations per element as is the case for Program 14. Nevertheless, the 2-way merge algorithm of Katajainen/Träff improves upon this with an optimisation that brings the amortised operation count down to 5.5 per element. An adaption of their implementation can be found in Program 15.

The trick is to start determining which of the two subarrays has the smallest tail element. The subarray with the smallest tail element is the last array to be emptied, and thus it sufficient to check for end condition only when an element from that subarray has been moved to the destination array.

Program 15: Katajainen/Träff 2-way merge

```

1  template<class RandomIterator, class OutputIterator>
2  OutputIterator merge(RandomIterator b1,
3                      RandomIterator e1,
4                      RandomIterator b2,
5                      RandomIterator e2,
6                      OutputIterator dest)
7  {
8      std::iterator_traits<RandomIterator>::value_type v1, v2, tv1, tv2;
9      RandomIterator t1, t2;
10     v1 = *b1; v2 = *b2;
11     t1 = e1; t1 = t1 - 1; tv1 = *t1;
12     t2 = e2; t2 = t2 - 1; tv2 = *t2;
13     if (tv1 > tv2) goto test_1;
14     goto test_2;
15 out_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
16 test_1: if (v1 <= v2) goto out_1;
17         b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
18         if (b2 < e2) goto test_1; /* hint: taken */
19 copy_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
20         if (b1 < e1) goto copy_1; /* hint: taken */
21         goto end;
22 out_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
23 test_2: if (v1 > v2) goto out_2;
24         b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
25         if (b1 < e1) goto test_2; /* hint: taken */
26 copy_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
27         if (b2 < e2) goto copy_2; /* hint: taken */
28 end:    return dest;
29 }
30

```

Property 22. *The merge, Program 15, is stable.*

Proof. It is obvious that the order is preserved for equal elements, if they come from the same subarray. For equal elements from different subarrays, notice that the selection at label `test_1` and `test_2` chooses elements from the first array over elements from the second array. Thus order is preserved. \square

If both subarrays have tail element of the same value, the algorithm uses the second inner loop (starting at label `test_2`), and this is for a good reason. The first inner loop keeps moving elements from the first subarray without checking for overrun as long as the head element is smaller than *or equal* to the head element of the second array. This would result in elements being moved beyond the last element of the first array, should the two arrays have tail elements of the same value. The second loop on the other hand, will also keep moving elements from first array as long as they are less than or equal to the head element of the second, but in addition it will also check for the end condition.

Property 23. *The pure-C operations count of Program 15 is $5.5n + O(1)$*

Proof. In the proof we will assume the case when the tail of second subarray has smaller value than the tail of the first subarray. The proof for the other case is symmetrical.

Line 10-14 are executed once with a total cost contribution of $O(1)$. The loop entry point is at line 16. For every element from $[b1, e1)$ line 15-16 are executed with a cost for each run of 5. For every element from $[b2, e2)$ line 16-18 are executed with a cost of 6. This sums up to $3 + 5\frac{n}{2} + 6\frac{n}{2} = 5.5n + O(1)$. \square

Property 24. *The average number of branch missing when running Program 15 is bounded above by $n/2 + O(1)$.*

Proof. There are two identical cases depending on which of the two input arrays has the lower bound. We shall only consider the case where $[b1, e1)$ has the lowest tail. There are three conditional branches:

- Line 18 : If this branch mispredicts, it is because all elements have been moved from $b2$. This can only happen once.
- Line 20 : Likewise, this branch mispredicts only when all remaining elements from $[b1, e1)$ have been moved in the final step of the algorithm.
- Line 16 : Prior to moving each element to `dist` it is decided at this line whether to move the element from position $b1$ or $b2$. This conditional branch has no hint, and since n decisions are made, the average number of branch mispredictions is $n/2$ (according to Theorem 2).

\square

Property 25. *The number of cache misses incurred when executing Program 14 or 15 is no more than $2\frac{n}{B} + O(1)$.*

Proof. Follows from Theorem 1 and the fact that `merge` reads/writes $2n$ elements sequentially. \square

6.2.3 The 3-way merge of Katajainen/Träff

The three-way merge program of Katajainen and Träff allows three subarrays to be merged with only a slightly higher number of pure-C operations than Program 15.

The two-way merge algorithm was divided in two cases depending on which of the two subarrays that had the largest tail. Likewise the three-way merge algorithm splits into 3 cases.

The algorithm is listed in Program 16. It is assumed that the tail of the first subarray is the smallest tail. The case where the second or the third subarray has the the smallest tail are similar to Program 16.

Program 16: Katajainen/Träff 3-way merge

```

1  template<class RandomIterator, class OutputIterator>
2  OutputIterator merge(RandomIterator b1,
3                      RandomIterator e1,
4                      RandomIterator b2,
5                      RandomIterator e2,
6                      RandomIterator b3,
7                      RandomIterator e3,
8                      OutputIterator dest)
9  {
10     std::iterator_traits<RandomIterator>
11         ::value_type v1, v2, v3;
12     v1 = *b1;
13     v2 = *b2;
14     v3 = *b3;
15     if (v2 <= v3) goto test_21;
16     goto test_31;
17 out_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
18     if (v2 > v3) goto test_31;
19 test_21: /* invariant: v2 <= v3 */
20     if (v2 < v1) goto out_2;
21     b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
22     if (b1 < e1) goto test_21; /* hint: branch taken */
23     goto exit;
24 out_3: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
25     if (v2 <= v3) goto test_21;
26 test_31: /* invariant: v2 > v3 */
27     if (v3 < v1) goto out_3;
28     b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
29     if (b1 < e1) goto test_31; /* hint: branch taken */
30 exit: /* final 2-way merge step */
31     return two_way::merge(b2, e2, b3, e3, dest);
32 }
33

```

Property 26. *The pure-C operation count of Program 16 is bounded above by $6n + O(1)$*

Proof. Line 12-16 are executed once with a total cost contribution of $O(1)$. As long as subarray three is non-empty, each iteration starts at either label `test_21` or `test_23`. There are four possible routes through each iteration. All routes have an operation count of 6 per element.

When subarray three is empty, the remaining elements in subarray one and two, are merged using the two-way merge algorithm that has an operation count of 5.5 per element. Thus no more than 6 operations are spent per element. \square

Property 27. *The average number of branch misses when running Program 16 is bounded above by $5n/6 + O(1)$.*

Proof. We shall start by considering the branches that contribute no more than a constant number of branch misses:

- The branch at line 15 is executed only once.
- If either the branch at line 22 or 29 fails, it is because all elements have been moved from b1. This happens only once, resulting in exactly one branch miss.

The remaining branches at line 18, 20, 25 and 27 are best analysed by examining the average contribution per element. First observe that any remaining elements after exhausting the first subarray, are merged at line 31, for no more than $1/2$ branch misses per element on an average. Furthermore observe that until the first subarray has been emptied, each iteration starts either at line 19 or 26 depending on which of v_2 and v_3 is larger.

Each of the elements from the first subarray are moved either in an iteration from line 19 to 22, or from line 26 to 29. The branch at line 20 has no hint and contributes on an average no more than $1/2$ branch misses. The number of elements in the first subarray is no greater than $\lceil n/3 \rceil$.

Each of the elements from the second or the third subarray are moved either in an iteration with route covering line 19-20 and 17-18 or route covering line 26-27 and 24-25, or they may be moved in the final 2-way merge step. Elements moved in the final 2-way merge has on an average no more than $1/2$ branch misses per element. The two routes when moving elements in the inner loop, passes two branches with-out branch hints, for an average of no more than 1 branch miss per element

Summing up: $1/2\lceil n/3 \rceil + 2\lceil n/3 \rceil = 5n/6 + O(1)$ □

Property 28. *The number of cache misses incurred when executing Program 16 is bounded above by $2\frac{n}{3} + O(1)$.*

Proof. Follows from Theorem 1 and the fact that merge reads/writes $2n$ elements sequentially. □

We can avoid splitting the implementation up into three different cases, by swapping the first subarray $[b_1, e_1)$ with the subarray that has the smallest tail element. Since we are only exchanging pointers and not actual data elements, this can be done in $O(1)$ pure-C operations, average branch misses and cache misses.

The trick of exchanging the first input array, with the smallest tail array, effectively reduces the code size to almost one third, since we do not need three different implementations for each case. However there is a catch to it. We loose the stability, since we can no longer rely on first input array actually being the first input array.

6.2.4 The 4-way merge of Katajainen/Träff

The 4-way merge of Katajainen and Träff merges four subarrays into one, with an average pure-C operations count only slightly higher than the 3-way merge algorithm.

As is the case with the 3-way merge, the algorithm starts out by determining which of the subarray has the smallest tail, and then splits out in 4 different cases. The algorithm listed in Program 17 is for the case the the first subarray has the smallest tail.

The trick of swapping the subarray with the smallest tail with the first subarray, can also be used here

Program 17: Katajainen/Träff 4-way merge

```

1  template<class RandomIterator, class OutputIterator>
2  OutputIterator merge(RandomIterator b1,
3                      RandomIterator e1,
4                      RandomIterator b2,
5                      RandomIterator e2,
6                      RandomIterator b3,
7                      RandomIterator e3,
8                      RandomIterator b4,
9                      RandomIterator e4,
10                     OutputIterator dest)
11 {
12     std::iterator_traits<RandomIterator>
13         ::value_type v1, v2, v3, v4;
14     v1 = *b1;
15     v2 = *b2;
16     v3 = *b3;
17     v4 = *b4;
18     if (v2 <= v3 && v4 < v1) goto test_24;
19     if (v2 <= v3 && v4 >= v1) goto test_21;
20     if (v2 > v3 && v4 < v1) goto test_34;
21     goto test_31;
22 out_24: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
23     if (v2 > v3) goto test_34;
24 test_24: /* invariant: v2 <= v3 && v4 < v1 */
25     if (v2 <= v4) goto out_24;
26     b4 = b4 + 1; *dest = v4; dest = dest + 1; v4 = *b4;
27     if (v4 < v1) goto test_24;
28     /* invariant: v2 <= v3 && v4 >= v1 */
29     if (v2 >= v1) goto out_12;
30 out_21: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
31     if (v2 > v3) goto test_31;
32 test_21: /* invariant: v2 <= v3 && v4 >= v1 */
33     if (v2 < v1) goto out_21;
34 out_12: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
35     if (b1 >= e1) goto exit; /* hint: not taken */
36     if (v4 >= v1) goto test_21;
37     goto test_24;
38 out_34: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
39     if (v2 <= v3) goto test_24;
40 test_34: /* invariant: v2 > v3 && v4 < v1 */
41     if (v3 <= v4) goto out_34;
42     b4 = b4 + 1; *dest = v4; dest = dest + 1; v4 = *b4;
43     if (v4 < v1) goto test_34;
44     /* invariant: v2 > v3 && v4 >= v1 */
45     if (v3 >= v1) goto out_13;
46 out_31: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
47     if (v2 <= v3) goto test_21;
48 test_31: /* invariant: v2 > v3 && v4 >= v1 */
49     if (v3 < v1) goto out_31;
50 out_13: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
51     if (b1 >= e1) goto exit; /* hint: not taken */
52     if (v4 >= v1) goto test_31;
53     goto test_34;
54 exit: /* merge remaining three subarrays */
55     return three_way::merge(b2, e2, b3, e3, b4, e4, dest);
56
57 }
58

```

Property 29. *The pure-C operation count of Program 17 is bounded above by $6.5n + O(1)$*

Proof. Lines 14-21 are executed once, requiring $O(1)$ pure-C operations. Observe that when all elements from $[b1, e1)$ has been moved, remaining elements from the three other arrays are merged using the 3-way merge algorithm with a cost of 6 pure-C operations per element. (see Property 26).

Each iterations starts at either label `test_24`, `test_21`, `test_34`, `test_31`, line 28, or line 44. In each iteration moving elements from either `b2`, `b3`, or `b4` takes 6 pure-C operations before getting to one of the loop entry points. Moving elements from $[b1, e1)$ requires no more than 8 pure-C operations.

The first subarray has $n/4$ elements requiring in total $2n$ operations. The cost of $3n/4$ elements of array 2, 3 and 4 is $4.5n$. \square

Property 30. *The number of cache misses incurred when executing Program 17 is bounded above by $2\frac{n}{B} + O(1)$.*

Proof. Follows from Theorem 1 and the fact that `merge` reads/writes $2n$ elements sequentially. \square

Property 31. *The average number of branch missing when running Program 17 is bounded above by $n + O(1)$.*

Proof. Lines 14-21, being executed more than once, contributes a constant number of branch misses on an average. The conditional branch of line 35 and 51, are taken only if the last element from $[b1, e1)$ has been moved. This happens only once, and thus these two branches contributes exactly one branch miss.

For each iteration where the first array is non-empty, observe that two conditional branches without hints are executed. According to Theorem 2 these contribute on an average no more than 1 branch miss.

Any remaining elements in array 2, 3 and 4, after $[b1, e1)$ is empty are merged using the 3-way merge sort. We know from Property 28 that the 3-way merge sort only causes no more than $5/6$ branch misses per element.

Thus the average number of branch misses per element is upper bounded by 1. \square

6.2.5 Back-to-back merge with conditional assignments

In this section a 2-way merge algorithm will be presented that is based on conditional assignments, and as a consequence only cause a constant number of branch misses on an average.

The algorithm is based on the merge algorithm presented by Robert Sedgewick in [Sed98, Page 339] and Donald Knuth in [Knu98, Section 162]. The interface of the algorithm varies from the other presented previously in the sense that it:

1. Assumes that the memory placement of the second subarray immediately follows the memory placement of the first.

2. Assumes that the first subarray is sorted to non-decreasing order, whereas the second is sorted to non-increasing order.
3. Takes a pointer to head of the first subarray, and a pointer to element after the tail of the second subarray.

The idea is in each iteration to move the smallest of the head and tail element of the input array (the combination of the two sorted subarrays). When the head pointer passes the tail pointer, all elements have been moved and algorithm terminates.

The algorithm could be referred to as a *bitonic merge* since the two consecutive input arrays, can be conceived of as a single *bitonic sequence*.² However in the literature, bitonic sorting and bitonic merging is Batcher's sorter (see [Knu98, pp. 230]).

Comparing with the simple merge (see Program 13) this approach uses only one conditional branch per iteration to check for end condition, and has no need for a final step to move any remaining elements.

The 2-way merge algorithm of Jyrki and Träff (Program 15) uses program state to reduce the number of pure-C operations. That is, we are only checking for end condition when elements are being moved from the subarray with the smallest tail.

Program 18: Back-to-back merge with conditional assignments

```

1  template<class RandomIterator, class OutputIterator>
2  OutputIterator merge(RandomIterator begin,
3                      RandomIterator end,
4                      OutputIterator dest)
5  {
6      std::iterator_traits<RandomIterator>
7          ::value_type v1, v2, tmp;
8          RandomIterator tmp0;
9
10     end = end - 1;
11     v1 = *begin;
12     v2 = *end;
13 loop:  tmp = v1 <= v2 ? v1 : v2;
14         *dest = tmp;
15         dest = dest + 1;
16         tmp0 = begin + 1;
17         begin = v1 <= v2 ? tmp0 : begin;
18         tmp0 = end - 1;
19         end = v1 <= v2 ? end : tmp0;
20         v1 = *begin;
21         v2 = *end;
22         if (begin <= end) goto loop /* hint: branch taken */;
23         return dest;
24 }
25
```

²A sequence a_1, a_2, \dots, a_n is said to be *bitonic*, if $a_1 \leq a_2 \leq \dots \leq a_k \geq \dots \geq a_{n-1} \geq a_n$ for some k , $1 \leq k \leq n$.

