A Portable Cache Profiler
Based on Source-Level Instrumentation

Noboru Obata
<obatan@cs.rpi.edu>
<noboru@ylug.org>

David R. Musser
<musser@cs.rpi.edu>

April 25, 2003

Contents

Abstract

The significance of cache-conscious algorithms has gradually become familiar to computer scientists. However, cache-conscious algorithms are not as widely studied as they ought to be, given that the processor-memory performance gap is steadily growing. The primary roadblock is difficulty in cache profiling. Various cache profiling methods and tools have been developed, but there has been no simple and portable solution to the study of basic cache performance. Many tools require a large amount of work to get started with them, and most are bound to a specific hardware architecture and operating system. This paper describes SMAT (STL Memory Access Tracer), a portable and easy-to-use cache profiling tool in the form of a trace-driven cache simulator, with traces collected by the source-level instrumentation. The trace-generating function is implemented as an iterator adaptor compatible with the C++ Standard Template Library (STL), so it requires little or no modification of the algorithm to be profiled. The difficulty in source-based trace generation is that it traces all objects, while object-based instrumentation, which is widely used, traces only objects which are not assigned to registers. In addition, source-level instrumentation potentially disables several important compile-time optimizations, such as copy propagation. How SMAT overcomes these complications and provides convincing cache profiling is a principal focus of this paper. Also detailed are applications of SMAT to profile STL sorting algorithms, including new cache-conscious versions of introsort and mergesort.

Chapter 1

Introduction

1.1 Caches

The performance of microprocessors is growing much faster than that of memory systems. To bridge this so-called processor-memory performance gap, most recently developed microprocessors have a *cache memory*, a small and fast memory located close to CPU.

The CPU first consults the cache upon each memory access request. If the requested data is found in the cache (a cache *hit*), the request is satisfied much faster than making a request to main memory. If the cache does not have the requested data, it must be fetched from main memory (a cache *miss*), but the data obtained is then stored in the cache, with the assumption that it is likely to be needed again soon. Caches achieve a significant speedup because this assumption is satisfied in many kinds of computations, especially those specificially tailored to take advantage of the cache—so-called "cache-conscious" algorithms.

A cache is conceptually an array of *blocks*. A block is the basic unit of access to main memory. The typical size of a block in most microprocessors today is 32 or 64 bytes, while the cache size ranges from a few kilobytes to several megabytes. Both the cache and block sizes are usually restricted to powers of two for efficiency. Which block the data should be stored is determined by the address of the data.

For example, suppose there is an 8KB cache with 32-byte blocks. The cache consists of 256 blocks in all ($2^{13}/2^5 = 2^8$). Then a 32-bit address is decomposed into three parts: a 19-bit *tag*, an 8-bit *index*, and a 5-bit *offset* (from the most-significant to the least-sigificant bit). The index is used to identify one of the 256 blocks. The offset locates the data in a block since a block always starts from the aligned addresses. That is, the contents of 32-byte blocks always start on a 32-byte boundary. In the simplest type of caches, the index part directly indicates the block where the data should go. This type of cache is called *direct mapped* since an address is mapped to only one block directly by its index part.

A more sophisticated type of address-block mapping is *set associative* mapping, in which an address can be mapped to a *set* of blocks. Again, the number of blocks in a set is usually a power of two. In the previous example, suppose the cache has sets of four blocks. Then the cache has 64 sets and so the upper 6 bits of the 8-bit index can be used to identify the set. This cache is called 4-way set associative because a set has four blocks and an address can be mapped to any one of them.

Since a cache is typically much smaller than main memory, it can hold only a small subset of main memory at any given time. Therefore, given a read access, it is highly likely that its corresponding block or set is already full. With a direct mapped cache, the old entry is simply replaced with the new entry. On the other hand, a choice must be made in a set associative cache which entry to replace. Several block replacement algorithms are used to choose the entry to discard: random, LRU (least-recently used), and FIFO (first in, first out).

Write accesses to main memory can be cached as well. There are two major strategies to handle write accesses, *write through* and *write back*. In a write through cache, a datum is written to both a block and main memory. In a write back cache, a datum is written to a block only, and thus the modified cache block must be written back to main memory when it is replaced. If there is no corresponding block in a cache at a write access, the block is allocated, possibly by replacement, and then the block is *filled* so that it reflects the main memory.

Each block should know which address it contains. The tag part of the address is used for this purpose, and the tag is stored to a dedicated area associated with a block. Reference time, or entry time, might be stored as well for a set associative cache. The write back cache has a bit called the *dirty bit* that tells whether the block must be written back to main memory at replacement.

The time taken by a cache hit is called the *hit time*, or *cache latency*, which is represented in terms of CPU clock cycles. The hit time is much smaller than the access time of main memory and it is only a few clock cycles in most microprocessors. On the other hand, the time taken by a cache miss is called the *miss penalty*, and the CPU is said to be *stalled* while it is waiting for the data from main memory. The term *memory stall cycles* refers to the number of clock cycles during which the CPU is stalled in a certain period.

Recent microprocessors have multi-level caches (two level or three level). On a cache miss in the higher level cache, the request is then forwarded to the next level cache, instead of accessing main memory immediately. Usually, caches closer to the CPU are smaller and faster.

1.2 Algorithms Should Be Cache-Conscious

Memory caches have achieved substantial speedup of software. However, further speedup is often possible by explicitly utilizing a cache.

Cache-conscious algorithms have been used for more than 10 years in scientific computation. The main application has been to operations on huge matrices, especially matrix multiplication. This is because matrix multiplication is not only quite useful in many scientific computations but is also susceptible to significant improvement by means of cache-conscious algorithms. It is not uncommon to have speedup by a factor of 10.

What about the other cache-conscious algorithms? LaMarca and Ladner have implemented cache-conscious mergesort and achieved an improvement of 15–40% in running time [13]. Even though it is not as significant as the improvement in matrix multiplication, 15–40% is a noticeable improvement in sorting.

The importance of cache-conscious algorithms is even growing because the processor-memory performance gap is growing. For example, the miss penalty on the Pentium 4 has reached 200 clock cycles; that is, one memory access is more expensive than dozens of integer operations on such microprocessors. Traditionally computer scientists have analyzed the performance of algorithms in terms of the number of operations executed. However, it is not too much to say that the rate-limiting factor on modern microprocessors is the number of cache misses. Therefore, in order to take full advantage of the recent powerful CPUs, algorithms should incur as few cache misses as possible.

1.3 Cache Profiling

So how does one make algorithms cache-conscious? How can one reduce cache misses? It is not impossible to estimate the number of cache misses by a close examination of the algorithm's memory access patterns, but it is usually a nontrivial task.

Classic profiling tools, such as `prof` and `gprof`, can figure out how much time a program spends in each function and how many times the function is called. They help identify bottlenecks, but they do not provide any information about cache misses. Cache behavior is sometime counterintuitive and it is hard to improve cache performance without analyzing when and where cache misses occur.

Cache profiling tools help address this problem, and quite a few cache profiling methods have been developed to study cache behavior and to analyze cache performance. Cache profiling methods are classified into two major categories, hardware performance counter based and software simulation based.

1.3.1 Hardware Performance Counter Based

Most modern microprocessors have a set of registers called performance counters, which count specific CPU events, such as TLB (Translation Look-aside Buffer) misses and cache misses. There are a number of performance profiling tools based on the hardware performance counter, and most of them provide cache performance information as a part of their functions.

Tru64 UNIX for HP Alpha is one of the few operating systems that natively supports profiling using hardware performance counters. The command line tool `uprofile` executes a specified command and counts CPU events occurring during its execution. HP Alpha architectures have two or three performance counters, and each of them has a set of CPU events it can measure. A user selects the events to measure from the available events. Another performance profiling tool based on performance counters is DCPI, the Digital (now Compaq) Continuous Profiling Infrastructure, which is a set of tools that provides low-overhead continuous profiling for all executable files, shared libraries, and even the kernel. It periodically collects performance statistics using performance counters and stores them into a database, which is analyzed later by various tools.

VTune Performance Analyzer [7] provides a comprehensive profiling environment for the Pentium processor family. The analyzer runs both on Windows and Linux, and supports the classic timer-based sampling and call graph analysis, as well as hardware performance counter based profiling. On the Windows platform, software development, debugging, performance profiling, and the tuning assistant are tightly integrated under the Visual Studio IDE. The tuning assistant is the interesting feature that provides source-level advice for multiple languages (C, C++, assembly, Fortran, and JAVA) to improve performance, using performance counter statistics. The accesses to performance counters are done inside the analyzer, and so users do not need to be aware of performance counters.

Brink and Abyss [21] are Pentium 4 performance counter tools for Linux. The performance monitoring feature on the Pentium 4 has been greatly extended from its predecessors. For example, the number of performance counters has increased from 2 to 18, and support for precise event-based sampling has added, which can identify the instruction precisely that caused a specific performance event even under out-of-order execution. Brink and Abyss place emphasis on utilizing these new performance monitoring features.

PAPI [4] (Performance Application Programming Interface) provides two levels of interfaces to access hardware performance counters. The high level interface is fairly portable and provides simple measurement. The low level is a fully programmable and architecture-dependent interface. Users explicitly insert these API calls in their application programs. The flexibility provided makes it possible to measure a specific part of the program. This is different from profiling tools above, which measure the entire program. PAPI supports a quite wide range of platforms: HP Alpha, Cray, IBM Power Series, Intel Pentium Series and Itenium, SGI (MIPS) R10000 and R12000, and Sun Microsystems Ultra I, II, and III. PAPI is not only used to measure the performance of a single program, but also widely used to develop performance profiling tools.

Hardware-based cache profiling is efficient in general because memory accesses are done by the actual hardware and reading performance counters is cheap. Therefore, it can be applied to relatively large programs. Also, it guarantees *real* improvement on the same hardware, while an improvement obtained by a software simulation is not necessarily effective on the actual hardware. But, at the same time, it is naturally bound to the cache model of underlying hardware. Consequently, hardware performance counter based methods are best suited for the practical case that a user wants to speed up a specific program on a specific architecture.

1.3.2 Software Simulation Based

Software simulation based cache profiling is preferred by people who are more interested in studying cache performance in general, than in tuning specific software on the specific hardware architecture. Software simulation makes it possible to perform analyses that are impossible by hardware based methods, such as changing cache parameters, and experimenting with new cache architectures which do not exist in the real world.

There are two major categories of software simulation: execution-driven and trace-driven. The execution-driven method simulates the CPU instructions and usually several I/O devices, and executes a program on a virtual CPU, by interpreting every instruction. On the other hand, the trace-driven tool only simulates a small part of hardware, e.g., caches and memory systems. A trace-driven simulator requires the "execution history" of a program, called a *trace*, to drive the simulation. For example, traces for cache profiling tools contain the history of memory accesses. Another simulator would require another kind of trace. Traces are usually generated by executing an *instrumented* version of the program, which is modified to output traces at defined control points. Trace-driven simulations are further divided into two types according to the way they collect traces: object-level instrumentation and source-level instrumentation.

Execution-Driven Simulators

The most accurate and most costly way in measuring cache performance by software simulation is to simulate the entire hardware system, including CPU, caches, memory, and I/O devices.

SimpleScalar [5] is a collection of powerful computer simulation tools, ranging from a fast functional simulator to a much more detailed simulator with out-of-order execution and non-blocking caches, and several of them include cache simulation. It is used widely in the study of computer architecture today. Simulation tools are highly portable and run on various host platforms, including Linux/x86, Windows 2000, Solaris, and others. They can emulate Alpha, PISA, ARM, and x86 instruction sets, where PISA (Portable Instruction Set Architecture) is a simple MIPS-like instruction set of its own. However, binary generation for the simulator is based on an old GNU C++ compiler, version 2.6.3, which did not fully support many of the template features needed for generic programming in C++.

SimOS [19] is a complete machine simulator based on the MIPS architecture, which simulates the microprocessor, memory system (including caches), console, disk, and even Ethernet. It first boots the operating system (IRIX 5.3) from the disk image, mounts another disk image which contains a executable file to simulate, and then runs the program. SimOS includes three simulators, Embra (fastest), Mipsy, and MXS (slowest), providing different speed and granularity of simulation. The simulator records all of the program's behavior during its execution into a single file, which is analyzed later by various script commands.

RSIM [18] (Rice Simulator for ILP Multiprocessors) is, as indicated by its name, a simulator for multiprocessors that exploit instruction-level parallelism (ILP). It places special emphasis on simulating ILP features, such as pipelining, out-of-order execution with branch prediction, speculative execution, and non-blocking caches. The target program must be compiled and linked for the SPARC V9 instruction set, and also must be linked with the library included in RSIM. The simulator runs on Solaris, IRIX 6.2 on SGI Power Challenge, and HP-UX version 10 on Convex Exemplar. Performance evaluation of every simulated component, including caches, is done by analyzing the statistics generated at the end of simulation. The target program can also generate the statistics explicitly by calling functions provided by RSIM.

Although execution-driven simulators provide precise results, they are quite slow in general. Getting started with them also requires considerable effort. Many simulators require a large development environment to be installed, including a compiler and various libraries. Some even require the disk image of operating systems. Such simulators may be overkill for people who just want to do cache profiling.

Trace-Driven Simulators — Collecting Traces by Object Modification

This category is quite popular in recent cache profiling studies. Tools in this category rewrite the compiled object or executable files to embed routines and function calls for profiling.

ATOM [22] (Analysis Tools with Object Modification) is a powerful framework for building customized program analysis tools that run under Tru64 UNIX on the HP Alpha architecture. One can develop one's own analysis tools with ATOM by implementing two kinds of routines: instrumenting routines and analysis routines. The instrumenting routines specify how the analysis routine calls are embedded into object code. Since users' analysis routines run in the same address space as the base application program, both *on-line* and *off-line* simulations are possible. The on-line simulator runs together with the base application, and thus the simulation routines should also be embedded into the object code. The off-line simulator, on the other hand, runs later, processing traces.

WARTS [2] (Wisconsin Architectural Research Tool Set) is a collection of tools for various program analyses, based on executable editing. EEL (Executable Editing Library) provides a rich set of functions to examine and modify executables. EEL works on Solaris and supports the executable format for SPARC processors, including UltraSPARCs with the V8+ instruction set. QPT (Quick Program Profiling and Tracing System) is a instrumentation tool built on EEL that embeds profiling routines and trace function calls into a program. The cache profiler CPROF, and cache simulators Tycho and DineroIII, perform cache performance analysis by reading traces collected by QPT.

Programs instrumented by object modification run much faster than execution-driven simulations since they run natively on the host architecture. In fact, the execution speed highly depends on the code instrumented. For a program that generates a huge trace, writing the trace can be a rate-limiting factor. Unfortunately, object-based instrumentation is highly object format-dependent and the actual tools are tied to a particular architecture. Once the trace is obtained, however, it can be simulated on other platforms.

Trace-Driven Simulators — Collecting Traces by Source-Level Instrumentation

Spork [20] has studied cache behavior of algorithms by manually embedding trace generation codes into algorithm implementations. Whether to trace an expression or not is completely up to a user. For example, if a user assumes that a certain object will be assigned to a register, the user can just leave the object uninstrumented. Though the approach is portable, it is difficult to apply this method to general algorithms without an automatic translation tool.

1.4 STL Memory Access Tracer

We have developed a new cache profiling tool based on source-level instrumentation. The tool can analyze cache performance of algorithms that operate on STL random access iterators, and we therefore named the tool SMAT (STL Memory Access Tracer). The memory access traces are collected via *adaptors*, which in STL terminology is a component that wraps another component and provides a modified interface or functionality. The SMAT adaptor provides the same interface as the base component with the additional function of generating traces.

SMAT is designed to analyze cache performance of an algorithm, not an entire program. This is often convenient for the study of cache-conscious algorithms. For example, it is common to generate test data and then process it with some algorithm in the same program. In such cases, people usually want to perform cache profiling only for the algorithm code, disregarding the data generation code.

Advantages of SMAT over other cache profiling tools can be summarized as follows.

- It is portable and independent of hardware architecture since it provides trace functionality at the source level.
- It requires little or no modification to the algorithm. However, SMAT expects that the implementation of the algorithm makes all accesses to data sequences through iterators and their associated types. When this is not the case (as may happen if the algorithm is not programmed as generically as it could be), the implementation should be modified to remove the non-iterator accesses.
- It is easy to get started with. SMAT only consists of one header file and six utility programs.

On the other hand, SMAT has the afore-mentioned drawback of source-based tracers, in that it traces *all* objects involved whereas the object modification based method only traces *real* memory accesses and *not* objects assigned to registers. That is, the source-based trace and object-based trace are collected at different levels of the memory hierarchy, as shown in Figure 1.1. Consequently, SMAT generates much longer traces than the object-based method. Besides, operations of the SMAT-adapted objects cannot be completely optimized by a compiler.

The primary theme of this paper is how to deal with these limitations. With a cache simulator that simulates registers as well, and a trace file optimizer that eliminates inefficient memory accesses that would in the absence of tracing be eliminated by the compiler, SMAT provides the convincing cache profiling.

The rest of the paper is organized as follows. Section 2 explains how to use SMAT, presenting real example source codes. Section 3 is the reference manual of SMAT adaptor library and utility programs. The design issues are discussed in Section 4. In Section 5, cache performance of STL sorting algorithms is analyzed using SMAT. Improvements of cache-conscious versions of introsort and mergesort over typical implementations are examined as well. Complete source codes of the

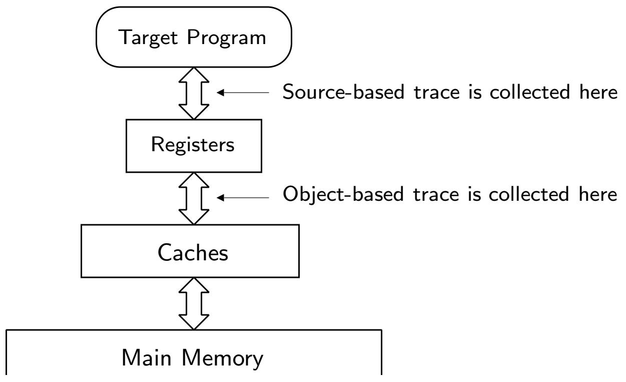

Target Program
Source-based trace is collected here
Registers
Object-based trace is collected here
Caches
Main Memory

Figure 1.1: Difference between the source-based and object-based trace collection.

SMAT adaptor library, cache simulator, and other utility programs are presented in Appendix A, and Appendix B contains their test suites and results.

Chapter 2

User's Guide

2.1 Overview

SMAT (STL Memory Access Tracer) is a trace-based cache profiling tool to analyze cache performance of algorithms that operate on STL random access iterators. The tool consists of the adaptor library (`smat.h`), cache simulator (`smatsim`), trace file optimizer (`smatopt`), operation counter (`smatcount`), trace file viewer (`smatview`), and trace file plotter (`smatplot`).

Figure 2.1 illustrates the flow using SMAT. The caller of the target algorithm is first modified so that the iterator arguments to the target algorithm are wrapped by the SMAT iterator adaptor. Then, the compiled program generates a trace file during its execution. Since the raw trace file may contain inefficient memory references, it is recommended to "optimize" it with the trace file optimizer `smatopt`. The optimized trace file is processed by the cache simulator `smatsim` to analyze cache performance. The operation counter `smatcount` provides the classic operation counting feature, implemented along principles similar to those of OCAL (Operation Counting Adaptor Library) [8], a more general operation counting facility. The trace file plotter `smatplot` generates a trace plot that exhibits how the algorithm has made memory references over time and which of them incurred cache misses. Such memory access trace plots are an extension of the kind of iterator trace plots produced by the ITRACE tool [14].

The SMAT adaptor library overloads almost all member functions of the base class and several global functions. Every overloaded function generates a memory access trace implicitly. SMAT can trace memory accesses not only of the iterator type but also of iterator's *associated types*. Every STL iterator type has associated types, and SMAT can trace four of them: value type, difference type, pointer type, and reference type. The value type is the type of object to which the iterator points, or the type of `*i` where `i` is an iterator. The difference type is a signed integer type that represents the distance between two iterators. The pointer and reference types are simply a pointer and a reference to the value type, respectively.

Almost every operation on iterators and associated type objects generates a trace. For example, a statement `v = *(i + 1)`, where `i` is an iterator and `v` is a value type object, generates the following trace. (The trace is actually in a binary format and not human-readable. The following output was converted to the human-readable format by the trace file viewer `smatview`. Comments were added manually.)

```
iaddi   4, (0xbffff8b4), (0xbffff884)  ; i + 1 -> tmp
readi   4, (0xbffff884)                ; *tmp
movv    4, (0xbffff8c8), (0xbffff894)  ; *tmp -> v
dtori   4, (0xbffff884)                ; destruct tmp
```

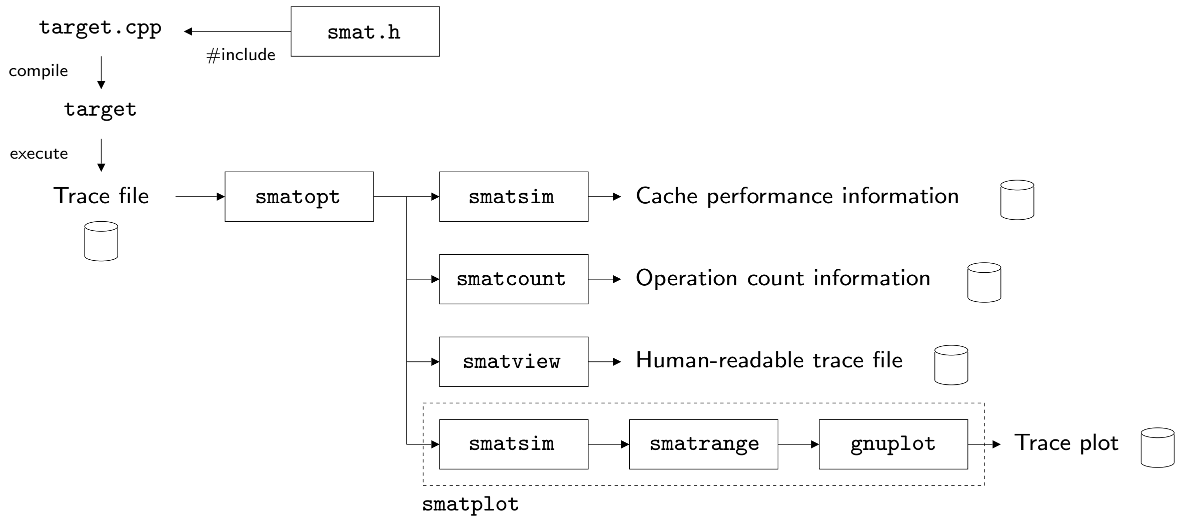


Figure 2.1: SMAT flow diagram.

The first trace corresponds to the subexpression `i + 1`, where `i`'s address is `0xbffff8b4`, and the result is stored into a temporary object at address `0xbffff884`. The mnemonic `iaddi` stands for "immediate addition of the iterator type." The last letter of the mnemonic, one of `i` (iterator type), `v` (value type), `d` (difference type), `p` (pointer type), represents the type of object involved. The number 4 after the mnemonic indicates the access length, or the size of the object. The second trace instruction is created by the dereference operation. In that trace, the temporary object is referenced to obtain the address to which the temporary object points. The third trace indicates the assignment, from the address `0xbffff8c8` to `v` (at address `0xbffff894`). The temporary object is no longer necessary and destructed in the last trace instruction.

2.2 Obtaining SMAT

The source archive of SMAT can be found at `http://www.cs.rpi.edu/~musser/gp/`. See `README` file in the archive how to configure, compile, and install the SMAT library and utility programs.

SMAT has been tested on the following compilers and operating systems.

- GCC version 3.2 on Linux 2.4.19, and
- GCC version 3.2 on Solaris 5.8.

The iterator adaptor (`smat.h`) requires the Boost Libraries [17], version 1.29.0 or newer. The trace file plotter `smatplot` requires Gnuplot [9], version 3.7 or newer, and it optionally uses the image compression utility `pnmtopng` of `netpbm` [1] to generate PNG format output.

2.3 Example 1 — Reverse a String

In this section, as a first example of how to use SMAT, we measure cache performance of the `std::reverse` algorithm. Example1a is a small program that reverses a fixed string and prints it out to the standard output.

"example1a.cpp" 10a ≡
```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s("Hello, this is SMAT.");
  reverse(s.begin(), s.end());
  cout << s << '\n';
  return 0;
}
```

2.3.1 Modifying Source for SMAT

Some changes must be made in the program to use SMAT, even though no change is necessary to the `std::reverse` algorithm itself. Example1b is the modified source code.

"example1b.cpp" 10b ≡
```
#include "smat.h"           // added
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string s("Hello, this is SMAT.");
  reverse(smat<string::iterator>(s.begin()),    // changed
          smat<string::iterator>(s.end()));
  cerr << s << '\n';        // cout -> cerr
  return 0;
}
```

Three changes have been made in `example1b.cpp`. First a header file, `smat.h`, is included. It is safe to include it before or after other STL header files. Secondly the arguments to the `std::reverse` algorithm are wrapped in a call of a constructor of `smat<string::iterator>`. Finally the output stream of the reversed string is changed from the standard output (`cout`) to the standard error stream (`cerr`). Note that these changes are in general required in every program when using SMAT.

Unlike first two changes, however, the last one is not essential. This change is made because the trace file is written to the standard output by default, and so the native output must be separated from the trace. It is possible to specify the output stream for the trace file by macro `SMAT_SET_OSTREAM`. For example, the trace will be written to the standard error stream by calling `SMAT_SET_OSTREAM(std::cerr)`. However, continuing to use the standard output stream would be convenient because other SMAT utility programs read a trace file from the standard input stream.

In this example, no modification is needed to the `std::reverse` algorithm because it is implemented “suitably” for SMAT. In order to trace all memory accesses with SMAT, the algorithm implementation must strictly observe STL generic programming conventions; that is, an algorithm can only use an iterator type and its associated types. See Section 2.5 for discussion of how to ensure algorithms follow the necessary type conventions.

2.3.2 Compiling a Program with SMAT

The file `example1b.cpp` is a complete example of using SMAT and so it can be compiled. When compiling, include paths to `smat.h` and Boost libraries must be specified. The first character `$` indicates the command line prompt, which we assume throughout this chapter.

```
$ g++ -Ipath_to_smat -Ipath_to_boost -o example1b example1b.cpp
```

If everything goes well, the executable file `example1b` with memory access tracing capability is ready to run.

2.3.3 Measuring Cache Performance

The executable `example1b` traces the memory accesses performed by the `std::reverse` algorithm and outputs the trace file to the standard output. The cache simulator `smatsim` reads a trace file from the standard input and outputs cache performance information.

Since the "raw" trace file may contain inefficient memory references, the recommended way is to filter it with the trace file optimizer `smatopt`.

```
$ ./example1b | smatopt | smatsim
.TAMS si siht ,olleH
4 hits, 6 misses, 202 cycles
```

The first line is the command line. The next line is the reversed string that `./example1b` has written to the standard error stream, and the last line is the output from the simulator. According to the report, the memory accesses performed by the `std::reverse` algorithm caused four cache hits and six cache misses, and took 202 clock cycles to complete.

An important note is that SMAT only counts clock cycles taken by memory references. Clock cycles taken by operations, such as arithmetic operations and comparisons, are *not* included in the number reported.

This simple diagnosis may be sufficient for some users, but `smatsim` gives more detailed outputs with the option `-v`. With another option `-t`, the simulator reports the diagnosis separately for the iterator type and its associated types. See Section 2.4.1 and 2.6.4 for these detailed cache performance information.

Cache Parameters

So what kind of cache does `smatsim` simulate? The option `-c` prints out the configurable cache parameters of the simulator. By default, the output is as follows.

```
$ smatsim -c
Number of registers     8 (32-bit)
L1 cache block size     32 bytes
L1 cache cache size     8192 bytes (256 blocks)
L1 cache associativity  2-way set associative
L1 cache write policy   write back
L1 cache hit time       1 clock(s)
Main memory hit time    32 clock(s)
```

The memory system has eight 32-bit registers, an 8KB level 1 cache with 32-byte blocks, and main memory. The cache is 2-way set associative and is write back. It only takes 1 clock on a cache hit, but accesses to main memory take 32 clock cycles per block. These parameters can be changed by recompiling `smatsim`. Section 2.6.3 explains how to change cache parameters.

The simulator does not have any notion of *parallelism*. It does not process the next trace instruction until the current instruction has been finished. The simulator does not have write buffers, and does not support non-blocking caches. See Section 4.1.1 for the rationale for these design decisions.

2.4 Example 2 — Finding the Minimum and Maximum Elements

More interesting features, detailed cache analysis, trace plotting, and operation counting, are presented in this section. Given a sequence, three algorithms find the minimum and maximum elements *at once*, and their cache performance is compared. Each algorithm takes the range of a sequence as arguments and return a pair (`std::pair`) of iterators that point to the minimum and maximum elements. See Appendix A.2.1 for the complete source code.

The first algorithm `minmax_twopass` calls `std::min_element` and `std::max_element` to find the minimum and maximum elements, respectively.

⟨Minmax in two passes. 12a⟩ ≡

```
template <typename T>
pair<T, T> minmax_twopass(const T& first, const T& last)
{
  return pair<T, T>(min_element(first, last), max_element(first, last));
}
```

Used in part 113a.

This algorithm is not as efficient as possible because it makes two passes. It is trivial to find both the minimum and the maximum in one pass, as in the next algorithm.

⟨Minmax in one pass. 12b⟩ ≡

```
template <typename T>
pair<T, T> minmax_onepass(T first, const T& last)
{
  T min = first, max = first;
  while (++first != last)
    {
      if (*first < *min)
        min = first;
      else if (*max < *first)
        max = first;
    }
  return pair<T, T>(min, max);
}
```

Used in part 113a.

This one pass algorithm still makes as many as $2N$ comparisons, where N is the number of elements. The third algorithm, `minmax_fewercmp`, reduces the number of comparisons to $1.5N$ by considering the numbers in pairs and updating the minimum and maximum using only three comparisons per pair instead of four.

⟨Minmax with fewer comparisons. 13⟩ ≡

```
template <typename T>
inline void minmax_helper(const T& small, const T& large, T& min, T& max)
{
  if (*small < *min)
    min = small;
  if (*max < *large)
    max = large;
}

template <typename T>
pair<T, T> minmax_fewercmp(T first, const T& last)
{
  T min = first, max = first;
  for (; first + 2 <= last; first += 2)
    {
      if (*first < *(first + 1))
        minmax_helper(first, first + 1, min, max);
      else
        minmax_helper(first + 1, first, min, max);
    }

  if (first + 1 == last)
    minmax_helper(first, first, min, max);

  return pair<T, T>(min, max);
}
```

Used in part 113a.

The program `example2` takes the name of the algorithm, one of `twopass`, `onepass`, or `fewercmp`, as the first command line argument, and the number of elements as the second argument. It generates the specified number of random numbers, and prints the minimum and maximum elements by calling the requested algorithm. The following example prints the minimum and maximum of 2000 random numbers, found using the one-pass algorithm.

```
$ ./example2 onepass 2000 | smatopt | smatsim
1698506, 4294861914
1755 hits, 256 misses, 10235 cycles
```

The second line `1698506, 4294861914` is the minimum and maximum values that the algorithm found, and the last line is the output from the cache simulator. Note that the cache performance output may differ slightly according to the alignment of the internal vector.

2.4.1 Detailed Cache Analysis

Figure 2.2 compares the number of clock cycles per element of three algorithms processing 100–100000 elements. Looking at clock cycles per element makes it easier to compare values each other and grasp the performance ratio. The most interesting point is that values of the two-pass algorithm differ clearly below and above approximately $N = 2000$. Since one element occupies 4 bytes, $N = 2000$ corresponds approximately to the size of cache, 8KB. This change in performance that occurs at this size analyzed in Section 2.4.2 by using trace plots.

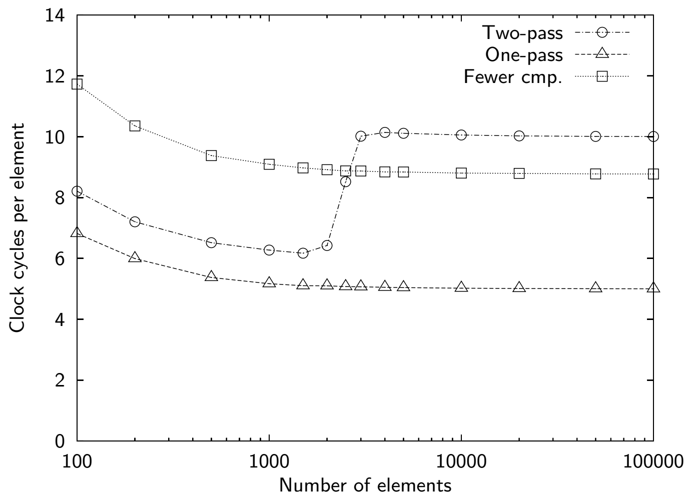


Figure 2.2: Three algorithms finding the minimum and maximum elements.

The graph shows that the two-pass and one-pass algorithms converge to about 10 and 5 clock cycles per element, respectively, and the fewer-comparisons algorithm converges to about 9 clock cycles per element. Observe that it takes more clock cycles when N is small, which indicates a fixed cost unrelated to N. This fixed cost can be measured by processing a very small number of elements.

```
$ ./example2 onepass 5 | smatopt | smatsim
303761048, 3607553667
5 hits, 5 misses, 170 cycles
```

Although the one-pass algorithm is supposed to take asymptotically 5 cycles per element, it takes 170 cycles in processing only 5 elements. It seems that roughly 150 cycles are the fixed cost. These cycles are, in fact, taken by the accesses to local variables on stack.

The algorithm `minmax_fewercmp` has made a poor showing. Though it performs only one pass, it takes more cycles than the two-pass algorithm in the range $N \leq 2000$, and even with $N > 2000$, the clock count is close to the two-pass algorithm. What makes the performance so bad?

Analysis by Type

Refinement of the analysis by considering operations on different types can help to explain the cache behavior. With the option `-t`, `smatsim` can help determine which type operations are dominant in causing cache misses by counting cache statistics separately for the iterator type and its associated types. The following are the output of one-pass algorithm and fewer-comparisons algorithm processing 10000 elements. (The native output from `example2`, values of the minimum and maximum elements, are now redirected to `/dev/null` because they are no longer interesting.)

```
$ ./example2 onepass 10000 2>/dev/null | smatopt | smatsim -t
Iterator type          3 hits, 4 misses, 135 clocks
Value type             8753 hits, 1251 misses, 50068 clocks
Difference type        0 hits, 0 misses, 0 clocks
Pointer type           0 hits, 0 misses, 0 clocks
```

```
Total                   8756 hits, 1255 misses, 50203 clocks

$ ./example2 fewercmp 10000 2>/dev/null | smatopt | smatsim -t
Iterator type           29435 hits, 17 misses, 29996 clocks
Value type              16997 hits, 1284 misses, 59369 clocks
Difference type         0 hits, 0 misses, 0 clocks
Pointer type            0 hits, 0 misses, 0 clocks
Total                   46432 hits, 1301 misses, 89365 clocks
```

The most notable difference is the cache hit count of the iterator type. Evidently when accessing iterators the fewer-comparisons algorithm has many more register misses and thus needs to access the cache much more often than the one-pass algorithm does. Further consideration of this performance problem is made in Section 2.6.4 and Section 4.2.2.

2.4.2 Trace File Plotting

A trace file plot can help provide intuitive understanding of the memory access pattern performed by an algorithm. The following commands generate trace plots for the two-pass algorithms processing 1000, 2500, and 5000 elements, respectively, which are shown in Figure 2.3.

```
$ ./example2 twopass 1000 | smatopt | smatplot -o fig-e2-1000.eps
$ ./example2 twopass 2500 | smatopt | smatplot -o fig-e2-2500.eps
$ ./example2 twopass 5000 | smatopt | smatplot -o fig-e2-5000.eps
```

Red and gray dots correspond to cache misses and cache hits, respectively. The x-axis represents the number of clock cycles, or time in the simulation, and the y-axis represents the address space. Note that the plot only shows memory accesses of the L1 cache. Memory accesses handled within registers do not appear in the trace plot.

In graph (a), 1000 elements, the first pass consistently causes cache misses but the second pass does not at all because the entire array is in the cache. So the second pass runs much faster than the first pass. This is the reason why the two-pass algorithm runs relatively faster for smaller N in Figure 2.2. The array does not fit into the cache in graph (c). Consequently it takes twice the time of the one-pass algorithm. Graph (b) shows behavior intermediate between that of (a) and (c). The second pass makes some cache misses, but some cache hits as well.

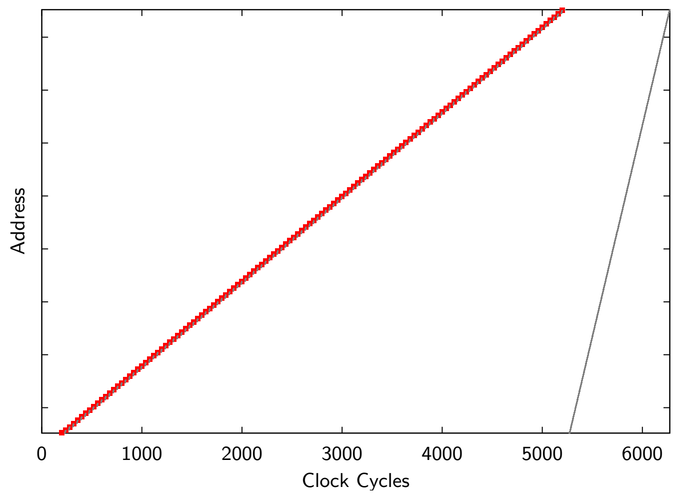
2.4.3 Operation Counting

Operation counting is a traditional method to measure the performance of algorithms. In SMAT, the command `smatcount` provides operation counting. The command reads the trace file from the standard input and outputs the operation counting information to the standard output stream. The number of operations is counted separately for the iterator type, value type, difference type, and pointer type. Here is the example of operation counting for the one-pass algorithm.

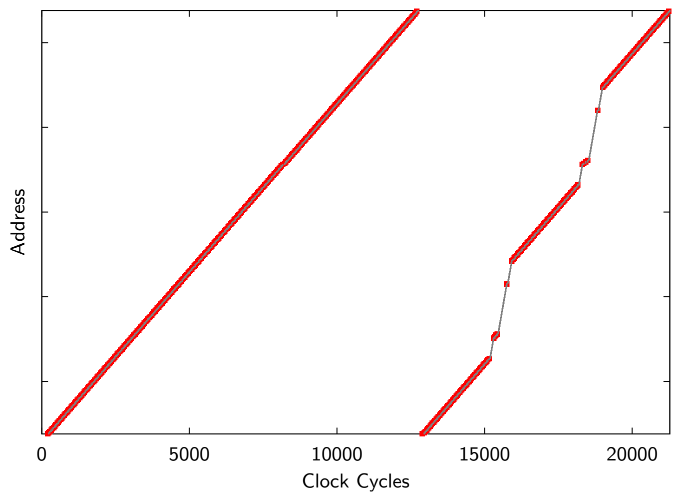
```
$ ./example2 onepass 10000 2>/dev/null | smatopt | smatcount
Iterator:
  Assignment        21
  Comparison        10000
  Arithmetic        10000

Value type:
  Assignment        0
```

(a)



(b)



(c)

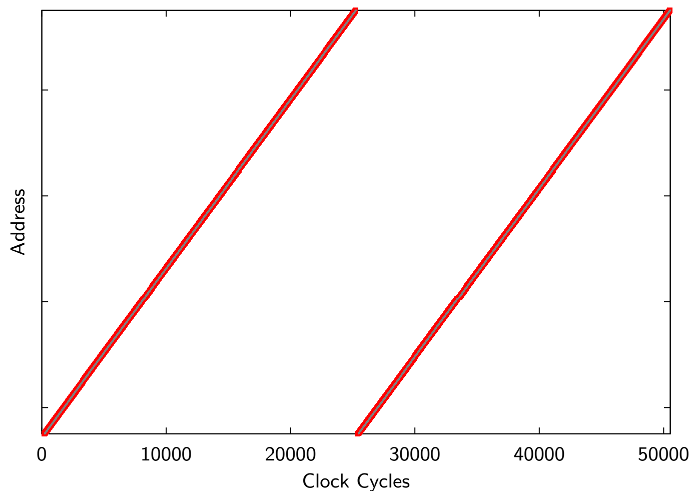


Figure 2.3: Trace plots of `minmax_twopass` algorithm, processing (a) 1000, (b) 2500, and (c) 5000 elements.

Table 2.1: Comparison of the number of operations.

Number of elements	Algorithm	Iterator			Value type		
		Assign.	Comp.	Arith.	Assign.	Comp.	Arith.
$N = 100$	Two-pass	16	202	200	0	198	0
	One-pass	12	100	100	0	196	0
	Fewer cmp.	12	52	202	0	150	0
$N = 1000$	Two-pass	23	2002	2000	0	1998	0
	One-pass	18	1000	1000	0	1993	0
	Fewer cmp.	17	502	2002	0	1500	0
$N = 10000$	Two-pass	26	20002	20000	0	19998	0
	One-pass	21	10000	10000	0	19990	0
	Fewer cmp.	20	5002	20002	0	15000	0
$N = 100000$	Two-pass	28	200002	200000	0	199998	0
	One-pass	23	100000	100000	0	199989	0
	Fewer cmp.	22	50002	200002	0	150000	0

```
    Comparison         19990
    Arithmetic         0

  Total:
    Assignment         21
    Comparison         29990
    Arithmetic         10000
```

The number of assignment, comparison, and arithmetic operations are counted separately for each type. The difference type and pointer type are not listed because there were no operations of these types. Section 2.6.5 explains a more detailed report that can be obtained using option `-v`.

Table 2.1 compares the operation counts of three algorithms processing 100–100000 elements. In the fewer-comparisons algorithm, the number of value-type comparisons is $1.5N$ as intended. The number of iterator type comparisons is fewer than other algorithms as well. The one-pass algorithm performs about a half the iterator comparisons and arithmetic operations of the two-pass algorithm.

At $N = 100000$, the numbers of operations overall are 4000012 in the one-pass algorithm and 400026 in the fewer-comparisons algorithm, showing little difference. However, in terms of clock cycles of the simulator, the one-pass algorithm has 43% fewer cycles than the fewer-comparisons algorithm. This shows that operation counting is not necessarily an appropriate index to compare performance when caches are involved.

2.5 Making the Best Use of SMAT

SMAT expects the algorithm to access memory through operations on iterator types and their associated types. Otherwise, SMAT would miss seeing some memory accesses, which would result in inaccurate cache analysis. For algorithms designed according to generic programming principles as exemplified by STL algorithms, no modification is needed. However, modification to meet the type requirements is sometimes necessary for user-written algorithms. This section gives advice how to modify algorithms so that SMAT can trace all memory accesses.

First of all, SMAT is designed to work on STL random access iterators. So the target algorithm must take STL random access iterators as parameters (and possibly others). STL containers that provide random access iterators, in addition to C-style arrays, are `std::vector` and `std::deque`[1]. SMAT could also be used on algorithms that access a user-defined container if the container provides iterators that meet the STL requirements for random access iterators.

2.5.1 Making Algorithms Analyzeable

Consider the following algorithm, `accumulate_stride`, which accumulates elements of the given vector at positions that are `stride` apart. Unlike `std::accumulate`, it does not take an `init` parameter.

The first version is intentionally written in the non-generic style, which is dedicated to container type `std::vector<int>`. This algorithm will be revised step by step.

⟨Accumulate stride algorithm (not generic). 18a⟩ ≡

```
int accumulate_stride_1(std::vector<int>& v, int stride)
{
  int sum = 0;
  for (size_t i = 0 ; i < v.size(); i += stride)
    sum += v[i];
  return sum;
}
```

Used in part 115b.

Iterators

The first thing that should be done is to parameterize in terms of iterators instead of the container. This is the standard way of the sequence-based generic algorithms of STL. Iterator parameterization provides additional flexibilities. With the following code, for example, a user can now compute the *partial* stride sum of a sequence instead of an entire sequence.

⟨Accumulate stride algorithm (iterators). 18b⟩ ≡

```
int accumulate_stride_2(std::vector<int>::iterator first,
                        std::vector<int>::iterator last, int stride)
{
  int sum = 0;
  for ( ; first < last; first += stride)
    sum += *first;
  return sum;
}
```

Used in part 115b.

[1]Small changes to the iterator adaptor will be required to trace `std::deque` iterator correctly because the size of `std::deque::iterator` is larger than that of `std::deque::difference_type`. Section 4.3 describes why this can be a problem.

Templates

However, just using iterators is not enough because the previous code can only process vector iterators. Making the argument types template parameters provide additional flexibility. In the following code, the iterator type and the type of `stride` are parameterized. The algorithm can now operate on other types of iterators.

⟨Accumulate stride algorithm (templates). 19a⟩ ≡

```
template <typename T, typename U>
int accumulate_stride_3(T first, T last, U stride)
{
  int sum = 0;
  for ( ; first < last; first += stride)
    sum += *first;
  return sum;
}
```

Used in part 115b.

Traits Types

Although the parameter types have been templated, the algorithm still assumes that the iterator's value type is convertible to `int`, which is not always true.

We can obtain the value type — the type of elements to which objects of iterator type `T` can point — using a library-provided *traits* class, `std::iterator_traits<T>::value_type`. The return type and the type of `sum` are corrected in the following code. The initialization of `sum` has changed so that it is now initialized by the first element of the sequence, because initializing by 0 is not generic. Another change is that the addition to `sum` is done by `+` instead of `+=` since it makes fewer requirements on the value type.

⟨Accumulate stride algorithm (traits types). 19b⟩ ≡

```
template <typename T, typename U>
typename std::iterator_traits<T>::value_type
accumulate_stride_4(T first, T last, U stride)
{
  typename std::iterator_traits<T>::value_type sum(*first);
  first += stride;
  for ( ; first < last; first += stride)
    sum = sum + *first;
  return sum;
}
```

Used in part 115b.

Another Iterator Traits Types

So far the algorithm has become fairly generic. It will receive a passing grade in most circumstances. However, SMAT has a stricter requirement. The type of `stride` is the problem. What will the type of `stride` be if the function is called as follows?

```
accumulate_stride(v.begin(), v.end(), 5);
```

The answer is `int`, which SMAT cannot trace. To trace the third argument, its type must be the iterator's *difference type*.

⟨Accumulate stride algorithm (final). 20⟩ ≡
```
template <typename T>
typename std::iterator_traits<T>::value_type
accumulate_stride_final(T first, T last,
  typename std::iterator_traits<T>::difference_type stride)
{
  typename std::iterator_traits<T>::value_type sum(*first);
  first += stride;
  for ( ; first < last; first += stride)
    sum = sum + *first;
  return sum;
}
```
Used in part 115b.

This technique, specifying the argument type by a traits type, ensures that all arguments are of an iterator type or its associated types.

2.5.2 More Tips

Using Difference Type

A user can naturally assume that the difference type is as "powerful" as `int` type. Therefore, the difference type can be used wherever `int` type works, and any arithmetic operations applicable to `int` type can be applied to the difference type. Use the difference type wherever possible and avoid using not-traceable raw types.

Making profiling faster

In the caller of the algorithm, it is recommended to put the following function calls before the first stream I/O operation. These calls make profiling run much faster (at a factor of 10) on some platforms.

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
```

2.6 Advanced Topics

2.6.1 Disabling SMAT

The memory access tracer can be disabled by replacing all occurrences of `smat<Iter>` to `Iter` manually. However, a much more convenient way is provided, in which a user wraps iterators by `smat<Iter>::type` instead of `smat<Iter>`.

The type `smat<Iter>::type` works the same as `smat<Iter>` by default. But if the macro `SMAT_DISABLE` is defined, `smat<Iter>::type` yields the raw type `Iter`. The `SMAT_DISABLE` also disables all SMAT macros, such as `SMAT_SET_OSTREAM`, by redefining them to do nothing. Therefore, a user can disable SMAT by just defining `SMAT_DISABLE`, without changing the source. But be sure to define `SMAT_DISABLE` *before* the header file `smat.h` is included.

2.6.2 Viewing Trace Files

The trace file viewer `smatview` converts a trace file into a human readable format, in which each line of the output corresponds to one trace instruction. A typical trace instruction would look as follows:

```
movv    4, (0x0807e234), (0x0807e20c)
```

The line consists of the following information. See Section 4.3 for how this information is actually encoded in the raw trace file.

- Instruction code — assignment (`movv`),
- Type of object — value type (`movv`),
- Access length — 4,
- Source address(es) — 0x0807e234, and
- Destination address — 0x0807e20c.

The first word, mnemonic, indicates the operation and the type of object. The last letter of the mnemonic, one of `i` (iterator type), `v` (value type), `d` (difference type), `p` (pointer type), represents the object type. The number that follows the mnemonic is the object size, or the length of the memory access. Hexadecimal numbers enclosed in parentheses are source or destination addresses. Source addresses always precede destination addresses. Every trace instruction takes a fixed number of source or destination addresses. The `mov` instruction takes one source and one destination address, for example.

Sections 3.1–3.5 describe the relationships between SMAT adaptor libraries and trace instructions, and Table 3.1 summarizes them.

2.6.3 Changing Cache Parameters

Cache parameters of the simulator can be changed by recompiling `smatsim`. A user can modify parameters either by modifying `Makefile` or specifying them as command line arguments of `make`.

The configurable parameters are as follows. Definitions in `Makefile` is shown in the parenthesis. The first four parameters must take powers of two, and the write policy must be either `true` or `false`. Hit time takes a non-negative integer value.

- Number of registers (`NREG = 8`)
- L1 cache capacity (`L1_SIZE = 8192`)
- L1 cache block size (`L1_BLOCK = 32`)
- L1 cache associativity (`L1_ASSOC = 2`)
- L1 cache hit time (`L1_LATENCY = 1`)
- L1 cache write policy (`L1_WB = true`)
- Main memory hit time (`MAIN_LATENCY = 32`)

To change the cache size to 16KB, for example, edit `Makefile` so that the block size parameter is `L1_SIZE = 16384`, then run `make` again.

```
$ make smatsim
```

If make command supports defining parameters on the command line, the following would do the same thing. The user does not have to edit Makefile in this case.

```
$ make smatsim L1_SIZE=16384
```

To avoid confusion, it is recommended give the new simulator a name distinguishable from other versions, for example, smatsim-16kb in this case.

2.6.4 More Cache Performance Information

The cache simulator smatsim outputs more detailed information when -v option is specified. For example, the one-pass algorithm introduced in Section 2.4 generates the following output.

```
$ ./example2 onepass 10000 2>/dev/null | smatopt | smatsim -v
Register hits           109994 hits (99977r + 10017w)
Register misses         10010 misses (10006r + 4w, 8.34%)
Register writebacks     1 (0.00999% of all misses)
Register reads          439932 bytes
Register writes         40084 bytes

L1 cache hits           8756 hits (8755r + 1w)
L1 cache misses         1255 misses (1255r + 0w, 12.5%)
L1 cache writebacks     1 (0.0797% of all misses)
L1 cache reads          40040 bytes
L1 cache writes         4 bytes

Main memory reads       40160 bytes
Main memory writes      32 bytes

Total clock cycles      50203 clocks
```

The output consists of four parts, the information about registers, the level 1 cache, main memory, and clock cycles. The register and L1 cache information look quite similar. In fact, the register set is implemented as the *level 0* cache, and so the same statistics are reported for it as for the L1 cache. Having registers in the simulator is essential in SMAT. See Section 4.1.1 for this design decision.

Figure 2.4 depicts where these cache statistics are collected in the memory hierarchy. For example, statistics Register reads and Register writes count how many read accesses and write accesses are made by the algorithm, respectively. These accesses cause hits and misses in the register set, and if requests cannot be satisfied within registers, they are forwarded to L1 cache. Statistics L1 cache reads and L1 cache writes count the number of these forwarded accesses. The requests are handled in L1 cache and some of them finally cause main memory accesses.

For the number of hits and misses, numbers in the parenthesis indicate how many of them are caused by read requests and write requests, respectively. For example,

```
L1 cache hits           8756 hits (8755r + 1w)
```

shows that 8755 hits out of 8756 hits are incurred by read requests. The number of misses has an additional number shown as a percentage.

```
L1 cache misses         1255 misses (1255r + 0w, 12.5%)
```

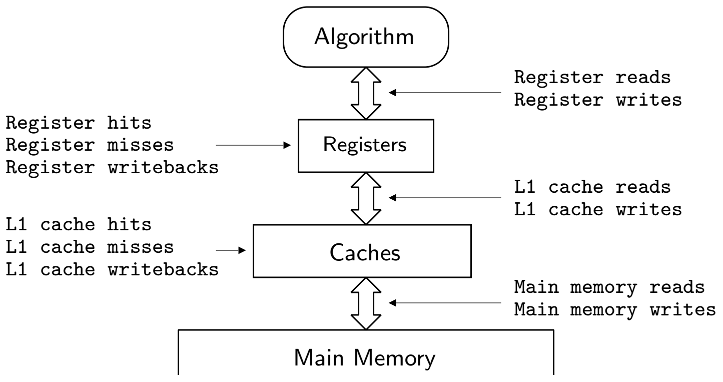


Figure 2.4: Levels in the memory hierarchy where cache statistics are collected.

The additional number is the *miss rate*, which is given by the following equation.

$$\frac{\text{Number of misses}}{\text{Number of cache accesses}} = \frac{\text{Number of misses}}{\text{Number of hits} + \text{Number of misses}}.$$

So $1255/(8756 + 1255) = 0.125$ in the example.

A writeback operation occurs in cache block replacement when the block pushed out is dirty. In a cache with write back policy, which is the default policy of `smatsim`, a writeback operation is the only operation that issues a write request to main memory. A write miss in L1 cache does *not* issue a write request to main memory. Instead, it issues a read request to bring data to the L1 cache, which is called cache *fill*. The number of writeback operations is shown in the statistics. The ratio to all the misses is shown as well.

```
    L1 cache writebacks     1 (0.0797% of all misses)
```

Note that writeback operations issue requests to main memory, which take exactly the same number of clock cycles as normal cache misses. So having a large number of writebacks does affect performance.

Figure 2.5 summarizes the number of requests issued in each level of the memory hierarchy. All numbers in the statistics and the relationship among them should now be clear. The figure also presents the intuition of how the clock cycles can be computed from the statistics and cache configuration parameters.

$$\begin{aligned} \text{Total clock cycles} = \;& (\text{Number of L1 misses and writebacks}) \times \text{Hit time}_{\text{Main}} + \\ & (\text{Number of L1 hits and misses}) \times \text{Hit time}_{\text{L1}} \end{aligned} \tag{2.1}$$

In the example, the right-hand side yields $(1255+1)\times 32+(8756+1255)\times 1 = 50203$, which agrees with the simulated clock cycles.

Back to The Fewer-Comparisons Algorithm

In Section 2.4.1, the fewer-comparisons algorithm showed poorer performance than expected. Can the detailed cache statistics help explain this result? Here is the output of the fewer-comparisons algorithm, processing the same number of elements as the one-pass algorithm.

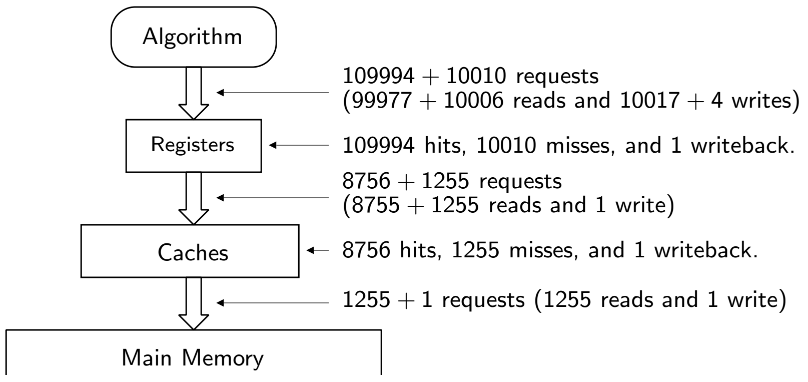

Algorithm
109994 + 10010 requests
(99977 + 10006 reads and 10017 + 4 writes)
Registers
109994 hits, 10010 misses, and 1 writeback.
8756 + 1255 requests
(8755 + 1255 reads and 1 write)
Caches
8756 hits, 1255 misses, and 1 writeback.
1255 + 1 requests (1255 reads and 1 write)
Main Memory

Figure 2.5: Number of requests issued at each level in the memory hierarchy.

```
$ ./example2 fewercmp 10000 2>/dev/null | smatopt | smatsim  -v
Register hits           62337 hits (57321r + 5016w)
Register misses         47713 misses (32707r + 15006w, 43.4%)
Register writebacks     20 (0.0419% of all misses)
Register reads          360112 bytes
Register writes         80088 bytes

L1 cache hits           46452 hits (46432r + 20w)
L1 cache misses         1281 misses (1281r + 0w, 2.68%)
L1 cache writebacks     1 (0.0781% of all misses)
L1 cache reads          190852 bytes
L1 cache writes         80 bytes

Main memory reads       40992 bytes
Main memory writes      32 bytes

Total clock cycles      88757 clocks
```

Compared to the output from one-pass algorithm, the fewer-comparisons algorithm clearly makes many register misses (10010 v.s. 47713), while total number of register reads and writes is comparable ($439932 + 40084 = 480016$ v.s. $360112 + 80088 = 440200$, in bytes).

This implies that the fewer-comparisons algorithm has less locality in memory references. Recalling that the fewer-comparisons algorithm made a large number of misses in iterator accesses, we conclude that the fewer-comparisons algorithm shows poorer performance because the iterator accesses have less locality. See Section 4.2.2 for how this can be improved.

2.6.5 Detailed Operation Counting

The operation counter `smatcount` provides more detailed information with the `-v` option. In the detailed format, operation counts are decomposed into the operator function level. For example, the one-pass algorithm processing 10000 elements generates the following.

```
$ ./example2 onepass 10000 2>/dev/null | smatopt | smatcount -v
Iterator:
  Ctor (copy)       2
  Ctor (base type)  2
```

```
    Dtor              4
    =                 17
    !=                10000
    ++                10000

  Value type:
    <                 19990

  Total:
    Ctor (copy)       2
    Ctor (base type)  2
    Dtor              4
    =                 17
    !=                10000
    <                 19990
    ++                10000
```

In the output, `Ctor (default)`, `Ctor (copy)`, `Ctor (base type)` and `Dtor` correspond to the default constructor, copy constructor, conversion constructor from the base type, and destructor, respectively. It should be clear that other symbols, such as `=`, `!=`, `++` and `<`, correspond to operator functions.

The option `-vv` genetates the output in much longer format, in which *all* operations are listed, even unused operators having 0 counts.

Chapter 3

Reference Manual

This chapter is the reference manual for four SMAT adaptor classes `smat`, `smat_d`, `smat_v`, `smat_p`, SMAT trace instruction codes, and five utility programs `smatsim`, `smatopt`, `smatcount`, `smatview`, and `smatplot`.

3.1 Iterator Adaptor

3.1.1 Files

```
#include "smat.h"
```

3.1.2 Class Declaration

```
template <typename T>
class smat
```

3.1.3 Description

SMAT iterator adaptor class `smat` wraps the base iterator type `T`, which is given as the template parameter, and emulates `T`. At the same time, it generates a memory access trace of almost all operations by overloading them. Overloaded functions (operators, member functions, and global operations) are assigned a trace instruction code that identifies the type of memory access. The trace code assigned is shown at the end of each function description. See Table 3.1 for the complete list of trace instruction codes.

3.1.4 Type Definitions

`value_type`

> Type of the object to which the iterator points. This type is also wrapped by the value type adaptor of SMAT. Defined as `smat_v<iterator_traits<T>::value_type>`.

`difference_type`

> Type that can be used to represent the distance between two iterators, or the number of elements in a range. Defined as `smat_d<iterator_traits<T>::difference_type>`.

`pointer`

Pointer type to `value_type`. Defined as `smat_p<iterator_traits<T>::value_type>`.

`reference`

Reference type to `value_type`. Defined as `smat_v<iterator_traits<T>::value_type>&`.

`iterator_category`

Iterator category, defined as `iterator_traits<T>::iterator_category`.

3.1.5 Constructors, Destructors, and Related Functions

`smat();`

Default constructor. `SMAT_CTOR`.

`smat(const smat<T>& x);`

Copy constructor. `SMAT_CCTOR`.

`smat(const T& x);`

Constructs a `smat<T>` object from the base iterator iterator `x`. `SMAT_BCTOR`.

`operator T() const;`

Conversion operator to the base iterator type. `SMAT_READ`.

`smat<T>& operator=(const smat<T>& x)`

Assignment operator. `SMAT_MOV`.

`~smat();`

Destructor. `SMAT_DTOR`.

3.1.6 Public Member Functions

`const T& base() const;`

Returns the const reference to the base iterator object.

```
smat<T>& operator++();
smat<T> operator++(int);
```

Prefix and postfix increment operator. `SMAT_INC`.

```
smat<T>& operator--();
smat<T> operator--(int);
```

Prefix and postfix decrement operator. `SMAT_DEC`.

```
smat<T>& operator+=(const difference_type& x);
smat<T>& operator-=(const difference_type& x);
```

Plus and minus assignment operators, which take a difference type argument. `SMAT_ADD` and `SMAT_SUB`, respectively.

```
template <typename U> smat<T>& operator+=(const U& x);
template <typename U> smat<T>& operator-=(const U& x);
```

Plus and minus assignment operators, which take an argument of other than a difference type. In fact, they assume that `x` is an integer constant. Therefore, operators are assigned the trace instruction code `SMAT_IADD` and `SMAT_ISUB`, respectively.

```
T operator->() const;
```

Dereferencing operator. Returns `base()`. `SMAT_READ`.

```
reference operator*() const;
```

Dereferencing operator. Returns `*base()`. `SMAT_READ`.

```
reference operator[](const difference_type& x) const;
template <typename U> reference operator[](const U& x) const;
```

Returns `*(*this + x)`.

3.1.7 Global Operations

```
template <typename T, typename U>
const smat<T> operator+(const smat<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat<T> operator+(const U& lhs, const smat<T>& rhs);
```

Returns `lhs.base() + rhs` and `lhs + rhs.base()`, respectively. `SMAT_IADD`.

```
template <typename T, typename U>
const smat<T> operator-(const smat<T>& lhs, const U& rhs);
```

Returns `lhs.base() - rhs`. `SMAT_ISUB`.

```
template <typename T>
const smat<T>
operator+(const smat<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs);
template <typename T>
const smat<T>
operator+(const smat_d<typename std::iterator_traits<T>::difference_type>& lhs,
          const smat<T>& rhs);
```

Returns `lhs.base() + rhs.base()`. `SMAT_ADD`.

```
template <typename T>
const smat<T>
operator-(const smat<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs);
```

Returns `lhs.base() - rhs.base()`. `SMAT_SUB`.

```
template <typename T>
const smat_d<typename std::iterator_traits<T>::difference_type>
operator-(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() - rhs.base()`. `SMAT_SUB`.

3.1.8 Equality and Ordering Predicates

```
template <typename T>
bool operator==(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() == rhs.base()`. `SMAT_EQ`.

```
template <typename T>
bool operator!=(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() != rhs.base()`. `SMAT_NEQ`.

```
template <typename T>
bool operator<(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() < rhs.base()`. `SMAT_LT`.

```
template <typename T>
bool operator>(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() > rhs.base()`. `SMAT_GT`.

```
template <typename T>
bool operator<=(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() <= rhs.base()`. `SMAT_LEQ`.

```
template <typename T>
bool operator>=(const smat<T>& lhs, const smat<T>& rhs);
```

Returns `lhs.base() >= rhs.base()`. `SMAT_GEQ`.

3.2 Difference Type Adaptor

3.2.1 Files

```
#include "smat.h"
```

3.2.2 Class Declaration

```
template <typename T>
class smat_d
```

3.2.3 Description

The difference type adaptor class `smat_d` wraps the base difference type `T`, which is given as the template parameter, and emulates `T`. At the same time, it generates a memory access trace of almost all operations by overloading them. The trace codes assigned to overloaded functions are shown at the end of each function description.

Since the base type `T` is usually defined as `int` or `long int`, `smat_d` overloads all possible operations that can be applied to these integral types.

3.2.4 Constructors, Destructors, and Related Functions

```
smat_d();
```

Default constructor. `SMAT_CTOR`.

```
smat_d(const smat_d& x);
```

Copy constructor. `SMAT_CCTOR`.

```
smat_d(const T& x);
```

Constructs a `smat_d<T>` object from the base difference type object `x`. `SMAT_BCTOR`.

```
operator T() const;
```

Conversion operator to the base difference type. `SMAT_READ`.

```
smat_d<T>& operator=(const smat_d<T>& x)
```

Assignment operator. `SMAT_MOV`.

```
~smat_d();
```

Destructor. `SMAT_DTOR`.

3.2.5 Public Member Functions

```
const T& base() const;
```

Returns the const reference to the base difference type object.

```
smat_d<T>& operator++();
smat_d<T> operator++(int);
```

Prefix and postfix increment operator. `SMAT_INC`.

```
smat_d<T>& operator--();
smat_d<T> operator--(int);
```

Prefix and postfix decrement operator. `SMAT_DEC`.

```
smat_d<T>& operator+=(const smat_d<T>& x);
```

Calls `base() += x.base()` and returns `*this`. `SMAT_ADD`.

```
smat_d<T>& operator-=(const smat_d<T>& x);
```

Calls `base() -= x.base()` and returns `*this`. `SMAT_SUB`.

```
smat_d<T>& operator*=(const smat_d<T>& x);
```

Calls `base() *= x.base()` and returns `*this`. `SMAT_MUL`.

```
smat_d<T>& operator/=(const smat_d<T>& x);
```

Calls `base() /= x.base()` and returns `*this`. `SMAT_DIV`.

```
smat_d<T>& operator%=(const smat_d<T>& x);
```

Calls `base() %= x.base()` and returns `*this`. `SMAT_MOD`.

```
smat_d<T>& operator<<=(const smat_d<T>& x);
```

Calls `base() <<= x.base()` and returns `*this`. `SMAT_LSHIFT`.

```
smat_d<T>& operator>>=(const smat_d<T>& x);
```

Calls `base() >>= x.base()` and returns `*this`. `SMAT_RSHIFT`.

```
smat_d<T>& operator&=(const smat_d<T>& x);
```

Calls `base() &= x.base()` and returns `*this`. `SMAT_AND`.

```
smat_d<T>& operator|=(const smat_d<T>& x);
```

Calls `base() |= x.base()` and returns `*this`. `SMAT_OR`.

```
smat_d<T>& operator^=(const smat_d<T>& x);
```

Calls `base() ^= x.base()` and returns `*this`. `SMAT_XOR`.

```
template <typename U> smat_d<T>& operator+=(const U& x);
```

Calls `base() += x` and returns `*this`. `SMAT_IADD`.

```
template <typename U> smat_d<T>& operator-=(const U& x);
```

Calls `base() -= x` and returns `*this`. `SMAT_ISUB`.

```
template <typename U> smat_d<T>& operator*=(const U& x);
```

Calls `base() *= x` and returns `*this`. `SMAT_IMUL`.

```
template <typename U> smat_d<T>& operator/=(const U& x);
```

Calls `base() /= x` and returns `*this`. `SMAT_IDIV`.