It is convenient to have a variant of `merge`, say `merge_rev` where the output array `dest` is being filled in reverse order. This can be accomplished by adding `end - begin` to `dest`, so that `dest` now points to the last element in the output array. Line 15 is modified so that `dest` is decremented in each iteration instead of being incremented.

We should consider the input for `merge` as being two non-decreasing arrays, where the latter array has been reversed. It is fairly obvious from the program that order is maintain in the sense,

Property 32. *The pure-C operation count of Program 18 is bounded above by $10n + O(1)$*

Proof. Line 10-12 are executed only once resulting in $O(1)$ operations being executed.

Line 13-22 are executed once for each element in the two subarrays, resulting in a contribution of $10n$. \square

Property 33. *The number of cache misses incurred when executing Program 18 is bounded above by $2\frac{n}{B} + O(1)$.*

Proof. Follows from Theorem 1 and the fact that `merge` reads/writes $2n$ elements sequentially. \square

Property 34. *The average number of branch missing when running Program 18 is $O(1)$.*

Proof. The branch at line 22 is the only conditional branch. The branch is correctly predicted for all iterations but the last. \square

6.2.6 Paired back-to-back merge with conditional assignments

The *paired merge algorithm* presented here, is basically just two back-to-back merges (see Program 18 performed at once).

The pure-C implementation is listed in Program 19. The algorithm merges two bitonic sequences `[begin1, end1)` and `[begin2, end2)` into two sorted sequences in respectively `dest1` and `dest2`.

Program 19: Paired back-to-back merge with conditional assignments

```

1  template<class RandomIterator, class OutputIterator>
2  std::pair<OutputIterator, OutputIterator>
3      merge(RandomIterator begin1,
4             RandomIterator end1,
5             RandomIterator begin2,
6             RandomIterator end2,
7             OutputIterator dest1,
8             OutputIterator dest2)
9  {
10     std::iterator_traits<RandomIterator>
11         ::value_type v11, v12, v21, v22, tmp1, tmp2;
12     RandomIterator tmp10, tmp20;
13
14     end1 = end1 - 1;           end2 = end2 - 1;
15     v11 = *begin1;           v21 = *begin2;
16     v12 = *end1;             v22 = *end2;
17     loop: tmp1 = v11 <= v12 ? v11 : v12;   tmp2 = v21 <= v22 ? v21 : v22;
18     *dest1 = tmp1;           *dest2 = tmp2;
19     dest1 = dest1 + 1;       dest2 = dest2 + 1;
20     tmp10 = begin1 + 1;     tmp20 = begin2 + 1;
21     begin1 = v11 <= v12 ? tmp10 : begin1;  begin2 = v21 <= v22 ? tmp20 : begin2;
22     tmp10 = end1 - 1;       tmp20 = end2 - 1;
23     end1 = v11 <= v12 ? end1 : tmp10;     end2 = v21 <= v22 ? end2 : tmp20;
24     v11 = *begin1;           v21 = *begin2;
25     v12 = *end1;             v22 = *end2;
26     if (begin1 <= end1) goto loop; /* hint: branch taken */
27     return std::make_pair(dest1, dest2);
28 }
29

```

Property 35. *The pure-C operation count of Program 19 is bounded above by $9.5n + O(1)$*

Proof. Line 14-16 is executed only once resulting in $O(1)$ operations being executed. Line 17-26 are executed once for each element in the two subarrays, resulting in a contribution of $10n$. \square

Property 36. *The average number of branch missing when running Program 19 is $O(1)$.*

Proof. The branch at line 26 is the only conditional branch. The branch is correctly predicted for all iterations but the last. \square

Property 37. *The number of cache misses incurred when executing Program 19 is bounded above by $2\frac{n}{B} + O(1)$.*

Proof. Follows from Theorem 1 and the fact that merge reads/writes $2n$ elements sequentially. \square

6.2.7 Summary of merge algorithms

Table 6.1 summarises the 6 merge algorithms presented.

Name	Basic / τ	Cache / τ_*	Branch / τ_b
Simple merge (two-way)	$6n + O(1)$	$n/B + O(1)$	$n/2 + O(1)$
Katajainen/Träff two-way merge	$5.5n + O(1)$	$n/B + O(1)$	$n/2 + O(1)$
Katajainen/Träff three-way merge	$6n + O(1)$	$n/B + O(1)$	$5n/6 + O(1)$
Katajainen/Träff four-way merge	$6.5n + O(1)$	$n/B + O(1)$	$n + O(1)$
Back-to-back merge with conditional assignments	$10n + O(1)$	$n/B + O(1)$	$O(1)$
Paired back-to-back merge with conditional assignments	$9.5n + O(1)$	$n/B + O(1)$	$O(1)$

Table 6.1: Summary of merge algorithms

6.3 Sorting

With the various implementations of merge, it is time to construct the sorting algorithm.

Mergesorting can be illustrated by a tree (see Figure 6.1 for the 2-way mergesorting of 16 elements). The leafs represents the unsorted array of elements, and each internal tree node, represents the result after merging children nodes. The root node is the sorted array, obtained after the final merge step.

There are two basic variations of mergesort, top-down and bottom-up. I will discuss both, but with emphasis on top-down mergesorting.

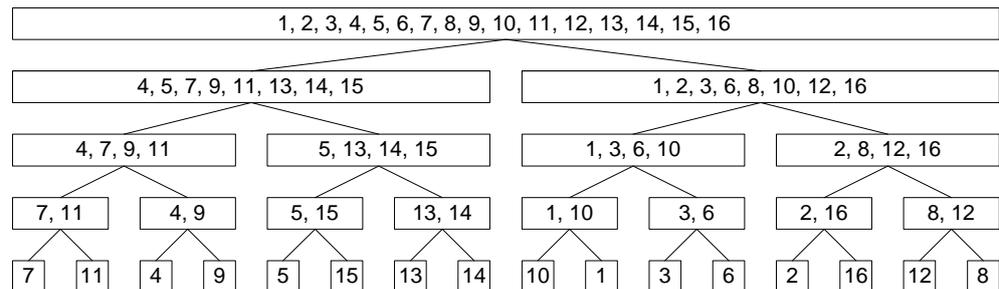


Figure 6.1: Illustration of 2-way mergesorting.

6.3.1 Top-down mergesorting

Top-down d-way mergesorting is a recursive, *divide-and-conquer* algorithm. The idea is to split the array in d subarrays of approximately equal length, sort each

sub array recursively, and then merge the results to one sorted array.

Using the tree analogy (see Figure 6.1), top-down mergesort is a *postorder traversal* of the tree nodes. The child nodes are being visited from left to right, before the current itself.

Program 20 implements a two-way top-down mergesort implementation using a two-way merge subroutine (such as Program 15). The algorithm sorts an array specified by *begin* and *end* into *dest* using temporary storage *tmp*.

Program 20: Top-down mergesort, version 1

```

1  template<class RandomIterator>
2  void mergesort(RandomIterator begin,
3                RandomIterator end,
4                RandomIterator dest,
5                RandomIterator tmp)
6  {
7      size_t len = end - begin;
8      if (len <= 0) return;
9      size_t half = len/2;
10     mergesort(begin, begin+half, tmp, dest);
11     mergesort(begin+half, end, tmp+half, dest);
12     merge(tmp, tmp+half, tmp+half, tmp+len, dest);
13 }
14
```

Space requirements

When analysing space requirements, we seek to find out how much extra space beyond the size of input is required for the execution of a program.

For many execution environments, such as Windows, the size of the stack is much smaller than the size of the heap. If too much space is consumed from the stack, a stack overflow will occur, and the program will terminate. I therefore find it very useful not only to know the general space requirements, but also the space requirements for the stack.

Property 38. *Recursion depth of Program 20 is $1 + \lceil \lg n \rceil$.*

Proof.

$$\begin{aligned}
 T(1) &= 1 \\
 T(n) &\leq 1 + T(\lceil \frac{n}{2} \rceil) \\
 &\leq 2 + \lceil \lg \lceil \frac{n}{2} \rceil \rceil \\
 &= 2 + \lceil \lg \frac{n}{2} \rceil \\
 &= 2 + \lceil \lg n - 1 \rceil \\
 &= 1 + \lceil \lg n \rceil
 \end{aligned}$$

□

Property 39. *The stack space requirement for Program 20 is $O(\lg n)$.*

Proof. This property follows from the fact that the stack depth is $O(\lg n)$ and only a constant number of variables are being allocated per recursive function call. \square

Although the stack size can be limited on some system, a program requiring $O(\lg n)$ stack space is not going to overflow the stack.

Property 40. *Program 20 uses $2n + O(\lg n)$ extra space.*

Proof. The recursion depth is $O(\lg n)$ and there is a constant number of variables that need to be pushed on the stack per recursive call. n words are being used for the destination array, and n words for the temporary array. \square

It is not very difficult to reduce the required extra space from $2n + O(\lg n)$ to $n + O(\lg n)$. The trick is to implement two different versions of the mergesort algorithm. The first one sorts input data “in-place”, using n elements of temporary storage. The second sorts input data to a specified destination array. The implementation for this can be seen in Program 21.

Program 21: Top-down mergesort, version 2

```

1  template<class RandomIterator>
2  void mergesort(RandomIterator begin,
3                RandomIterator end,
4                RandomIterator tmp)
5  {
6      size_t len = end - begin;
7      if (len <= 0) return;
8      size_t half = len/2;
9      mergesort_to(begin, begin+half, tmp);
10     mergesort_to(begin+half, end, tmp+half);
11     merge(tmp, tmp+half, tmp+half, tmp+len, begin);
12 }
13
14 template<class RandomIterator>
15 void mergesort_to(RandomIterator begin,
16                  RandomIterator end,
17                  RandomIterator tmp)
18 {
19     size_t len = end - begin;
20     if (len <= 0) return;
21     size_t half = len/2;
22     mergesort(begin, begin+half, dest);
23     mergesort(begin+half, end, dest+half);
24     merge(begin, begin+half, begin+half, begin+len, dest);
25 }
26
```

`mergesort` sorts the input array (`begin`, `end`) inplace. This is done by splitting the input in two, and sorting each subarray independently using `mergesort_to` to the temporary array. Finally the two sorted subarrays (in `tmp`) are merged.

`mergesort_to` starts by splitting the input array in two, and then sort each range inplace using `mergesort` (and using `dest` as temporary storage). Final step is merging the two subarrays starting at `begin` and `begin+half` into the destination array `dest`.

`mergesort` and `mergesort_to` does nothing if the input array is empty.

d-way mergesorting

It is easy to generalise the 2-way mergesort to d-way mergesorting. The input array is being split in d subarrays. Each subarray is then being sorted recursively, and a final d-way merge completes the algorithm.

The stack depth of d-way mergesorting is upper bounded by \lg_n , and the algorithm uses $n + O(\lg n)$ elements of extra space.

Back-to-back mergesorting

Top-down mergesorting using the back-to-back merge algorithm (see Section 6.2.5) is slightly more complicated. The implementation (see Program 22) consists of two mergesort algorithms. `mergesort` behaves the same way as Program 20, whereas `mergesort_rev` stores the sorted array reversed. They both split the input array in two subarrays of approximately equal size and sort the two arrays. The second subarray however is sorted with `mergesort_rev`, so that the two subarrays form a bitonic sequence. The final step of the algorithm is to merge the two sequences. The final merge is the only difference between `mergesort_rev` and `mergesort`.

Program 22: Top-down back-to-back mergesort

```

1  template<class RandomIterator>
2  void mergesort(RandomIterator begin,
3                RandomIterator end,
4                RandomIterator dest,
5                RandomIterator tmp)
6  {
7      size_t len = end - begin;
8      if (len <= 0) return;
9      size_t half = len/2;
10     mergesort(begin, begin+half, tmp, dest);
11     mergesort_rev(begin+half, end, tmp+half, dest);
12     merge(tmp, tmp+len, dest);
13 }
14
15 template<class RandomIterator>
16 void mergesort_rev(RandomIterator begin,
17                   RandomIterator end,
18                   RandomIterator dest,
19                   RandomIterator tmp)
20 {
21     size_t len = end - begin;
22     if (len <= 0) return;
23     size_t half = len/2;
24     mergesort(begin, begin+half, tmp, dest);
25     mergesort_rev(begin+half, end, tmp+half, dest);
26     merge_rev(tmp, tmp+len, dest);
27 }
28

```

The same trick used on Program 20 for reducing space requirements can be applied to Program 22. This however means that we will need two versions of both `mergesort` and `mergesort_rev`. The implementation that is used for experimentation uses this optimisation, and can be found in Appendix A.2.

Optimisation for small arrays

For small arrays it is likely that a $O(n^2)$ sorting routine, like `insertionsort`, is going to be faster than `mergesort`. The `mergesort` algorithms can possibly be improved by switching to `insertionsort` for small arrays.

The technique is also used in the SGI and STLport implementation of `STL::std::sort`, and in the `quicksort` implementation found in [PTVF92]. The SGI and STLport implementation switches to `insertionsort` for sequences smaller than 16.

Analysis

In order to analyse the running we need a Lemma that solve the recursion formula implied by 2-way mergesorting:

Lemma 2. *The recursion $T(n) = \alpha n + \beta + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$ has solutions of the form $T(n) = \alpha n \lceil \lg n \rceil + \beta(dn - 1)$.*

Proof.

$$\begin{aligned}
 T(n) &= \alpha n + \beta + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) \\
 &= \alpha n + \beta + \alpha \lceil \frac{n}{2} \rceil \lceil \lg \lceil \frac{n}{2} \rceil \rceil + \beta(d \lceil \frac{n}{2} \rceil - 1) + \alpha \lfloor \frac{n}{2} \rfloor \lceil \lg \lfloor \frac{n}{2} \rfloor \rceil + \beta(d \lfloor \frac{n}{2} \rfloor - 1) \\
 &= \alpha n + \alpha n \lceil \lg \frac{n}{2} \rceil + \beta(dn - 1) \\
 &= \alpha n + \alpha n \lceil \lg n - 1 \rceil + \beta(dn - 1) \\
 &= \alpha n \lceil \lg n \rceil + \beta(dn - 1)
 \end{aligned}$$

□

For d -way mergesorting ($d > 2$) we shall be analysing under the assumption that n is a multiple of d . The following Lemma solves the recursion formula implied by d -way mergesorting:

Lemma 3. *Assuming that n is a multiple of d , the recursion $T(n) = \alpha n + \beta + kT(\frac{n}{d})$ has solutions of the form $T(n) = \alpha n \lg_d n + \beta(dn - \frac{2}{d})$.*

Proof.

$$\begin{aligned}
 T(n) &= \alpha n + \beta + kT(\frac{n}{d}) \\
 &= \alpha n + \beta + k(\alpha n \lg_d \frac{n}{d} + \beta(d \frac{n}{d} - \frac{2}{d})) \\
 &= \alpha n + \beta + \alpha n (\lg_d n - 1) + \beta(dn - 2) \\
 &= \alpha n \lg_d n + \beta(dn - 1)
 \end{aligned}$$

□

It is evident from both Lemmas that a d -way merge algorithm with a cost contribution of $\alpha n + O(1)$ results in a mergesort with a cost of $\alpha n \lg_n + O(n)$.

Using Lemma 2 and 3 it is straight forward to analyse the pure-C operations count and the upper bound for the average number of branch misses. The cost bounds can be found in 6.2.

For the number of cache misses we have to be more careful, since the cache miss count calculated for the various merge algorithms assumes that no elements from input or output is currently in the cache.