```
template <typename U> smat_d<T>& operator%=(const U& x);
```

Calls `base() %= x` and returns `*this`. `SMAT_IMOD`.

```
template <typename U> smat_d<T>& operator<<=(const U& x);
```

Calls `base() <<= x` and returns `*this`. `SMAT_ILSHIFT`.

```
template <typename U> smat_d<T>& operator>>=(const U& x);
```

Calls `base() >>= x` and returns `*this`. `SMAT_IRSHIFT`.

```
template <typename U> smat_d<T>& operator&=(const U& x);
```

Calls `base() &= x` and returns `*this`. `SMAT_IAND`.

```
template <typename U> smat_d<T>& operator|=(const U& x);
```

Calls `base() |= x` and returns `*this`. `SMAT_IOR`.

```
template <typename U> smat_d<T>& operator^=(const U& x);
```

Calls `base() ^= x` and returns `*this`. `SMAT_IXOR`.

```
smat_d<T>* operator&();
const smat_d<T>* operator&() const;
```

Address-of operator. This operator does not have a trace instruction code because no memory reference is involved in returning a pointer.

```
smat_d<T> operator+() const;
```

Returns `smat_d<T>(base())`. The assigned trace instruction code is `SMAT_READ`, since it is not that interesting to trace the unary plus operator.

```
smat_d<T> operator-() const;
```

Returns `smat_d<T>(-base())`. `SMAT_MINUS`.

```
smat_d<T> operator~() const;
```

Returns `smat_d<T>(~base())`. `SMAT_CMPL`.

3.2.6 Global Operations

```
template <typename T>
const smat_d<T> operator+(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Addition operator between two `smat_d<T>` objects. Returns `lhs.base() + rhs.base()`. `SMAT_ADD`.

```
template <typename T, typename U>
const smat_d<T> operator+(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator+(const U& lhs, const smat_d<T>& rhs);
```

Addition operators between `smat_d<T>` and an integral constant. Return `lhs.base() + rhs` and `lhs + rhs.base()`, respectively. `SMAT_IADD`.

```
template <typename T>
const smat_d<T> operator-(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() - rhs.base()`. `SMAT_SUB`.

```
template <typename T, typename U>
const smat_d<T> operator-(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator-(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() - rhs` and `lhs - rhs.base()`, respectively. `SMAT_ISUB`.

```
template <typename T>
const smat_d<T> operator*(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() * rhs.base()`. `SMAT_MUL`.

```
template <typename T, typename U>
const smat_d<T> operator*(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator*(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() * rhs` and `lhs * rhs.base()`, respectively. `SMAT_IMUL`.

```
template <typename T>
const smat_d<T> operator/(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() / rhs.base()`. `SMAT_DIV`.

```
template <typename T, typename U>
const smat_d<T> operator/(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator/(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() / rhs` and `lhs / rhs.base()`, respectively. `SMAT_IDIV`.

```
template <typename T>
const smat_d<T> operator%(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() % rhs.base()`. `SMAT_MOD`.

```
template <typename T, typename U>
const smat_d<T> operator%(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator%(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() % rhs` and `lhs % rhs.base()`, respectively. `SMAT_IMOD`.

```
template <typename T>
const smat_d<T> operator<<(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() << rhs.base()`. `SMAT_LSHIFT`.

```
template <typename T, typename U>
const smat_d<T> operator<<(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator<<(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() << rhs` and `lhs << rhs.base()`, respectively. `SMAT_ILSHIFT`.

```
template <typename T>
const smat_d<T> operator>>(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() >> rhs.base()`. `SMAT_RSHIFT`.

```
template <typename T, typename U>
const smat_d<T> operator>>(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator>>(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() >> rhs` and `lhs >> rhs.base()`, respectively. `SMAT_IRSHIFT`.

```
template <typename T>
const smat_d<T> operator&(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() & rhs.base()`. `SMAT_AND`.

```
template <typename T, typename U>
const smat_d<T> operator&(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator&(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() & rhs` and `lhs & rhs.base()`, respectively. `SMAT_IAND`.

```
template <typename T>
const smat_d<T> operator|(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() | rhs.base()`. `SMAT_OR`.

```
template <typename T, typename U>
const smat_d<T> operator|(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator|(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() | rhs` and `lhs | rhs.base()`, respectively. `SMAT_IOR`.

```
template <typename T>
const smat_d<T> operator^(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() ^ rhs.base()`. `SMAT_XOR`.

```
template <typename T, typename U>
const smat_d<T> operator^(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U>
const smat_d<T> operator^(const U& lhs, const smat_d<T>& rhs);
```

Return `lhs.base() ^ rhs` and `lhs ^ rhs.base()`, respectively. `SMAT_IXOR`.

3.2.7 Equality and Ordering Predicates

```
template <typename T>
bool operator==(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() == rhs.base()`. `SMAT_EQ`.

```
template <typename T>
bool operator!=(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() != rhs.base()`. `SMAT_NEQ`.

```
template <typename T>
bool operator<(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() < rhs.base()`. `SMAT_LT`.

```
template <typename T>
bool operator>(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() > rhs.base()`. `SMAT_GT`.

```
template <typename T>
bool operator<=(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() <= rhs.base()`. `SMAT_LEQ`.

```
template <typename T>
bool operator>=(const smat_d<T>& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() >= rhs.base()`. `SMAT_GEQ`.

```
template <typename T, typename U> bool operator==(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator==(const U& lhs, const smat_d<T>& rhs);
```

Equality test between integral constants. Returns `lhs.base() == rhs` and `lhs == rhs.base()`, respectively. `SMAT_IEQ`.

```
template <typename T, typename U> bool operator!=(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator!=(const U& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() != rhs` and `lhs != rhs.base()`, respectively. `SMAT_INEQ`.

```
template <typename T, typename U> bool operator<(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator<(const U& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() < rhs` and `lhs < rhs.base()`, respectively. `SMAT_ILT`.

```
template <typename T, typename U> bool operator>(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator>(const U& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() > rhs` and `lhs > rhs.base()`, respectively. `SMAT_IGT`.

```
template <typename T, typename U> bool operator<=(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator<=(const U& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() <= rhs` and `lhs <= rhs.base()`, respectively. `SMAT_ILEQ`.

```
template <typename T, typename U> bool operator>=(const smat_d<T>& lhs, const U& rhs);
template <typename T, typename U> bool operator>=(const U& lhs, const smat_d<T>& rhs);
```

Returns `lhs.base() >= rhs` and `lhs >= rhs.base()`, respectively. `SMAT_IGEQ`.

3.3 Value Type Adaptor

3.3.1 Files

```
#include "smat.h"
```

3.3.2 Class Declaration

```
template <typename T>
class smat_v
```

3.3.3 Description

The value type adaptor class `smat_v` wraps the base value type `T`, and emulates `T`. At the same time, it generates a memory access trace of almost all operations by overloading them.

Further description is omitted because the interface of `smat_v` is exactly the same as the difference type adaptor `smat_d`, except that `smat_d` has the addition operators with iterator types and that the subtraction between iterators yields type `smat_d`.

3.4 Pointer Type Adaptor

3.4.1 Files

```
#include "smat.h"
```

3.4.2 Class Declaration

```
template <typename T>
class smat_p
```

3.4.3 Description

The pointer type adaptor class `smat_p` wraps the base pointer type `T` and emulates `T`. At the same time, it generates a memory access trace of almost all operations by overloading them.

Most of the descriptions of functions of this class are omitted because they are identical to those of `smat`. Only functions that are not defined in `smat` are described below.

3.4.4 Public Member Functions

```
smat_p<T>& operator=(smat_v<T>* x)
```

Assignment operator from the pointer of the value type object. `SMAT_MOV`.

3.5 Trace Instruction Code

3.5.1 Files

```
#include "smat.h"
```

3.5.2 Class Declaration

```
enum smat_code
```

3.5.3 Description

The `smat_code` enumeration defines constants that identify each trace instruction. Although this enumeration is only used internally, the correspondence between the trace instruction codes and SMAT adaptor library functions is important to understanding how SMAT works.

Briefly, every SMAT adaptor library function described in preceding sections has a corresponding trace instruction code and generates the trace of the assigned code at every invocation.

Some instructions have their "immediate version" counterparts. For example, `SMAT_IADD` is the immediate version of `SMAT_ADD`. The immediate version is generated by the expression in which one of the operands is an immediate value, or a constant. For example, an expression `i + 1`, where `i` is an iterator, generates the instruction code `SMAT_IADD`.

3.5.4 Enumerator Constants

`SMAT_CCTOR`

> Generated by copy constructors.

`SMAT_BCTOR`

> Generated by conversion operators from the base type objects.

`SMAT_CTOR`

> Generated by default constructors. This instruction is not printed by `smatview`. (Use option `-v` to print).

`SMAT_DTOR`

> Generated by destructors.

`SMAT_MOV`

> Generated by assignment operators.

`SMAT_AND`
`SMAT_IAND`

> Generated by bitwise and operators. `SMAT_IAND` is generated when one of the operands is an immediate value.

`SMAT_OR`
`SMAT_IOR`

> Generated by bitwise or operators.

`SMAT_XOR`
`SMAT_IXOR`

> Generated by bitwise xor operators.

`SMAT_INC`

Generated by both prefix and postfix increment operators.

`SMAT_DEC`

Generated by both prefix and postfix decrement operators.

`SMAT_CMPL`

Generated by complement operators.

`SMAT_MINUS`

Generated by unary minus operators.

`SMAT_ADD`
`SMAT_IADD`

Generated by addition operators.

`SMAT_SUB`
`SMAT_ISUB`

Generated by subtraction operators.

`SMAT_MUL`
`SMAT_IMUL`

Generated by multiplication operators.

`SMAT_DIV`
`SMAT_IDIV`

Generated by division operators.

`SMAT_MOD`
`SMAT_IMOD`

Generated by modulo operators.

`SMAT_LSHIFT`
`SMAT_ILSHIFT`

Generated by left shift operators.

`SMAT_RSHIFT`
`SMAT_IRSHIFT`

Generated by right shift operators.

`SMAT_EQ`
`SMAT_IEQ`

Generated by equality operators.

`SMAT_NEQ`
`SMAT_INEQ`

Generated by inequality operators.

`SMAT_GT`
`SMAT_IGT`

Generated by greater-than operators.

`SMAT_GEQ`
`SMAT_IGEQ`

Generated by greater-than-or-equal-to operators.

`SMAT_LT`
`SMAT_ILT`

Generated by less-than operators.

`SMAT_LEQ`
`SMAT_ILEQ`

Generated by less-than-or-equal-to operators.

`SMAT_NOP`

Generated when the trace file optimizer has eliminated an instruction. This instruction is not printed by `smatview`. (Use option `-v` to print.)

`SMAT_READ`

Indicates arbitrary memory read. Generated by dereference, subscripting, arrow, unary plus operators, and conversion operators to the base types.

`SMAT_WRITE`

Indicates arbitrary memory write. Currently not used.

`SMAT_FMARK`

Places some mark in trace files. Currently not used.

3.5.5 Trace Instruction Summary

Table 3.1 is the complete list of trace instructions. The table consists of the instruction code, mnemonic (used in `smatview`), the number of source addresses and destination addresses, corresponding operator function, and notes.

Table 3.1: Trace instruction summary.

Code	Mnemonic	Src.[†]	Dest.[‡]	Operator	Notes
SMAT_CCTOR	cctor	1	1		copy constructor
SMAT_BCTOR	bctor	1	1		conversion from the base type
SMAT_CTOR	ctor	0	0		default constructor
SMAT_DTOR	dtor	1	0		destructor
SMAT_MOV	mov	1	1	=	assignment
SMAT_AND	and	2	1	&	bitwise and
SMAT_OR	or	2	1	|	bitwise or
SMAT_XOR	xor	2	1	^	bitwise xor
SMAT_IAND	iand	1	1	&	bitwise and with immediate value
SMAT_IOR	ior	1	1	|	bitwise or with immediate value
SMAT_IXOR	ixor	1	1	^	bitwise xor with immediate value
SMAT_INC	inc	1	1	++	increment (both pre and post)
SMAT_DEC	dec	1	1	--	decrement (both pre and post)
SMAT_CMPL	cmpl	1	1	~	unary bitwise complement
SMAT_MINUS	minus	1	1	- (unary)	unary minus
SMAT_ADD	add	2	1	+	addition
SMAT_SUB	sub	2	1	-	subtraction
SMAT_MUL	mul	2	1	*	multiplication
SMAT_DIV	div	2	1	/	division
SMAT_MOD	mod	2	1	%	modulo
SMAT_LSHIFT	sl	2	1	<<	left shift
SMAT_RSHIFT	sr	2	1	>>	right shift
SMAT_IADD	iadd	1	1	+	addition with immediate value
SMAT_ISUB	isub	1	1	-	subtraction with immediate value
SMAT_IMUL	imul	1	1	*	multiplication with immediate value
SMAT_IDIV	idiv	1	1	/	division with immediate value
SMAT_IMOD	imod	1	1	%	modulo with immediate value
SMAT_ILSHIFT	isl	1	1	<<	left shift with immediate value
SMAT_IRSHIFT	isr	1	1	>>	right shift with immediate value
SMAT_EQ	eq	2	0	==	equality
SMAT_NEQ	neq	2	0	!=	inequality
SMAT_GT	gt	2	0	>	greater than
SMAT_GEQ	geq	2	0	>=	greater than or equal to
SMAT_LT	lt	2	0	<	less than
SMAT_LEQ	leq	2	0	<=	less than or equal to
SMAT_IEQ	ieq	1	0	==	comparison with immediate value
SMAT_INEQ	ineq	1	0	!=	likewise
SMAT_IGT	igt	1	0	>	likewise
SMAT_IGEQ	igeq	1	0	>=	likewise
SMAT_ILT	ilt	1	0	<	likewise
SMAT_ILEQ	ileq	1	0	<=	likewise
SMAT_NOP	nop	0	0		do nothing
SMAT_READ	read	1	0	*, [], ->	read operation
SMAT_WRITE	write	0	1		not used
SMAT_FMARK	mark	0	0		not used

[†] The number of source addresses.

[‡] The number of destination addresses.

3.5.6 Expression Examples and Corresponding Traces

For a number of statements and expressions, their corresponding traces are presented. “Pseudo” trace instructions are used in which the address of object `i` is shown as `(i)` instead of its raw address. Additionally the following assumptions are made.

- `Iter` denotes the SMAT-adapted iterator type.
- `i` and `j` denote variables of type `Iter`.
- `d` and `n` denote variables of `Iter`’s difference type.
- `v` denotes a variable of `Iter`’s value type.

```
Iter i(j);

     cctori  4, (j), (i)             ; copy j to i

j = i + 1;

     iaddi   4, (i), (tmp)           ; i + 1 -> tmp
     movi    4, (tmp), (j)           ; tmp -> j
     dtori   4, (tmp)                ; destruct tmp

j >= i + 100

     iaddi   4, (i), (tmp)           ; i + 100 -> tmp
     gteqi   4, (j), (tmp)           ; j >= tmp
     dtori   4, (tmp)                ; destruct tmp

d = j - i;

     subi    4, (j), (i), (tmp)      ; j - i -> tmp
     movd    4, (tmp), (d)           ; tmp -> d
     dtord   4, (tmp)                ; destruct tmp

n = d * 2;

     imuld   4, (d), (tmp)           ; d * 2 -> tmp
     movd    4, (tmp), (n)           ; tmp -> n
     dtord   4, (tmp)                ; destruct tmp

d *= 2;

     imuld   4, (d), (d)             ; d * 2 -> d

v = *i;

     readi   4, (i)                  ; read from i
     movv    4, (*i), (v)            ; copy *i to v

v = *(i + 1);

     iaddi   4, (i), (tmp)           ; i + 1 -> tmp
     readi   4, (tmp)                ; read from tmp
     movv    4, (*tmp), (v)          ; copy *tmp to v
     dtori   4, (tmp)                ; destruct tmp
```

3.6 Cache Simulator

3.6.1 Synopsis

```
smatsim [OPTION]... < TRACEFILE
```

3.6.2 Description

The cache simulator for SMAT. It reads a trace file from the standard input, performs the cache simulation, and then outputs the cache statistics to the standard output stream. Cache parameters are given at compile-time. See Section 2.6.3 for how to change cache parameters.

3.6.3 Options

`-c`

Display cache parameters and exit.

`-d`

Generate debug outputs. With `-d` option, it prints every trace instruction simulated, as well as the current clock count. With `-dd` option, it also dumps all non-empty cache block information after each step, in a format like `*bffff780-9f:60:238`. The first asterisk indicates the block is dirty. The next hexadecimal numbers are the address range to which the block corresponds. The next decimal number (`60`) is the set index. The last number (`238`) is the time in clock cycles when the block is last used. Some entries might have `<--` at the end of the line, which means that the entry is the oldest and will be replaced next.

`-g FILE`

Output time-position information to `FILE`, which is for use by `smatrange`, a helper utility program of `smatplot`.

`-h`

Display a help message and exit.

`-t`

Perform type analysis. The numbers of cache misses, cache hits, and clock cycles are counted and reported separately (in the simple form) for the iterator type and its associated types.

`-v`

Generate detailed cache statistics. Section 2.6.4 explains the detailed output.

3.7 Trace File Optimizer

3.7.1 Synopsis

```
smatopt [OPTION]... < TRACEFILE
```

3.7.2 Description

The trace file optimizer `smatopt` eliminates inefficient memory accesses in a trace file. It reads a trace file from the standard input stream, and outputs the optimized trace file to the standard output stream. Section 4.2 explains the optimizations it performs.

3.7.3 Options

`-h`

Display a help message and exit.

3.8 Operation Counter

3.8.1 Synopsis

```
smatcount [OPTION...] < TRACEFILE
```

3.8.2 Description

The operation counter `smatcount` counts the numbers of operations in a trace file and reports them separately for the iterator type and its associated types. It reads a trace file from the standard input stream and outputs the report to the standard output stream.

By default, `smatcount` classifies operations into three categories: assignment, comparison, and arithmetic operations. The option `-v` generates more detailed information and `-vv` generates even more. Section 2.6.5 presents an example of using `-v` option. It reports only for types with non-zero operation counts. However, if one of options `-I`, `-D`, `-V`, `-P`, and `-T` is specified, the program enters a selective mode, in which the only specified types are reported.

3.8.3 Options

`-D`

Enter a selective mode and output operation counts for a difference type.

`-I`

Enter a selective mode and output operation counts for an iterator type.

`-h`

Display the help and exit.

`-P`

Enter a selective mode and output operation counts for a pointer type.

`-T`

Enter a selective mode and output total operation counts.

`-V`

Enter a selective mode and output operation counts for a value type.

`-v`

Generate detailed operation counting information. Specifying `-vv` generates more.

3.9 Trace File Viewer

3.9.1 Synopsis

```
smatview [OPTION]... < TRACEFILE
```

3.9.2 Description

Trace file viewer `smatview` inputs the trace file and outputs every trace instruction in a human-readable format. Section 2.6.2 explains the output format.

3.9.3 Options

`-h`

Display the help and exit.

`-v`

Output `SMAT_NOP` and `SMAT_CTOR` as well, which may not be interesting to most users.

3.10 Trace File Plotter

3.10.1 Synopsis

```
smatplot [OPTION]... < TRACEFILE
```

3.10.2 Description

The trace file plotter `smatplot` generates a trace plot for the given trace file. It reads a trace file from the standard input and generates the plot file `tmp.eps` in the EPS (Encapsulated PostScript) format, by using `gnuplot`'s postscript driver. The output file and format can be changed by option `-o`. The supported formats are the EPS, PBM, and PNG format. For example, with option `-o x.png`, it would generate the output file `x.png` in the PNG format.

The trace may contain several address regions, such as the heap and stack. The plotter `smatplot` by default generates a plot for the address range with the *smallest* address. It usually generates plots for the heap area, since the address of the heap area is usually smaller than that of stack. If the result does not seem to be correct, try option `-a` to examine address regions of a trace file, and then try option `-A` to explicitly specify the address region.

With option `-c`, `smatplot` outputs the command file and data file for `gnuplot`. So the advanced user can edit the command file and run `gnuplot` to create another plot.

By default, it plots the entire clock and address ranges. With options `-x` and `-y`, the clock range and address range can be specified.

3.10.3 Options

`-A PREFIX`

Plot the address range which has `PREFIX`. For example, if `-A bf` is specified, it plots traces in a range `0xbf000000-0xbfffffff`, if any. The prefix `PREFIX` must be two hexadecimal digits. The next option `-a` provides a hint as to what address to specify.

`-a`

Display address ranges that the given trace file contains. Each line of the output looks like `bf: 0xbfffec40-0xbffff7a8`, so a user can specify `-A bf` to plot that range.

`-c NAME`

Generate command file `NAME.cmd` and data file `NAME.dat` for `gnuplot`. A user can edit the command file and run `gnuplot` to produce another plot from the data file.

`-h`

Display the help and exit.

`-o FILE.EXT`

Specify the output file and its format, instead of the default output file `tmp.eps`. The extension must be one of `eps`, `pbm`, and `png`.

`-x FROM:TO`

Specify the clock range in decimal integers, `-x 20000:25000`, for example.

`-y FROM:TO`

Specify the address range in hexadecimal numbers, `-y 0x08082290:0x08083a00`, for example. This option precedes option `-A`.

`-X NUM`

Specify xtics of a graph.

Chapter 4

Design Issues

This chapter discusses the design issues of SMAT. There have been two primary difficulties. The first concerns the cache simulator, which is discussed in Section 4.1. The second concerns inefficiencies resulting from the source-level instrumentation. Section 4.2 describes the types of inefficiencies and how the can in part be eliminated by the trace file optimizer. The trace file format and its size problem are discussed in Section 4.3. Miscellaneous issues related to the use the GCC version 3.2 compiler are discussed in Section 4.4.

4.1 Cache Simulator

The cache simulator is one of the most important components in SMAT. The simulator reads a trace file and simulates cache behavior that would be caused by memory accesses in the trace file. Since a trace file only contains the starting address and access length, the simulator does not perform actual memory transfer. Instead, it simulates the cache behavior just by updating the address tag and other information associated with cache blocks. At the end of the simulation, it reports the cache statistics it has gathered, such as the number of clock cycles, cache misses, etc.

4.1.1 Design Decisions

Many design decisions were made during the development of cache simulator. Two of the most important decisions are discussed below.

Simulating Registers

SMAT unlike most trace-driven cache simulators available today, simulates registers. To understand why, first note that most trace-driven cache simulators generate traces by object modification, which embeds profiling routines and trace function calls into an executable file. This method does not trace objects that are assigned to registers, and only traces objects that are not assigned to registers, that is, the real memory accesses. Thus the cache simulator only simulates caches and real memory.

On the other hand, source-level instrumentation, on which SMAT is based, traces *all* objects. Therefore, in order to make the source-based cache simulation comparable to object-based cache simulation, the simulator must emulate registers as well as cache memory. Otherwise, a source-based trace would appear to exhibit very poor cache performance.

Registers in the simulator are modeled, in fact, simply as a *level zero* cache, which is fully associative and has 4-byte blocks and the write back policy. Having zero clock hit time, it costs nothing at all if the requested address is found in a register set. The register allocation algorithm is, however, Least Recently Used, so it might not be as efficient as the optimized register allocation performed by a compiler. SMAT cache simulator tries to improve the register utilization by the explicit register invalidation, which is described later in this section.

Simple Model

The SMAT cache simulator only provides a simple, one level cache, whereas most microprocessors today have multi-level, non-blocking caches with write buffers. A user may doubt whether such a simple cache could be used to evaluate cache performance accurately.

The answer is yes, or rather, the simple cache model can be more suitable for cache-conscious algorithm studies than the more powerful, complicated cache models. Multi-level caches, non-blocking caches, and write buffers are all techniques to reduce the cache miss penalty [10, p. 449]. Caches with these techniques may decrease or hide the miss cost partially. On the other hand, these techniques hide the inefficient memory accesses of an algorithm as well and make them harder to be observed. With a simple cache, the inefficient accesses directly cause cache misses which helps to analyze cache behavior. Therefore, the simple cache model often suffices in basic cache-conscious algorithm studies, especially for novice users.

However, more advanced users will no doubt be interested in the behavior of more sophisticated caches. For this reason, the implementation of the cache simulator was designed to be independent of the SMAT adaptors, so that it can be improved independently. We would like to see improvements of the simulator carried out in future work.

4.1.2 How It Works

The cache simulator interprets a trace file simply as a sequence of requests of three kinds: read request, write request, and register invalidation request. For example, trace instructions

```
iaddi   4, (0xbffff8b4), (0xbffff884)
readi   4, (0xbffff884)
movv    4, (0xbffff8c8), (0xbffff894)
dtori   4, (0xbffff884)
```

are interpreted as follows.

1. Read 4 bytes from `0xbffff8b4`.
2. Write 4 bytes to `0xbffff884`.
3. Read 4 bytes from `0xbffff884`.
4. Read 4 bytes from `0xbffff8c8`.
5. Write 4 bytes to `0xbffff894`.
6. Invalidate the register which has 4 bytes from `0xbffff884`.

Read Requests

In the simulator, a read request is satisfied by "transferring" the requested memory region into the register set. As mentioned, however, the simulator does not perform actual memory transfer. Instead, it updates the tag field in the cache block to simulate memory transfer.

A read request is, very briefly, processed as follows. For simplicity, it is assumed that the address starts from a 4-byte boundary and the access length is also four bytes.

```
 1: Find the requested address in the register set.
 2: if found then
 3:   return
 4: else
 5:   Allocate a register for the request. (Discard the least-recently-used register)
 6: end if
 7: Find the requested address in the cache.
 8: if found then
 9:   Transfer the requested data from the cache to the allocated register.
10:   return
11: else
12:   Allocate a block in the set for the request. (Discard the least-recently-used block)
13: end if
14: Transfer the requested data from main memory to the allocated cache block.
15: Transfer the requested data from the block to the allocated register as well.
```

At first, the requested address is searched for in the register set (line 1). If found, the access is satisfied immediately (lines 3). If not found, the requested data should be brought to one of the registers. Then, a register must be allocated (line 5). The requested address is searched for in the cache next (line 7). If found in the cache, it is "transferred" to the register and the request is satisfied (lines 9–10). If it is not found in the cache, a cache block must be allocated to hold requested address block (line 12). The access request is finally sent to main memory, and the allocated cache block is filled with the requested memory region (line 14). The access completes by further transferring requested memory to the register (line 15).

Note that the allocations performed in lines 5 and 12 might involve a writeback operation because if the least-recently-used register (block) is dirty, it must be written back to the lower-level memory. Otherwise, the change made in the register (block) would be lost.

More precisely, a read request is handled by three layers of functions: CACHE-READ-REGISTER, CACHE-READ-L1, and CACHE-READ-MAIN.

The function CACHE-READ-REGISTER performs a lookup in the register set and forwards the request to L1 cache if not found. It takes the starting address *addr*, access length *len* and the current time *time* represented in clocks, and returns the finishing time. The implicit parameters $\mathsf{hit\_time}_{\mathsf{reg}}$ and $\mathsf{block\_size}_{\mathsf{reg}}$ are constants given at compile time, and $\mathsf{hit\_time}_{\mathsf{reg}} \leftarrow 0$, $\mathsf{block\_size}_{\mathsf{reg}} \leftarrow 4$, by default. This function assumes that the access does not span the ($\mathsf{block\_size}_{\mathsf{reg}}$)-byte boundary.

```
CACHE-READ-REGISTER($addr, len, time$)
 1: $tag \leftarrow$ a tag field of $addr$
 2: $r \leftarrow$ lookup $tag$ in the register set
 3: if $r \neq$ NIL then
 4:   $r$.reftime $\leftarrow time + \mathsf{hit\_time}_{\mathsf{reg}}$
 5: else
 6:   $r \leftarrow$ the least-recently-used register
 7:   if $r$.dirty = true then
 8:     $rtop \leftarrow$ the address which $r$ contains
 9:     $time \leftarrow$ CACHE-WRITE-L1($rtop, \mathsf{block\ size}_{\mathsf{reg}}, time$)
```

```
10:     end if
11:     top ← addr rounded down to a multiple of $\mathsf{block\_size_{reg}}$
12:     time ← CACHE-READ-L1($top$, $\mathsf{block\_size_{reg}}$, $time$)
13:     $r$.tag ← $tag$
14:     $r$.dirty ← false
15:     $r$.reftime ← $time + \mathsf{hit\_time_{reg}}$
16: end if
17: return $time + \mathsf{hit\_time_{reg}}$
```

The function starts by computing a tag (line 1), and the register lookup is performed using the tag (line 2). If found, it updates the reference time of the register (line 4). If not found, the least-recently-used register to be replaced is located (line 6). The contents of the register is written back to the L1 cache if it is dirty (lines 7–10), in which case the write request is issued *blockwise*. That is, the starting address of the write request starts from the ($\mathsf{block\_size_{reg}}$)-byte boundary and the access length is the block size. The parameter *time* is passed to CACHE-WRITE-L1, which updates and returns *time*. Then, the read request is issued to L1 cache, again blockwise (lines 11–12). The tag field, dirty flag, and reference time are updated (lines 13–15), and the function returns the updated *time*.

The next function CACHE-READ-L1 looks quite similar to CACHE-READ-REGISTER but there are some differences. It computes the index field as well as the tag field, and the block lookup is performed in the ith set (lines 1–3). The read and write requests are now issued to main memory, which are handled by functions CACHE-READ-MAIN and CACHE-WRITE-MAIN, respectively. These two functions just return $time + \mathsf{hit\ time_{main}}$, in which the $\mathsf{hit\ time_{main}}$ is a hit time of main memory represented in clocks per block. This works because CACHE-READ-L1 issues accesses blockwise.

```
CACHE-READ-L1($addr$, $len$, $time$)
 1: $tag$ ← a tag field of $addr$
 2: $i$ ← an index field of $addr$
 3: $r$ ← lookup $tag$ in the $i$th set
 4: if $r \neq$ NIL then
 5:     $r$.reftime ← $time + \mathsf{hit\_time_{L1}}$
 6: else
 7:     $r$ ← the least-recently-used block in the $i$th set
 8:     if $r$.dirty = true then
 9:         $rtop$ ← the address which $r$ contains
10:         $time$ ← CACHE-WRITE-MAIN($rtop$, $\mathsf{block\_size_{L1}}$, $time$)
11:     end if
12:     $top$ ← $addr$ rounded down to the multiple of $\mathsf{block\_size_{L1}}$
13:     $time$ ← CACHE-READ-MAIN($top$, $\mathsf{block\_size_{L1}}$, $time$)
14:     $r$.tag ← $tag$
15:     $r$.dirty ← false
16:     $r$.reftime ← $time + \mathsf{hit\_time_{L1}}$
17: end if
18: return $time + \mathsf{hit\_time_{L1}}$
```

Write Requests

To handle a write request in the cache, the cache block must have the corresponding data. If not, the memory region shoud be transferred from main memory first, which is called cache *fill*. Consequently, the write request functions are very similar to the read request counterparts.

For example, the function CACHE-WRITE-L1 shown below is the same as CACHE-READ-L1 except that the block is turned dirty in cache hits (line 5) and the allocated block is also turned dirty (line 16). The function CACHE-WRITE-REGISTER has the same changes from CACHE-READ-REGISTER.

```
CACHE-WRITE-L1($addr$, $len$, $time$)
 1: $tag$ ← a tag field of $addr$
 2: $i$ ← an index field of $addr$
 3: $r$ ← lookup $tag$ in the $i$th set
 4: if $r \neq$ NIL then
 5:     $r$.dirty ← true
 6:     $r$.reftime ← $time + \mathsf{hit\_time_{L1}}$
 7: else
 8:     $r$ ← the least-recently-used block in the $i$th set
 9:     if $r$.dirty = true then
10:         $rtop$ ← the address which $r$ contains
11:         $time$ ← CACHE-WRITE-MAIN($rtop$, $\mathsf{block\_size_{L1}}$, $time$)
12:     end if
13:     $top$ ← $addr$ rounded down to the multiple of $\mathsf{block\_size_{L1}}$
14:     $time$ ← CACHE-READ-MAIN($top$, $\mathsf{block\_size_{L1}}$, $time$)
15:     $r$.tag ← $tag$
16:     $r$.dirty ← true
17:     $r$.reftime ← $time + \mathsf{hit\_time_{L1}}$
18: end if
19: return $time + \mathsf{hit\_time_{L1}}$
```

Register Invalidation

Destructors in C++ are called when an object is no longer used. The objects wrapped by SMAT adaptors generate trace instruction `SMAT_DTOR` to tell the simulator that the memory region used by the object can be discarded. The simulator uses this information to improve register utilization.

Given a trace instruction `SMAT_DTOR`, the simulator finds the specified address in the register set. If found, the reference time of the corresponding block is set to 0, and the dirty flag is cleared. This block is then given preference over the other cache blocks for block replacement, and replacement can be done without writeback.

What will happen if register invalidation is not performed? Then the register containing destructed data will eventually be chosen at block replacement, which may cause a costly writeback operation. The register invalidation prevents such a useless writeback by explicitly freeing registers. In experiments, this technique has reduced total clock cycles by as much as 45%.

4.2 Trace File Optimizer

Source-level instrumentation prevents some compile-time optimizations. For example, a compiler cannot assign SMAT-adapted objects to registers because traces are obtained by applying the address-of operator, `operator&()`, which forces objects to be placed in main memory. For another example, compiler optimizations cannot eliminate any operations that generate traces since the trace-generation is a side effect that the compiler is unaware is inessential to the underlying computation. Consequently, temporary objects are not eliminated at all by optimization, and all pass-by-value parameters cause the invocation of copy constructors at every function call.

These problems are quite troublesome. Since such compiler optimizations are no longer very effective, traces may contain many inefficient memory references. To alleviate these problems, we designed SMAT to include post-processing of the trace file, by the trace file optimizer, to eliminate or rewrite inefficient trace instructions.

4.2.1 Optimizing Trace Files

Several types of inefficiencies described above can be eliminated by the trace file optimizer `smatopt`. It analyzes the lifetime, and numbers of read and write references, for *all* addresses that appear in a trace file. And at the destruction of an address, it performs several optimizations described below.

Temporary Object Elimination

A memory reference to temporary object has a unique trace instruction sequence, and so it is relatively easy to identify such a reference. The following pseudo trace instructions, which are generated by a statement `j = i + 1`, where `i` and `j` are iterators, show how temporary object elimination works.

```
iaddi  4, (i), (tmp)         iaddi  4, (i), (j)
movi   4, (tmp), (j)   ->    nop
dtori  4, (tmp)              nop
```

In the original instructions (left-hand side), the result of `i + 1` is once written to the temporary object, and then copied to `j`, followed by a destruction of the temporary object. References to the temporary object are useless and so the instructions can be optimized as in the right-hand side, in which the result of `i + 1` is directly written to `j` and the references to the temporary object are eliminated.

Early Destruction

"Early destruction" is the optimization that destructs objects as early as possible. Specifically, it moves destructors just after the trace instruction in which the destructed object is last used. Since the destructor may "free" a register in the simulation, this optimization can improve register utilization. The following is the example of early destruction. In the left-hand side, it is assumed that `x` is no longer used after the first `imuld` instruction and `y` is last used in `ineqd` instruction. Then, in the right-hand side, destructors of `x` and `y` are moved just after the instruction in which they are last used.

```
imuld  4, (x), (y)         imuld  4, (x), (y)
 :                         dtord  4, (x)
 :                          :
ineqd  4, (y)         ->   ineqd  4, (y)
 :                         dtord  4, (y)
 :                          :
dtord  4, (y)              nop
dtord  4, (x)              nop
```

Copy Propagation

If an object `y` is written once by the assignment-like trace instruction (`SMAT_MOV`, `SMAT_CCTOR`, and `SMAT_BCTOR`) and just referred to many times thereafter, read accesses to `y` can possibly be eliminated by copy propagation. The following pseudo instructions illustrates the idea.

```
cctord 4, (x), (y)         nop
 :                          :
subd   4, (y), ...         subd   4, (x), ...
 :                    ->    :
eqd    4, (y), ...         eqd    4, (x), ...
 :                          :
dtord  4, (y)              nop
```

In the left-hand side, an object `y` is copied from object `x`, and is read many times by other instructions. This sequence can be optimized as in the right-hand side. The read references to `y` are replaced by references to `x`. The instructions to create and destroy `y` are eliminated as well.

4.2.2 Weakness — Lack of Common Subexpression Removal

The trace optimizer, however, does not necessarily eliminate all ineffective memory accesses. One of its significant weaknesses is that it cannot eliminate common subexpressions.

Common subexpression elimination (CSE) is an important compile-time optimization in which each common subexpression is computed only once, then the result is referred to from other contexts. However, CSE does not work for expressions of SMAT-adapted objects since eliminating them has side effects. Unfortunately, the trace file optimizer `smatopt` cannot perform CSE neither. Users are therefore advised to avoid writing algorithm code in which the same subexpression appears many times, by referring instead to a temporary variable introduced to hold the value of the expression.

The algorithm `minmax_fewercmp` introduced in Section 2.4 is an example for which manual CSE is effective. In `minmax_fewercmp`, the subexpression `first + 1` is computed twice per loop (once in the `if` statement and once as the parameter of `minmax_helper`), and `first + 2` is computed every time in the conditional expression of the `for` statement.

The following is a modified version of `minmax_fewercmp`. The subexpression `first + 1` is eliminated by introducing a new variable `first_plus_one`. The conditional expression in the `for` loop `first + 2 <= last` is replaced with `first <= last`, by computing `last -= 2` in advance.

⟨Minmax with fewer comparisons and CSE. 52⟩ ≡

```
template <typename T>
pair<T, T> minmax_fewercmp_cse(T first, T last)
{
  T min = first, max = first;
```

```
    T first_plus_one = first + 1;
    last -= 2;
    for (; first <= last; first += 2, first_plus_one += 2)
      {
        if (*first < *first_plus_one)
          minmax_helper(first, first_plus_one, min, max);
        else
          minmax_helper(first_plus_one, first, min, max);
      }

    if (first_plus_one == last + 2)
      minmax_helper(first, first, min, max);

    return pair<T, T>(min, max);
  }
```
Used in part 113a.

With this modification, the clock cycles to process 100000 elements have decreased from 877317 to 750239, which is the improvement of 14%. This example shows that indeed one may need to pay attention to the existence of common subexpressions and carry out manual CSE in order to improve the accuracy of cache simulations performed with SMAT.

4.3 Trace File Format

The trace file is a sequence of trace instructions, in a fixed 16 byte format as shown in Figure 4.1. The first 4 bytes consist of several fields, followed by two source addresses and one destination address. Some trace instructions take only one source address, or no destination address. Unused address fields should be filled with 0's.

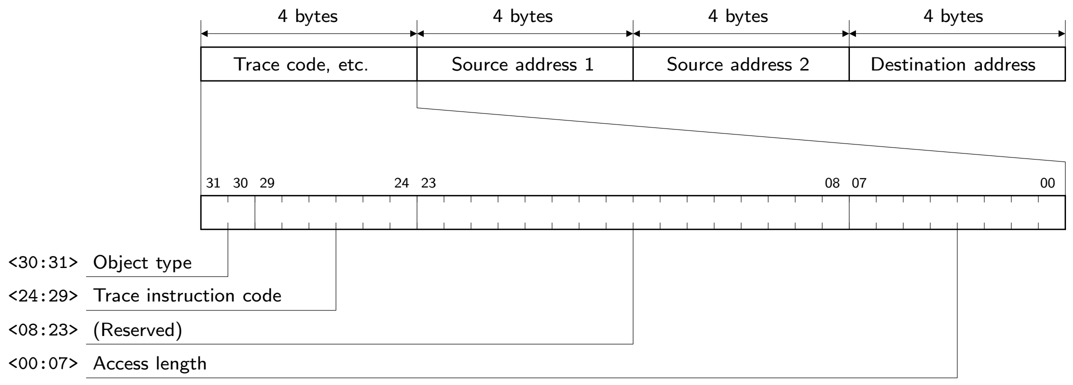


Figure 4.1: Trace file format.

The object type (2 bits) takes values 0 to 3, each of which corresponds to the iterator type, value type, difference type, and pointer type, respectively. The trace instruction code (6 bits) is one of the codes listed in Table 3.1. The access length (8 bits) is the number of bytes read from source addresses and written to the destination address (if any).

Note that the access length applies to all address fields. That is, the number of bytes read from source addresses and written to the destination addresses must be the same. The consequence of this restriction is that the difference type must have the same size as the iterator type, which SMAT assumes to be true.

4.3.1 Space-Time Problems

It is not uncommon that the trace file becomes more than 100MB long. The trace file of the heapsort algorithm sorting 100000 integers requires 894MB. Some kind of compression is definitely necessary for the practical use of SMAT. Table 4.1 compares the compression time, compress ratio, and uncompress time of three major compress utility programs compressing the 894MB trace file.

Table 4.1: Compressing and uncompressing a trace file of 894MB.

Command	Compression (sec)	Result (bytes)	Ratio	Uncompression (sec)
`gzip -1`	26.2	48457498	19.4	11.1
`gzip -9`	228.8	24637966	38.6	8.3
`bzip2 -1`	635.9	21823459	43.0	132.1
`bzip2 -9`	1503.4	11604355	80.8	240.4
`compress`	257.3	65821079	14.3	73.7

Very good compression ratios are obtained by `bzip2`. However, the long uncompression time make its use impractical. One recommended way is to compress with `gzip -1` first, and later, at midnight for example, recompress (uncompress and then compress again) by `gzip -9`. This would provide quicker turnaround time and lower storage consumption.

It is worth noting that there are several compression algorithms dedicated to the trace file compression. They might be a help when the volume really matters. Burtscher and Jeeradit [6] have presented an interesting trace file compression algorithm based on value-prediction. Unfortunately, their algorithm cannot be applied directly to SMAT trace files due to differences in format.

4.4 GCC-Specific Issues

Several problems are specific to the compiler currently being used, GCC version 3.2.

4.4.1 STL Temporary Buffer

In the GCC implementation of the C++ standard library, mergesort, or `std::stable_sort`, allocates a temporary buffer in the beginning. References to the temporary buffer are expressed directly using pointers to the associated value type of the iterator type, rather than the iterator type's associated pointer type. That is, it uses

```
std::iterator_traits<_ForwardIterator>::value_type*
```

instead of

```
std::iterator_traits<_ForwardIterator>::pointer
```

where `_ForwardIterator` is the iterator type passed to the algorithm. Although these types usually work out to be the same type, only the latter type is traceable by SMAT. For this reason we modified the temporary buffer implementation (`stl_tempbuf.h`) to use the associated pointer type. See Appendix A.11 for the modified header file.

4.4.2 Local Variable Alignment

Local variables wrapped by SMAT adaptors do not seem to be tightly aligned. This is bad because they can incur extra cache misses which might be avoided if they are tightly aligned.

The following program declares six local variables and prints their names, addresses, and sizes.

"localaddr.cpp" 55a ≡
```
#include "smat.h"
#include <iostream>

int main()
{
  unsigned a;
  smat_d<int> b, c, d; // smat difference type adaptor.
  int e, f;
  std::cerr << "a: " << &a << ": " << sizeof(a) << '\n'
            << "b: " << &b << ": " << sizeof(b) << '\n'
            << "c: " << &c << ": " << sizeof(c) << '\n'
            << "d: " << &d << ": " << sizeof(d) << '\n'
            << "e: " << &e << ": " << sizeof(e) << '\n'
            << "f: " << &f << ": " << sizeof(f) << '\n';
  return 0;
}
```

All six variables are naturally expected to be packed tightly without any padding between them. However, the output shows that there are gaps between adapted-`int` variables, even though they have the same size as the usual `int` variables.

```
a: 0xbffff800: 4
b: 0xbffff824: 4
c: 0xbffff814: 4
d: 0xbffff804: 4
e: 0xbffff7fc: 4
f: 0xbffff7f8: 4
```

Three variables `b`, `c`, and `d` are aligned at every 16 bytes but this does not mean SMAT-adapted objects require 16-byte alignment since their addresses are not multiple of 16 bytes. Several trials to change the alignment by specifying `-mpreferred-stack-boundary` or `-falign-*` compile options have all failed.

4.4.3 Missing Destructor Calls

Sometimes, temporary objects created as function parameters are not destructed. In the following code, for example, the parameter `bar(i) + 2` to a function `bar` creates a temporary object, but the temporary object is not destructed if compiled with the optimization levels `-O2`, `-O1`, and `-O0`. It *is* destructed with `-O3`.

"lackdtor.cpp" 55b ≡
```
#include "smat.h"

template <typename T>
T bar(T a) { return a; }
```

```
int main()
{
  smat_d<int> i(7);
  return bar(bar(i) + 2);
}
```

Here is the output of operation counting. Constructors are called for $1 + 3 + 1 = 5$ times, while destructors are called only four times. Although this bug is not critical, it can introduce some inaccuracy in cache performance measurements because the simulation treats destructor calls as an opportunity to reuse registers. Compilation with optimization level `-O3` is thus recommended.

```
$ ./lackdtor | smatcount -vT
Total:
  Ctor (default)    1
  Ctor (copy)       3
  Ctor (base type)  1
  Dtor              4
  +                 1
```

Chapter 5

Analyzing Sorting Algorithms

Sorting algorithms are of particular interest in view of their practical application and their varied algorithmic behavior. LaMarca and Ladner have studied the cache performance characteristics of sorting algorithms using the object modification based cache profiling tool ATOM [22], with results demonstrating that there is still a chance for significant improvement in sorting algorithms [13].

In this chapter, the cache performance of three STL sorting algorithms, introsort, mergesort, and heapsort, is analyzed using SMAT. Experiments are done with GCC 3.2 and its own version of STL. Since different compiler developers have different STL implementations, the results presented in this chapter are compiler-dependent. However, they do *not* depend on the hardware architecture or operating system.

Section 5.1 describes the sorting algorithms using pseudo-code. STL sorting algorithm implementations are worth studying because they are tuned carefully. The cache performance of three sorting algorithms are measured and compared in Section 5.2. Trace plots are presented as well to observe when and where cache misses tend to occur. In Section 5.3, the influences of various cache parameters upon cache performance are examined. Cache-conscious versions of introsort and mergesort are explained and experimented with in Sections 5.4 and 5.5, respectively. Finally, in Section 5.6, the performance data obtained is compared with the actual running time of the three sorting algorithms on two different hardware platforms, the Pentium 4 and the UltraSparc-IIi.

Throughout this chapter, algorithms sort 100–100000 elements of 32-bit uniformly distributed random integers in each experiment. Their cache performance is discussed using three performance indices, the number of clock cycles per element, the number of cache misses per element, and the number of cache hits per element.

5.1 Commentary on STL Sorting Algorithms

The GCC (GNU Compiler Collection) version 3.2 has a good STL implementation based on the SGI Standard Template Library [12].

In the following description, algorithms take a pair of iterators, in most cases *first* and *last*, as a range to sort. The first iterator points the first element and the second iterator points *one element past* the end of range. This is the way STL represents a range, which is often written as [*first*, *last*). The notation *val*[*first*] denotes the value to which iterator *first* points.

5.1.1 Introsort

Introsort [16] is a hybrid of quicksort and heapsort that solves the main problem with quicksort: that, although on average it runs in $O(n \lg n)$ time with a low constant coefficient, it can take $\Omega(n^2)$ time in the worst-case, due to many occurrences of bad partitions. Each bad partition leaves as a subproblem a large block still to be sorted, and many repetitions of bad partitioning can result in a subproblem depth that is linear in the size of the original sequence. Introsort keeps track of subproblem depth and switches from quicksort to heapsort if the depth exceeds a certain threshold. By making the threshold proportional to the log of the problem size, the time spent before the switch-over is limited to $O(n \log n)$, and so is the time to complete the sort since heapsort has an $O(n \log n)$ worst-case bound. Introsort thus achieves a worst-case time bound of $O(n \log n)$, and it does so with essentially no increase in the actual time taken in the average case.

In the following pseudo-code, Introsort calls Introsort-Loop followed by Final-Insertion-Sort. Introsort-Loop sorts the sequence roughly, and Final-Insertion-Sort finishes it completely. The subproblem depth threshold used here is $2 \lg N$, where N is the number of elements.

```
INTROSORT($first$, $last$)
1: if $first \neq last$ then
2:    INTROSORT-LOOP($first$, $last$, $2\lg(last - first)$)
3:    FINAL-INSERTION-SORT($first$, $last$)
4: end if
```

Introsort-Loop partitions a sequence by taking a median-of-3 as a pivot. If the sequence size is smaller than or equal to 16 elements, then it does not do anything and the given sequence is left unsorted, which is why we say this function sorts the sequence “roughly.” After the partition in line 8, the second half [*middle*, *last*) is processed by the recursive call and the first half [*first*, *middle*) is sorted by the **while** loop. In both cases, *depth* is decremented by one, and the algorithm switches to heapsort if *depth* became zero. The function Median returns the median value of three arguments.

```
INTROSORT-LOOP($first$, $last$, $depth$)
1: while $last - first > 16$ do
2:    if $depth = 0$ then
3:       HEAPSORT($first$, $last$)
4:       return
5:    end if
6:    $depth \leftarrow depth - 1$
7:    $pivot \leftarrow$ MEDIAN($val[first], val[first + \lfloor (last - first)/2 \rfloor], val[last - 1]$)
8:    $middle \leftarrow$ UNGUARDED-PARTITION($first$, $last$, $pivot$)
9:    INTROSORT-LOOP($middle$, $last$, $depth$)
10:   $last \leftarrow middle$
11: end while
```

The algorithm Unguarded-Partition is based on Hoare’s partitioning algorithm [11]. Iterator *first* moves from the beginning towards the end, and *last* moves from the end towards the beginning. Iterator *first* skips all elements that are smaller than *pivot* (lines 2–4), and *last* skips all elements that are larger than *pivot* (lines 5–8). If these two iterators have met or passed by each other, the partitioning is done and it returns *first* (lines 9–11). Otherwise, it exchanges the two elements that *first* and *last* point to, and continues the outer **while** loop.

```
UNGUARDED-PARTITION($first$, $last$, $pivot$)
 1: while true do
 2:    while $val[first] < pivot$ do
 3:       $first \leftarrow first + 1$
 4:    end while
 5:    $last \leftarrow last - 1$
 6:    while $pivot < val[last]$ do
 7:       $last \leftarrow last - 1$
 8:    end while
 9:    if $first \geq last$ then
10:       return $first$
11:    end if
12:    ITER-SWAP($first$, $last$)    ▷ Swap two values that $first$ and $last$ point to.
13:    $first \leftarrow first + 1$
14: end while
```

FINAL-INSERTION-SORT sorts the first 16 elements by INSERTION-SORT, and the rest (if any) by UNGUARDED-INSERTION-SORT. This distinction is made because the latter is slightly more efficient since it can omit a boundary check that is necessary in INSERTION-SORT.

```
FINAL-INSERTION-SORT($first$, $last$)
 1: if $last - first > 16$ then
 2:    INSERTION-SORT($first$, $first + 16$)
 3:    UNGUARDED-INSERTION-SORT($first + 16$, $last$)
 4: else
 5:    INSERTION-SORT($first$, $last$)
 6: end if
```

INSERTION-SORT might look more complicated than expected. But the basic idea is the same as the classic insertion sort algorithm. The first $i - 1$ elements are sorted when it processes the ith element, and ith element is inserted into the sorted sequence so that the first i elements are sorted. It makes a distinction between two cases, whether the ith element comes to the beginning of the sorted subsequence (lines 8–9), or not (line 11). In the first case, the sorted subsequence so far, $[first, i)$, is moved forward by one element (COPY-BACKWARD), and the former ith element v is stored at the beginning. In the second case, the ith element is inserted into the appropriate position by UNGUARDED-LINEAR-INSERT.

```
INSERTION-SORT($first$, $last$)
 1: if $first = last$ then
 2:    return
 3: end if
 4: for $i \leftarrow first + 1$ to $last$ do
 5:    $v \leftarrow val[i]$
 6:    if $v < val[first]$ then
 7:       COPY-BACKWARD($first, i, i + 1$)
 8:       $val[first] \leftarrow v$
 9:    else
10:       UNGUARDED-LINEAR-INSERT($i, v$)
```

```
11:     end if
12: end for
```

Unguarded-Linear-Insert inserts a value v into the appropriate position of the subsequence before *last*. The boundary check is omitted and so this function can be used only if it is guaranteed that there is an element somewhere before *last* that is smaller than or equal to v. Unguarded-Insertion-Sort uses Unguarded-Linear-Insert to sort a sequence.

```
Unguarded-Linear-Insert($last$, $v$)
1: while $v < val[last - 1]$ do
2:     $val[last] \leftarrow val[last - 1]$
3:     $last \leftarrow last - 1$
4: end while
5: $val[last] \leftarrow v$
```

```
Unguarded-Insertion-Sort($first$, $last$)
1: while $first \neq last$ do
2:     Unguarded-Linear-Insert($first$, $val[first]$)
3:     $first \leftarrow first + 1$
4: end while
```

5.1.2 Mergesort

The Mergesort algorithm first allocates a temporary buffer of the same size as the given sequence, then calls Stable-Sort-Adaptive to sort it. Although the original algorithm is designed to work even if only the smaller size is allocated, the following pseudocode assumes for simplicity that it could allocate the requested size. In the STL of GCC version 3.2, algorithm `std::stable_sort` is implemented based on this Mergesort algorithm.

```
Mergesort($first$, $last$)
1: $buf \leftarrow$ pointer to the temporary buffer of size $last - first$
2: Stable-Sort-Adaptive($first$, $last$, $buf$, $last - first$)
```

Stable-Sort-Adaptive sorts the sequence $[first, last)$ using temporary buffer *buf* of size *buf_size*. It separately sorts the first half and the second half of the sequence (lines 2–3), and then merges them into one sequence (lines 4–5). Line 4 copies the first sorted subsequence to the temporary buffer, and line 5 merges that subsequence and the second sorted subsequence back into the original sequence.

```
Stable-Sort-Adaptive($first$, $last$, $buf$, $buf\_size$)
1: $middle \leftarrow first + \lfloor (last - first + 1)/2 \rfloor$
2: Merge-Sort-With-Buffer($first$, $middle$, $buf$)
3: Merge-Sort-With-Buffer($middle$, $last$, $buf$)
4: $first\_half\_end \leftarrow$ Copy($first$, $middle$, $buf$)
5: Merge($buf$, $first\_half\_end$, $middle$, $last$, $first$)
```

Merge-Sort-With-Buffer sorts the given sequence by calling Merge-Sort-Loop repeatedly. The sequence is first pre-sorted in units of 7 elements (line 4). These pre-sorted subsequences are merged in the **while** loop of lines 5–10, doubling the unit size of subsequences. Line 6 merges from the sequence to the temporary buffer, and line 8 merges from the temporary buffer back to the original sequence. Note that these two merge steps are indivisible because the sorted sequence at the end must be in the original sequence and not in the temporary buffer.

```
MERGE-SORT-WITH-BUFFER(first, last, buf)
 1: $len \leftarrow last - first$
 2: $buf\_last \leftarrow buf + len$
 3: $step\_size \leftarrow 7$
 4: CHUNK-INSERTION-SORT(first, last, step_size)
 5: while $step\_size < len$ do
 6:    MERGE-SORT-LOOP(first, last, buf, step_size)
 7:    $step\_size \leftarrow step\_size \times 2$
 8:    MERGE-SORT-LOOP(buf, buf_last, first, step_size)
 9:    $step\_size \leftarrow step\_size \times 2$
10: end while
```

MERGE-SORT-LOOP is costly because it reads or writes all elements of the sequence. How many times is MERGE-SORT-LOOP called in this function for a sequence of size M? The **while** loop in lines 5–10 is repeated while *step_size* is smaller than M. The variable *step_size* is initialized as 7 and multiplied by 4 in every loop iteration. So the number of iterations executed is the smallest integer x such that $7 \cdot 4^x \geq M$. Solving this equation yields $x = \lceil \log_4 \lceil M/7 \rceil \rceil$, and then MERGE-SORT-LOOP is executed $2x = 2\lceil \log_4 \lceil M/7 \rceil \rceil$ times. But remember that MERGE-SORT-WITH-BUFFER is called twice, each for the first and second halves of the entire sequence. So letting N be the size of the entire sequence, $M = N/2$ (if N is even). Finally, the number of calls for MERGE-SORT-LOOP is given by $4\lceil \log_4 \lceil N/14 \rceil \rceil$, which is used in the further analysis.

CHUNK-INSERTION-SORT effectively splits the given sequence [*first*, *last*) into subsequences of size *chunk_size*, and performs insertion sort on each subsequence. This step is called pre-sorting.

```
CHUNK-INSERTION-SORT(first, last, chunk_size)
1: while $last - first \geq chunk\_size$ do
2:    INSERTION-SORT(first, first + chunk_size)
3:    $first \leftarrow first + chunk\_size$
4: end while
5: INSERTION-SORT(first, last)
```

MERGE-SORT-LOOP assumes that the sequence is pre-sorted in units of *step_size* elements. That is, it assumes the first *step_size* elements are sorted within themselves, and the second *step_size* elements are similarly sorted, and so on. The **while** loop of lines 2–5 repeatedly merges two sorted subsequences into one sorted sequence, and lines 6–7 takes care of the remainder.

```
MERGE-SORT-LOOP(first, last, result, step_size)
1: $two\_step \leftarrow step\_size \times 2$
2: while $last - first \geq two\_step$ do
3:    $result \leftarrow$ MERGE(first, first + step_size, first + step_size, first + two_step, result)
4:    $first \leftarrow first + two\_step$
5: end while
6: $step\_size \leftarrow$ MIN(last − first, step_size)
```

```
7: MERGE($first, first + step\_size, first + step\_size, last, result$)
```

The MERGE algorithm merges two sorted subsequences by the straightforward method. The loop of lines 1–11 is repeated until either sequence [*first1*, *last1*) or [*first2*, *last2*) becomes empty. Lines 12–13 just copy the non-empty sequence at the end of *result*.

```
MERGE($first1, last1, first2, last2, result$)
 1: while $first1 \neq last1$ and $first2 \neq last2$ do
 2:   if $val[first2] < val[first1]$ then
 3:     $val[result] \leftarrow val[first2]$
 4:     $result \leftarrow result + 1$
 5:     $first2 \leftarrow first2 + 1$
 6:   else
 7:     $val[result] \leftarrow val[first1]$
 8:     $result \leftarrow result + 1$
 9:     $first1 \leftarrow first1 + 1$
10:   end if
11: end while
12: $result \leftarrow$ COPY($first1, last1, result$)
13: return COPY($first2, last2, result$)
```

5.1.3 Heapsort

HEAPSORT is an algorithm based on properties of the heap data structure: linear time construction, logarithmic time insertion, and logarithmic time deletion of the largest element (in the case of a max-heap, or the smallest value in the case of a min-heap). These properties can be achieved with binary trees, and by representing the tree in an array, the heap operations can be done in place and without any need for explicit pointers. The array representation of the tree used here puts the left child of the i element in position $2i + 1$ if it exists in the array (i.e., if $2i + 1 < N$) and the right child in position $2i + 2$ (if $2i + 2 < N$). MAKE-HEAP makes the entire sequence into a max-heap: each element is greater than or equal to its children. Thus the element at position 0 is the largest element of the entire sequence after MAKE-HEAP executes. SORT-HEAP repeatedly exchanges the first (maximum) element and the last element of the sequence, and then restores the heap property in the shortened heap structure, so that at each iteration the current heap's maximum element is removed and settled in its final position.

```
HEAPSORT($first, last$)
1: MAKE-HEAP($first, last$)
2: SORT-HEAP($first, last$)
```

MAKE-HEAP constructs a binary max-heap array by calling ADJUST-HEAP which adjusts the data structure locally under a certain heap subtree. ADJUST-HEAP is called in bottom-up order for all nodes that have a child.

```
MAKE-HEAP($first, last$)
1: if $last - first < 2$ then
2:   return
3: end if
```

```
 4: $len \leftarrow last - first$
 5: $parent \leftarrow (len - 2)/2$   $\triangleright$ a parent of the last element
 6: **while true do**
 7:    ADJUST-HEAP($first, parent, len, val[first + parent]$)
 8:    **if** $parent = 0$ **then**
 9:       **return**
10:    **end if**
11:    $parent \leftarrow parent - 1$
12: **end while**
```

SORT-HEAP sorts the sequence by utilizing the max-heap property, in which the first element has the maximum value. Line 2 saves the value of the last element, and line 3 copies the maximum value to the last element. The heap structure is then adjusted in line 4, but only for $last - first - 1$ elements, by calling ADJUST-HEAP function. After every **while** loop, one element is placed at its final position and the sequence is shorten by one element.

```
SORT-HEAP($first, last$)
 1: **while** $last - first > 1$ **do**
 2:    $v \leftarrow val[last - 1]$
 3:    $val[last - 1] \leftarrow val[first]$
 4:    ADJUST-HEAP($first, 0, last - first - 1, v$)
 5:    $last \leftarrow last - 1$
 6: **end while**
```

ADJUST-HEAP adjusts the heap structure locally under the element indexed by *hole*. In lines 3–14, the hole is moved top-down. Lines 4–6 make *second_child* point the child with larger value, and lines 7–9 move the hold downward. Lines 11–14 handles a special case in which *second_child* points just one past the last element. Then, the heap is adjusted bottom-up by PUSH-HEAP.

```
ADJUST-HEAP($first, hole, len, v$)
 1: $top \leftarrow hole$
 2: $second\_child \leftarrow 2 \times hole + 2$
 3: **while** $second\_child < len$ **do**
 4:    **if** $val[first + second\_child] < val[first + second\_child - 1]$ **then**
 5:       $second\_child \leftarrow second\_child - 1$
 6:    **end if**
 7:    $val[first + hole] \leftarrow val[first + second\_child]$
 8:    $hole \leftarrow second\_child$
 9:    $second\_child \leftarrow 2 \times second\_child + 2$
10: **end while**
11: **if** $second\_child = len$ **then**
12:    $val[first + hole] \leftarrow val[first + second\_child - 1]$
13:    $hole \leftarrow second\_child - 1$
14: **end if**
15: PUSH-HEAP($first, hole, top, v$)
```

PUSH-HEAP stores a value v into the appropriate position in the binary heap array. The search for the final position starts from the element indexed by *hole* and proceeds bottom-up direction, which is bounded by index *top*.