Lemma 4. *If $M > 2n + O(1)$, no more than $2n + O(1)$ cache misses will be incurred when executing Program 21 (or Program 22).*

Proof. The mergesort algorithm requires $n + O(1)$ extra space beyond the n elements used for the input. So when $n < M/2 + O(1)$, the total number of required elements is smaller than the cache size. There are d input arrays. Each

Name	Basic / τ	Branch / τ_b
Simple merge (two-way)	$6n \lg n + O(n)$	$0.5n \lg n + O(n)$
Katajainen/Träff two-way merge	$5.5n \lg n + O(n)$	$0.5n \lg n + O(n)$
Katajainen/Träff three-way merge	$3.79n \lg n + O(n)$	$0.53n \lg n + O(n)$
Katajainen/Träff four-way merge	$3.25n \lg n + O(n)$	$0.5n \lg n + O(n)$
Back-to-back merge with conditional assignments	$10n \lg n + O(n)$	$O(\lg n)$
Paired back-to-back merge with conditional assignments	$9.5n \lg n + O(n)$	$O(\lg n)$

Table 6.2: Summary of top-down mergesort algorithms using various merge algorithms

array is a consecutive sequence of no more than $\lceil n/d \rceil$ elements, and thus cover no more than $1 + \lceil n/(db) \rceil$ cache blocks. The input array cover in total $d(1 + \lceil n/(db) \rceil) = n + O(1)$ cache block. The output array cover $n/B + O(1)$ cache blocks. Since the cache size is large enough to contain all arrays, no more than one cache miss can occur per cache block. \square

Now let us consider a call to mergesort in the case when $M < 2n + O(1)$. Using Lemma 4 and Lemma 3, we can easily find an upper bound for the number of cache misses by solving the recurrence:

$$T(n) \leq 2n + O(1), \text{ if } M > 2n + O(1)$$

$$T(n) \leq \alpha n + \beta + kT\left(\frac{n}{k}\right), \text{ if } M < 2n + O(1)$$

Table 6.3 summarises the upper bound for the number of incurred branch misses top-down mergesorting using various merge algorithms.

Name	Cache / τ_*
Simple merge (two-way)	$2n/B(\lg n - \lg M/2) + O(n)$
Katajainen/Träff two-way merge	$2n/B(\lg n - \lg M/2) + O(n)$
Katajainen/Träff three-way merge	$1.26n/B(\lg n - \lg M/2) + O(n)$
Katajainen/Träff four-way merge	$n/B(\lg n - \lg M/2) + O(n)$
Back-to-back merge with conditional assignments	$2n/B(\lg n - \lg M/2) + O(n)$
Paired back-to-back merge with conditional assignments	$2n/B(\lg n - \lg M/4) + O(n)$

Table 6.3: Summary of pure-C cache performance top-down mergesort algorithms using various merge algorithms

6.3.2 Bottom-up mergesort

Bottom-up mergesorting is the classical mergesort algorithm as described by Donald E. Knuth in [Knu98, Section 5.2.4]. The algorithm is iterative, where in each iteration the number of sorted subsequences is halved. So in the first iteration the n elements are merged one by one into $n/2$ subarrays of length 2. In the the second iteration the $n/2$ subarrays are merged two by two into $n/4$ subarrays, and so forth.

The principle can be generalised to d -way bottom-up mergesorting. In the first round groups of d elements are merged, reducing the n subarrays to n/d sub-array, and so forth.

If the input array is not a power of d , there will be iterations where the last subarray does not have same length of the others. Bottom-up mergesort implementations need to take special care of this. From now on it will be assumed that the input array size is a power of d .

Following the tree analogy, the bottom-up mergesort correspond to a *reversed breadth-first traversal*, where all siblings are visited before the parent node.

It obvious that bottom-up mergesort execute the exact same merges as top-down mergesort (assuming n is power of d), and thus the total pure-C operations count and average branch miss count from merges is the same.

The same is not true for the pure-C cost of cache misses. If $2n < M$ then all elements and the auxiliary array can fit in the cache, and more than $2n/B$ cache misses will occur (same argument as for Lemma 4).

Let us now assume that $2n > M$. In each iteration several merges will be performed consisting of n sequential element load and n writes. Since only M elements can be in the cache, we will get at least $2n - M$ cache misses. This sums up to a lower bound of $(2n - M) \lg_d n = 2n \lg_d n - M \lg_d n$ cache misses, whereas top-down has an upper bound of $2n \lg_d n - 2n \lg_d M$.

In summary, bottom-up mergesort is a non-recursive variant of mergesort. On the positive side bottom-up mergesort is iterative requiring only a constant amount of stack space. On the negative side bottom-up mergesort it is not as cache efficient as top-down.

6.3.3 Natural mergesort

Natural mergesort is a variation of bottom-up mergesort. The idea is instead of dividing the input array up into singleton subarrays, to divide into up into non-decreasing runs.

Property 41. *Let m specify the number of non-decreasing runs in the input sequence. The pure-C operations count for a d -way natural mergesort is then bounded by $O(n \lg_d m)$*

Proof. In each iteration $O(n)$ work is spent reducing the remaining x sorted subarrays to x/d sorted subarrays. We have n subarrays to begin with and after

$\lg_d n$ iteration, $x/d^{\lg_d n} = 1$ arrays remain. Thus no more than $O(n) \lg_d n = O(n \lg_d n)$ work is spent. \square

For random permutations of $1..n$, the average length of increasing sequences is approximately 2. Obviously for such input data, natural mergesort has little advantage over regular bottom-up mergesorting.

6.3.4 Tiled bottom-up mergesort

Tiled bottom-up mergesorting is a cache optimised version of bottom-up mergesorting where block suitable for the cache size are sorting first in order to avoid unnecessary cache misses. Tiled bottom-up mergesorting can be credited to LaMarca [LL97] and [LaM96]. Tiled mergesort is also analysed by Spork in his masters thesis [Spo99].

The idea is to split the data up into blocks of size $M/2$ elements, and sort each block independently using bottom-up mergesorting. The $2n/M$ sorted blocks are then merged into one array, using bottom-up mergesorting.

The algorithm assumes a two-level memory hierarchy, with only one cache level between CPU registers and memory. However most machines today have at least 2 levels of cache.

Tiled mergesorting can easily be extended to handle two-level caches. Let us consider the case with two levels of cache of sizes M_1 and M_2 , where $M_1 < M_2$. The algorithm would divide the elements into blocks of size $M_2/2$, each block is then divided into subblocks of size $M_1/2$. We then sort each block, using the normal tiled bottom-up mergesorting. Finally all of the $2n/M_2$ sorted blocks are merged into one sorted array. It should be obvious that principle generalises for multi-level caches.

Tiled bottom-up mergesort avoids recursion, but maintain the good cache performance of top-down mergesorting. The caveat however is that the algorithm requires knowledge of:

- The number of cache levels of the machine.
- The size of each cache level.

6.4 Experiments

I have benchmarked top-down mergesorting for the following merge algorithms:

- Two-way merge of Katajainen and Träff (Program 15)
- Four-way merge of Katajainen and Träff (Program 17)
- Back-to-back merge with conditional assignments (Program 18)
- Paired back-to-back merge with conditional assignments (Program 19)

The various mergesorting algorithm are compared with STL `std::sort` as reference. The `std::sort` from STLport switches to `std::__insertion_sort`, when less than 16 elements remain. I have implemented all top-down mergesort programs, so that they also switch to insertion sort at the same threshold as `std::sort`. For input data I use random data of type `long`, generated with library function `rand` from `<cstdlib>`. I benchmark with data sizes from 256 and up to $2097152 = 2^{21}$, with a size progression factor of 2^{-4} . The data is being created just prior to running the sort program, hence the data will be in the memory cache, when starting the sort program. This is what usually is referred to as a warm start³.

The program is implemented, so that when running in Debug mode, it is checked that the output of each mergesort program is a sorted sequence. The full test program containing all sorting algorithms can be found in Appendix A.2

6.4.1 Expectations

We have seen for the small example `limit_data` on page 22 that the cost of branch misprediction is very high. It was estimated that τ_b is roughly 15τ . Assuming that $\tau_b = 15\tau$, Table 6.4 gives estimates for the running time measured in τ . The table does not include the cost of cache misses. For small input data, close to none cache misses will occur, and it is expected that cost of Table 6.4 will be reflected in the actual running time of the programs. The paired back-to-back mergesort is not included in the table, since its pure-C running is practically identical to normal back-to-back mergesort.

Name	Cost / τ
Katajainen/Träff two-way merge	$13n \lg n + O(n)$
Katajainen/Träff four-way merge	$10.75n \lg n + O(n)$
Back-to-back merge with conditional assignments	$10n \lg n + O(n)$

Table 6.4: Summary of top-down mergesort algorithms using various merge algorithms

Considering the tremendous effect of pairing straight-line binary search (see Section 5.3.2), it is expected that we will get a performance gain in paired back-to-back mergesort (see Program 19) over paired back-to-back mergesort (see Program 18). It is going be interesting to see if it will be able to compete with `std::sort` of STLport.

According to [LL97], a quicksort program with random input has an expected number of cache misses equal to

$$2n/B \log n/M + O(n) = 1.39n/B \lg n/M + O(n).$$

Since four-way mergesorting has an upper-bound for cache misses of $n/B \lg n/M$, I would expect four-way mergesorting to be competitive with `std::sort` for large data sizes.

³Not to be confused with Bentley's use of "warm start" in [Ben00a].

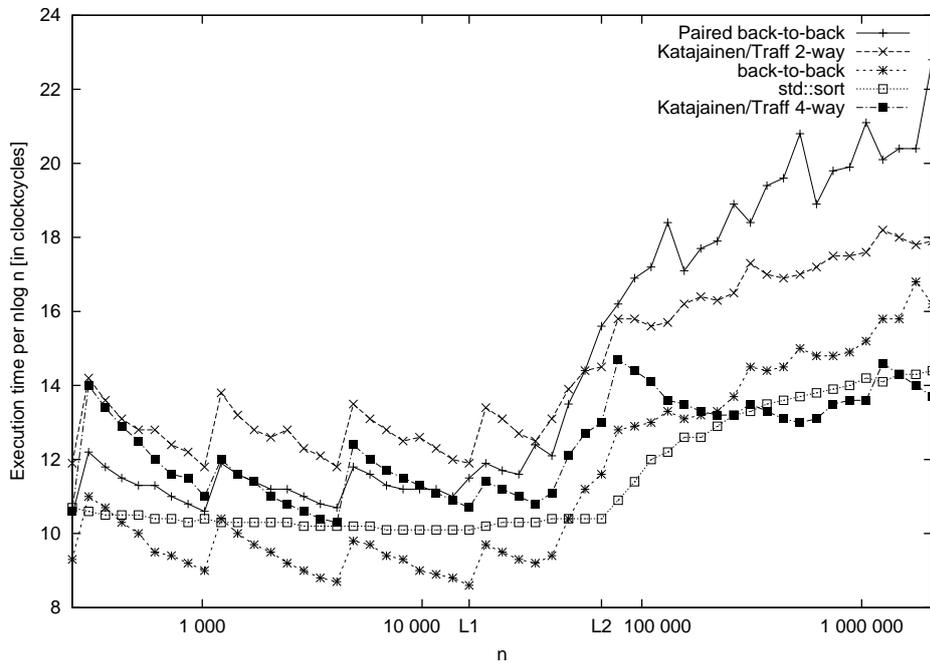


Figure 6.2: Mergesort on 1000 MHz Athlon

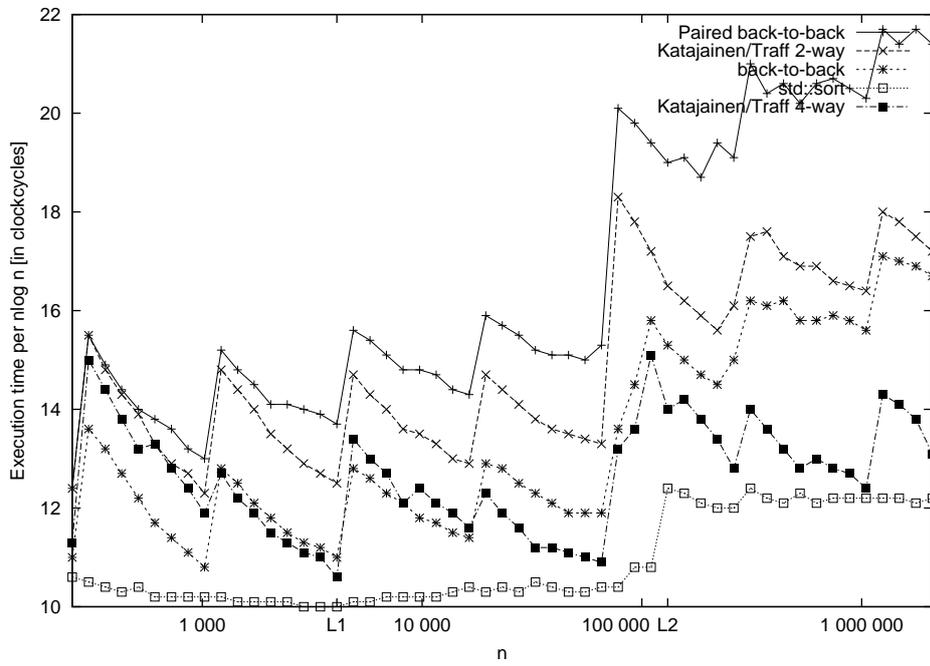


Figure 6.3: Mergesort on 450 MHz Pentium 2

6.4.2 Results

The results can be seen in Figure 6.2 and 6.3. The graphs display running time in clock cycles per $n \lg n$ as function of n . Figure 6.2 shows the results for the 1000 MHz Athlon, and Figure 6.3 the results for the 450 MHz Pentium 2.

The first thing to notice for both graphs, is the knee bends in the running time when the required memory of the program gets larger than the cache size. It is easiest to see for STL `std::sort` because it has smooth running times. The knee bend occurs at the limit for the size of the L2 cache. The knee bend for the L1 cache is hardly noticeable. The mergesort algorithm use n elements of extra space, whereas quicksort only uses $O(\lg n)$ elements of extra space. Observe how this is reflected in the running time by the fact that the bend of mergesort programs comes a little sooner than for the quicksort program.

On the Athlon it can be seen that the four-way mergesort (Program 17) faster than the two-way mergesort. This is expected as seen in Table 6.4, and from the fact that the four-way mergesort has better cache performance than the two-way (see Table 6.3). For small inputs where all data fits in the cache, the back-to-back mergesort is faster than the four-way mergesort, as predicted in Table 6.4. For small input, the back-to-back mergesort is even faster than `std::sort`. For large input, the four-way mergesort is on par with `std::sort`.

On the Pentium 2, the four-way mergesort is faster than the two-way as expected. For small inputs, the four-way mergesort and the back-to-back mergesort have practically the same running time. This however is not entirely unexpected since their estimated running times are very close. The STL implementation `std::sort` is faster than all mergesort algorithms for all values of n .

On both processors the paired back-to-back mergesort disappoints by having a much slower running time than normal back-to-back mergesort. It was expected that the paired back-to-back mergesort would be faster than back-to-back mergesort. However, practice has shown that it is slower. Inspecting the generated assembler code for Program 19 revealed that some of variables were being moved back and forth from memory to register. It appeared that the number of registers were too small. The x86 instruction set has only 7 general registers, and the program requires far more.