```
PUSH-HEAP($first$, $hole$, $top$, $v$)
1: $parent \leftarrow (hole - 1)/2$
2: while $hole > top$ and $val[first + parent] < v$ do
3:     $val[first + hole] \leftarrow val[first + parent]$
4:     $hole \leftarrow parent$   ▷ the hole moves up
5:     $parent \leftarrow (hole - 1)/2$   ▷ parent index moves up as well
6: end while
```

5.2 Cache Performance

Figure 5.1(a) compares the total clock cycles per element of three sorting algorithms, introsort, mergesort, and heapsort. Introsort is the winner, as expected, and mergesort is the second best. Heapsort takes longer than the others, especially with larger sequences. Every plot changes its shape around $N = 2000$, where the sequence becomes the same size as the cache. This point is called the *cache saturation point*. The saturation point in mergesort starts earlier, around $N = 1500$, which is because the mergesort uses a temporary buffer and has a larger working set.

The number of clock cycles per element taken by introsort and mergesort is almost constant before the saturation point. Then it increases slowly afterward in introsort. Mergesort shows a huge gap between $N = 2000$ and $N = 5000$, followed by moderate growth like that of introsort. The growth in heapsort is mild before the saturation point, but it grows fairly steep thereafter. Since the values shown in the graph are the clock cycles *per element* and the x-axis is in a logarithmic scale, linear growth indicates $O(N \lg N)$ behavior, and a constant value indicates $O(N)$. This applies to all other graphs in this chapter.

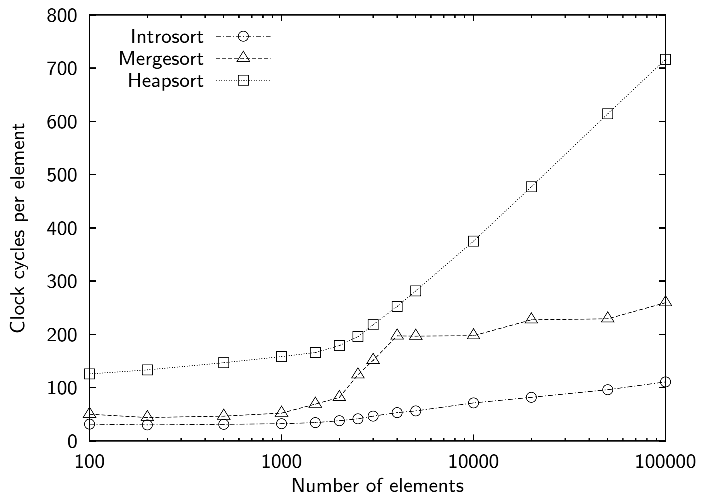
The number of cache misses per element is shown in Figure 5.1(b). The three plots are very close to each other until $N = 1000$. In fact, the plots are decreasing in this range, which implies that each algorithm involves some fixed number of cache misses unrelated to N. The plots after the saturation point show shapes similar to graph (a), which is quite natural because the number of cache misses directly affects the total clock cycles, as shown in equation 2.1. What is different from graph (a) is the relative position of heapsort. The number of misses of heapsort is smaller than mergesort until $N = 10000$, while the clock count of heapsort is consistently greater than mergesort. Why?

The last graph (c) answers to this question. Heapsort causes far more cache hits than the other two algorithms. According to equation 2.1, the number of hits does affect the total clock cycles, though less significantly. In cache performance studies, the number of cache hits is rarely mentioned. However, these graphs show consideration of the number of cache hits is sometimes necessary to explain the total clock cycles. Therefore, the number of hits are shown throughout this chapter and mentioned if necessary.

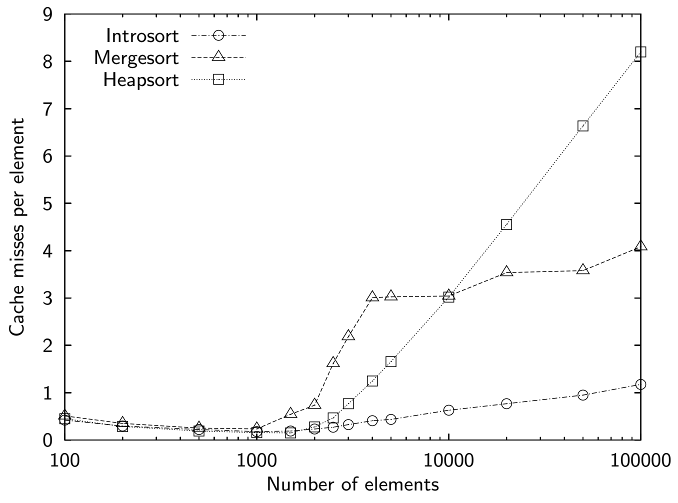
5.2.1 Type Analysis

Figure 5.3 is an excerpt from Figure 5.1 and shows values only for introsort. Plots are drawn separately for the iterator type and its associated types. For example, in Figure 5.3(a), the lowest circle plots indicate the clock cycles taken by the value type operation. The *difference* between the circle and triangle plots represents the clock counts taken by both the iterator type and pointer type. Similarly, the gap between the triangle and square plots, which is very narrow, corresponds to the difference type operation. The solid line with the square plots itself represents the total clock

(a)



(b)



(c)

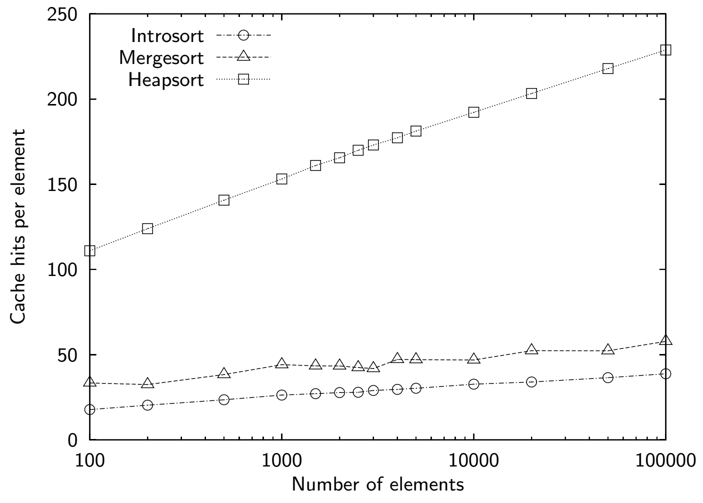


Figure 5.1: Cache performance of STL sorting algorithms. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

cycles taken by all operations. Figure 5.2 illustrates the idea of type decomposition and the same discussion applies to Figure 5.4 and 5.5.

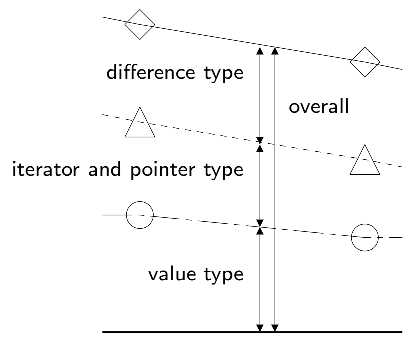


Figure 5.2: How Figure 5.3, 5.4, and 5.5 are decomposed into types.

Introsort

Although it is not so clear from Figure 5.1(a), the number of clock cycles in introsort also increases steadily after the saturation point ($N = 2000$) in Figure 5.3(a). In terms of types, the value type is taking most of the clock cycles. The iterator type is the second. The iterator type seems to incur a constant number of clock cycles after the saturation point. And the difference type costs very little.

According to Figure 5.3(b), the number of cache misses caused by value type accesses does not change so much before the saturation point, and the compulsory misses are mainly caused by iterator type operations. In the range $N \leq 2000$, the entire sequence fits into the cache, and so optimistically, each element is brought from main memory only once. Since one cache block can have eight elements, the theoretical number of cache misses per element caused by the value type is $1/8 = 0.125$. However, the observed values are about 0.2. So there must be other sources of value-type misses, probably references to local variables. After the saturation point, the cache misses caused by the iterator and difference types are small, and the value type is dominant.

Unlike cache misses, the number of cache hits in Figure 5.3(c) does not change its shape before and after the saturation point. The number of hits incurred by the iterator and difference type operations are kept constant regardless the number of elements.

Mergesort

A similar discussion applies to mergesort, as shown in Figure 5.4. Clock cycles taken by the value type operations are dominant, iterator (and pointer) type operations are almost constant, and difference type operations are insignificant. The compulsory misses observed before the threshold are due to the iterator type operations.

Why, one might ask, do the clock counts increase by discrete jumps after the saturation point? This is due to the number of merge steps executed. As derived in Section 5.1.2, the number of calls of function MERGE-SORT-LOOP is $4\lceil \log_4 \lceil N/14 \rceil \rceil$. Thus the numbers of merge steps are as follows. The clock counts grow by discrete jumps because they increase almost in proportion to the number of merge steps.

$$\begin{aligned}
4\lceil \log_4 \lceil 5000/14 \rceil \rceil &= 4\lceil \log_4 358 \rceil = 4\lceil 4.241\ldots \rceil = 20,\\
4\lceil \log_4 \lceil 10000/14 \rceil \rceil &= 4\lceil \log_4 715 \rceil = 4\lceil 4.740\ldots \rceil = 20,\\
4\lceil \log_4 \lceil 20000/14 \rceil \rceil &= 4\lceil \log_4 1429 \rceil = 4\lceil 5.240\ldots \rceil = 24,\\
4\lceil \log_4 \lceil 50000/14 \rceil \rceil &= 4\lceil \log_4 3572 \rceil = 4\lceil 5.901\ldots \rceil = 24,\\
4\lceil \log_4 \lceil 100000/14 \rceil \rceil &= 4\lceil \log_4 7143 \rceil = 4\lceil 6.401\ldots \rceil = 28.
\end{aligned}$$

Heapsort

Heapsort, Figure 5.5, shows very different characteristics. Not only the value-type operations, but also the iterator and difference type operations take a large number of clock cycles in graph (a). However, it is the value type accesses that still increase after the saturation point. Graph (b) clearly shows that the value type operations are predominant in terms of cache misses. Concerning the number of cache hits, graph (c), the number of hits caused by the difference type operation is the largest, by the iterator type is second, and by the value type third. It is surprising that the difference type and iterator type operations cause such a large number of cache misses. The number of cache hits caused by the value type, which is slightly less than 50 at $N = 100000$, is not quite as bad.

5.2.2 Trace Plot

Figure 5.6 shows two iterator trace plots[1] of introsort, sorting 1500 and 5000 elements. In the first plot (a), the partitioning process in which two iterators move towards each other is clearly observed in the clock range 0–10000. Then, the upper half is sorted recursively (10000–30000), followed by the lower half (30000–38000). The sort is finished by Final-Insertion-Sort (38000–). The number of elements is 1500, which is smaller than the cache size, but there are several cache misses after the first partition.

There are more cache misses in plot (b), sorting 5000 elements. Not only the first partitioning, but also the second and third partitioning cause cache misses. At the end of every partitioning, the elements near the partition boundary are supposed to be in the cache. So there are few cache misses at the beginning of sequence in the second and third partitioning. Once the partitioned sequence becomes smaller than a certain size, cache misses are rarely observed, just as in plot (a). The plot also indicates that the first partitioning is not very good. Processing the upper half takes 170000 clock cycles (40000–210000) while the lower half only takes 10000 cycles (210000–220000). Processing the upper half first seems to be a good strategy because elements at the beginning of the sequence are supposed to be in the cache at the start of the final insertion sort.

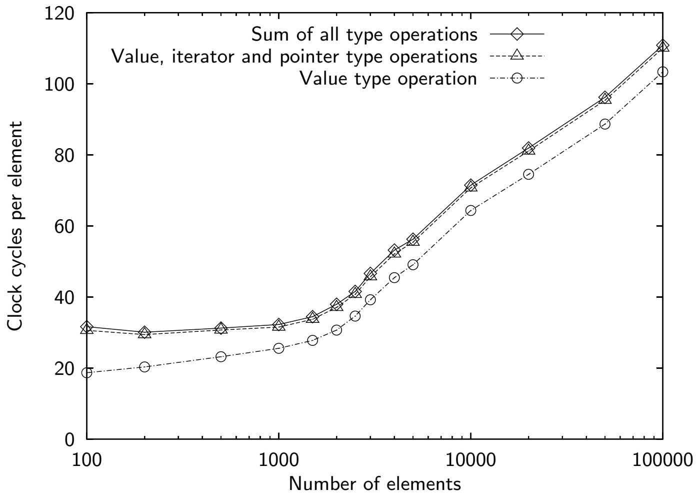
The trace plots for mergesort, Figure 5.7, contain both the sequence and the temporary buffer, and their respective ranges of address space are shown in the right side of plots. The address space can be split into three parts at almost the same intervals. The lower two parts are the sequence, and the upper part corresponds to the temporary buffer. By the way, the temporary buffer is allocated with the same size as the sequence at most, but only its first half is used. The first half of the sequence is pre-sorted in the clock range 0–10000, followed by eight merge steps (10000–50000). The remaining half of the sequence is pre-sorted (50000–60000), and then merged (60000–84000). Finally, the two halves are merged (84000–). The lower plot (b), sorting 5000 elements, looks quite similar to plot (a) except the frequency of cache misses and the number of merge steps.

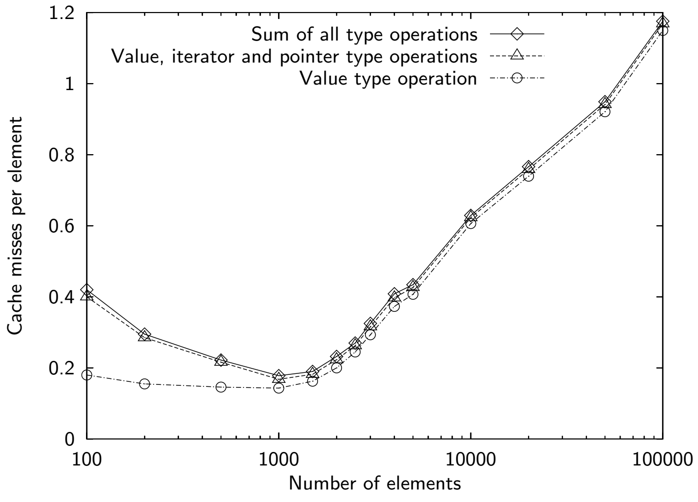
[1]Such memory access trace plots are an extension of the iterator trace plots produced by the itrace tool [14].

When merging the first half in plot (a), groups of cache misses are observed periodically. Since they are in the fixed range of addresses, they are supposed to be conflict misses. In fact, they should not be observed since the whole sequence must fit into the cache. Further experimentation revealed that these misses are due to the conflict between the first half of the sequence, the temporary buffer, and local variables placed on the stack (not shown in the plot).

The trace plots of heapsort are completely different from the other algorithms, as shown in Figure 5.8. Plot (a) incurs relatively few cache misses, but it has a large number of cache hits. Plot (b) reveals quite a large number of cache misses and hits. The first, orderly part corresponds to the MAKE-HEAP operation, and the rest corresponds to the SORT-HEAP operation. In the SORT-HEAP operation, the maximum element stored at the beginning of sequence is moved to the end, and then the data structure is reorganized so that it regains the heap property. This involves quite discontinuous memory accesses over the whole sequence. Because of this characteristic, heapsort exhibits poor cache performance, especially on large sequences.

Comparing plots (a) and (b) for the three algorithms, it turns out that introsort exhibits self-similarity in cache behavior. That is, the cache behavior similar to plot Figure 5.6(a) is found several times in plot (b). This is a consequence of its divide-and-conquer strategy. On the other hand, mergesort and heapsort have almost the same structures in plots (a) and (b), just the plot (b) having more cache misses. Also note that the plots' shape in mergesort and heapsort may not be affected much by the specific sequence, while in introsort the partitioning and the following memory access pattern will be significantly affected by the specific sequence.

(a)


(b)


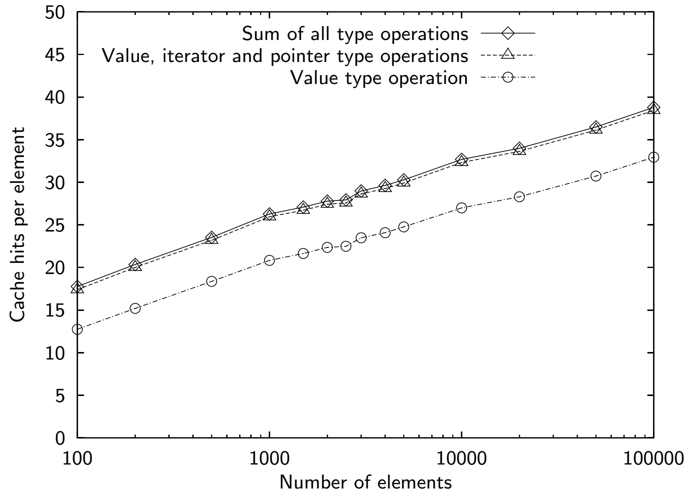
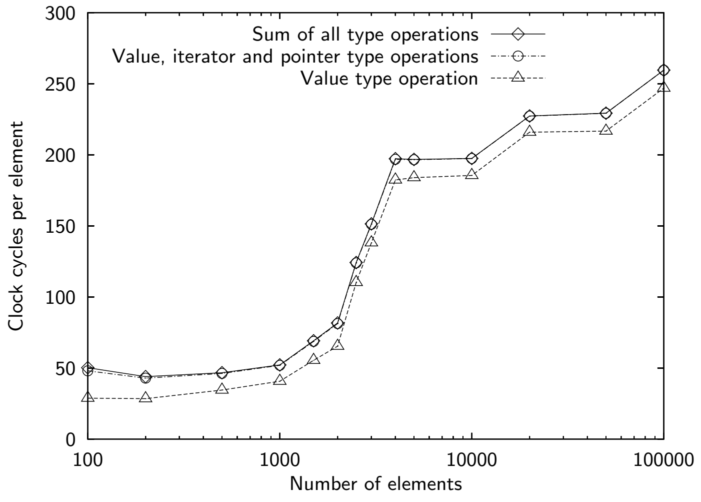
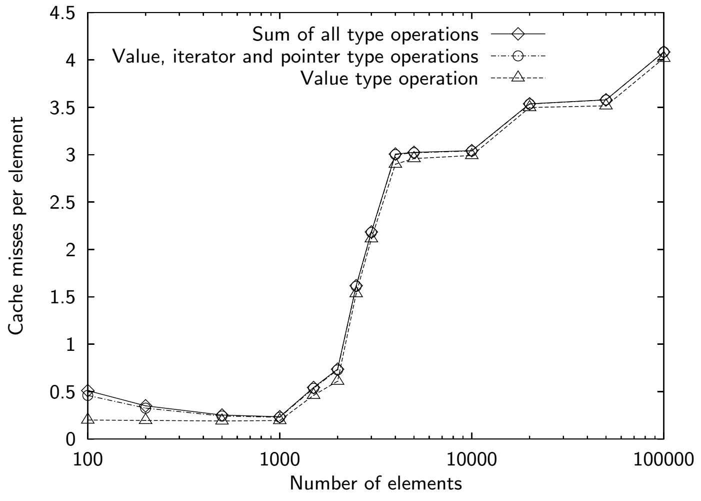
(c)


Figure 5.3: Type analysis of introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

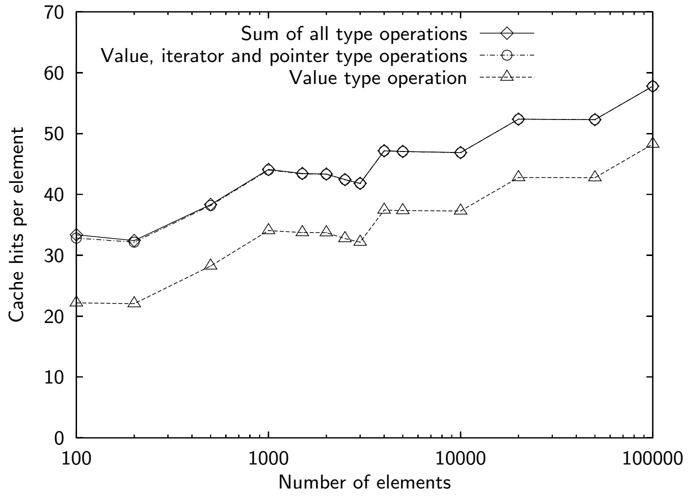
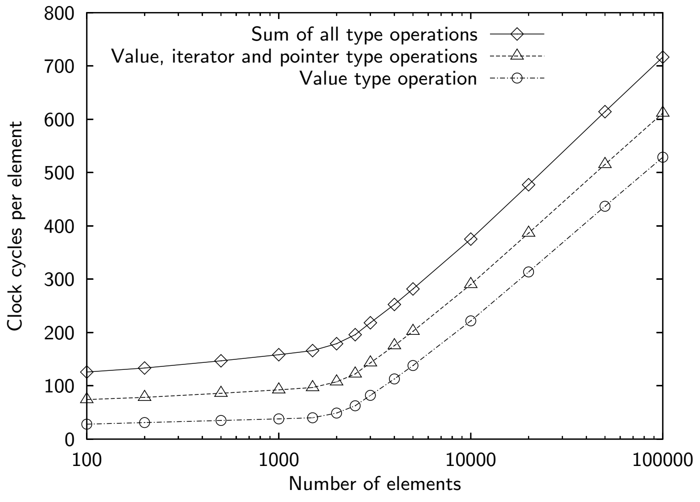
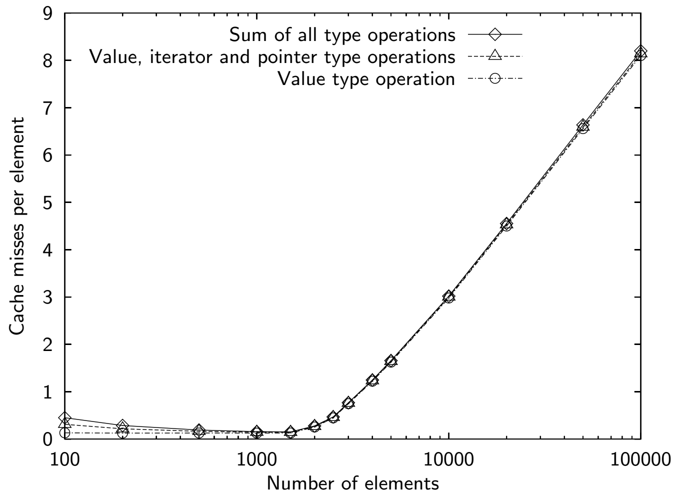


Figure 5.4: Type analysis of mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

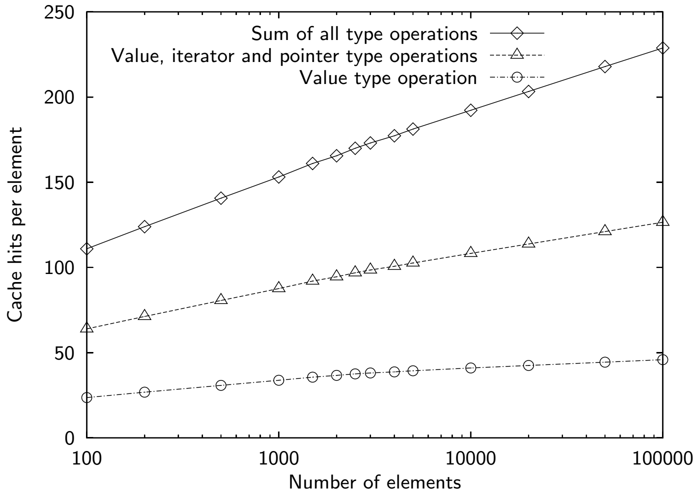
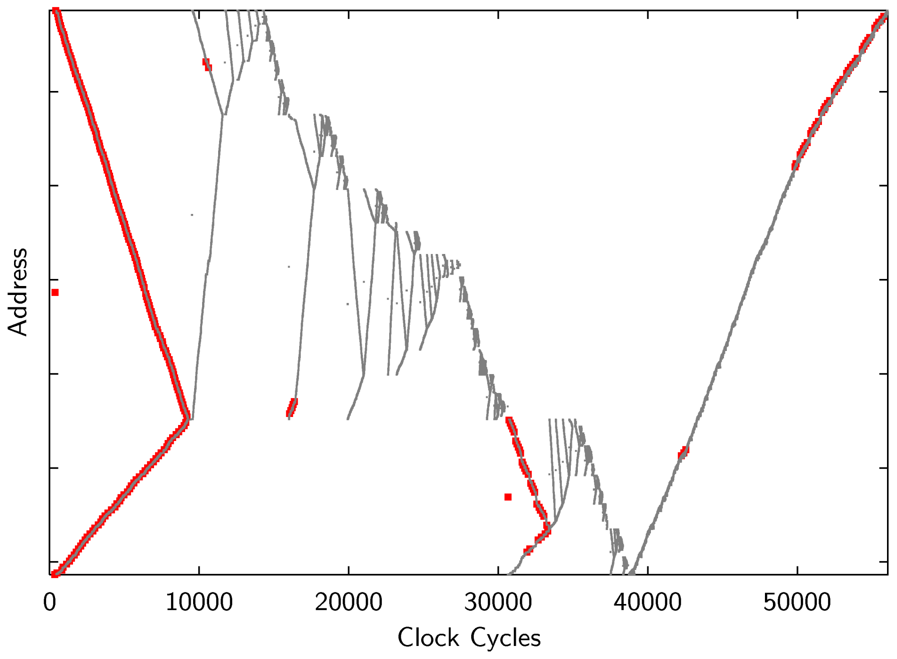


Figure 5.5: Type analysis of heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)

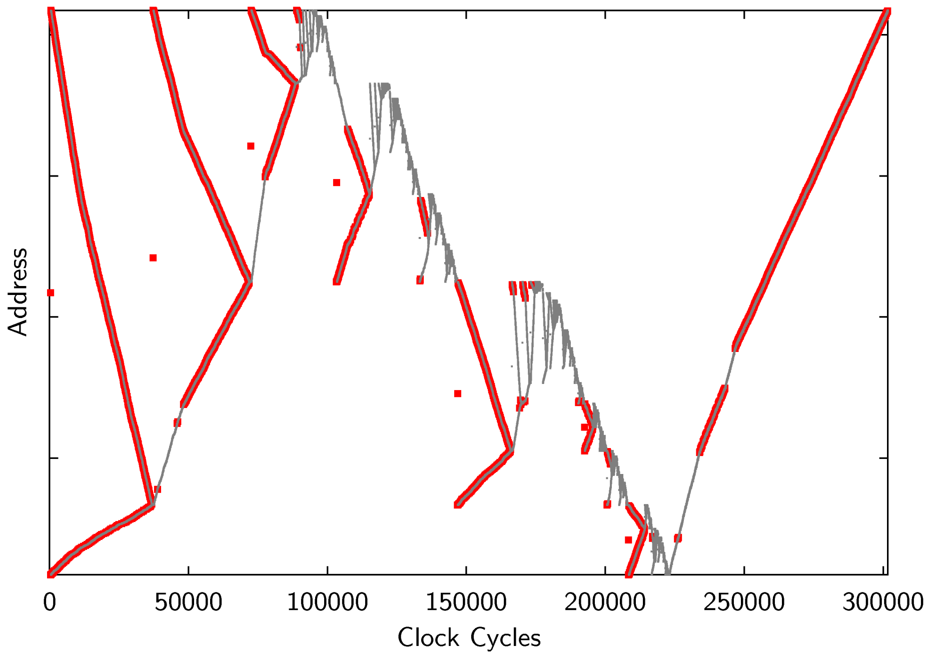
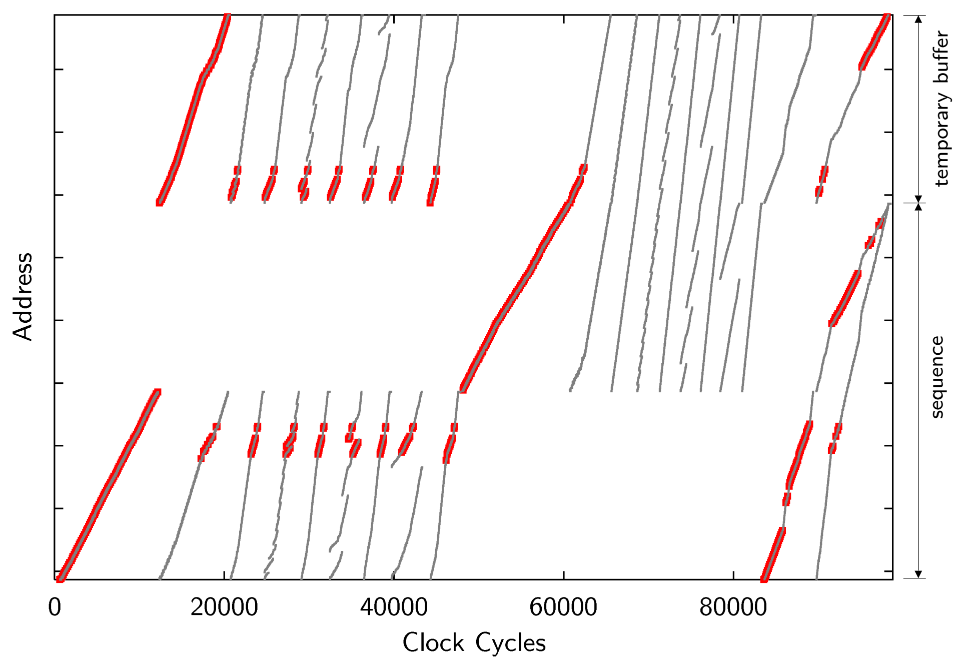


Figure 5.6: Trace plots of introsort, sorting (a) 1500 and (b) 5000 elements.

(a)



(b)

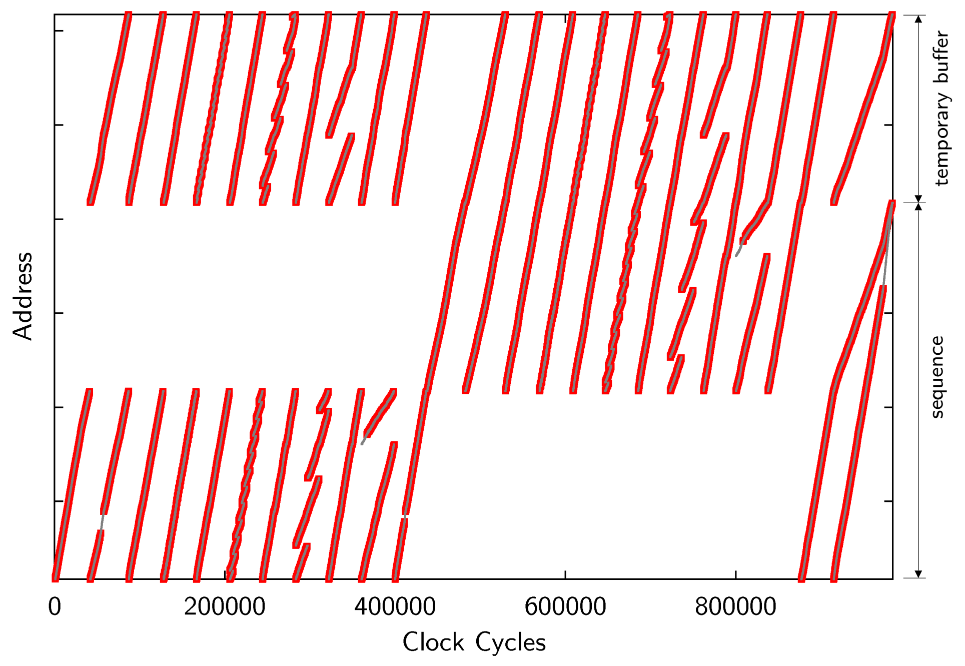
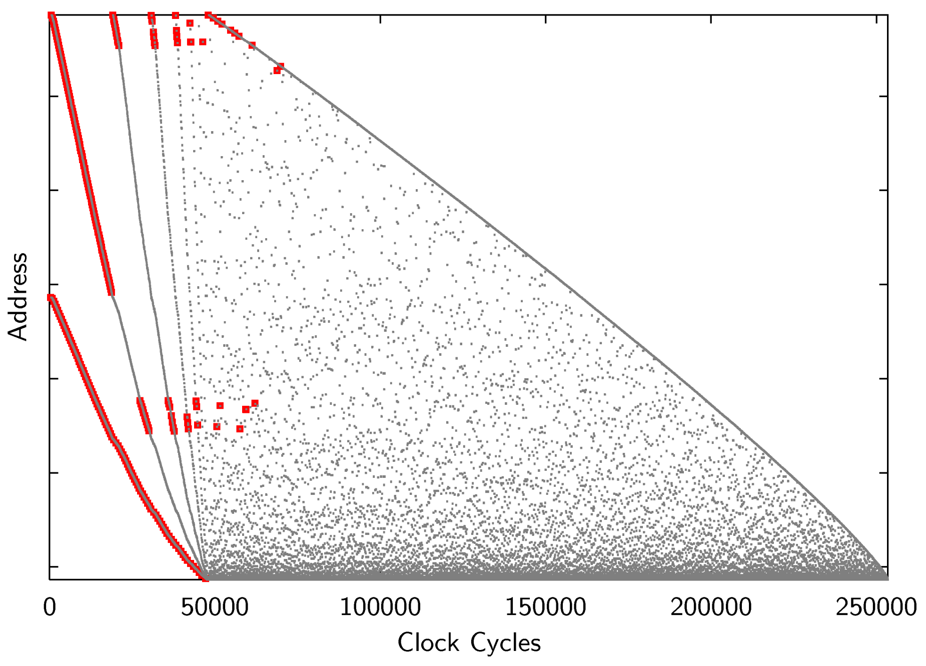


Figure 5.7: Trace plots of mergesort, sorting (a) 1500 and (b) 5000 elements.

(a) 

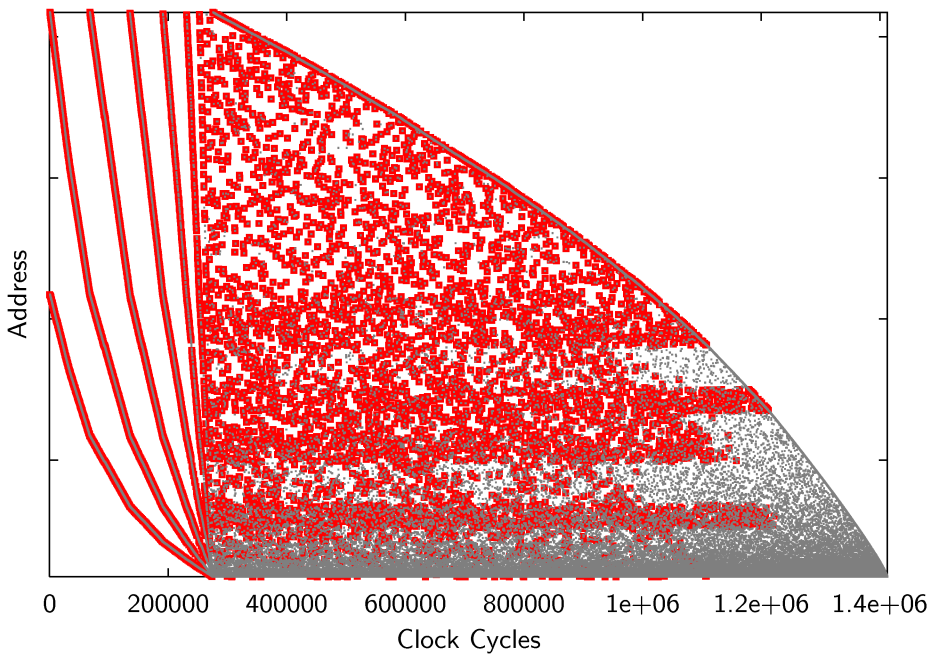
(b) 

Figure 5.8: Trace plots of heapsort, sorting (a) 1500 and (b) 5000 elements.

5.3 Significance of Cache Parameters

In this section, the influence of cache parameters on cache performance is examined by changing configurable cache parameters one by one. Parameters and values tested are as follows. The underlined values are the default values.

- Cache size — 1, 2, 4, 8, 16, 32 KB.
- Associativity — Direct mapped, 2-way, 4-way, 8-way, 16-way set associative.
- Block size — 16, 32, 64, 128 bytes.
- Number of registers — 1, 2, 4, 8, 16, 32.
- Latency of Main Memory — 8, 16, 32, 64, 128 clocks.

Only one parameter is changed in each experiment, keeping other parameters at their default values, except for the block size in which the latency of main memory is also changed. Since the latency value indicates the clock cycles per *block*, it should be altered as well in order to keep the latency per byte constant. For example, the latency of main memory is changed to be 128 clocks when the block size is 128 bytes.

5.3.1 Cache Size

It is not uncommon for high-end systems to have caches of more than 1 MB. For example, the Intel Xeon processor MP family can have up to a 2 MB level 3 cache, and the HP Alpha 21264 can have up to an 8MB level 2 cache. It is easy to see that larger caches are advantageous.

The influence of changing the cache size on introsort is shown in Figure 5.9. Graph (a) clearly suggests that the algorithm takes fewer clock cycles with large caches. The plot shifts rightward as the cache size increases, which makes sense because the saturation point is given by $N_{\text{thres}} =$ (Cache size/4). The slopes of plots after the threshold are close to each other. The variation in graph (b) is similar to graph (a).

It should be obvious that the changing cache sizes does not affect the memory access pattern of the algorithm. Therefore, the total number of memory accesses, or the summation of the hit counts and miss counts, is not affected by the cache sizes. This observation explains graph (c), which shows the small differences between plots. In fact, these differences correspond to the number of cache misses. That is, (b) + (c) is kept constant for different cache sizes, which is also true in Figure 5.10 and 5.11.

The basic discussion of the shift of the saturation point can be applied also to mergesort. Figure 5.10(a) shows the gaps are shifting rightward as the cache size becomes larger. However, what is different from introsort is that at larger N the plots converge on almost the same value again (except the plot of cache size = 1KB). This means that the performance is not affected much if the sequence size is big enough. This is because each merge step reads the whole sequence again from main memory.

The shift is also observed in heapsort, as in Figure 5.11(a), which looks somewhat like the same graph of introsort. However, the performance degradation is severe with cache sizes 1KB and 2KB, and their plots grow faster than the other plots.

5.3.2 Associativity

Figure 5.12 shows the clock cycles per element taken by introsort with various cache associativities. Except for the direct mapped case, all plots are very close to each other in graph (a) and (b). This implies that associativities higher than 2-way do not improve performance much.

In mergesort, shown in Figure 5.13, the effect of increasing associativity is less significant. However, there is one plot, $N = 100000$ of direct mapped, that stands out. Closer examination reveals that, in this particular case, the starting addresses of the sequence and temporary buffer share the same cache block. Therefore, accessing the ith element of the sequence and of the temporary buffer conflicts with each other, resulting in extremely poor cache performance.

The effect on heapsort is shown in Figure 5.14. Similar to introsort, all plots except for the direct mapped case are very close to each other.

5.3.3 Block Size

Although the 32-byte cache block has been popular among many architectures, several recent microprocessors have larger block sizes, such as 64 bytes and 128 bytes. On the actual hardware, transferring in the larger unit is more efficient due to burst transfer. Thus, in practice, the latency per block decreases with larger blocks, especially with memory that has high latency and high bandwidth. These conditions are the motivation to adopt caches with large block sizes.

In this experiment, however, the latency per byte is kept constant. Assume that the algorithm just reads all elements only once from the beginning to the end. Then, no difference in performance would be observed between different block sizes. The difference is noticeable when the algorithm makes jumpy accesses through the sequence. In such a case, a smaller block size would outperform a larger one.

Figure 5.15 shows the effect of increasing block sizes on introsort. With large block sizes, the number of cache misses has decreased significantly, but the number of clock cycles has increased slightly. Since the increase is small, it is conjectured that a large block size would pay off in practice if the latency per block decreased.

The variation of performance is moderate in mergesort, as shown in Figure 5.16. This implies that most memory accesses performed by mergesort are sequential. On the other hand, heapsort, Figure 5.17, shows severe performance deterioration in large block sizes. The number of cache misses is not decreased, unlike in other algorithms. This is evidently because its memory access pattern is quite irregular.

5.3.4 Number of Registers

The number of registers is a cache parameter specific to the SMAT cache simulator. Having many registers would improve performance because it increases the number of objects that can be accessed without cost.

Figure 5.18(a) clearly shows the advantage in introsort of increasing the number of registers. The number of clock cycles decreases significantly as the number of registers increases to 8. It turns out that the default value, 8 registers, is favorable. Having 16 registers seems to achieve some additional improvement, but having 32 registers yields little improvement.

It might be counterintuitive that the number of registers does *not* affect the miss counts, as shown in graph (b). Why is the number of cache misses not affected? This question is best answered by discussing how the requested address is found in a register set and an L1 cache. There are four cases to consider.

1. Found both in a register set and in the cache. The memory access request is satisfied by the register set, and so this case is *not* counted as a cache hit.

2. Not found in a register set but found in the cache. This case is counted as a cache hit.

3. Found in a register set but not in the cache. This combination cannot happen because the register set is always a subset of the cache.

4. Not found in a register set or in the cache. This case is counted as a cache miss.

Out of the four cases, only the last causes a cache miss. Since the register set is a subset of L1 cache, the register set cannot have addresses that are not in L1 cache, regardless of the number of registers. The number of cache misses only depends on whether the L1 cache has the data or not, and so having many registers does not contribute to reducing cache misses.

On the other hand, the number of registers is a factor in the first two cases. If the requested address is in the cache, the request is handled by either case 1 or 2. The more registers there are, the more accesses are satisfied by case 1, and so the number of accesses satisfied by case 2 decreases. And it is the number of cache hits in this experiment that affects the total clock cycles.

The effects of having many registers on mergesort and heapsort are almost the same as that on introsort. However, in heapsort, to increase from 8 to 16 registers seems to be more effective than in the other algorithms. Moreover, the improvement is still observed in heapsort when changing from 16 to 32. This implies that heapsort accesses elements that were last referenced a fairly long time before.

5.3.5 Latency of Main Memory

In general, microprocessors with higher clock speed have longer main memory latency. For example, the Pentium 4 3.06GHz takes more than 200 CPU clock cycles to access main memory. It is easy to see that the longer latency has a severe impact on performance.

The influence of memory latency on introsort is shown in Figure 5.21. The number of clock cycles, graph (a), is affected by the memory latency, but the number of cache misses and cache hits, graphs (b) and (c), are not affected at all because the memory latency is a factor that plays a role only *after* cache misses.

Recall that equation 2.1, which derives the number of total clock cycles from the number of cache hits, misses, writebacks and the main memory latency, contains the term

$$\text{Number of L1 misses} \times \text{Hit time}_{\text{Main}}.$$

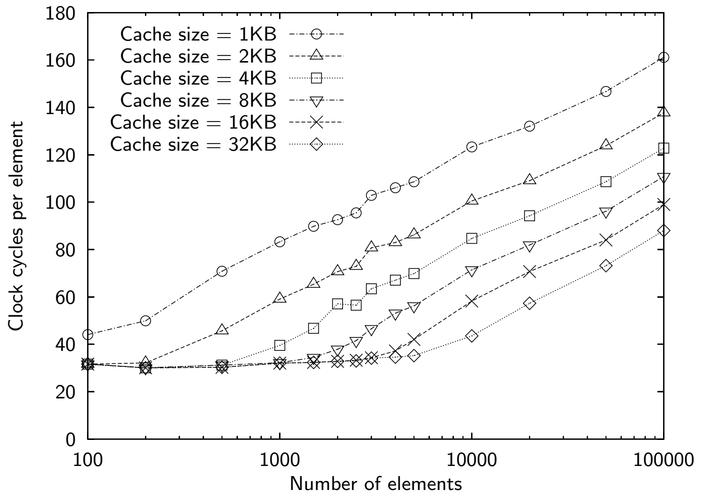
Therefore, the clock cycle count is affected more in cases of larger N where the algorithm incurs many cache misses. And the variation of the clock cycles is small around $N = 1000$ where the algorithm has fewer cache misses.

This observation can be applied to mergesort and heapsort as well, which are shown in Figure 5.22 and 5.23, respectively.

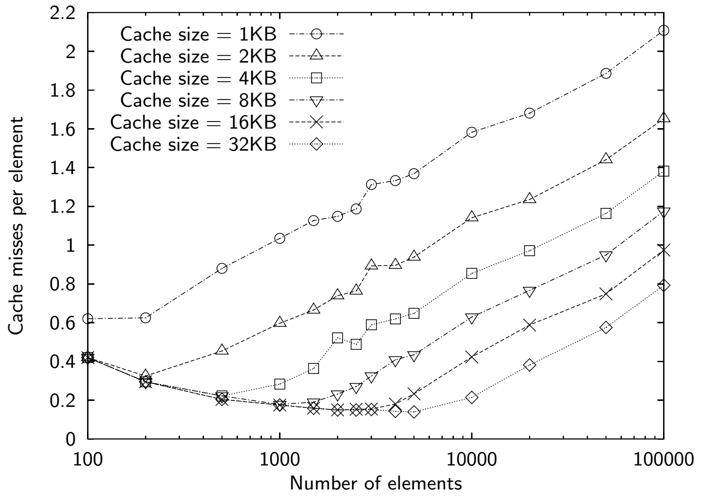
5.4 Improvements Over Introsort

Although introsort is confirmed to be quite efficient in most cases, there is at least a small possibility of improvement in terms of cache performance. Experiments with two variations of introsort are described in this section. The first algorithm tries to reduce cache misses by eliminating Final-Insertion-Sort. The second, less promising one employs a different partition algorithm called Lomuto's partition.

(a)



(b)



(c)

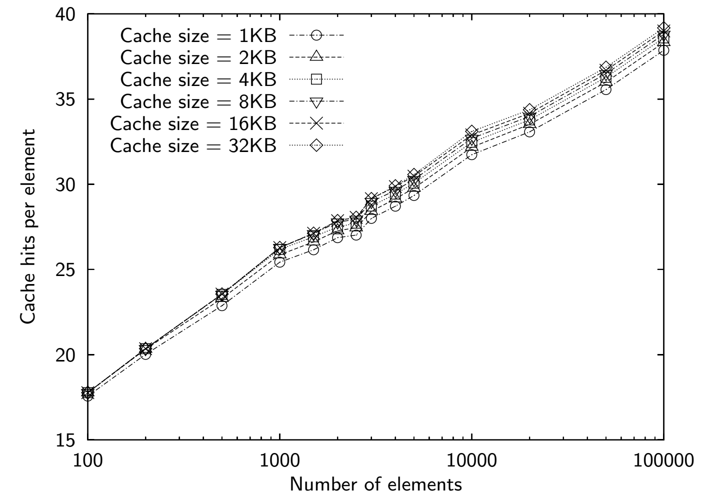
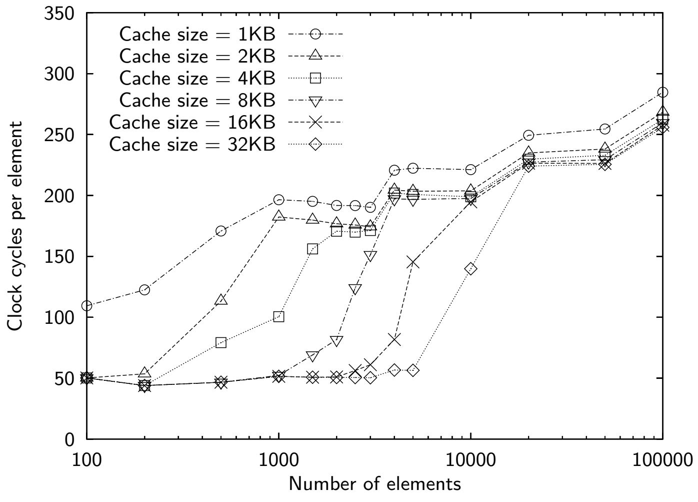
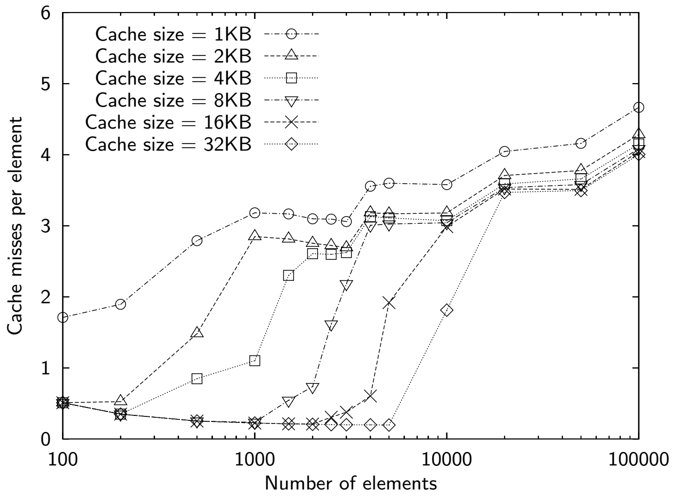


Figure 5.9: Influence of the cache size on introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

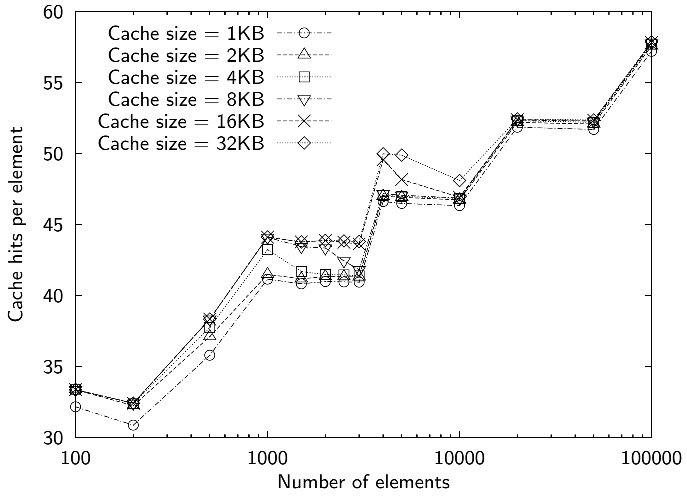


Figure 5.10: Influence of the cache size on mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

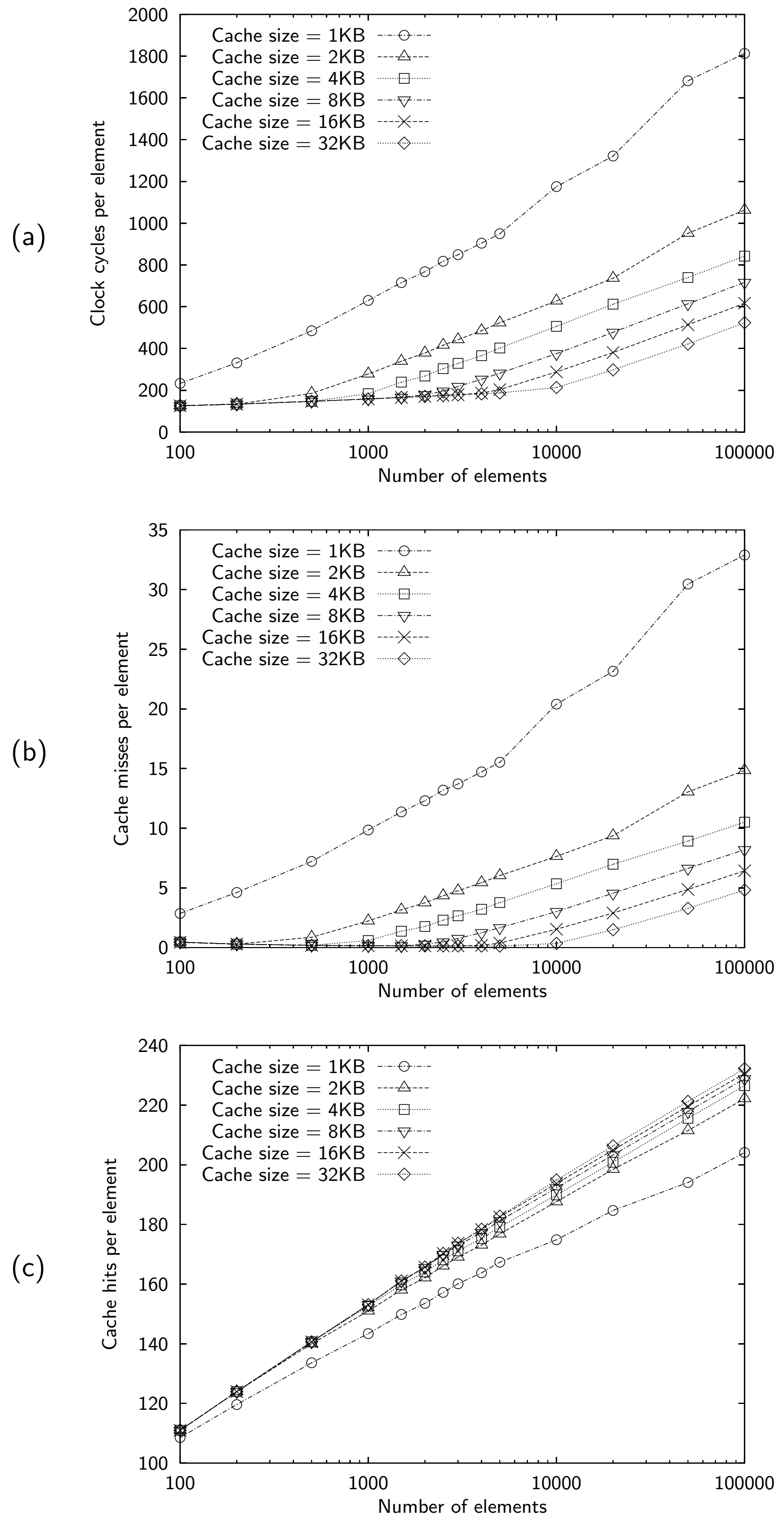
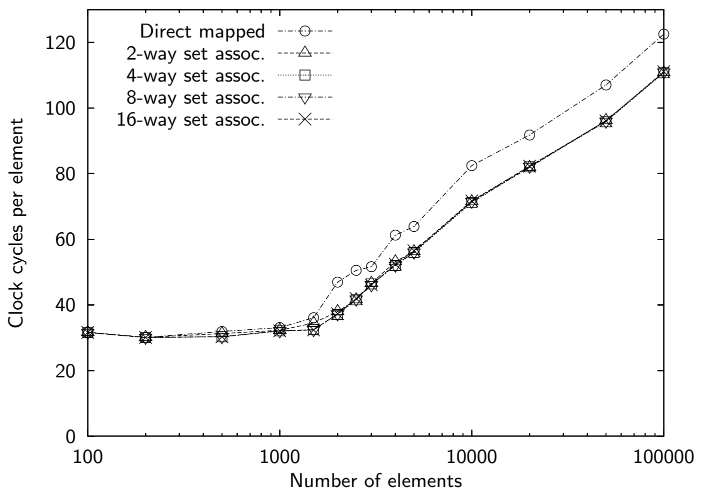
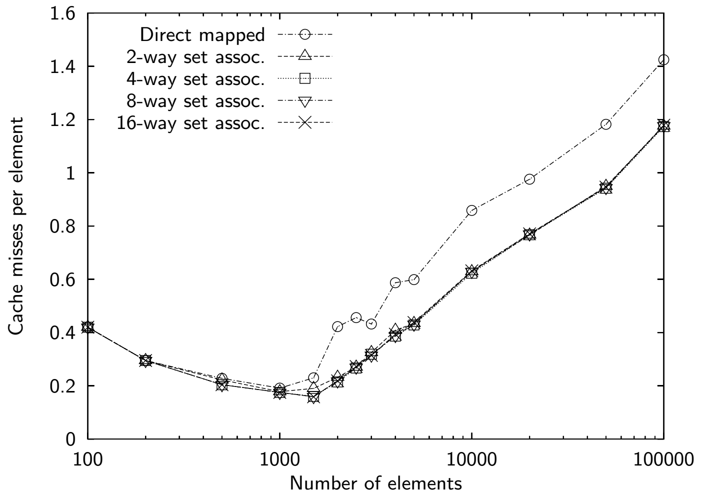


Figure 5.11: Influence of the cache size on heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

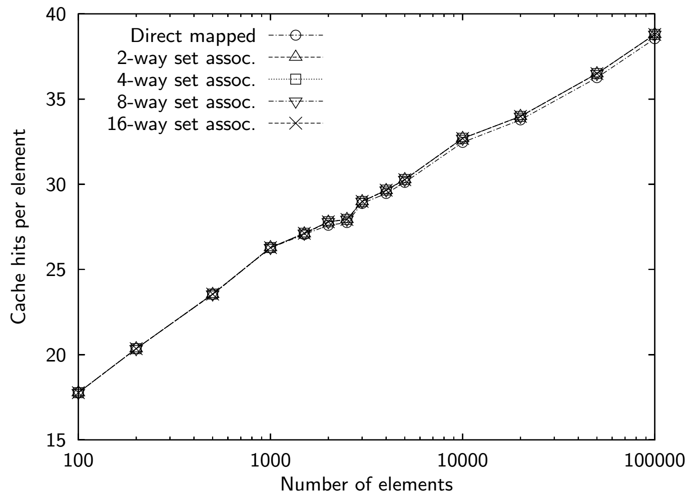
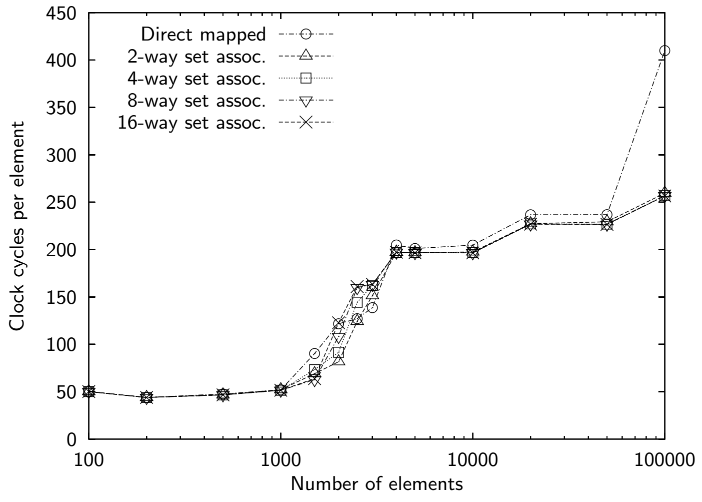
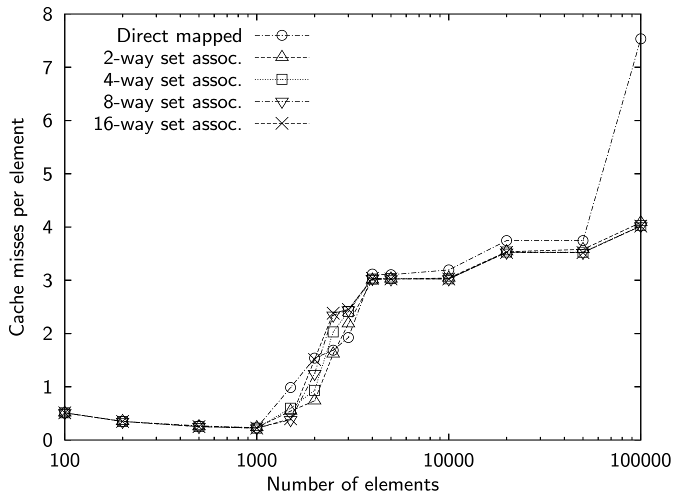


Figure 5.12: Influence of the associativity on introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

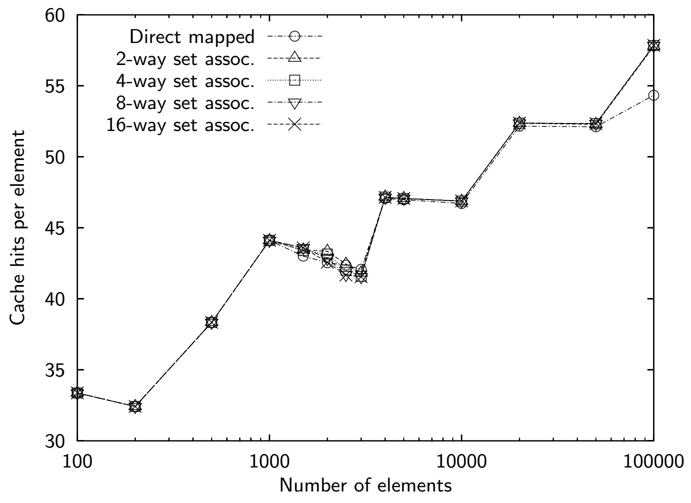
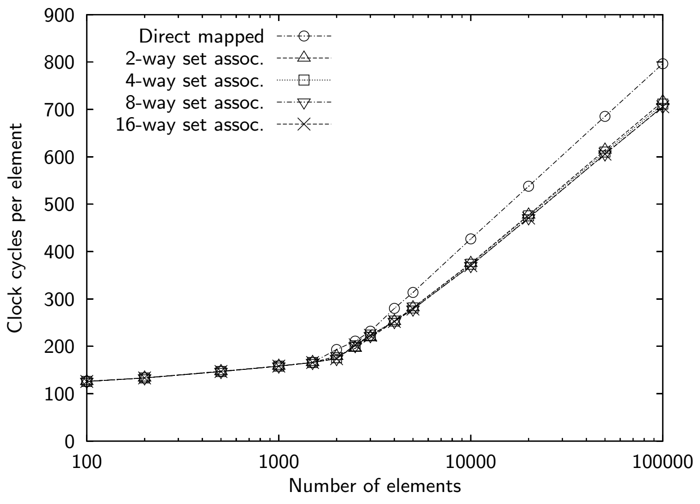
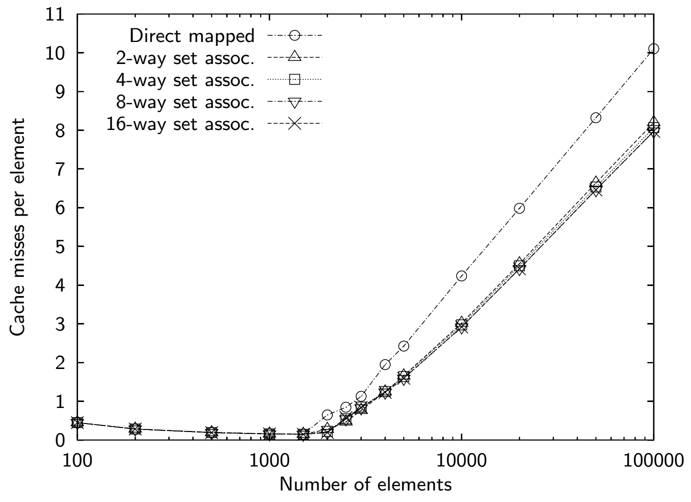


Figure 5.13: Influence of the associativity on mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

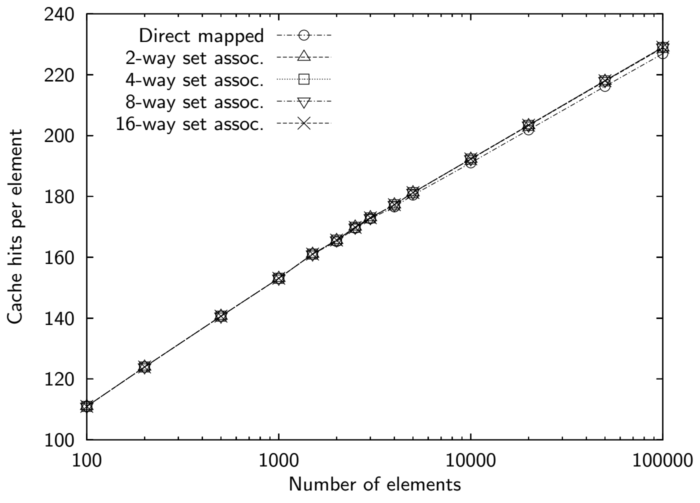
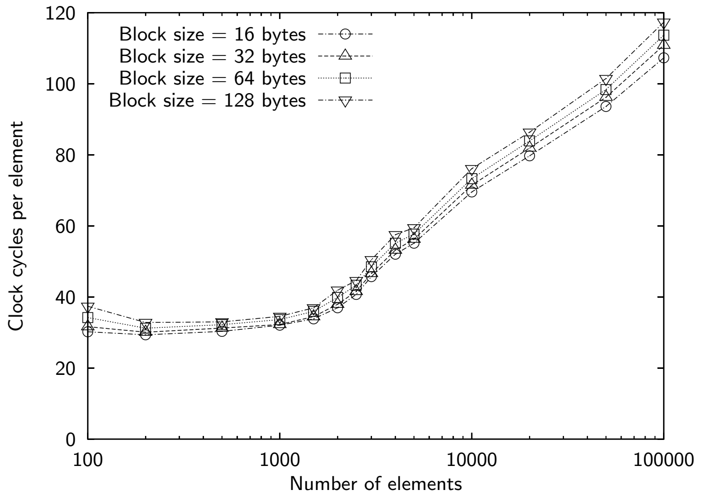
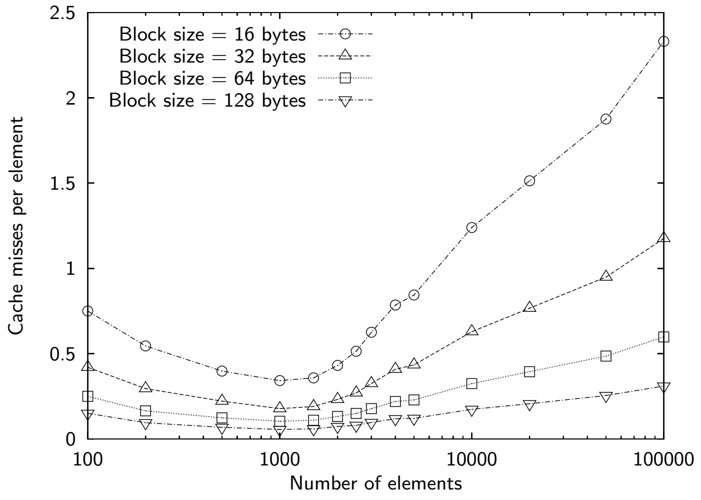


Figure 5.14: Influence of the associativity on heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

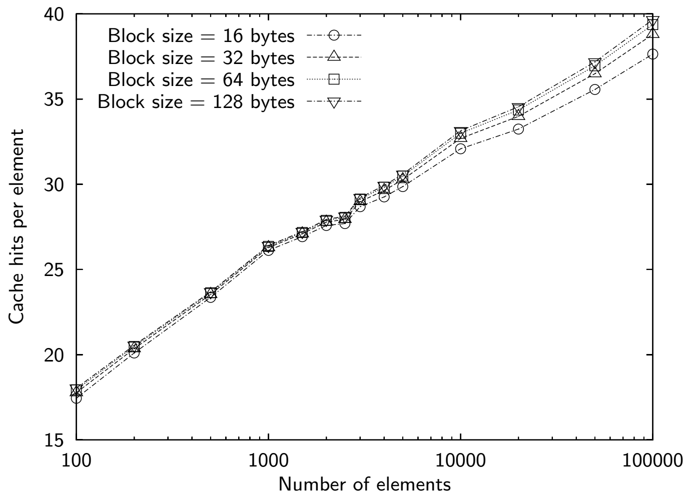
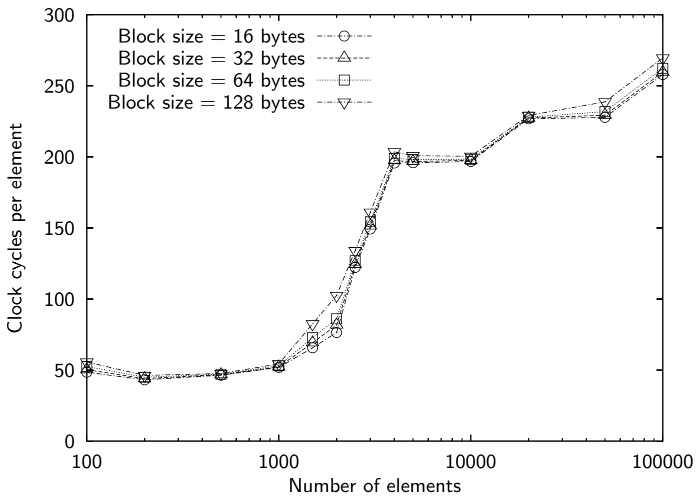
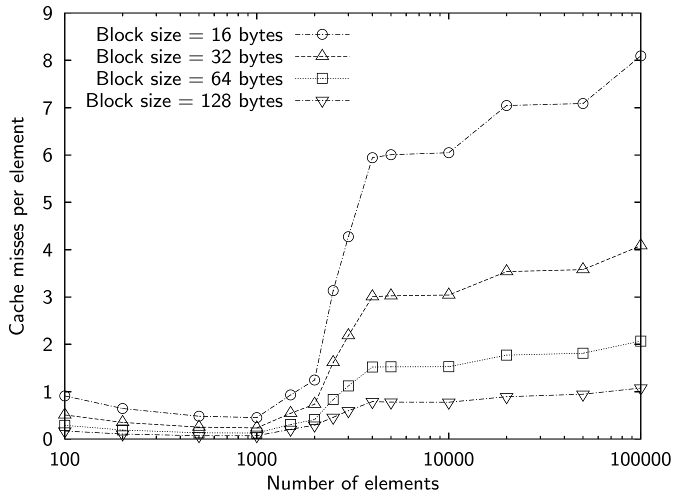


Figure 5.15: Influence of the block size on introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

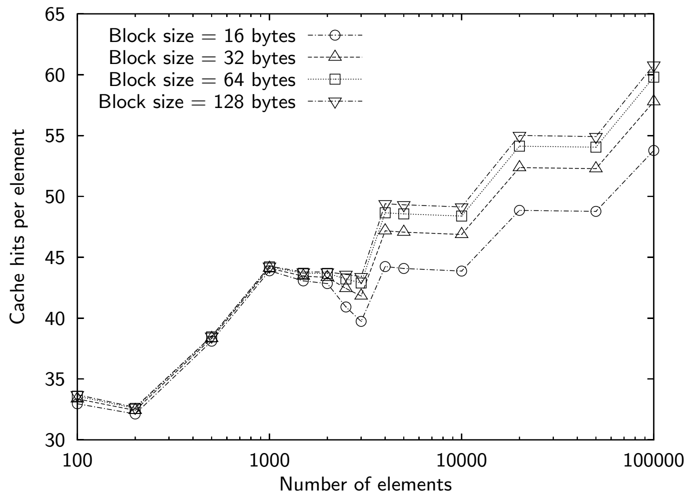


Figure 5.16: Influence of the block size on mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

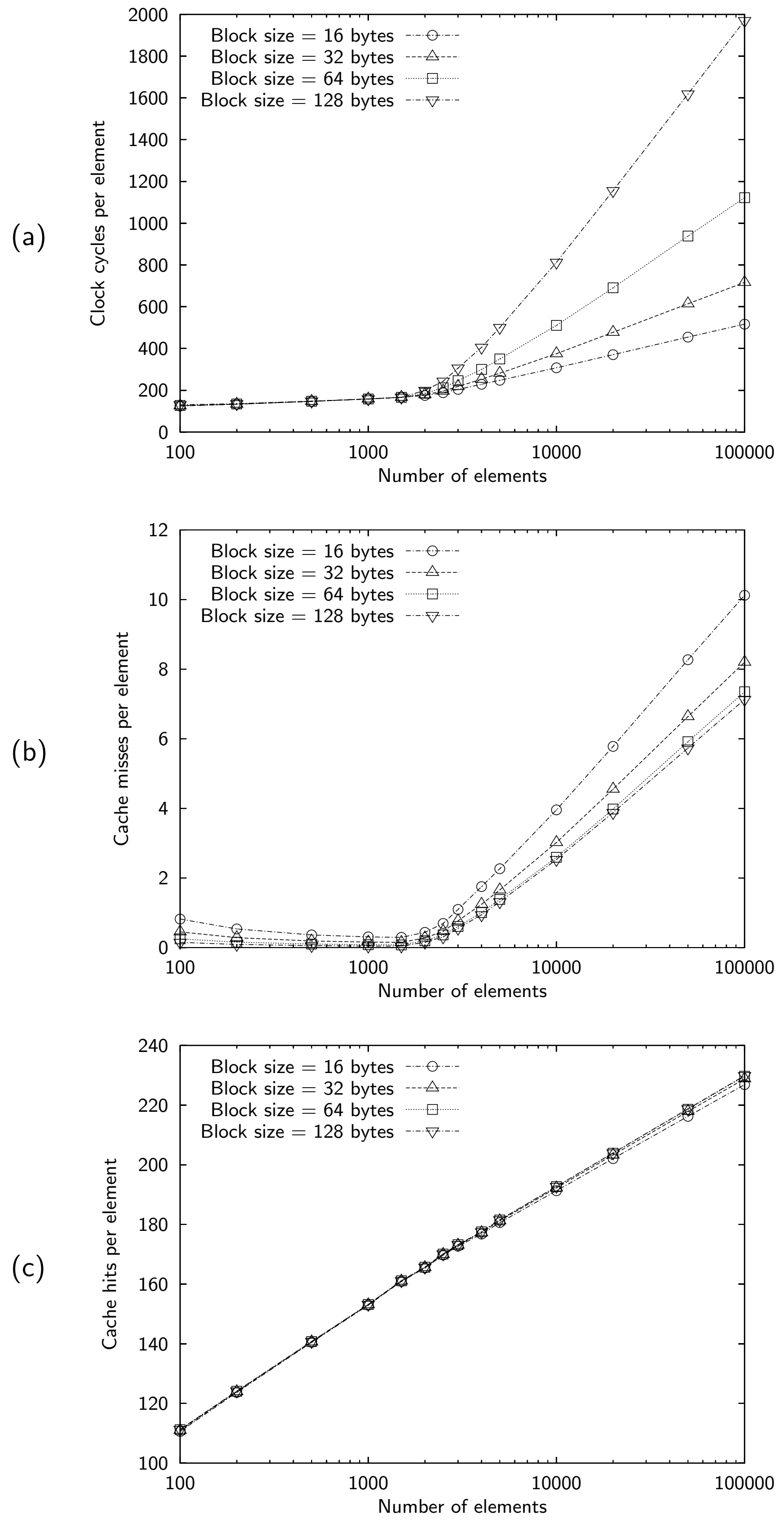
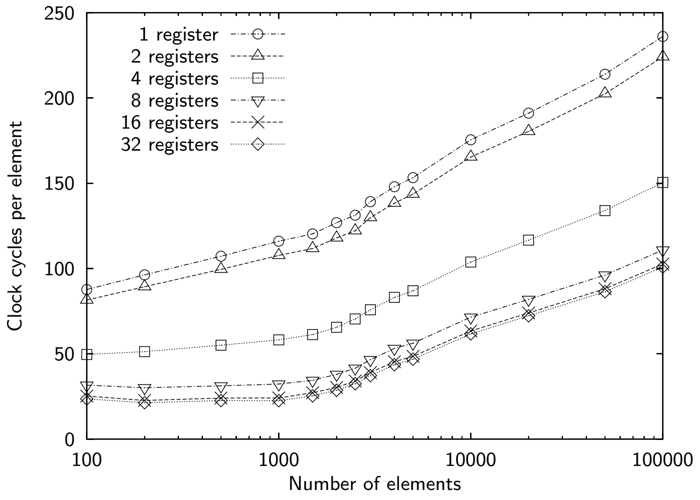
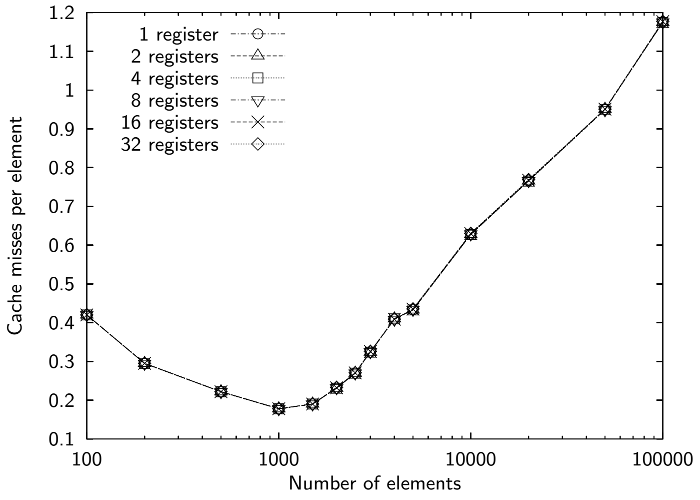


Figure 5.17: Influence of the block size on heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

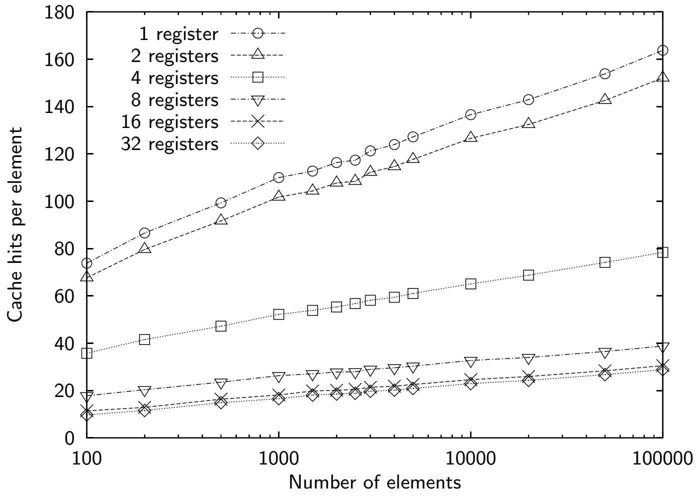
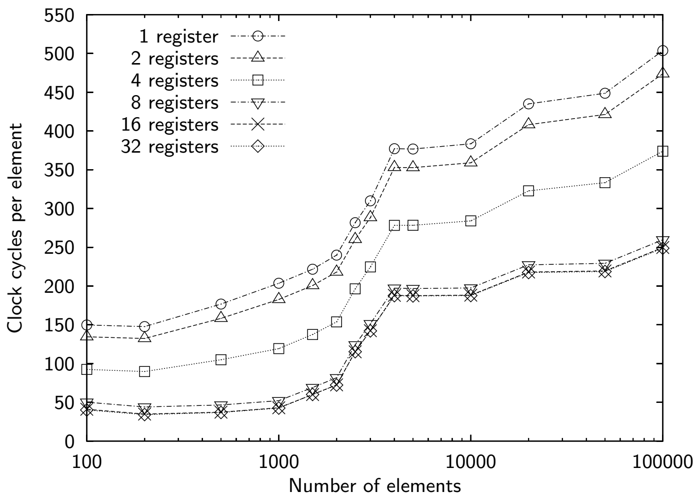
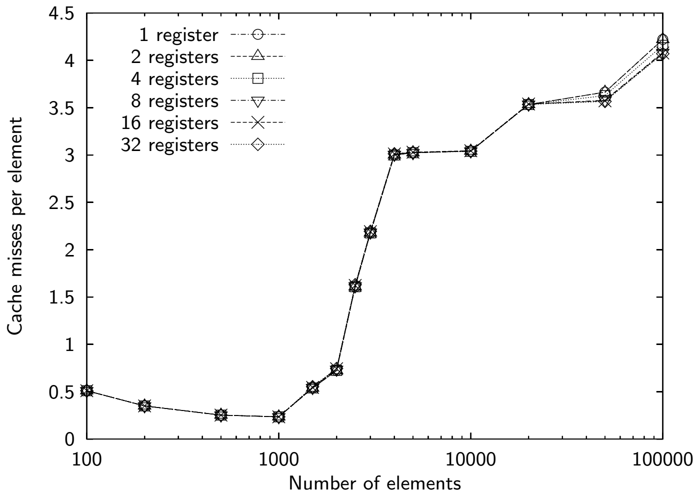


Figure 5.18: Influence of the number of registers on introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

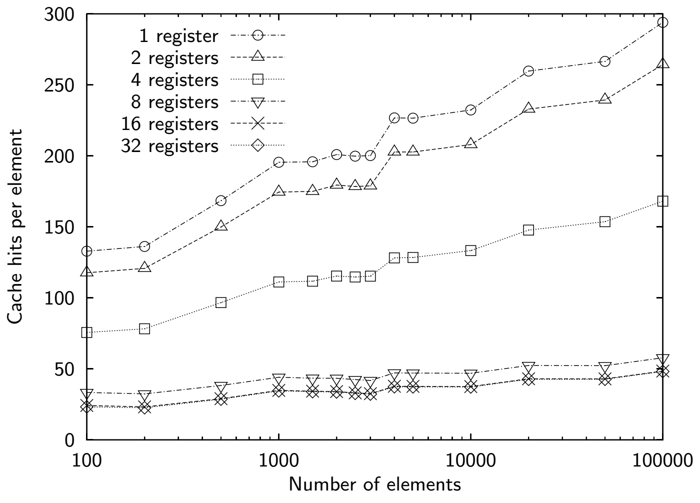
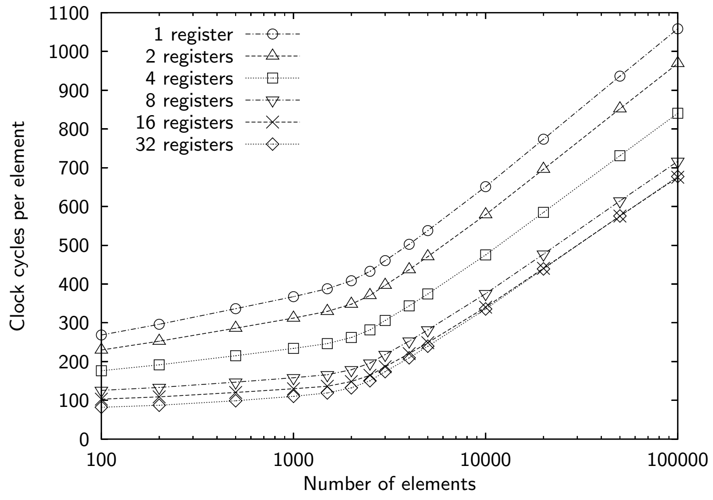
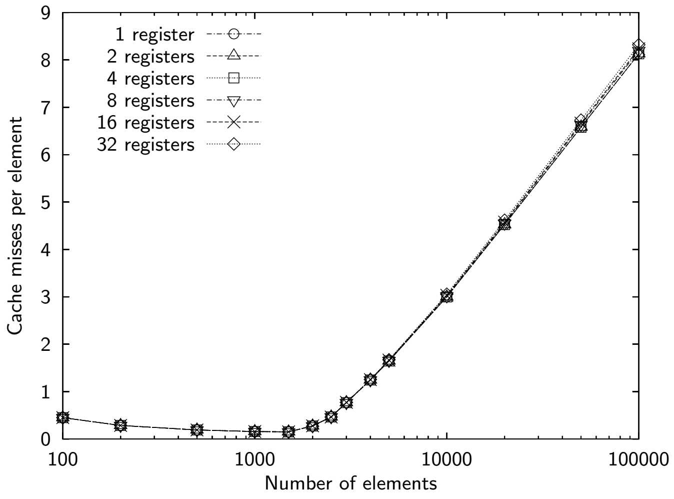


Figure 5.19: Influence of the number of registers on mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

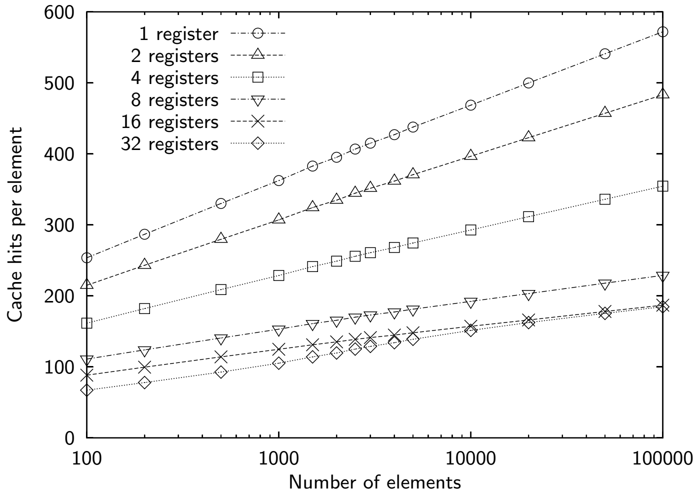
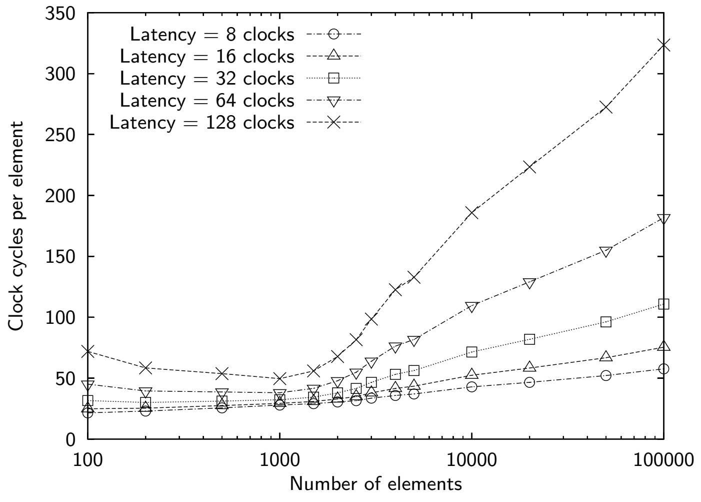
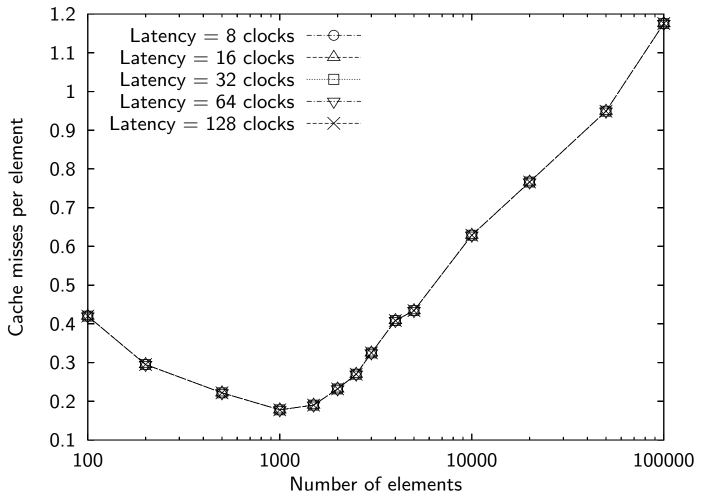


Figure 5.20: Influence of the number of registers on heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

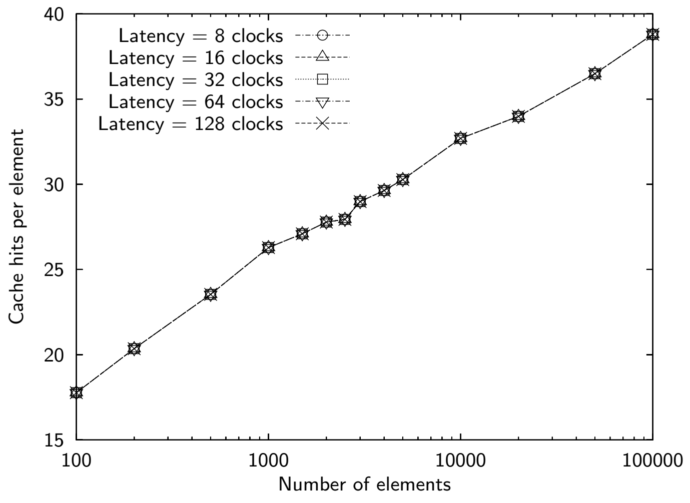
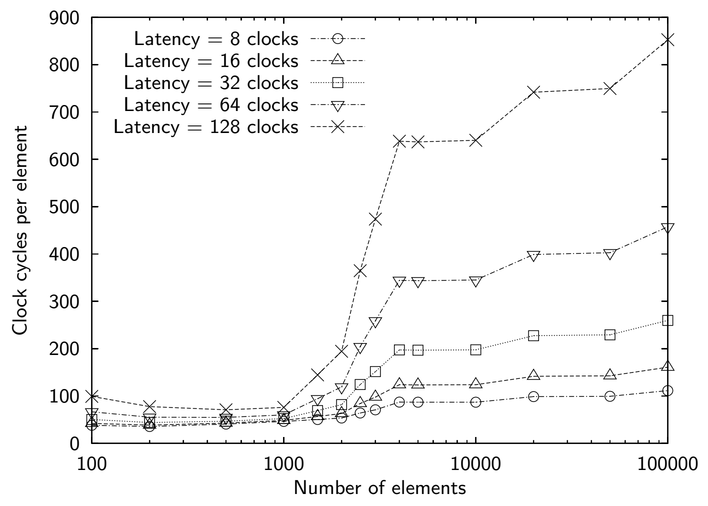
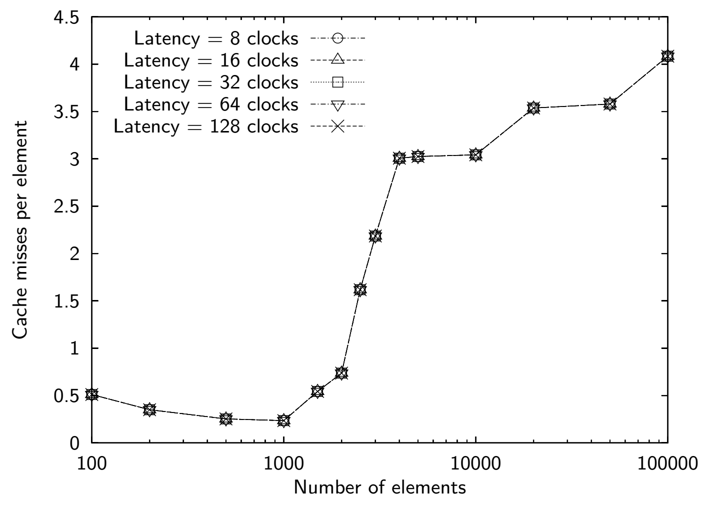


Figure 5.21: Influence of the latency of main memory on introsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

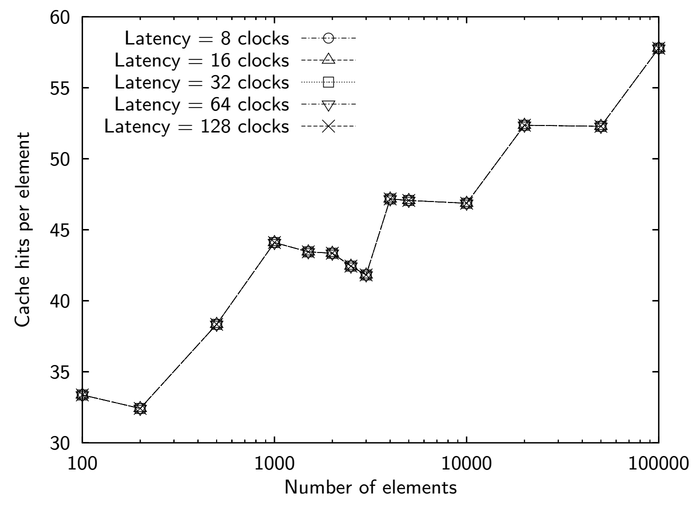
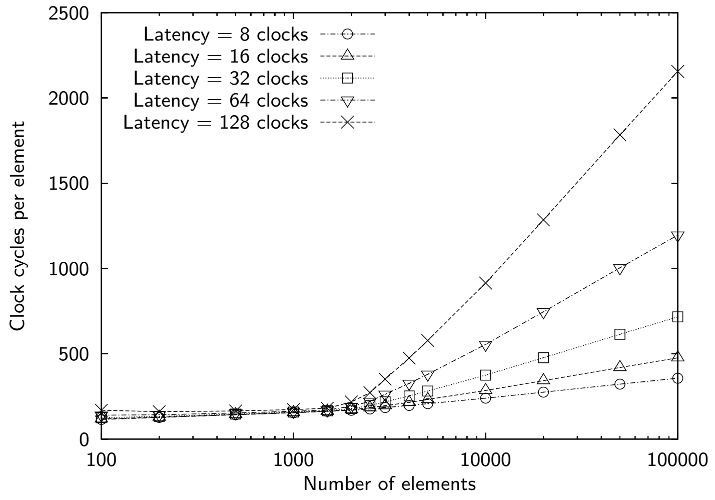
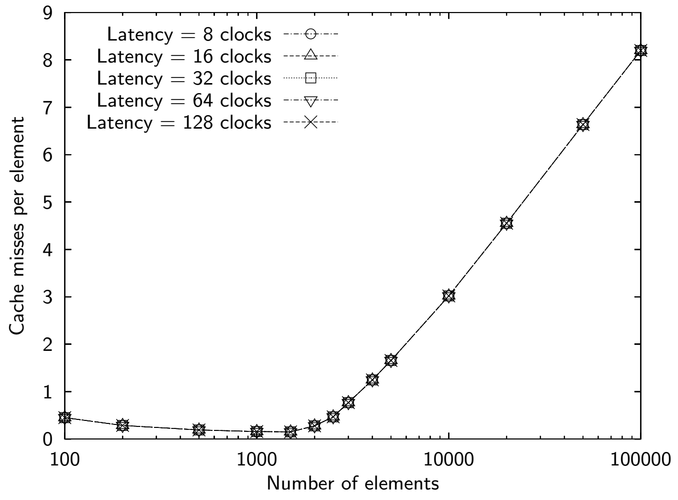


Figure 5.22: Influence of the latency of main memory on mergesort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

(a)



(b)



(c)

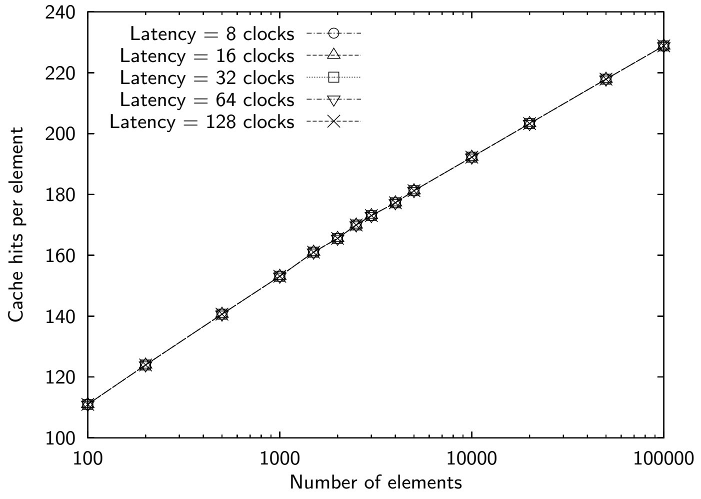


Figure 5.23: Influence of the latency of main memory on heapsort. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per element.

5.4.1 Early Finalization

In the original Introsort, the FINAL-INSERTION-SORT performed at the very end may read almost all elements again from main memory when sorting a large sequence. With "early finalization," insertion sort is performed at the end of *each* partitioning. Since the subarray at the end of partitioning is small (only of 9–16 elements), it is highly likely that all the elements are in the cache. So sorting them may not incur any cache misses. FINAL-INSERTION-SORT is thus eliminated by this optimization and cache misses caused by FINAL-INSERTION-SORT are expected to be eliminated as well.

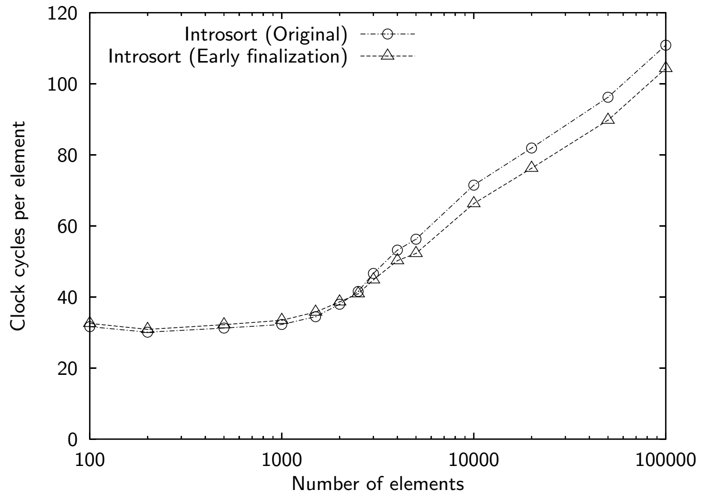
The algorithm EARLY-INTROSORT calls EARLY-INTROSORT-LOOP if the given sequence is not empty. EARLY-INTROSORT-LOOP is quite similar to INTROSORT-LOOP except that it performs the insertion sort after the **while** loop and that it takes an extra argument *real_first*, which simply holds the very first element of the entire sequence and remains unchanged throughout the execution. The new argument *real_first* is used to decide which algorithm to call INSERTION-SORT or UNGUARDED-INSERTION-SORT. Since the latter is slightly efficient, it is preferred if the current region [*first*, *last*) is *not* the beginning of the entire sequence.