Since the Pentium 2 and Athlon allegedly has more register internally than they employ with register renaming, I have tried to fine tune the paired back-to-back mergesort to get better performance. What I have done is to instead of running two merges (A and B) simultaneously, I run one iteration of B immediately after an iteration of A. By doing so, I hope that the dynamic scheduling of the processors will cause instructions from merge B to be executed simultaneously with instructions from merge A. The implementation can be found in namespace `tuned_paired_back_to_back` in Appendix A.2.

The tuned version is a little faster than normal back-to-back mergesort, but only with a very margin, and only for small data that fits in the cache. It must be concluded that the attempt to parallelise two merges has failed.

Part of the explanation why paired back-to-back mergesort fails can be found in the source for back-to-back mergesort. Following is a manual rescheduling for parallelism of loop of Program 18:

```

13 loop: tmp = v1<=v2 ? v1 : v2; tmp_b = begin + 1; tmp_e = end - 1;
14 *dest = tmp; begin = v1<=v2 ? tmp_b : begin; end = v1<=v2 ? end : tmp_e;
15 dest = dest + 1; v1 = *begin; v2 = *end;
16 if (begin <= end) goto loop /* hint: branch taken */;
17

```

It demonstrates how that there are really three independent threads of executions through the loop. It is probable that the dynamic scheduler of the Athlon and Pentium 2 is capable of finding instructions to be executed simultaneously. This may explain why the pairing fails, because the ILP capabilities of the processor are already being used for the normal back-to-back merge program.

Still with the tuned paired back-to-back algorithm being the fastest for small sequences and 4-way mergesort looking promising for large sequences, I decide to try combining them to achieve an efficient sorting program, called *hybrid mergesort*. The principle is simply to recursively sort using 4-way mergesort, and when the remaining elements are less than 32000, I switch to the fine-tuned paired back-to-back mergesort program. The hybrid mergesort is 5-10% faster for most values of n . The results can be seen in Figure 6.4, where the sorting algorithms are compared with `std::sort` as index.

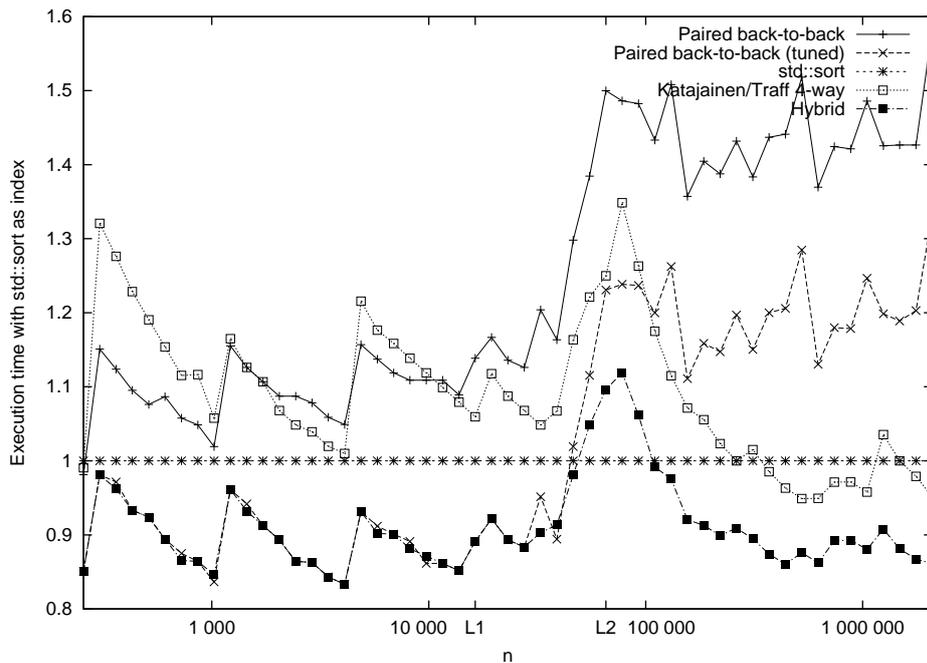


Figure 6.4: Mergesort compared with hybrid mergesort on 1000 MHz Athlon

6.5 Summary

Several mergesort algorithms have been analysed under pure-C refined to account for branch misprediction and cache misses. The experimental results are

in agreement with the theoretical model. Without the modelling for branch mispredictions the pure-C cost model would not have been able to explain why back-to-back mergesort runs faster than both the 2-way and 4-way mergesort algorithms of Katajainen and Träff.

The attempt to parallelise two back-to-back merges has failed. It is not entirely clear whether this is because register shortage, or the fact that the normal back-to-back merge is already taking advantage of execution resources available.

Finally a *hybrid* sorting algorithm has been constructed, combining the cache efficiency of 4-way mergesort with the branch efficiency of back-to-back mergesort.

Chapter 7

Conclusion

The main contribution of this work is the refinement of the pure-C cost model to account for the latencies incurred by branch mispredictions. The model has been used to analyse the efficiency of binary search and mergesort programs. The findings have been tested in practice on two different machines, and has shown to be somewhat accurate. The test results indicate that the refinement has increased the accuracy of the pure-C cost model.

Furthermore instruction level parallelism has been discussed, and an alternative model, the ILP pure-C cost model, has been suggested. An ILP efficient range search program has been implemented, the paired straight-line range search, which exploits the parallelism to execute two binary searches simultaneously at the cost of one. This result is a clear indication that the pure-C cost model idiom of counting instructions is not sufficient in the modelling of program execution on today's superscalar processors. The attempt with the paired back-to-back merge to parallelise two merges has failed on the two test machines. It is suggested that the experiment failed either because of lack of registers, or that the normal back-to-back merge program might already be exploiting the instruction level parallelism available on the test machines to the fullest.

Further work

The findings in this thesis are only backed by experimental results from two different machines, the AMD Athlon and the Pentium 2. The experiments should be performed on more machines to validate the model.

It is mentioned in Section 2.4 that some modern processors have machine instructions that let us prefetch memory data to the cache without blocking other instructions, and instructions to write to memory without polluting the cache. It would be interesting to use these in practice to see if the programs presented herein can be improved.

Bibliography

- [Adv99] Advanced Micro Devices. *AMD Athlon Processor - Technical Brief*, December 1999. Publication 22054, Revision D.
- [Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1st edition, 1986.
- [Ben00a] Jon Bentley. Binary search: Algorithm, code, and caching. *Dr. Dobbs's Journal*, (311):111–116, April 2000.
- [Ben00b] Jon Bentley. *Programming Pearls*. Addison-Wesley, 2nd edition, 2000.
- [BKS00] Jesper Bojesen, Jyrki Katajainen, and Maz Spork. Performance engineering case study: heap construction. *The ACM Journal of Experimental Algorithmics*, to appear, 2000.
- [Boj00] Jesper Bojesen. Managing memory hierarchies. Master's thesis, Department of Computer Science, University of Copenhagen, 2000.
- [CLR94] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1994.
- [DS98] Kevin Dowd and Charles Severance. *High Performance Computing*. O'Reilly, 2nd edition, 1998.
- [GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics, 2nd edition*. Addison-Wesley, 1994.
- [HP96] John Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.
- [Int] Intel. *The Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture*. Order Number 243190.
- [Jon97] Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. The MIT Press, 1997.
- [Jos99] Nicolai M. Josuttis. *The C++ Standard Library*. Addison-Wesley, 1999.
- [Knu98] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison Wesley Longman, 2nd edition, 1998.
- [KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language, 2nd edition*. Prentice Hall, 1988.

- [KT97] Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science 1203*, volume 1203, pages 217–228. Springer-Verlag, 1997.
- [LaM96] Anthony G. LaMarca. *Caches and Algorithms*. PhD thesis, University of Washington, 1996.
- [LL97] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.
- [MP95] Rajeev Motwani and Raghaven Prabhakar. *Randomized Algorithms*. Cambridge, 1995.
- [Nil96] Stefan Nilsson. *Radix Sorting & Searching*. PhD thesis, Department of Computer Science, Lund University, 1996.
- [Nil00] Stefan Nilsson. The fastest sorting algorithm? *Dr. Dobbs's Journal*, April 2000.
- [Pat95] David A. Patterson. Microprocessors in 2020. *Scientific American*, 273:48–51, September 1995.
- [Ped99] Morten Nicolaj Pedersen. A study of the practical significance of word ram algorithms for internal integer sorting. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, 1999.
- [PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.
- [Sed98] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 3rd edition, 1998.
- [Spo99] Maz Spork. Design and analysis of cache-conscious programs. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.
- [Str95] Bjarne Stroustrup. *The C++ Programming Language, 3rd edition*. Addison-Wesley, 1995.

Appendix A

Program source code

A.1 Binary Search

```
1  #define __STL_NO_SGI_IOSTREAMS
    #include <cstdio>
    #include <cstdlib>
5  #include <cmath>
    #include <ctime>

    #include <algorithm>
    #include <vector>
10 #include <string>
    #include <map>
    #include <iostream>

    #include <ls/microtime.h>
15 using std::swap;

    const size_t M1_MIN = 1024;
    const size_t M1_MAX = 1024;
20
    const size_t M2_MIN = 63;
    const size_t M2_MAX = 63;

    const size_t data_max = 256*256*256;
25 long data[data_max];

    unsigned char log_table[256] = {
30     0xff, // <--- rogue value
        0,
        1, 1,
        2, 2, 2, 2,
        3, 3, 3, 3, 3, 3, 3, 3,
        4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
35     5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
        5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
40     6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
45     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
```

```

7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
50 };
// calculates floor( log2( n ) )
inline size_t log2(unsigned long n)
{
55     long rv = 0;
    if (n & 0xffff0000) { rv += 16; n >>= 16; }
    if (n & 0xff00) { rv += 8; n >>= 8; }
    return rv + log_table[n];
}

60 namespace dummy {

    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound(
65         ForwardIterator begin,
        ForwardIterator end,
        const T& val)
    {
        return ForwardIterator();
    }

70     template<typename ForwardIterator, typename T>
    std::pair<ForwardIterator, ForwardIterator> range_search(
        ForwardIterator begin,
75         ForwardIterator end,
        const T& val1,
        const T& val2)
    {
        return std::make_pair(ForwardIterator(), ForwardIterator());
    }

80 }

namespace stl_version {

85     template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound(ForwardIterator begin,
                                ForwardIterator end, const
                                T& val)
    {
90         std::iterator_traits<ForwardIterator>::difference_type half;
        ForwardIterator middle;

        size_t len = std::distance(begin, end);

95         while (len > 0)
            {
                half = len >> 1;
                middle = begin;
                std::advance( middle, half );
100                 if (*middle < val)
                    {
                        begin = middle;
                        ++begin;
                        len = len - half - 1;
105                 }
                else
                    len = half;
            }
        return begin;
110    }

    namespace cold_start {

115        template<class RandomIterator, class T>
        std::pair<RandomIterator, RandomIterator>
        range_search(RandomIterator begin,
                    RandomIterator end,
                    const T& val1,
                    const T& val2)
120        {

```

```

        return std::make_pair(lower_bound(begin,end,val1),
                               lower_bound(begin,end,val2));
    }
125 }

    namespace warm_start {
        template<class ForwardIterator, class T>
        std::pair<ForwardIterator, ForwardIterator>
130 range_search(ForwardIterator begin,
                ForwardIterator end,
                const T& lower,
                const T& upper)
        {
135     ForwardIterator it = std::lower_bound(begin, end, lower);
        return std::make_pair( it, std::lower_bound(it, end, upper) );
        }
    }

140 namespace purec {

        template<class RandomIterator, class T>
        RandomIterator lower_bound(RandomIterator begin,
145     RandomIterator end,
        const T& val)
        {
            RandomIterator middle;
            std::iterator_traits<RandomIterator>::value_type middle_value;
150     ptrdiff_t len = end - begin;
            ptrdiff_t half;
            goto loop_start;
recurse_right_half:
            begin = middle;
155     begin = begin + 1;
            len = len - half;
            len = len - 1;
loop_start:
            if (len == 0) goto exit_loop; /* hint: not taken */
160     half = len >> 1;
            middle = begin + half;
            middle_value = *middle;
            if (middle_value < val) goto recurse_right_half;
            len = half;
            goto loop_start;
165 exit_loop:
            return begin;
        }

170     template<class RandomIterator, class T>
        std::pair<RandomIterator, RandomIterator>
        range_search(RandomIterator begin,
                    RandomIterator end,
                    const T& val1,
175     const T& val2)
        {
            return std::make_pair(lower_bound(begin,end,val1),
                                   lower_bound(begin,end,val2));
        }
180     }
    }

185 namespace optimised {

        template<typename RandomIterator, typename T>
        RandomIterator lower_bound(RandomIterator begin,
190     RandomIterator end,
        const T& val)
        {
            std::iterator_traits<RandomIterator>::difference_type n = end - begin;

            unsigned long i = (1 << log2(n)) - 1;

```

```

195         begin = begin[i] < val ? begin + (n - i) : begin;
           while (i > 0) {
               i = i >> 1;
               begin = begin[i] < val ? begin + i + 1: begin;
200         }
           return begin;
       }

       namespace purec {
205     template<class RandomIterator, class T>
       RandomIterator lower_bound(
           RandomIterator begin,
           RandomIterator end,
           const T& val)
210     {
           ptrdiff_t n = end - begin;
           ptrdiff_t i = (1 << log2(n)) - 1; // 0(log sizeof(ptrdiff_t))

           RandomIterator begin_right, middle;
215     std::iterator_traits<RandomIterator>::value_type middle_element;

           middle = begin + i;
           middle_element = *middle;
           begin_right = begin + n;
220     begin_right = begin_right - i;
           begin = middle_element < val ? begin_right : begin;

       loop:
           i = i >> 1;
225     middle = begin + i;
           middle_element = *middle;
           begin_right = begin + i;
           begin_right = begin_right + 1;
           begin = middle_element < val ? begin_right : begin;
230     if (i > 0) goto loop;

           return begin;
       }
235     }
}

       namespace straight_line {
240     template<class RandomIterator, class T>
       RandomIterator lower_bound(RandomIterator begin,
           RandomIterator end,
           const T& val)
245     {
           std::iterator_traits<RandomIterator>::
               difference_type n, logn, i;
           n = end - begin;
           logn = log2(n);
250     i = (1 << logn) - 1;
           begin = begin[i] < val ? begin + (n - i) : begin;

           switch (logn) {
255     case 31:
               begin = begin[1073741824 - 1] < val
                   ? begin + 1073741824 : begin;
           case 30:
               begin = begin[536870912 - 1] < val
                   ? begin + 536870912 : begin;
260     case 29:
               begin = begin[268435456 - 1] < val
                   ? begin + 268435456 : begin;
           case 28:
               begin = begin[134217728 - 1] < val
265     ? begin + 134217728 : begin;
           case 27:
               begin = begin[67108864 - 1] < val
                   ? begin + 67108864 : begin;