```
EARLY-INTROSORT($first, last$)
1: if $first \neq last$ then
2:     EARLY-INTROSORT-LOOP($first, last, 2\lg(last - first), first$)
3: end if
```

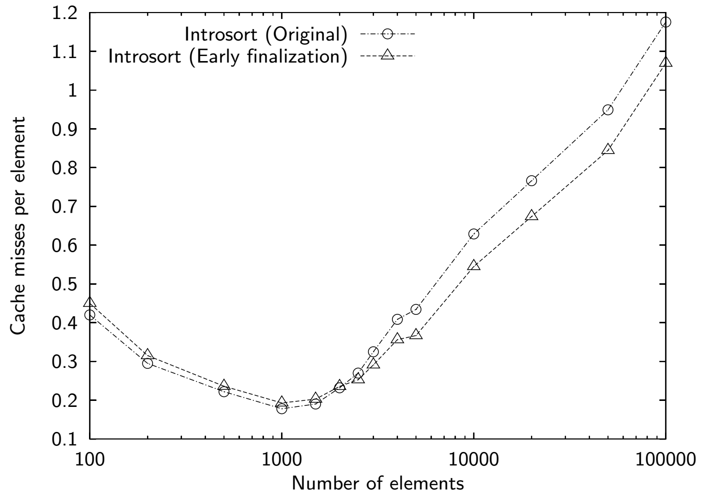
```
EARLY-INTROSORT-LOOP($first, last, depth\_limit, real\_first$)
 1: while $last - first > 16$ do
 2:     if $depth\_limit = 0$ then
 3:         HEAPSORT($first, last$)
 4:         return
 5:     end if
 6:     $depth\_limit \leftarrow depth\_limit - 1$
 7:     $m \leftarrow$ MEDIAN($val[first], val[first + \lfloor (last - first)/2 \rfloor], val[last - 1]$)
 8:     $cut \leftarrow$ UNGUARDED-PARTITION($first, last, m$)
 9:     EARLY-INTROSORT-LOOP($cut, last, depth\_limit, real\_first$)
10:     $last \leftarrow cut$
11: end while
12: if $first = real\_first$ then
13:     INSERTION-SORT($first, last$)
14: else
15:     UNGUARDED-INSERTION-SORT($first, last$)
16: end if
```

Figure 5.24 shows the improvement of the early finalization. The number of cache misses in graph (b) has decreased slightly in larger sequences. The theoretical reduction of the miss count is $1/8 = 0.125$ because FINAL-INSERTION-SORT may read the entire sequence from main memory, causing one cache miss per 32-byte block. The observed reduction of the miss count at $N = 100000$ is 0.11, which is close to the theoretical value. The clock cycles, graph (a), also shows a small improvement with large sequences, about 7% with $N = 100000$.

(a)



(b)



(c)

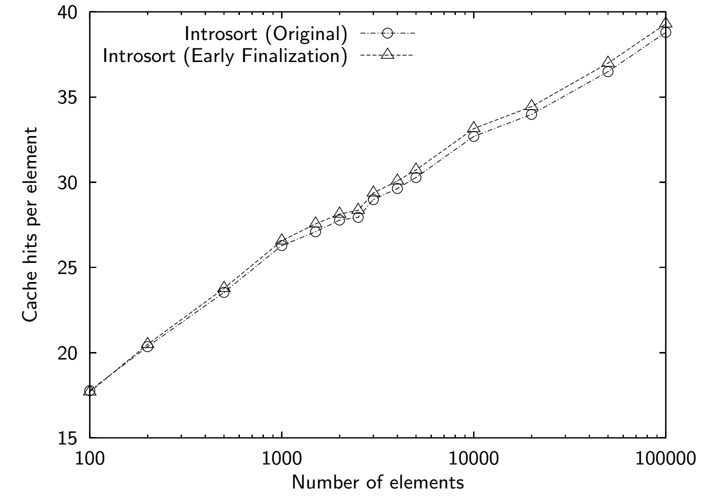
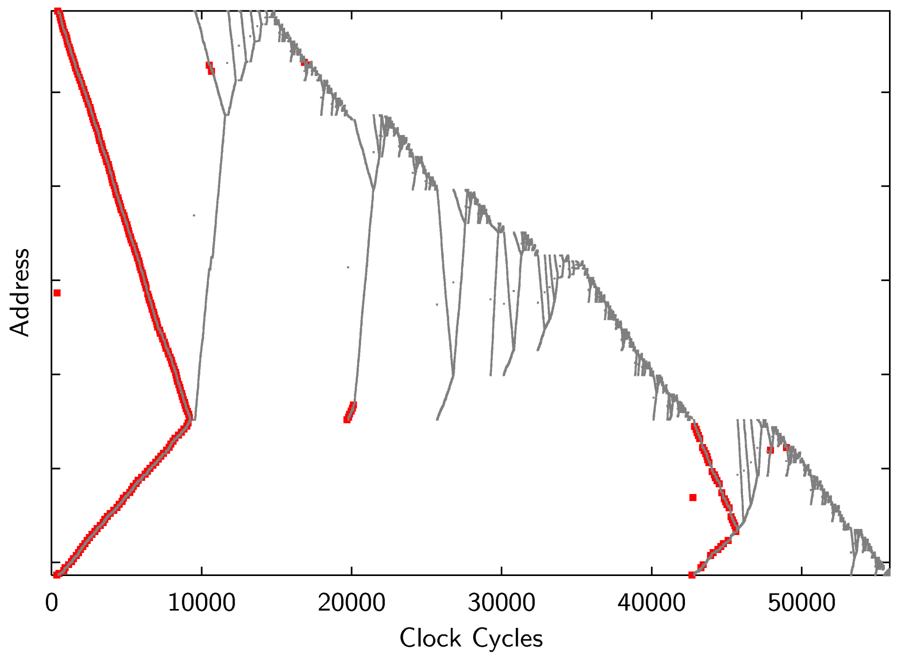


Figure 5.24: Improvement of the early finalization. (a) clock cycles per element, (b) cache misses per element, and (c) cache hits per elements.

(a)



(b)

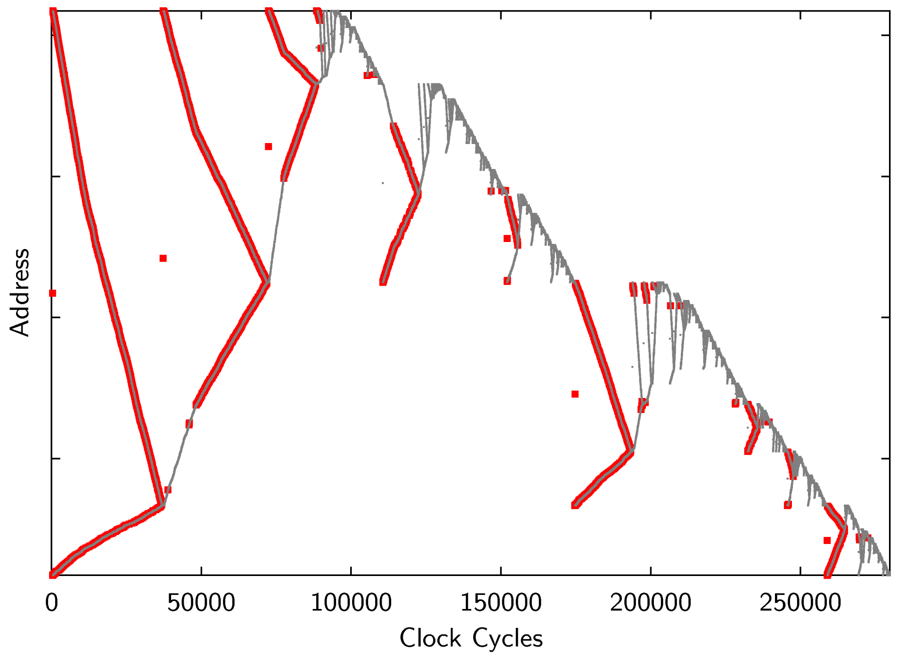


Figure 5.25: Trace plots of introsort with early finalization, sorting (a) 1500 and (b) 5000 elements.

5.4.2 Lomuto's Partition

It turns out that early finalization reduces the number of cache misses per element by at most 0.125. That is, the most cache misses are incurred by partitioning. Then, how can partitioning be improved? In fact, it is difficult because UNGUARDED-PARTITION is already quite efficient in terms of cache performance.

The partitioning algorithm UNGUARDED-PARTITION, based on the famous Hoare's partitioning algorithm, reads and writes each element at most once. To read once is mandatory to obtain its value. And the value is overwritten if the value is not already in a suitable position. Since the read and write is done successively, an element is brought from main memory only once, which is optimal. Cache performance can be improved only when an algorithm makes inefficient memory accesses, reading the same element many times from main memory, for example. Since UNGUARDED-PARTITION makes minimal accesses to the elements, it seems to be impossible to improve the algorithm further. However, it is worth trying other algorithms because it tells us how UNGUARDED-PARTITION is efficient.

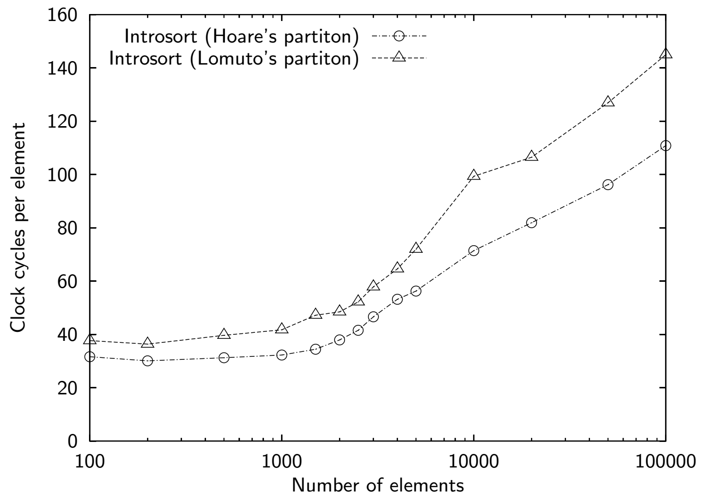
The algorithm used here is called Lomuto's partition, which Jon Bentley introduced in his book [3]. The algorithm LOMUTO-PARTITION-PIVOT places all elements in a sequence $[first, last)$ that are smaller than *pivot* before all elements that are greater than or equal to *pivot*, and then returns the first position of the latter subsequence. The pivot is explicitly given to the algorithm, unlike several implementations available that use the first or the last element as a pivot. In Lomuto's algorithm, two pointers *lessthan* and *first* both move from the beginning of the sequence towards the end, whereas in Hoare's algorithm, two pointers *first* and *last* start from both ends of the sequence and move towards each other.

```
LOMUTO-PARTITION-PIVOT($first$, $last$, $pivot$)
1: $lessthan \leftarrow first - 1$
2: while $first \neq last$ do
3:     if $val[first] < pivot$ then
4:         $lessthan \leftarrow lessthan + 1$
5:         ITER-SWAP($lessthan, first$)    ▷ Swap two values that $lessthan$ and $first$ point to.
6:     end if
7:     $first \leftarrow first + 1$
8: end while
9: return $lessthan + 1$
```

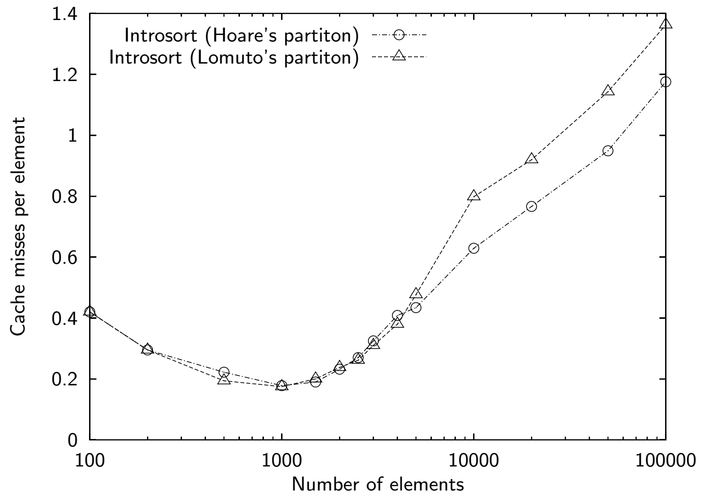
Figure 5.26 compares introsort using Lomuto's partition with the original one which is based on Hoare's partition. The original introsort has a clear advantage in terms of the number of clock cycles, as shown in graph (a). The number of cache misses is competitive when $N \leq 4000$, but Lomuto's partition incurs more cache misses thereafter. Graph (c) shows that Lomuto's partition consistently makes more cache hits, and the difference grows as N becomes larger.

The trace plot in Figure 5.27(b) points out the weakness of Lomuto's partition. Look at the first partition performed in the clock range 0–40000. Two pointers start from the beginning of the sequence. It is natural that the pointer ahead (*first*) causes cache misses throughout the first partitioning, but the latter pointer (*lessthan*) does also. That is, some elements are read twice from main memory, which is not optimal behavior.

(a)



(b)



(c)

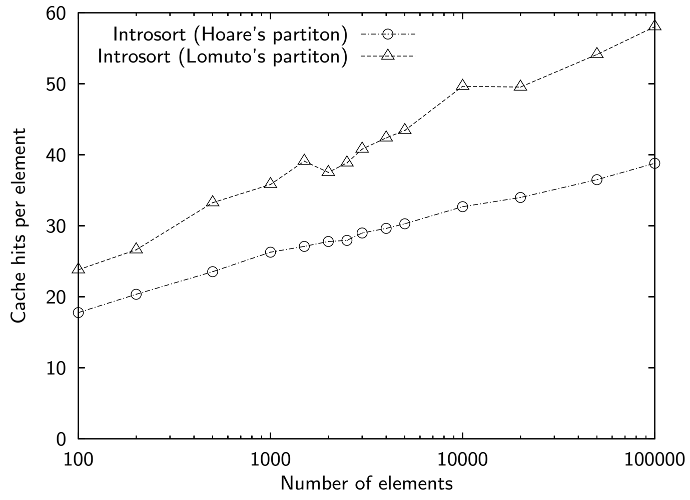
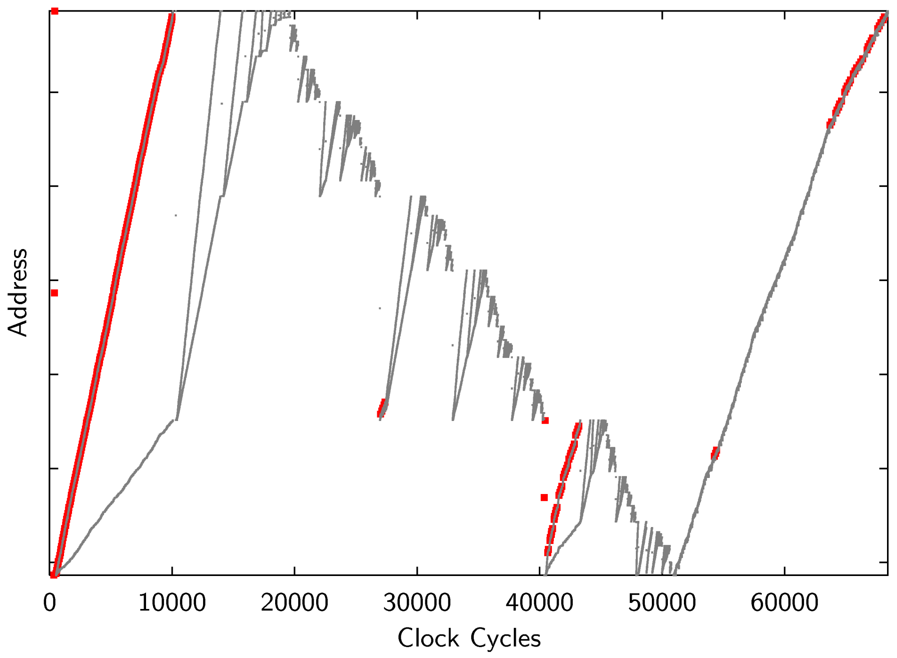


Figure 5.26: Hoare's partition vs. Lomuto's partition.

(a)



(b)

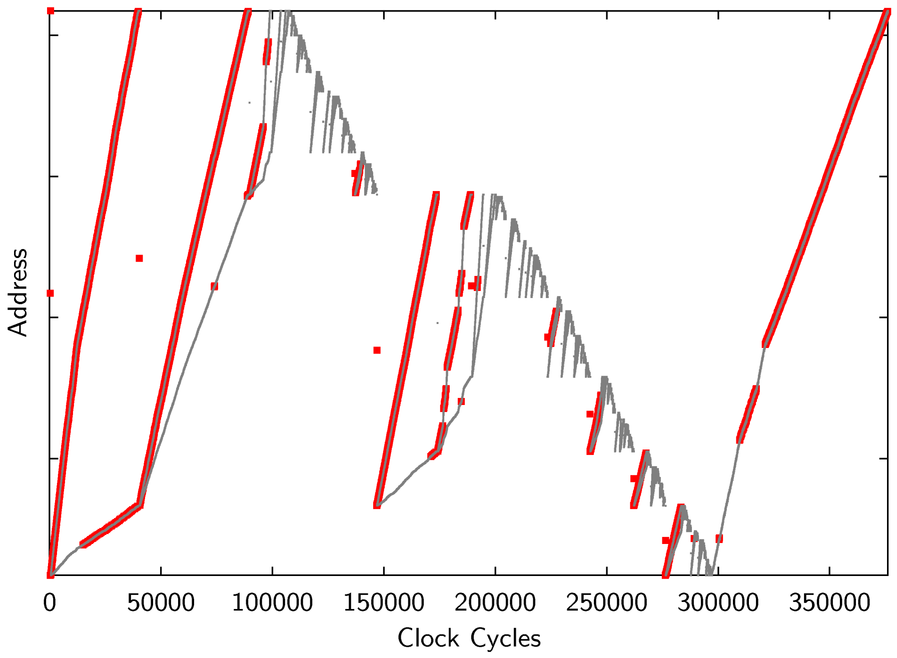


Figure 5.27: Trace plots of introsort with Lomuto's partition, sorting (a) 1500 and (b) 5000 elements.

5.5 Improvements Over Mergesort

As mentioned earlier, the performance of the mergesort is almost determined by the number of merge steps, or the number of calls of function MERGE-SORT-LOOP. Therefore, one of the optimization strategies with mergesort is to reduce the number of merge steps. The Mergesort in its original GCC version starts merging from subarrays of 7 elements, which are pre-sorted using insertion sort.

"Tiled mergesort" is based on the same idea, but starts merging from much larger subarrays, half the cache size. This unit is called a *tile* and corresponds to 1000 elements with the default cache simulator. The pre-sorting is expected to be done efficiently since all the elements fit into the cache. But note that it should be stable. Therefore, mergesort is again used to pre-sort inside a tile, which starts merging from subarrays of 16 elements. The reason why the tile size is a half the cache size is that the other half is used by a temporary buffer during the pre-sorting. The number of merge passes at $N = 100000$ is reduced from 28 to 8 by this optimization.

TILED-MERGESORT allocates a temporary buffer and calls TILED-MERGESORT-INTERNAL. The algorithm fails immediately if the requested size could not be allocated, unlike the original mergesort which manages to work with a temporary buffer smaller than requested.