```

```
270     case 26:
        begin = begin[33554432 - 1] < val
            ? begin + 33554432 : begin;
        case 25:
        begin = begin[16777216 - 1] < val
            ? begin + 16777216 : begin;
275     case 24:
        begin = begin[8388608 - 1] < val
            ? begin + 8388608 : begin;
        case 23:
        begin = begin[4194304 - 1] < val
            ? begin + 4194304 : begin;
280     case 22:
        begin = begin[2097152 - 1] < val
            ? begin + 2097152 : begin;
        case 21:
        begin = begin[1048576 - 1] < val
            ? begin + 1048576 : begin;
285     case 20:
        begin = begin[524288 - 1] < val
            ? begin + 524288 : begin;
        case 19:
        begin = begin[262144 - 1] < val
            ? begin + 262144 : begin;
290     case 18:
        begin = begin[131072 - 1] < val
            ? begin + 131072 : begin;
        case 17:
        begin = begin[65536 - 1] < val
            ? begin + 65536 : begin;
295     case 16:
        begin = begin[32768 - 1] < val
            ? begin + 32768 : begin;
        case 15:
        begin = begin[16384 - 1] < val
            ? begin + 16384 : begin;
300     case 14:
        begin = begin[8192 - 1] < val
            ? begin + 8192 : begin;
        case 13:
        begin = begin[4096 - 1] < val
            ? begin + 4096 : begin;
305     case 12:
        begin = begin[2048 - 1] < val
            ? begin + 2048 : begin;
        case 11:
        begin = begin[1024 - 1] < val
            ? begin + 1024 : begin;
310     case 10:
        begin = begin[512 - 1] < val
            ? begin + 512 : begin;
        case 9:
        begin = begin[256 - 1] < val
            ? begin + 256 : begin;
315     case 8:
        begin = begin[128 - 1] < val
            ? begin + 128 : begin;
        case 7:
        begin = begin[64 - 1] < val
            ? begin + 64 : begin;
320     case 6:
        begin = begin[32 - 1] < val
            ? begin + 32 : begin;
        case 5:
        begin = begin[16 - 1] < val
            ? begin + 16 : begin;
325     case 4:
        begin = begin[8 - 1] < val
            ? begin + 8 : begin;
        case 3:
        begin = begin[4 - 1] < val
            ? begin + 4 : begin;
330     case 2:
        begin = begin[2 - 1] < val
```



```

begin2 = begin2[16777216 - 1] < val2
    ? begin2 + 16777216 : begin2;
420 case 24:
begin1 = begin1[8388608 - 1] < val1
    ? begin1 + 8388608 : begin1;
begin2 = begin2[8388608 - 1] < val2
    ? begin2 + 8388608 : begin2;
425 case 23:
begin1 = begin1[4194304 - 1] < val1
    ? begin1 + 4194304 : begin1;
begin2 = begin2[4194304 - 1] < val2
    ? begin2 + 4194304 : begin2;
430 case 22:
begin1 = begin1[2097152 - 1] < val1
    ? begin1 + 2097152 : begin1;
begin2 = begin2[2097152 - 1] < val2
    ? begin2 + 2097152 : begin2;
435 case 21:
begin1 = begin1[1048576 - 1] < val1
    ? begin1 + 1048576 : begin1;
begin2 = begin2[1048576 - 1] < val2
    ? begin2 + 1048576 : begin2;
440 case 20:
begin1 = begin1[524288 - 1] < val1
    ? begin1 + 524288 : begin1;
begin2 = begin2[524288 - 1] < val2
    ? begin2 + 524288 : begin2;
445 case 19:
begin1 = begin1[262144 - 1] < val1
    ? begin1 + 262144 : begin1;
begin2 = begin2[262144 - 1] < val2
    ? begin2 + 262144 : begin2;
450 case 18:
begin1 = begin1[131072 - 1] < val1
    ? begin1 + 131072 : begin1;
begin2 = begin2[131072 - 1] < val2
    ? begin2 + 131072 : begin2;
455 case 17:
begin1 = begin1[65536 - 1] < val1
    ? begin1 + 65536 : begin1;
begin2 = begin2[65536 - 1] < val2
    ? begin2 + 65536 : begin2;
460 case 16:
begin1 = begin1[32768 - 1] < val1
    ? begin1 + 32768 : begin1;
begin2 = begin2[32768 - 1] < val2
    ? begin2 + 32768 : begin2;
465 case 15:
begin1 = begin1[16384 - 1] < val1
    ? begin1 + 16384 : begin1;
begin2 = begin2[16384 - 1] < val2
    ? begin2 + 16384 : begin2;
470 case 14:
begin1 = begin1[8192 - 1] < val1
    ? begin1 + 8192 : begin1;
begin2 = begin2[8192 - 1] < val2
    ? begin2 + 8192 : begin2;
475 case 13:
begin1 = begin1[4096 - 1] < val1
    ? begin1 + 4096 : begin1;
begin2 = begin2[4096 - 1] < val2
    ? begin2 + 4096 : begin2;
480 case 12:
begin1 = begin1[2048 - 1] < val1
    ? begin1 + 2048 : begin1;
begin2 = begin2[2048 - 1] < val2
    ? begin2 + 2048 : begin2;
485 case 11:
begin1 = begin1[1024 - 1] < val1
    ? begin1 + 1024 : begin1;
begin2 = begin2[1024 - 1] < val2
    ? begin2 + 1024 : begin2;
490 case 10:
begin1 = begin1[512 - 1] < val1

```

```

        ? begin1 + 512 : begin1;
begin2 = begin2[512 - 1] < val2
        ? begin2 + 512 : begin2;
495     case 9:
        begin1 = begin1[256 - 1] < val1
        ? begin1 + 256 : begin1;
        begin2 = begin2[256 - 1] < val2
        ? begin2 + 256 : begin2;
500     case 8:
        begin1 = begin1[128 - 1] < val1
        ? begin1 + 128 : begin1;
        begin2 = begin2[128 - 1] < val2
        ? begin2 + 128 : begin2;
505     case 7:
        begin1 = begin1[64 - 1] < val1
        ? begin1 + 64 : begin1;
        begin2 = begin2[64 - 1] < val2
        ? begin2 + 64 : begin2;
510     case 6:
        begin1 = begin1[32 - 1] < val1
        ? begin1 + 32 : begin1;
        begin2 = begin2[32 - 1] < val2
        ? begin2 + 32 : begin2;
515     case 5:
        begin1 = begin1[16 - 1] < val1
        ? begin1 + 16 : begin1;
        begin2 = begin2[16 - 1] < val2
        ? begin2 + 16 : begin2;
520     case 4:
        begin1 = begin1[8 - 1] < val1
        ? begin1 + 8 : begin1;
        begin2 = begin2[8 - 1] < val2
        ? begin2 + 8 : begin2;
525     case 3:
        begin1 = begin1[4 - 1] < val1
        ? begin1 + 4 : begin1;
        begin2 = begin2[4 - 1] < val2
        ? begin2 + 4 : begin2;
530     case 2:
        begin1 = begin1[2 - 1] < val1
        ? begin1 + 2 : begin1;
        begin2 = begin2[2 - 1] < val2
        ? begin2 + 2 : begin2;
535     case 1:
        begin1 = begin1[1 - 1] < val1
        ? begin1 + 1 : begin1;
        begin2 = begin2[1 - 1] < val2
        ? begin2 + 1 : begin2;
540     default:
        break;
    }
    return std::make_pair(begin1, begin2);
}

545

}

550 void initdata(unsigned long n) {
    for (size_t i=0; i<n; ++i) data[i] = rand();
    std::sort(data, data+n);
}

555 void touch_data(unsigned long n) {
    for (size_t i=0; i<n; ++i) if (data[i] > RAND_MAX) data[i] = 0;
}

560 void main_rs()
{
    FILE *file;
    file = fopen("range_search.dat", "w");

    typedef std::pair<long*, long*>(*LPRS)(long*, long*, const long&, const long&);
    LPRS rs[] = { dummy::range_search,

```

```

565             stl_version::cold_start::range_search,
                stl_version::warm_start::range_search,
                straight_line::range_search_not_paired,
                straight_line::range_search };

570     const size_t nors = sizeof(rs)/sizeof(LPRS);

        size_t M1 = M1_MAX;
        size_t M2 = M2_MAX;
        size_t n1 = 4;

575     for (size_t i=4; i<20; ++i, n1*=2, M1/=2, M2/=2) {
        size_t n = (1 << i) - 1;
        if (n>data_max) break;
        if (M1 < M1_MIN) M1 = M1_MIN;
580         if (M2 < M2_MIN) M2 = M2_MIN;

        __int64 elapsed_time;
        float ftime[nors];

585         std::pair<long*, long*> res_old[M2_MAX];

        for (size_t k=0; k<nors; ++k) ftime[k] = 0.0;
        for (size_t j=0; j<M1; ++j) {
            initdata(n);

590             long val1[M2_MAX];
            long val2[M2_MAX];
            for (size_t l=0; l<M2; ++l) {
                val1[l] = rand();
595                 val2[l] = rand();
                if (val1[l] > val2[l]) std::swap(val1[l], val2[l]);
            }
            for (k=0; k<nors; ++k) {
                touch_data(n);

600                 float ftime2[M2_MAX];
                for (size_t l=0; l<M2; ++l) {

605                     microtime::start();
                        std::pair<long*, long*> res = rs[k](data, data+n, val1[l], val2[l]);
                        microtime::stop(elapsed_time);

                        ftime2[l] = float(elapsed_time);

610                     if (k==1) res_old[l] = res;
                        else if (k>1) {
                            if (res != res_old[l]) {
                                printf("error\n");
                            }
                        }
                    }
                }
                std::sort(ftime2, ftime2 + M2);
                ftime[k] += ftime2[M2/2];
            }
        }
        for (k=0; k<nors; ++k) ftime[k] /= float(M1);
        fprintf(file, "%d", n);
        printf("%d", n);
        float offset = ftime[0];
625         printf("\t%.1f", offset);
        for (k=1; k<nors; ++k) {
            fprintf(file, "\t%.1f", ftime[k] - offset);
            printf("\t%.1f", (ftime[k] - offset)/(log(n+1)/log(2)));
        }
630         fprintf(file, "\n");
        printf("\n");
    }
    fclose(file);

635 }

void main_bs()
{
    FILE *file;

```

```

640     file = fopen("lower_bound.dat", "w");

typedef long>(*LPBS)(long*, long*, const long&);
LPBS bs[] = { dummy::lower_bound,
              std::lower_bound,
              stl_version::lower_bound, stl_version::purec::lower_bound,
645             optimised::lower_bound, optimised::purec::lower_bound,
              straight_line::lower_bound };

const size_t nobs = sizeof(bs)/sizeof(LPBS);

650     size_t M1 = M1_MAX;
        size_t M2 = M2_MAX;
        size_t n1 = 4;
        for (size_t i=4; i<20; ++i, n1*=2, M1/=2, M2/=2) {
            size_t n = (1 << i) - 1;
655             if (n>data_max) break;
            if (M1 < M1_MIN) M1 = M1_MIN;
            if (M2 < M2_MIN) M2 = M2_MIN;

            __int64 elapsed_time;
660             float ftime[nobs];

            long* res_old[M2_MAX];

            for (size_t k=0; k<nobs; ++k) ftime[k] = 0.0;
            for (size_t j=0; j<M1; ++j) {
665                 initdata(n);
                long val[M2_MAX];
                for (size_t l=0; l<M2; ++l) val[l] = rand();

                for (k=0; k<nobs; ++k) {
670                     float ftime2[M2_MAX];
                    touch_data(n);
                    for (size_t l=0; l<M2; ++l) {
675                         long* res;
                        {
                            microtime::start();
680                             res = bs[k](data,data+n, val[l]);
                            microtime::stop(elapsed_time);
                        }

                        ftime2[l] = float(elapsed_time);

685                         if (k==1) res_old[l] = res;
                        else if (k>1) {
                            if (res != res_old[l]) {
                                printf("error\n");
                            }
                        }
                    }

                    std::sort(ftime2, ftime2 + M2);
                    ftime[k] += ftime2[M2/2];
                }
            }

695             for (k=0; k<nobs; ++k) ftime[k] /= float(M1);
            fprintf(file, "%d", n);
            printf("%d", n);
700             float offset = ftime[0];
            printf("\t%.1f", offset);
            for (k=1; k<nobs; ++k) {
                fprintf(file, "\t%.1f", ftime[k] - offset);
                printf("\t%.1f", (ftime[k] - offset)/(log(n+1)/log(2)));
705             }
            fprintf(file, "\n");
            printf("\n");
        }
        fclose(file);
710     }

void main()

```

```

    {
715         srand( (unsigned)time( NULL ) );
        microtime::calibrate();
        main_bs();
        main_rs();
    }

```

A.2 Mergesort

```

1  #define __STL_NO_SGI_IOSTREAMS

    #include <cstdio>
    #include <cstdlib>
5  #include <cmath>
    #include <ctime>

    #include <algorithm>
    #include <vector>
10  #include <string>
    #include <map>
    #include <iostream>
    #include <iterator>

15  #include <ls/microtime.h>

    using std::swap;

    #ifndef NDEBUG
20  #define ASSERT_SORTED
    #pragma message("*** Debug build ***")
    #endif

    const size_t threshold = 17;
25  const size_t cache_threshold = 32000;

    const size_t data_max = 32*256*256;
    long* data;
    long* tmp;
30  const size_t NUMBER_OF_RUNS = 25;

    #ifdef ASSERT_SORTED
35  #define assert(x) if (!(x)) __asm int 3;
    #else
    #define assert(x)
    #endif

40  template<class RandomIterator>
    inline void assert_sorted(RandomIterator begin, RandomIterator end)
    {
    #ifdef ASSERT_SORTED
45  bool fail = false;
        if (begin == end) {
            fail = true;
        }
        if (fail) __asm int 3
50  for (++begin; begin != end; ++begin) {
            if (begin[-1] > *begin) {
                __asm int 3
            }
        }
    #endif
55  }

    template<class RandomIterator>
    inline void assert_sorted_rev(RandomIterator begin, RandomIterator end)
60  #ifdef ASSERT_SORTED

```

```

bool fail = false;
if (begin == end) {
    fail = true;
}
65 if (fail) __asm int 3
    for (++begin; begin != end; ++begin) {
        if (begin[-1] < *begin) {
            __asm int 3
70         }
    }
#endif
}

namespace kt {
75     namespace two_way {

        template<class RandomIterator, class OutputIterator>
            OutputIterator merge(RandomIterator b1,
80                RandomIterator e1,
                RandomIterator b2,
                RandomIterator e2,
                OutputIterator dest)
        {
85            if (e2[-1] < e1[-1]) {
                std::swap(b1, b2);
                std::swap(e1, e2);
            }

90            std::iterator_traits<RandomIterator>::value_type v1, v2, tv1, tv2;
            RandomIterator t1, t2;
            v1 = *b1; v2 = *b2;
            t1 = e1; t1 = t1 - 1; tv1 = *t1;
            t2 = e2; t2 = t2 - 1; tv2 = *t2;
95            if (tv1 > tv2) goto test_1;
            goto test_2;
        out_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
        test_1: if (v1 <= v2) goto out_1;
            b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
100        if (b2 < e2) goto test_1; /* hint: taken */
        copy_1: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
            if (b1 < e1) goto copy_1; /* hint: taken */
            goto end;
        out_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
105        test_2: if (v1 > v2) goto out_2;
            b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
            if (b1 < e1) goto test_2; /* hint: taken */
        copy_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
            if (b2 < e2) goto copy_2; /* hint: taken */
110        end: return dest;
    }

    template<class RandomIterator>
        void sort(RandomIterator begin, RandomIterator end, RandomIterator dest)
115    {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;

        if (n > threshold) {
120            RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;

            sort_to(begin, mid, dest);
            sort_to(mid, end, dest_mid);

125            merge(dest, dest_mid, dest_mid, dest+n, begin);
        } else {
            std::_insertion_sort(
                begin, end,
130                std::less<std::iterator_traits<RandomIterator>::value_type>());
        }
    }

    template<class RandomIterator>
        void sort_to(RandomIterator begin, RandomIterator end, RandomIterator dest)

```

```

135     {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;

        if (n > threshold) {
            RandomIterator mid = begin + n/2;
140         RandomIterator dest_mid = dest + n/2;

            sort(begin, mid, dest);
            sort(mid, end, dest);

145         merge(begin, mid, mid, end, dest);
        } else {
            std::_insertion_sort(
                begin,
                end,
150         std::less<std::iterator_traits<RandomIterator>::value_type>());
            memcpy(dest,
                begin,
                (end - begin)*sizeof(std::iterator_traits<RandomIterator>::value_type));
        }
155     }

    void sort(long* begin, long* end)
    {
        sort(begin, end, tmp);
160     }
}

namespace three_way {
165     template<class RandomIterator, class OutputIterator>
        OutputIterator merge(RandomIterator b1,
            RandomIterator e1,
            RandomIterator b2,
            RandomIterator e2,
170         RandomIterator b3,
            RandomIterator e3,
            OutputIterator dest)
        {
175         RandomIterator dest0 = dest;

            if (e2[-1] < e1[-1]) {
                std::swap(b1, b2);
                std::swap(e1, e2);
180         }

            if (e3[-1] < e1[-1]) {
                std::swap(b1, b3);
                std::swap(e1, e3);
185         }

            std::iterator_traits<RandomIterator>
                ::value_type v1, v2, v3;
190         v1 = *b1;
            v2 = *b2;
            v3 = *b3;
            if (v2 <= v3) goto test_21;
            goto test_31;
195     out_2: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
            if (v2 > v3) goto test_31;
        test_21: /* invariant: v2 <= v3 */
            if (v2 < v1) goto out_2;
            b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
            if (b1 < e1) goto test_21; /* hint: branch taken */
            goto exit;
200     out_3: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
            if (v2 <= v3) goto test_21;
        test_31: /* invariant: v2 > v3 */
            if (v3 < v1) goto out_3;
205         b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
            if (b1 < e1) goto test_31; /* hint: branch taken */
        exit: /* final 2-way merge step */
            assert_sorted(dest0, dest);

```

```

return two_way::merge(b2, e2, b3, e3, dest);
210 }

template<class RandomIterator>
void sort(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
215     std::iterator_traits<RandomIterator>::difference_type n = end - begin;

    if (n > threshold) {
        std::iterator_traits<RandomIterator>::value_type third = n/3;
220         RandomIterator mid1 = begin + third;
        RandomIterator dest_mid1 = dest + third;

        RandomIterator mid2 = begin + 2*third;
225         RandomIterator dest_mid2 = dest + 2*third;

        sort_to(begin, mid1, dest);
        sort_to(mid1, mid2, dest_mid1);
        sort_to(mid2, end, dest_mid2);

230         merge(dest, dest_mid1, dest_mid1, dest_mid2, dest_mid2, dest+n, begin);
    } else {
        std::__insertion_sort(
            begin,
235             end,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
    }
}

template<class RandomIterator>
240 void sort_to(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
    std::iterator_traits<RandomIterator>::difference_type n = end - begin;

    if (n > threshold) {
245         std::iterator_traits<RandomIterator>::value_type third = n/3;

        RandomIterator mid1 = begin + third;
        RandomIterator dest_mid1 = dest + third;

250         RandomIterator mid2 = begin + 2*third;
        RandomIterator dest_mid2 = dest + 2*third;

        sort(begin, mid1, dest);
        sort(mid1, mid2, dest);
255         sort(mid2, end, dest);

        merge(begin, mid1, mid1, mid2, mid2, end, dest);
    } else {
260         std::__insertion_sort(
            begin,
            end,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
        memcpy(
265             dest,
            begin,
            (end - begin)*sizeof(std::iterator_traits<RandomIterator>::value_type));
    }
}

270 void sort(long* begin, long* end)
{
    sort(begin, end, tmp);
}

275 }

namespace four_way {

280 template<class RandomIterator, class OutputIterator>
    OutputIterator merge(RandomIterator b1,
        RandomIterator e1,
        RandomIterator b2,

```

```

285     RandomIterator e2,
        RandomIterator b3,
        RandomIterator e3,
        RandomIterator b4,
        RandomIterator e4,
        OutputIterator dest)
290     {
        OutputIterator dest0 = dest;

        assert_sorted(b1, e1);
        assert_sorted(b2, e2);
295     assert_sorted(b3, e3);
        assert_sorted(b4, e4);

        if (e2[-1] < e1[-1]) {
300             std::swap(b1, b2);
             std::swap(e1, e2);
        }

        if (e3[-1] < e1[-1]) {
305             std::swap(b1, b3);
             std::swap(e1, e3);
        }

        if (e4[-1] < e1[-1]) {
310             std::swap(b1, b4);
             std::swap(e1, e4);
        }

        std::iterator_traits<RandomIterator>
315             ::value_type v1, v2, v3, v4;
        v1 = *b1;
        v2 = *b2;
        v3 = *b3;
        v4 = *b4;
320     if (v2 <= v3 && v4 < v1) goto test_24;
        if (v2 <= v3 && v4 >= v1) goto test_21;
        if (v2 > v3 && v4 < v1) goto test_34;
        goto test_31;
    out_24: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
        if (v2 > v3) goto test_34;
325     test_24: /* invariant: v2 <= v3 && v4 < v1 */ assert( v2 <= v3 && v4 < v1 );
        if (v2 <= v4) goto out_24;
        b4 = b4 + 1; *dest = v4; dest = dest + 1; v4 = *b4;
        if (v4 < v1) goto test_24;
        /* invariant: v2 <= v3 && v4 >= v1 */ assert( v2 <= v3 && v4 >= v1 );
330     if (v2 >= v1) goto out_12;
    out_21: b2 = b2 + 1; *dest = v2; dest = dest + 1; v2 = *b2;
        if (v2 > v3) goto test_31;
    test_21: /* invariant: v2 <= v3 && v4 >= v1 */ assert( v2 <= v3 && v4 >= v1 );
        if (v2 < v1) goto out_21;
335     out_12: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
        if (b1 >= e1) goto exit; /* hint: not taken */
        if (v4 >= v1) goto test_21;
        goto test_24;
    out_34: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
340     if (v2 <= v3) goto test_24;
    test_34: /* invariant: v2 > v3 && v4 < v1 */ assert( v2 > v3 && v4 < v1 );
        if (v3 <= v4) goto out_34;
        b4 = b4 + 1; *dest = v4; dest = dest + 1; v4 = *b4;
        if (v4 < v1) goto test_34;
345     /* invariant: v2 > v3 && v4 >= v1 */ assert( v2 > v3 && v4 >= v1 );
        if (v3 >= v1) goto out_13;
    out_31: b3 = b3 + 1; *dest = v3; dest = dest + 1; v3 = *b3;
        if (v2 <= v3) goto test_21;
    test_31: /* invariant: v2 > v3 && v4 >= v1 */ assert( v2 > v3 && v4 >= v1 );
350     if (v3 < v1) goto out_31;
    out_13: b1 = b1 + 1; *dest = v1; dest = dest + 1; v1 = *b1;
        if (b1 >= e1) goto exit; /* hint: not taken */
        if (v4 >= v1) goto test_31;
        goto test_34;
355     exit: /* merge remaining three subarrays */
            assert_sorted(dest0, dest);

```

```

return three_way::merge(b2, e2, b3, e3, b4, e4, dest);
}
360
template<class RandomIterator>
void sort(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
365     std::iterator_traits<RandomIterator>::difference_type n = end - begin;
    if (n > threshold) {
        std::iterator_traits<RandomIterator>::value_type fourth = n/4;

370         RandomIterator mid1 = begin + 1*fourth;
        RandomIterator dest_mid1 = dest + 1*fourth;

        RandomIterator mid2 = begin + 2*fourth;
        RandomIterator dest_mid2 = dest + 2*fourth;

375         RandomIterator mid3 = begin + 3*fourth;
        RandomIterator dest_mid3 = dest + 3*fourth;

        sort_to(begin, mid1, dest);
        sort_to(mid1, mid2, dest_mid1);
380         sort_to(mid2, mid3, dest_mid2);
        sort_to(mid3, end, dest_mid3);

        merge(dest, dest_mid1,
              dest_mid1, dest_mid2,
385             dest_mid2, dest_mid3,
              dest_mid3, dest+n,
              begin);
    } else {
390         std::__insertion_sort(
            begin,
            end,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
    }
    assert_sorted(begin, end);
395 }

template<class RandomIterator>
void sort_to(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
400     std::iterator_traits<RandomIterator>::difference_type n = end - begin;
    if (n > threshold) {
        std::iterator_traits<RandomIterator>::value_type fourth = n/4;

405         RandomIterator mid1 = begin + 1*fourth;
        RandomIterator dest_mid1 = dest + 1*fourth;

        RandomIterator mid2 = begin + 2*fourth;
        RandomIterator dest_mid2 = dest + 2*fourth;

410         RandomIterator mid3 = begin + 3*fourth;
        RandomIterator dest_mid3 = dest + 3*fourth;

        sort(begin, mid1, dest);
        sort(mid1, mid2, dest);
415         sort(mid2, mid3, dest);
        sort(mid3, end, dest);

        merge(begin, mid1,
              mid1, mid2,
420             mid2, mid3,
              mid3, end,
              dest);
    } else {
425         std::__insertion_sort(
            begin,
            end,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
        memcpy(
430             dest,

```

```

        begin,
        (end - begin)*sizeof(std::iterator_traits<RandomIterator>::value_type));
    }
    assert_sorted(dest, dest+n);
435 }

void sort(long* begin, long* end)
{
440     sort(begin, end, tmp);
}

}

445 namespace back_to_back {

    template<class RandomIterator, class OutputIterator>
    OutputIterator merge(RandomIterator begin,
450                       RandomIterator end,
                       OutputIterator dest)
    {
        std::iterator_traits<RandomIterator>
455             ::value_type v1, v2, tmp;
        RandomIterator tmp0;

        end = end - 1;
        v1 = *begin;
        v2 = *end;
460 loop: tmp = v1 <= v2 ? v1 : v2;
        *dest = tmp;
        dest = dest + 1;
        tmp0 = begin + 1;
        begin = v1 <= v2 ? tmp0 : begin;
465 tmp0 = end - 1;
        end = v1 <= v2 ? end : tmp0;
        v1 = *begin;
        v2 = *end;
470 if (begin <= end) goto loop /* hint: branch taken */;
        return dest;
    }

    template<class RandomIterator, class OutputIterator>
    OutputIterator merge_rev(RandomIterator begin,
475                          RandomIterator end,
                          OutputIterator dest)
    {
        std::iterator_traits<RandomIterator>
480             ::value_type v1, v2, tmp;
        RandomIterator tmp0;

        end = end - 1;
        v1 = *begin;
        v2 = *end;
485 loop: tmp = v1 <= v2 ? v1 : v2;
        *dest = tmp;
        dest = dest - 1;
        tmp0 = begin + 1;
        begin = v1 <= v2 ? tmp0 : begin;
490 tmp0 = end - 1;
        end = v1 <= v2 ? end : tmp0;
        v1 = *begin;
        v2 = *end;
495 if (begin <= end) goto loop /* hint: branch taken */;
        return dest;
    }

    template<class RandomIterator>
    void sort_to(RandomIterator begin, RandomIterator end, RandomIterator dest)
    {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;
        if (n > threshold) {

```

```

505         RandomIterator mid = begin + n/2;
           RandomIterator dest_mid = dest + n/2;
           sort( begin, mid, dest );
           sort_rev( mid, end, dest_mid );

510     } else {
           merge( begin, end, dest );
           std::__insertion_sort(
515             begin,
           end,
           std::less<std::iterator_traits<RandomIterator>::value_type>());
           memcpy(
           dest,
           begin,
           (end - begin)*sizeof(std::iterator_traits<RandomIterator>::value_type));
520     }
           assert_sorted(dest, dest+n);
       }

template<class RandomIterator>
525 void sort(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
       std::iterator_traits<RandomIterator>::difference_type n = end - begin;

           if (n > threshold) {
530             RandomIterator mid = begin + n/2;
             RandomIterator dest_mid = dest + n/2;
             sort_to(begin, mid, dest );
             sort_to_rev( mid, end, dest_mid );

535             merge( dest, dest+n, begin );
           } else {
           std::__insertion_sort(
           begin,
540             end,
           std::less<std::iterator_traits<RandomIterator>::value_type>());
           }
           assert_sorted(begin, end);
       }

545 template<class RandomIterator>
void sort_to_rev(RandomIterator begin, RandomIterator end, RandomIterator dest)
{
       std::iterator_traits<RandomIterator>::difference_type n = end - begin;

550           if (n > threshold) {
           RandomIterator mid = begin + n/2;
           RandomIterator dest_mid = dest + n/2;
           sort( begin, mid, dest );
           sort_rev( mid, end, dest );

555             merge_rev( begin, end, dest+n-1 );
           } else {
           #ifndef STABLE_SORT
           std::__insertion_sort(
560             begin,
           end,
           std::greater<std::iterator_traits<RandomIterator>::value_type>());
           #else
           std::__insertion_sort(
565             begin,
           end,
           std::less<std::iterator_traits<RandomIterator>::value_type>());
           std::reverse(begin, end);
           #endif
570             memcpy(
           dest,
           begin,
           (end - begin)*sizeof(std::iterator_traits<RandomIterator>::value_type));
           }
575             assert_sorted_rev(dest, dest+n);
       }

template<class RandomIterator>

```

```

580     void sort_rev(RandomIterator begin, RandomIterator end, RandomIterator dest)
        {
            std::iterator_traits<RandomIterator>::difference_type n = end - begin;

            if (n > threshold) {
                RandomIterator mid = begin + n/2;
585                RandomIterator dest_mid = dest + n/2;
                sort_to( begin, mid, dest );
                sort_to_rev( mid, end, dest_mid );

                merge_rev( dest, dest+n, end-1 );
590            } else {
#ifdef STABLE_SORT
                std::__insertion_sort(
                    begin,
595                    end,
                    std::greater<std::iterator_traits<RandomIterator>::value_type>());
#else
                std::__insertion_sort(
                    begin,
600                    end,
                    std::less<std::iterator_traits<RandomIterator>::value_type>());
                std::reverse(begin, end);
#endif
            }
            assert_sorted_rev(begin, end);
605        }

        void sort(long* begin, long* end)
        {
            sort(begin, end, tmp);
610        }
    }

    namespace paired_back_to_back {
615        template<class RandomIterator, class OutputIterator>
        std::pair<OutputIterator, OutputIterator>
        merge(RandomIterator begin1,
              RandomIterator end1,
              RandomIterator begin2,
620              RandomIterator end2,
              OutputIterator dest1,
              OutputIterator dest2)
        {
            std::iterator_traits<RandomIterator>
                ::value_type v11, v12, v21, v22, tmp1, tmp2;
            RandomIterator tmp10, tmp20;

            end1 = end1 - 1;                end2 = end2 - 1;
            v11 = *begin1;                  v21 = *begin2;
            v12 = *end1;                    v22 = *end2;
630        loop: tmp1 = v11 <= v12 ? v11 : v12;    tmp2 = v21 <= v22 ? v21 : v22;
            *dest1 = tmp1;                    *dest2 = tmp2;
            dest1 = dest1 + 1;                dest2 = dest2 + 1;
            tmp10 = begin1 + 1;                tmp20 = begin2 + 1;
635            begin1 = v11 <= v12 ? tmp10 : begin1; begin2 = v21 <= v22 ? tmp20 : begin2;
            tmp10 = end1 - 1;                tmp20 = end2 - 1;
            end1 = v11 <= v12 ? end1 : tmp10;    end2 = v21 <= v22 ? end2 : tmp20;
            v11 = *begin1;                    v21 = *begin2;
            v12 = *end1;                      v22 = *end2;
640            if (begin1 <= end1) goto loop; /* hint: branch taken */
            return std::make_pair(dest1, dest2);
        }

        template<class RandomIterator, class OutputIterator>
645        std::pair<OutputIterator, OutputIterator>
        merge_rev(RandomIterator begin1,
                 RandomIterator end1,
                 RandomIterator begin2,
                 RandomIterator end2,
                 OutputIterator dest1,
650                 OutputIterator dest2)
        {

```

```

        std::iterator_traits<RandomIterator>
            ::value_type v11, v12, v21, v22, tmp1, tmp2;
655     RandomIterator tmp10, tmp20;

        end1 = end1 - 1;                end2 = end2 - 1;
        v11 = *begin1;                  v21 = *begin2;
        v12 = *end1;                    v22 = *end2;
660     tmp1;                             tmp2;
        tmp10;                          tmp20;
loop:   tmp1 = v11 <= v12 ? v11 : v12;    tmp2 = v21 <= v22 ? v21 : v22;
        *dest1 = tmp1;                  *dest2 = tmp2;
        dest1 = dest1 - 1;              dest2 = dest2 - 1;
665     tmp10 = begin1 + 1;              tmp20 = begin2 + 1;
        begin1 = v11 <= v12 ? tmp10 : begin1; begin2 = v21 <= v22 ? tmp20 : begin2;
        tmp10 = end1 - 1;               tmp20 = end2 - 1;
        end1 = v11 <= v12 ? end1 : tmp10; end2 = v21 <= v22 ? end2 : tmp20;
670     v11 = *begin1;                  v21 = *begin2;
        v12 = *end1;                    v22 = *end2;
        if (begin1 <= end1) goto loop; /* hint: branch taken */
        return std::make_pair(dest1, dest2);
    }

675     template<class RandomIterator, class OutputIterator>
        std::pair<OutputIterator, OutputIterator>
        merge_both(RandomIterator begin1,
680                 RandomIterator end1,
                 RandomIterator begin2,
                 RandomIterator end2,
                 OutputIterator dest1,
                 OutputIterator dest2)
    {
        std::iterator_traits<RandomIterator>
            ::value_type v11, v12, v21, v22, tmp1, tmp2;
685     RandomIterator tmp10, tmp20;

        end1 = end1 - 1;                end2 = end2 - 1;
        v11 = *begin1;                  v21 = *begin2;
        v12 = *end1;                    v22 = *end2;
690     tmp1;                             tmp2;
        tmp10;                          tmp20;
loop:   tmp1 = v11 <= v12 ? v11 : v12;    tmp2 = v21 <= v22 ? v21 : v22;
        *dest1 = tmp1;                  *dest2 = tmp2;
695     dest1 = dest1 + 1;                dest2 = dest2 + 1;
        tmp10 = begin1 + 1;              tmp20 = begin2 + 1;
        begin1 = v11 <= v12 ? tmp10 : begin1; begin2 = v21 <= v22 ? tmp20 : begin2;
        tmp10 = end1 - 1;               tmp20 = end2 - 1;
700     end1 = v11 <= v12 ? end1 : tmp10; end2 = v21 <= v22 ? end2 : tmp20;
        v11 = *begin1;                  v21 = *begin2;
        v12 = *end1;                    v22 = *end2;
        if (begin1 <= end1) goto loop; /* hint: branch taken */
        return std::make_pair(dest1, dest2);
    }

705     template<class RandomIterator>
        void sort_to(RandomIterator begin,
                    RandomIterator end,
                    RandomIterator dest,
710                    std::iterator_traits<RandomIterator>::difference_type offset )
    {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;

        if (n > threshold) {
715             RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;
            sort( begin, mid, dest, offset );
            sort_rev( mid, end, dest, offset );

720             merge( begin, end, begin+offset, end+offset, dest, dest+offset );
        } else {
            std::__insertion_sort(
                begin,
                end,
725                 std::less<std::iterator_traits<RandomIterator>::value_type>());
            memcpy(

```

```

        dest,
        begin,
        n*sizeof(std::iterator_traits<RandomIterator>::value_type));
730     std::__inversion_sort(
        begin+offset,
        end+offset,
        std::less<std::iterator_traits<RandomIterator>::value_type>());
735     memcpy(
        dest+offset,
        begin+offset,
        n*sizeof(std::iterator_traits<RandomIterator>::value_type));
    }
    assert_sorted(dest, dest+n);
740     assert_sorted(dest+offset, dest+n+offset);
}

template<class RandomIterator>
void sort(RandomIterator begin,
745         RandomIterator end,
        RandomIterator dest,
        std::iterator_traits<RandomIterator>::difference_type offset )
{
    std::iterator_traits<RandomIterator>::difference_type n = end - begin;
750
    if (n > threshold) {
        RandomIterator mid = begin + n/2;
        RandomIterator dest_mid = dest + n/2;
        sort_to( begin, mid, dest, offset );
755         sort_to_rev( mid, end, dest_mid, offset );

        merge( dest, dest+n, dest+offset, dest+n+offset, begin, begin+offset );
    } else {
760         std::__inversion_sort(
            begin,
            end,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
        std::__inversion_sort(
765             begin+offset,
            end+offset,
            std::less<std::iterator_traits<RandomIterator>::value_type>());
    }
    assert_sorted(begin, end);
770     assert_sorted(begin+offset, end+offset);
}

template<class RandomIterator>
void sort_to_rev(RandomIterator begin,
775                 RandomIterator end,
                RandomIterator dest,
                std::iterator_traits<RandomIterator>::difference_type offset )
{
    std::iterator_traits<RandomIterator>::difference_type n = end - begin;
780
    if (n > threshold) {
        RandomIterator mid = begin + n/2;
        RandomIterator dest_mid = dest + n/2;
        sort( begin, mid, dest, offset );
785         sort_rev( mid, end, dest, offset );

        merge_rev( begin, end,
                   begin+offset, end+offset,
                   dest+n-1, dest+n-1+offset );
    } else {
790         std::__inversion_sort(
            begin,
            end,
            std::greater<std::iterator_traits<RandomIterator>::value_type>());
795         memcpy(
            dest,
            begin,
            n*sizeof(std::iterator_traits<RandomIterator>::value_type));
        std::__inversion_sort(
800             begin+offset,
            end+offset,

```

```

        std::greater<std::iterator_traits<RandomIterator>::value_type>());
        memcpy(
            dest+offset,
            begin+offset,
            n*sizeof(std::iterator_traits<RandomIterator>::value_type));
805     }
        assert_sorted_rev(dest, dest+n);
        assert_sorted_rev(dest+offset, dest+n+offset);
810     }
template<class RandomIterator>
void sort_rev(RandomIterator begin,
             RandomIterator end,
             RandomIterator dest,
             std::iterator_traits<RandomIterator>::difference_type offset )
815 {
    std::iterator_traits<RandomIterator>::difference_type n = end - begin;

    if (n > threshold) {
820         RandomIterator mid = begin + n/2;
        RandomIterator dest_mid = dest + n/2;
        sort_to( begin, mid, dest, offset );
        sort_to_rev( mid, end, dest_mid, offset );

825         merge_rev( dest, dest+n, dest+offset, dest+n+offset, end-1, end-1+offset );
    } else {
        std::__insertion_sort(
            begin,
            end,
830             std::greater<std::iterator_traits<RandomIterator>::value_type>());
        std::__insertion_sort(
            begin+offset,
            end+offset,
            std::greater<std::iterator_traits<RandomIterator>::value_type>());
835     }
        assert_sorted_rev(begin, end);
        assert_sorted_rev(begin+offset, end+offset);
    }
840 void sort(long* begin, long* end)
    {
        size_t n = end - begin;
        size_t offset = n / 2;
        size_t half = offset / 2;
845         long* b1 = begin;
        long* e1 = begin + half;
        long* b2 = begin + half;
        long* e2 = begin + offset;

850         sort(b1, e1, tmp, offset);
        sort_rev(b2, e2, tmp, offset);

        merge_both(b1, e2, b1+offset, e2+offset, tmp, tmp+n-1);
855         //         assert_sorted(tmp, tmp+half);
        //         assert_sorted(tmp+half, tmp+half);
        //         ::back_to_back::merge(tmp, tmp+n, begin);
860         assert_sorted(begin, end);
    }
}

865 namespace tuned_paired_back_to_back
    {
        template<class RandomIterator>
        void merge( RandomIterator begin1,
870                 RandomIterator end1,
                 RandomIterator dest1,
                 size_t offset )
        {
            RandomIterator begin2 = begin1 + offset;

```

```

875     RandomIterator end2 = end1 + offset;
        RandomIterator dest2 = dest1 + offset;

        end1--;
        end2--;
880     while (begin1 + 4 < end1) {
            std::iterator_traits<RandomIterator>::value_type u1, u2;
            std::iterator_traits<RandomIterator>::value_type v1, v2;

            u1 = *begin1;
            v1 = *end1;
885     dest1[0] = u1 <= v1 ? u1 : v1;
            begin1 = u1 <= v1 ? begin1+1 : begin1;
            end1 = u1 <= v1 ? end1 : end1-1;

890     u2 = *begin2;
            v2 = *end2;
            dest2[0] = u2 <= v2 ? u2 : v2;
            begin2 = u2 <= v2 ? begin2+1 : begin2;
            end2 = u2 <= v2 ? end2 : end2-1;

895     u1 = *begin1;
            v1 = *end1;
            dest1[1] = u1 <= v1 ? u1 : v1;
            begin1 = u1 <= v1 ? begin1+1 : begin1;
900     end1 = u1 <= v1 ? end1 : end1-1;

            u2 = *begin2;
            v2 = *end2;
            dest2[1] = u2 <= v2 ? u2 : v2;
905     begin2 = u2 <= v2 ? begin2+1 : begin2;
            end2 = u2 <= v2 ? end2 : end2-1;

            u1 = *begin1;
            v1 = *end1;
            dest1[2] = u1 <= v1 ? u1 : v1;
            begin1 = u1 <= v1 ? begin1+1 : begin1;
910     end1 = u1 <= v1 ? end1 : end1-1;

            u2 = *begin2;
            v2 = *end2;
            dest2[2] = u2 <= v2 ? u2 : v2;
            begin2 = u2 <= v2 ? begin2+1 : begin2;
915     end2 = u2 <= v2 ? end2 : end2-1;

            u1 = *begin1;
            v1 = *end1;
            dest1[3] = u1 <= v1 ? u1 : v1;
            begin1 = u1 <= v1 ? begin1+1 : begin1;
920     end1 = u1 <= v1 ? end1 : end1-1;

            u2 = *begin2;
            v2 = *end2;
            dest2[3] = u2 <= v2 ? u2 : v2;
            begin2 = u2 <= v2 ? begin2+1 : begin2;
925     end2 = u2 <= v2 ? end2 : end2-1;

            dest1 += 4;
            dest2 += 4;
930     }

935     while (begin1 <= end1) {
            std::iterator_traits<RandomIterator>::value_type u1, u2;
            std::iterator_traits<RandomIterator>::value_type v1, v2;

940     u1 = *begin1;
            v1 = *end1;
            dest1[0] = u1 <= v1 ? u1 : v1;
            begin1 = u1 <= v1 ? begin1+1 : begin1;
            end1 = u1 <= v1 ? end1 : end1-1;

945     u2 = *begin2;
            v2 = *end2;
            dest2[0] = u2 <= v2 ? u2 : v2;

```

```

begin2 = u2 <= v2 ? begin2+1 : begin2;
end2 = u2 <= v2 ? end2 : end2-1;

dest1++;
dest2++;
}
955 }

template<typename RandomIterator>
void merge_both( RandomIterator begin1,
960             RandomIterator end1,
             RandomIterator begin2,
             RandomIterator end2,
             RandomIterator dest1,
             RandomIterator dest2 )
965 {
    end1--;
    end2--;
    while (begin1 + 4 < end1) {
970         std::iterator_traits<RandomIterator>::value_type u1, u2;
         std::iterator_traits<RandomIterator>::value_type v1, v2;

         u1 = *begin1;
         v1 = *end1;
         dest1[0] = u1 <= v1 ? u1 : v1;
975         begin1 = u1 <= v1 ? begin1+1 : begin1;
         end1 = u1 <= v1 ? end1 : end1-1;

         u2 = *begin2;
         v2 = *end2;
         dest2[0] = u2 < v2 ? u2 : v2;
980         begin2 = u2 < v2 ? begin2+1 : begin2;
         end2 = u2 < v2 ? end2 : end2-1;

         u1 = *begin1;
         v1 = *end1;
         dest1[1] = u1 <= v1 ? u1 : v1;
985         begin1 = u1 <= v1 ? begin1+1 : begin1;
         end1 = u1 <= v1 ? end1 : end1-1;

         u2 = *begin2;
         v2 = *end2;
         dest2[-1] = u2 < v2 ? u2 : v2;
990         begin2 = u2 < v2 ? begin2+1 : begin2;
         end2 = u2 < v2 ? end2 : end2-1;

         u1 = *begin1;
         v1 = *end1;
         dest1[2] = u1 <= v1 ? u1 : v1;
995         begin1 = u1 <= v1 ? begin1+1 : begin1;
         end1 = u1 <= v1 ? end1 : end1-1;
1000         u2 = *begin2;
         v2 = *end2;
         dest2[-2] = u2 < v2 ? u2 : v2;
1005         begin2 = u2 < v2 ? begin2+1 : begin2;
         end2 = u2 < v2 ? end2 : end2-1;

         u1 = *begin1;
         v1 = *end1;
         dest1[3] = u1 <= v1 ? u1 : v1;
1010         begin1 = u1 <= v1 ? begin1+1 : begin1;
         end1 = u1 <= v1 ? end1 : end1-1;

         u2 = *begin2;
         v2 = *end2;
         dest2[-3] = u2 < v2 ? u2 : v2;
1015         begin2 = u2 < v2 ? begin2+1 : begin2;
         end2 = u2 < v2 ? end2 : end2-1;

         dest1 += 4;
1020         dest2 -= 4;
    }
    while (begin1 <= end1) {

```

```

1025     std::iterator_traits<RandomIterator>::value_type u1, u2;
        std::iterator_traits<RandomIterator>::value_type v1, v2;

        u1 = *begin1;
        v1 = *end1;
        dest1[0] = u1 <= v1 ? u1 : v1;
1030     begin1 = u1 <= v1 ? begin1+1 : begin1;
        end1 = u1 <= v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
        dest2[0] = u2 <= v2 ? u2 : v2;
1035     begin2 = u2 <= v2 ? begin2+1 : begin2;
        end2 = u2 <= v2 ? end2 : end2-1;

        dest1++;
        dest2--;
1040     }
    }

template<typename RandomIterator>
1045     void merge_rev( RandomIterator begin1,
                    RandomIterator end1,
                    RandomIterator dest1,
                    size_t offset )
    {
1050     RandomIterator begin2 = begin1 + offset;
        RandomIterator end2 = end1 + offset;
        RandomIterator dest2 = dest1 + offset;
        end1--;
        end2--;
1055     while (begin1 + 4 < end1) {
        std::iterator_traits<RandomIterator>::value_type u1, u2;
        std::iterator_traits<RandomIterator>::value_type v1, v2;

        u1 = *begin1;
        v1 = *end1;
1060     dest1[0] = u1 < v1 ? u1 : v1;
        begin1 = u1 < v1 ? begin1+1 : begin1;
        end1 = u1 < v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
1065     dest2[0] = u2 < v2 ? u2 : v2;
        begin2 = u2 < v2 ? begin2+1 : begin2;
        end2 = u2 < v2 ? end2 : end2-1;

        u1 = *begin1;
        v1 = *end1;
1070     dest1[-1] = u1 < v1 ? u1 : v1;
        begin1 = u1 < v1 ? begin1+1 : begin1;
        end1 = u1 < v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
1075     dest2[-1] = u2 < v2 ? u2 : v2;
        begin2 = u2 < v2 ? begin2+1 : begin2;
        end2 = u2 < v2 ? end2 : end2-1;

        u1 = *begin1;
        v1 = *end1;
1080     dest1[-2] = u1 < v1 ? u1 : v1;
        begin1 = u1 < v1 ? begin1+1 : begin1;
        end1 = u1 < v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
1085     dest2[-2] = u2 < v2 ? u2 : v2;
        begin2 = u2 < v2 ? begin2+1 : begin2;
        end2 = u2 < v2 ? end2 : end2-1;

        u1 = *begin1;
        v1 = *end1;
1090     dest1[-3] = u1 < v1 ? u1 : v1;
        begin1 = u1 < v1 ? begin1+1 : begin1;
        end1 = u1 < v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
1095     dest2[-3] = u2 < v2 ? u2 : v2;
        begin2 = u2 < v2 ? begin2+1 : begin2;
        end2 = u2 < v2 ? end2 : end2-1;
    }
}

```

```

begin1 = u1 < v1 ? begin1+1 : begin1;
end1 = u1 < v1 ? end1 : end1-1;

1100     u2 = *begin2;
        v2 = *end2;
        dest2[-3] = u2 < v2 ? u2 : v2;
        begin2 = u2 < v2 ? begin2+1 : begin2;
        end2 = u2 < v2 ? end2 : end2-1;

1105     dest1 -= 4;
        dest2 -= 4;
    }
    while (begin1 <= end1) {
1110         std::iterator_traits<RandomIterator>::value_type u1, u2;
        std::iterator_traits<RandomIterator>::value_type v1, v2;

        u1 = *begin1;
        v1 = *end1;
1115         dest1[0] = u1 <= v1 ? u1 : v1;
        begin1 = u1 <= v1 ? begin1+1 : begin1;
        end1 = u1 <= v1 ? end1 : end1-1;

        u2 = *begin2;
        v2 = *end2;
1120         dest2[0] = u2 <= v2 ? u2 : v2;
        begin2 = u2 <= v2 ? begin2+1 : begin2;
        end2 = u2 <= v2 ? end2 : end2-1;

1125         dest1--;
        dest2--;
    }
}

1130     template<class RandomIterator>
    void sort_to(RandomIterator begin,
                RandomIterator end,
                RandomIterator dest,
                std::iterator_traits<RandomIterator>::difference_type offset )
1135     {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;

        if (n > threshold) {
1140             RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;
            sort( begin, mid, dest, offset );
            sort_rev( mid, end, dest, offset );

            merge( begin, end, dest, offset );
1145         } else {
            std::__insertion_sort(
                begin,
                end,
                std::less<std::iterator_traits<RandomIterator>::value_type>());
1150             memcpy(
                dest,
                begin,
                n*sizeof(std::iterator_traits<RandomIterator>::value_type));
            std::__insertion_sort(
1155                 begin+offset,
                end+offset,
                std::less<std::iterator_traits<RandomIterator>::value_type>());
            memcpy(
1160                 dest+offset,
                begin+offset,
                n*sizeof(std::iterator_traits<RandomIterator>::value_type));
        }
        assert_sorted(dest, dest+n);
        assert_sorted(dest+offset, dest+n+offset);
1165     }

    template<class RandomIterator>
    void sort(RandomIterator begin,
              RandomIterator end,
1170              RandomIterator dest,

```

```

        std::iterator_traits<RandomIterator>::difference_type offset )
    {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;
1175     if (n > threshold) {
            RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;
            sort_to( begin, mid, dest, offset );
            sort_to_rev( mid, end, dest_mid, offset );
1180     } else {
            merge( dest, dest+n, begin, offset );
        } else {
            std::__inversion_sort(
1185                 begin,
                    end,
                    std::less<std::iterator_traits<RandomIterator>::value_type>());
            std::__inversion_sort(
                begin+offset,
                end+offset,
1190                 std::less<std::iterator_traits<RandomIterator>::value_type>());
        }
        assert_sorted(begin, end);
        assert_sorted(begin+offset, end+offset);
1195     }

    template<class RandomIterator>
    void sort_to_rev(RandomIterator begin,
                   RandomIterator end,
                   RandomIterator dest,
1200                   std::iterator_traits<RandomIterator>::difference_type offset )
    {
        std::iterator_traits<RandomIterator>::difference_type n = end - begin;

1205     if (n > threshold) {
            RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;
            sort( begin, mid, dest, offset );
            sort_rev( mid, end, dest, offset );

1210     } else {
            merge_rev( begin, end, dest+n-1, offset );
        } else {
            std::__inversion_sort(
1215                 begin,
                    end,
                    std::greater<std::iterator_traits<RandomIterator>::value_type>());
            memcpy(
                dest,
                begin,
                n*sizeof(std::iterator_traits<RandomIterator>::value_type));
1220            std::__inversion_sort(
                begin+offset,
                end+offset,
                std::greater<std::iterator_traits<RandomIterator>::value_type>());
            memcpy(
1225                 dest+offset,
                    begin+offset,
                    n*sizeof(std::iterator_traits<RandomIterator>::value_type));
        }
        assert_sorted_rev(dest, dest+n);
        assert_sorted_rev(dest+offset, dest+n+offset);
1230     }

    template<class RandomIterator>
    void sort_rev(RandomIterator begin,
1235                   RandomIterator end,
                   RandomIterator dest,
                   std::iterator_traits<RandomIterator>::difference_type offset )
    {
1240     std::iterator_traits<RandomIterator>::difference_type n = end - begin;

        if (n > threshold) {
            RandomIterator mid = begin + n/2;
            RandomIterator dest_mid = dest + n/2;
            sort_to( begin, mid, dest, offset );

```

```

1245         sort_to_rev( mid, end, dest_mid, offset );
           merge_rev( dest, dest+n, end-1, offset );
       } else {
1250         std::__insertion_sort(
           begin,
           end,
           std::greater<std::iterator_traits<RandomIterator>::value_type>());
1255         std::__insertion_sort(
           begin+offset,
           end+offset,
           std::greater<std::iterator_traits<RandomIterator>::value_type>());
       }
       assert_sorted_rev(begin, end);
1260       assert_sorted_rev(begin+offset, end+offset);
   }

   void sort(long* begin, long* end)
   {
1265       size_t n = end - begin;
       size_t offset = n / 2;
       size_t half = offset / 2;

       long* b1 = begin;
1270       long* e1 = begin + half;
       long* b2 = begin + half;
       long* e2 = begin + offset;

       sort(b1, e1, tmp, offset);
1275       sort_rev(b2, e2, tmp, offset);

       merge_both(b1, e2, b1+offset, e2+offset, tmp, tmp+n-1);

       //       assert_sorted(tmp, tmp+half);
1280       //       assert_sorted(tmp+half, tmp+half);
       //       ::back_to_back::merge(tmp, tmp+n, begin);

       assert_sorted(begin, end);
   }

1285 }

namespace hybrid {

1290     using kt::four_way::merge;

     template<class RandomIterator>
     void sort(RandomIterator begin,
1295         RandomIterator end,
         RandomIterator dest)
     {
         std::iterator_traits<RandomIterator>::difference_type n = end - begin;

1300         if (n > cache_threshold) {
             std::iterator_traits<RandomIterator>::value_type fourth = n/4;

             RandomIterator mid1 = begin + 1*fourth;
             RandomIterator dest_mid1 = dest + 1*fourth;

1305             RandomIterator mid2 = begin + 2*fourth;
             RandomIterator dest_mid2 = dest + 2*fourth;

             RandomIterator mid3 = begin + 3*fourth;
1310             RandomIterator dest_mid3 = dest + 3*fourth;

             sort_to(begin, mid1, dest);
             sort_to(mid1, mid2, dest_mid1);
             sort_to(mid2, mid3, dest_mid2);
             sort_to(mid3, end, dest_mid3);

1315             merge(dest, dest_mid1,
                 dest_mid1, dest_mid2,
                 dest_mid2, dest_mid3,

```

```

1320         dest_mid3, dest+n,
           begin);
           } else {
           tuned_paired_back_to_back::sort(begin, end);
           }
1325     }
           assert_sorted(begin, end);

           template<class RandomIterator>
           void sort_to(RandomIterator begin,
1330                 RandomIterator end,
                   RandomIterator dest)
           {
           std::iterator_traits<RandomIterator>::difference_type n = end - begin;

1335           {
                   std::iterator_traits<RandomIterator>::value_type fourth = n/4;

                   RandomIterator mid1 = begin + 1*fourth;
                   RandomIterator dest_mid1 = dest + 1*fourth;

1340                   RandomIterator mid2 = begin + 2*fourth;
                   RandomIterator dest_mid2 = dest + 2*fourth;

                   RandomIterator mid3 = begin + 3*fourth;
                   RandomIterator dest_mid3 = dest + 3*fourth;

1345                   sort(begin, mid1, dest);
                   sort(mid1, mid2, dest);
                   sort(mid2, mid3, dest);
                   sort(mid3, end, dest);

1350                   merge(begin, mid1,
                           mid1, mid2,
                           mid2, mid3,
                           mid3, end,
1355                           dest);

                   }
                   assert_sorted(dest, dest+n);
           }

1360     void sort(long* begin, long* end)
           {
           sort(begin, end, tmp);
           }

1365 }

           void initdata(size_t n) {
           for (size_t i=0; i<n; ++i) data[i] = rand();
1370 }

           void main_sort()
           {
1375     FILE* file = fopen("merge.dat", "w");

           data = new long[data_max];
           tmp = new long[data_max];

           typedef void(*LPSORT)(long*, long*);
1380     LPSORT sort[] = { std::sort,
                       kt::two_way::sort, kt::three_way::sort, kt::four_way::sort,
                       back_to_back::sort, paired_back_to_back::sort,
                       tuned_paired_back_to_back::sort,
                       hybrid::sort };

1385     const size_t number_of_functions = sizeof(sort)/sizeof(LPSORT);

           const double prog_factor = 1.1892071150027210667174999705605;
           for (double fsz = 256; fsz <= data_max; fsz *= prog_factor)
1390     {
           int sz = floor(fsz);
           if (sz&1) sz++;

```

```

1395     printf("%10d", sz);
        fprintf(file, "%10d", sz);
        for (int f_idx = 0; f_idx < number_of_functions; ++f_idx)
        {
            float time[NUMBER_OF_RUNS];

1400             for (int i = 0; i < NUMBER_OF_RUNS; ++i) {
                __int64 elapsed_time;
                initdata(sz);

                microtime::start();
                sort[f_idx](data, data+sz);
1405                 microtime::stop(elapsed_time);
                time[i] = float(elapsed_time);

                assert_sorted(data, data+sz);

1410             }
            std::sort(time, time+NUMBER_OF_RUNS);
            printf("\t%.1f",
                time[NUMBER_OF_RUNS/2]/(sz*log(sz)/log(2)));
1415             fprintf(file,
                "\t%.1f", time[NUMBER_OF_RUNS/2]/(sz*log(sz)/log(2)));
        }
        printf("\n");
        fprintf(file, "\n");
1420     }
        fclose(file);
    }

1425 void main()
    {
        #ifdef NDEBUG
            srand( (unsigned)time( NULL ) );
        #endif

1430         microtime::calibrate();

            main_sort();
    }

```

A.3 limit_data

```

1  #define __STL_NO_SGI_IOSTREAMS

    #include <ls/microtime.h>
    #include <algorithm>
5  #include <cmath>

    typedef unsigned long ELEMENT_TYPE;

10     namespace normal_version {
        template<typename IT>
        void limit_array_1(IT begin, IT end)
        {
15             std::iterator_traits<IT>::value_type v;
                goto test;

        loop:
            v = *begin;
            if (v <= 100) goto skip_if;
            *begin = 100;
20         skip_if:
            begin = begin + 1;

        test:
            if (begin < end) goto loop;
25     }

```

```

// We need to code the "normal version" in assembly
// The compiler was smart enough to use conditional moves for limit_array_1
void limit_array(ELEMENT_TYPE * restrict begin, ELEMENT_TYPE * restrict end)
30 {
    __asm {
        mov eax, begin           ;
        mov edx, end             ;
        jmp loop_begin           ;
35
    loop:
        cmp DWORD PTR [eax], 100 ;
        jbe skip_if              ;
        mov DWORD PTR [eax], 100 ;
40 skip_if:
        add eax, 4                ;
    loop_begin:
        cmp eax, edx              ;
        jb loop                   ;
45     }
}

50 namespace ca_version {

    template<typename IT> void limit_array(IT begin, IT end)
    {
55         std::iterator_traits<IT>::value_type v;
        goto test;
    loop:
        v = *begin;
        v = v > 100 ? 100 : v;
60         *begin = v;
        begin = begin + 1;
    test:
        if (begin < end) goto loop;
65     }
}

namespace cpp_version {

70     template<typename IT> void limit_array(IT begin, IT end)
    {
        size_t len = std::distance(begin, end);
        for (size_t i = 0; i < len; ++i)
        {
75             if (begin[i] > 100) begin[i] = 100;
        }
    }

    const size_t data_size = 1024;
80     const size_t NUMBER_OF_RUNS = 16*4096;

    void initdata(ELEMENT_TYPE* data, size_t n, size_t percentage) {
        for (size_t i=0; i<n; ++i) data[i] = (rand() % 100) + percentage;
    }
85     void main_test()
    {
        FILE* file = fopen("test_ca.dat", "w");

90         ELEMENT_TYPE* data = new ELEMENT_TYPE[data_size];

        typedef void(*LPFUNC)(ELEMENT_TYPE*, ELEMENT_TYPE*);
        LPFUNC func[] = { normal_version::limit_array, ca_version::limit_array };

95         const size_t number_of_functions = sizeof(func)/sizeof(LPFUNC);

        for (size_t percentage = 0; percentage <= 100; percentage += 5)
        {
            printf("%10d", percentage);

```



```

30      __asm {
          pushad                ;
          CPUID                 ;
          RDTSC                 ;
35      mov     cycles_high, edx ;
          mov     cycles_low, eax ;
          popad                ;
          pushad                ;
          CPUID                 ;
40      RDTSC                 ;
          popad                ;
          pushad                ;
          CPUID                 ;
45      RDTSC                 ;
          mov     cycles_high, edx ;
          mov     cycles_low, eax ;
          popad                ;
          pushad                ;
50      CPUID                 ;
          RDTSC                 ;
          popad                ;
          pushad                ;
55      CPUID                 ;
          RDTSC                 ;
          mov     cycles_high, edx ;
          mov     cycles_low, eax ;
60      popad                ;
          pushad                ;
          CPUID                 ;
          RDTSC                 ;
65      sub     eax, cycles_low ;
          mov     base_extra, eax ;
          popad                ;
          pushad                ;
70      CPUID                 ;
          RDTSC                 ;
          mov     cycles_high, edx ;
          mov     cycles_low, eax ;
          popad                ;
75      pushad                ;
          CPUID                 ;
          RDTSC                 ;
80      sub     eax, cycles_low ;
          mov     base, eax ;
          popad                ;
          } // End inline assembly
85      // The following provides insurance for the above code,
          // in the instance the final test causes a miss to the
          // instruction cache.
          if (base_extra < base)
90          base = base_extra;
          base += 1;
          base_ = base;
          return base;
95      }
      inline void start()
      {
100     __asm {
          pushad
          CPUID
;

```