```
TILED-MERGESORT($first$, $last$, $tile\_size$)
 1: $buf \leftarrow$ pointer to the temporary buffer of size $last - first$
 2: $tile\_elem \leftarrow$ number of elements in a tile
 3: TILED-MERGESORT-INTERNAL($first$, $last$, $buf$, $tile\_elem$)
```

The function TILED-MERGESORT-INTERNAL is called from two different contexts in the algorithm. First, it is called from TILED-MERGESORT, as mentioned above. At that time, *chunk_elem*, the number of element in one chunk, is 1000 (with the default cache parameters). Then it calls CHUNK-MERGESORT to pre-sort inside the chunk (line 4), and enters the merge loop (lines 8–15). The second context is a call from CHUNK-MERGESORT, which will be explained shortly. In the second context, *chunk_elem* is 16 and thus insertion sort is used to pre-sort the chunk (line 6). The first context sorts the entire sequence and the second context pre-sorts each tile.

```
TILED-MERGESORT-INTERNAL($first$, $last$, $buf$, $chunk\_elem$)
 1: $nelem \leftarrow first - last$
 2: $buf\_last \leftarrow buf + nelem$
 3: if $16 < chunk\_elem$ then
 4:     CHUNK-MERGESORT($first$, $last$, $buf$, $chunk\_elem$)
 5: else
 6:     CHUNK-INSERTION-SORT($first$, $last$, $chunk\_elem$)
 7: end if
 8: $nchunk \leftarrow (nelem + chunk\_elem - 1)/chunk\_elem$
 9: while $nchunk > 1$ do
10:     MERGE-SORT-LOOP($first$, $last$, $buf$, $chunk\_elem$)
11:     $chunk\_elem \leftarrow chunk\_elem \times 2$
12:     MERGE-SORT-LOOP($buf$, $buf\_last$, $first$, $chunk\_elem$)
13:     $chunk\_elem \leftarrow chunk\_elem \times 2$
14:     $nchunk \leftarrow (nchunk + 3)/4$
15: end while
```

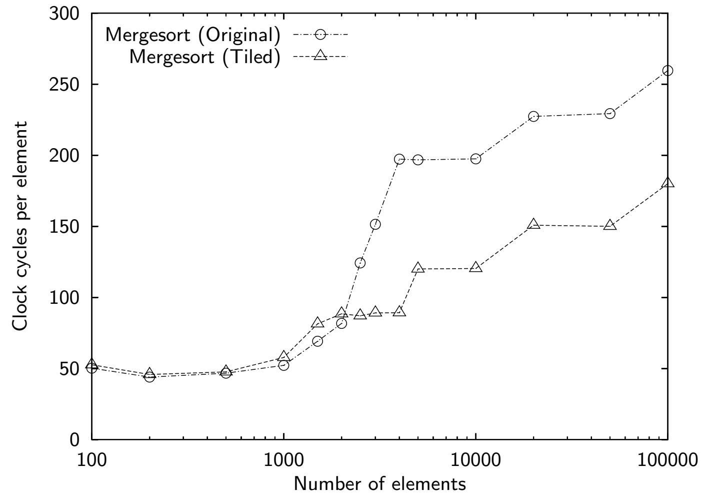
Pre-sorting inside each tile is done by CHUNK-MERGESORT, which is similar to CHUNK-INSERTION-SORT. This function calls TILED-MERGESORT-INTERNAL with a chunk size small enough to make it enter the second context.

```
CHUNK-MERGESORT($first$, $last$, $buf$, $chunk\_elem$)
1: while $first + chunk\_elem \leq last$ do
2:     TILED-MERGESORT-INTERNAL($first$, $first + chunk\_elem$, $buf$, 16)
3:     $first \leftarrow first + chunk\_elem$
4: end while
5: if $first < last$ then
6:     TILED-MERGESORT-INTERNAL($first$, $last$, $buf$, 16)
7: end if
```

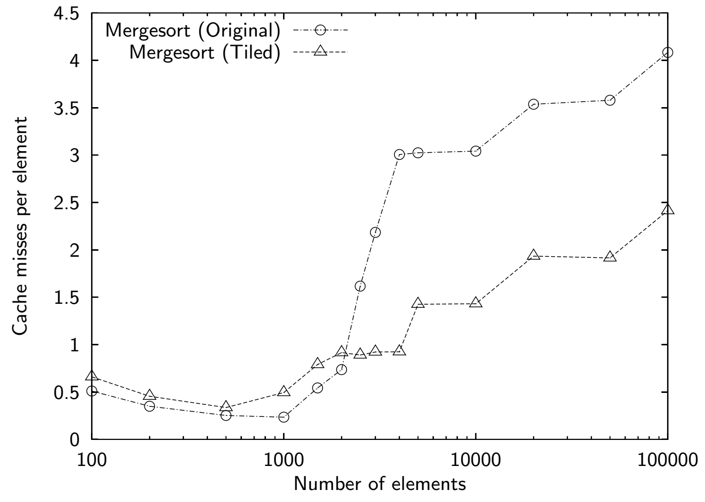
Figure 5.28 shows the significant speedup of the tiled mergesort. The number of clock cycles per element in graph (a) is competitive before $N = 2000$, and superior thereafter. The sudden increase in the original mergesort is no longer observed. When $N \geq 5000$, the difference between two plots looks constant. The number of cache misses in graph (b) shows a similar improvement. However, with smaller N, the tiled mergesort has slightly more misses. The number of cache hits in graph (c) grows in stair steps. The tiled mergesort has fewer hits for $N \leq 3000$, but there are no big differences for larger N.

The trace plot of the tiled mergesort is shown in Figure 5.29. The lower half of the address space is the given sequence and the upper half is the temporary buffer. The difference from the original mergesort is not quite noticeable in graph (a), but graph (b) clearly shows the pre-sorting step in $0 \leq x \leq 300000$, in which five tiles are sorted separately. The five pre-sorted tiles are then merged as usual.

(a)



(b)



(c)

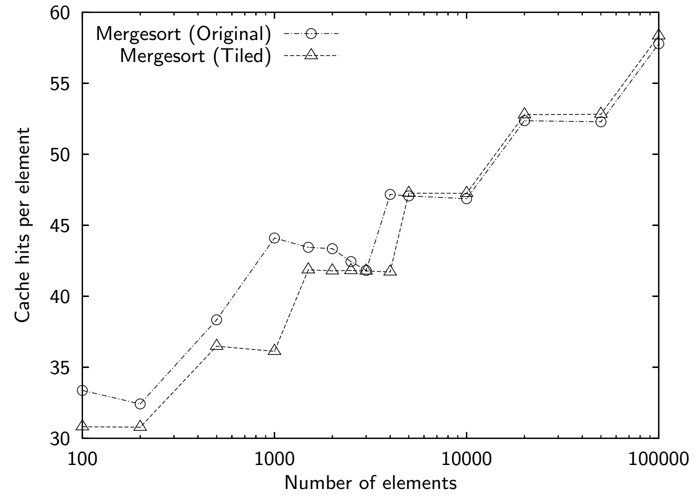
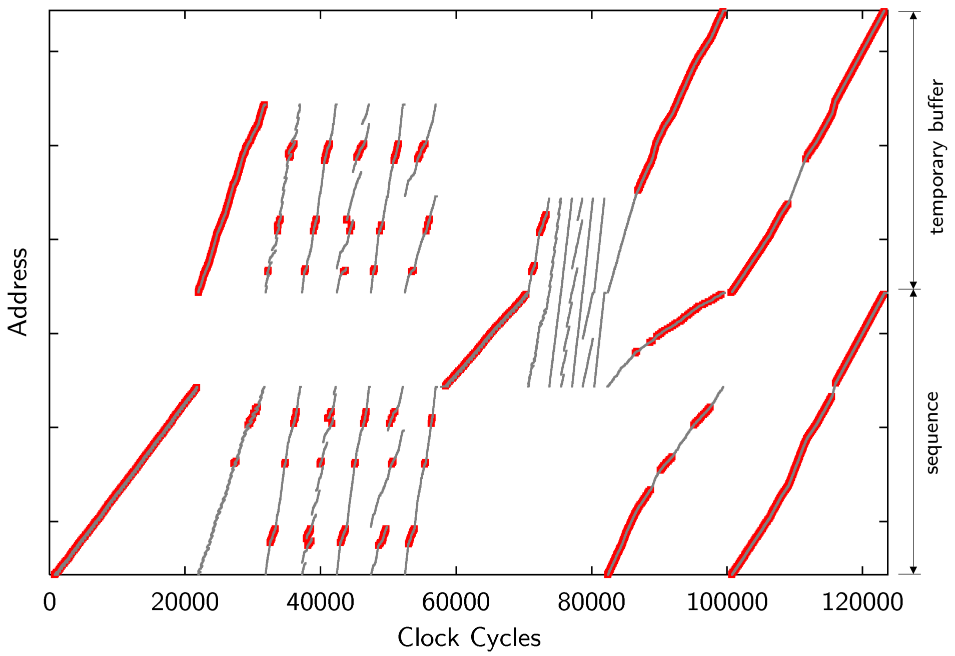


Figure 5.28: Improvements by tiled mergesort. (tile size = 4KB)

(a)



(b)

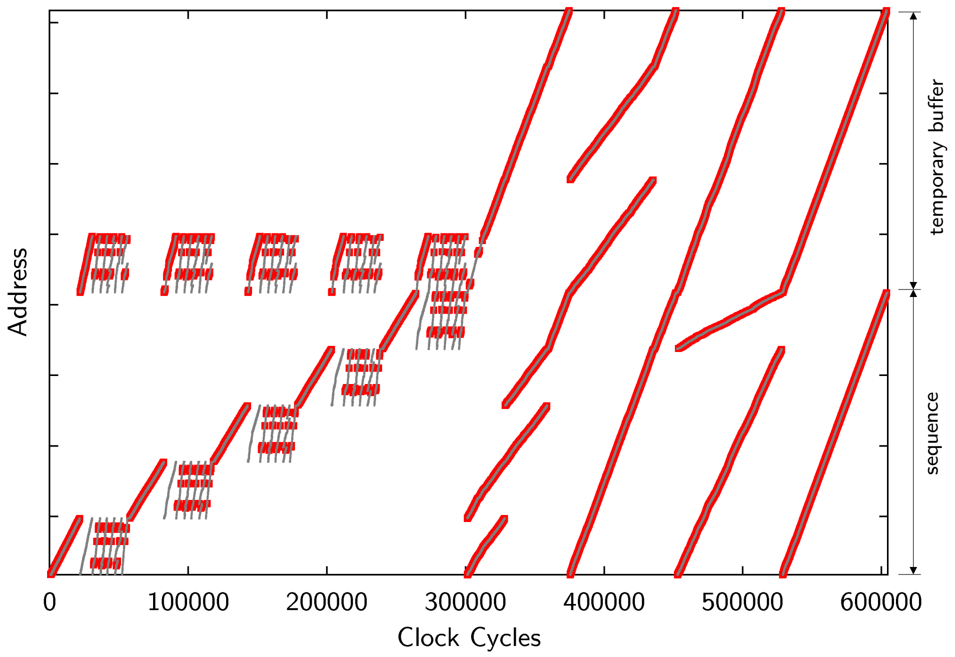


Figure 5.29: Trace plots of tiled mergesort, sorting (a) 1500 and (b) 5000 elements.

5.6 Running Time on Actual Hardware

Developers of cache-conscious algorithms will be interested in the performance on actual hardware platforms as well as on the cache simulator.

Figure 5.30 shows the running time of ordinary STL sorting algorithms executed on the Pentium 4 1.5GHz (L1 = 8KB, L2 = 256KB) and UltraSparc-IIi 333HMz (L1 = 16KB, L2 = 2MB). Saturation points are around $N = 65000$ on the Pentium 4, and around $N = 500000$ on the UltraSparc-IIi.

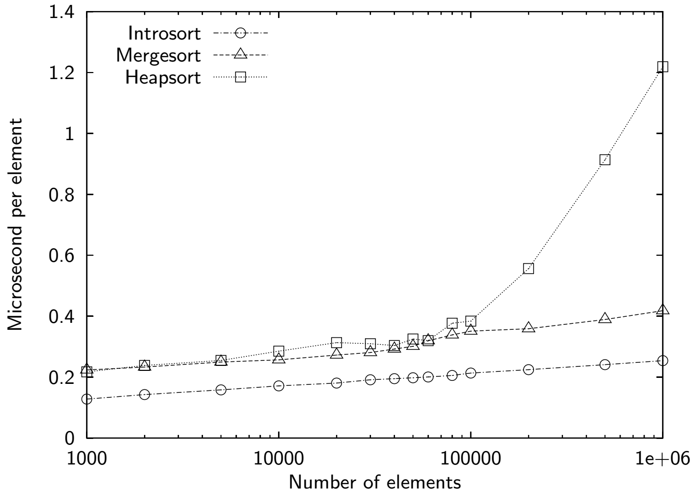
On the Pentium 4, graph (a), the plot of introsort is almost linear throughout the range. The slope of mergesort is similar to introsort, and the gap observed on the simulator is not so apparent. The large growth in heapsort is still there, and it starts from around $N = 100000$, a little after the saturation point. On UltraSparc-IIi, graph (b), introsort is almost linear as well. Mergesort shows a noticeable increase near the end of the plot. It is not clear whether this increase corresponds to the gap observed on the simulator. Heapsort increases linearly until the saturation point, and then shows steep growth. The common points in both graphs are (1) introsort is the fastest, (2) performance of mergesort and heapsort is close to each other before the saturation point, and (3) after the saturation point, mergesort beats heapsort.

Figure 5.31 shows the results on the Pentium 4 of each of the attempts to improve the sorting algorithms as discussed in the last two sections. Unfortunately, the early finalization shown in graph (a) does a little worse than the original introsort. The difference between the two is small and almost fixed. Lomuto's partition, graph (b), presents a similar result, though it seems to be a little better than indicated by the the simulation results. An advantage can be clearly observed in graph (c), tiled mergesort. Incidentally, in the last graph, small gaps can be seen on both plots around $N = 60000$.

Results of the improvement attempts on the UltraSparc-IIi are shown in Figure 5.32. Early finalization, graph (a), does not achieve a meaningful improvement on this platform either. Effectiveness of tiled mergesort in graph (c), as was seen in the simulator and on the Pentium 4, is also observed on the UltraSparc. What is surprising is Lomuto's partition. Contrary to previous results on the simulator and the Pentium 4, on the UltraSparc-IIi it beats Hoare's partition by a significant margin.

Microprocessors hide cache miss latency by sophisticated techniques, such as non-blocking cache and hardware prefetching. Consequently, performance degradation on these two architectures overall is moderate compared to that on the simulator. The improvement of tiled mergesort is modest as well. This means that progression and regression of performance are less obvious on these actual hardware platforms than on the simulator. It is not easy to explain cache behavior without the detailed hardware-level profiling. Cache behavior is often counterintuitive, as observed in Lomuto's partition on the UltraSparc. As for this particular result, no rational explanation is readily apparent, so further examination is needed.

(a)



(b)

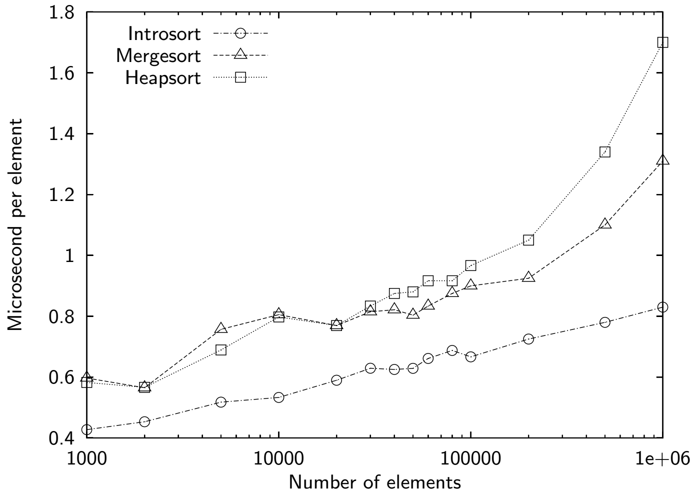
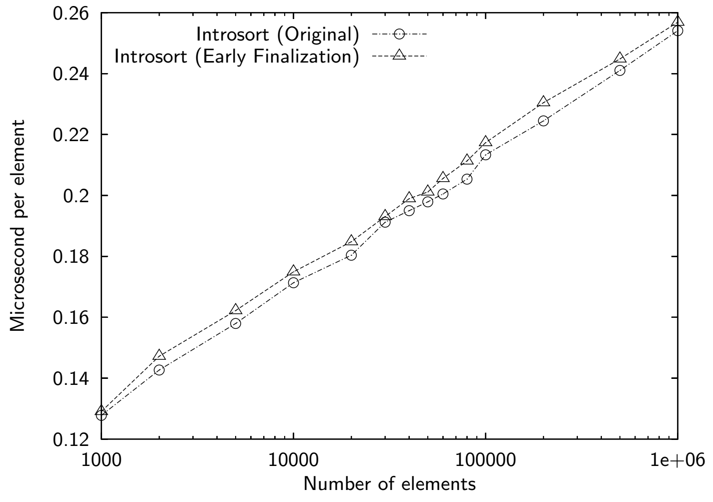
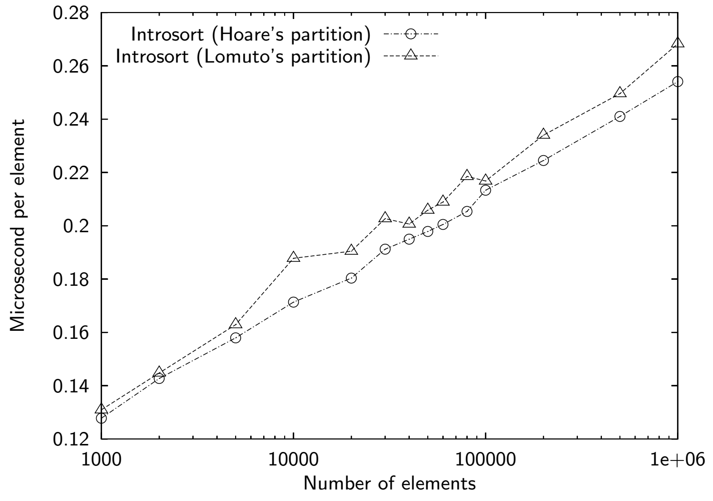


Figure 5.30: Running time of STL sorting algorithms on (a) Pentium 4 and (b) UltraSparc-IIi.

(a)



(b)



(c)

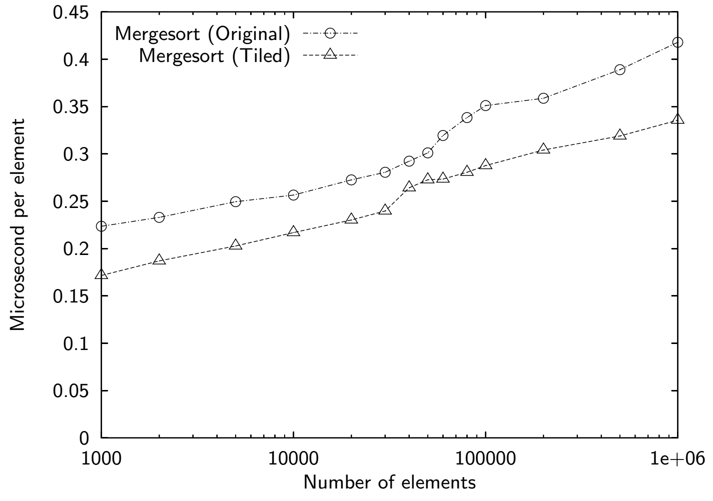
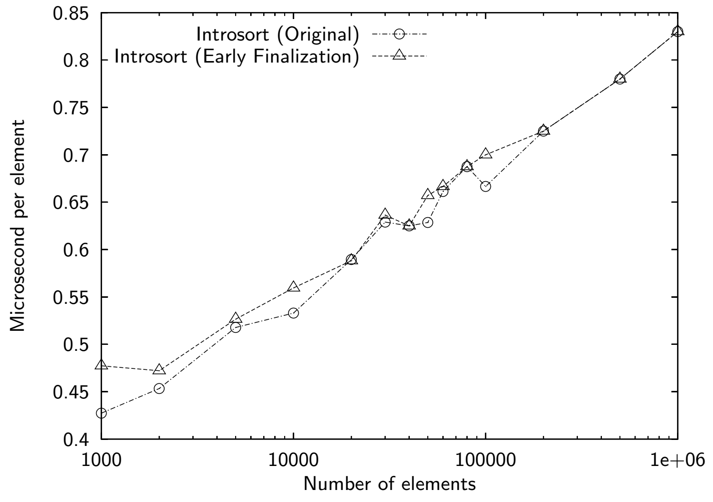
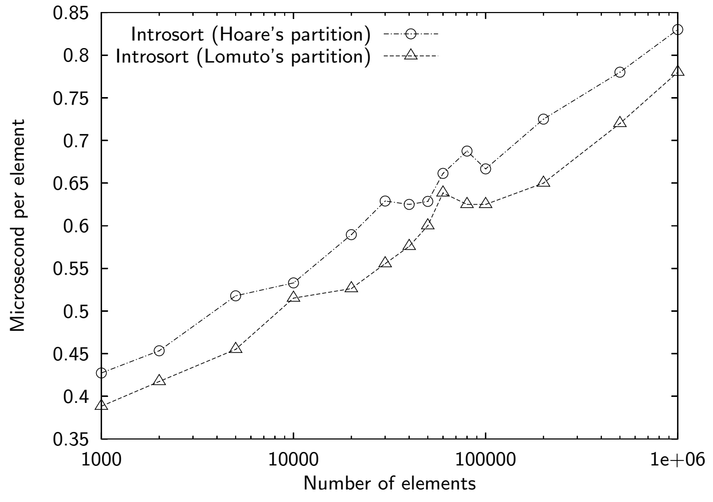


Figure 5.31: Improvements over STL sorting algorithms on Pentium 4 by (a) Early finalization, (b) Lomuto's partition, and (c) Tiled mergesort.

(a)



(b)



(c)

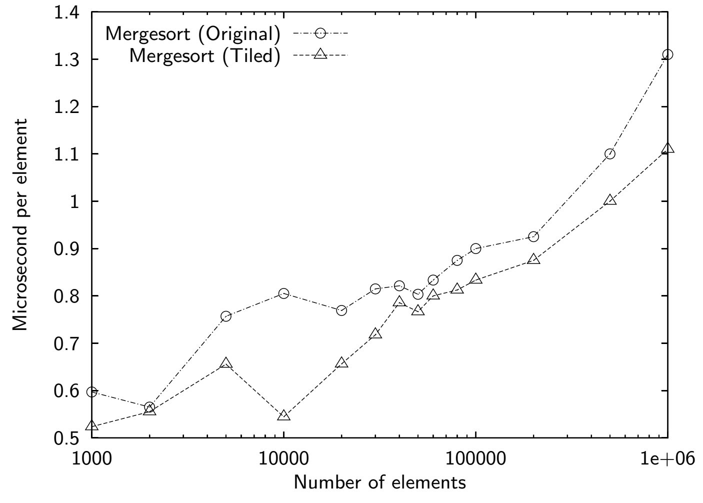


Figure 5.32: Improvements over STL sorting algorithms on UltraSparc-IIi by (a) Early finalization, (b) Lomuto's partition, and (c) Tiled mergesort.

Chapter 6

Conclusion

A portable cache profiling tool based on source-level instrumentation has been developed and experimented with. The tool, named SMAT (STL Memory Access Tracer), implements trace function calls as STL adaptors. Thus SMAT does not require trace functions to be embedded into the target algorithm, unlike other source-based approaches.

SMAT traces *all* objects involved in the computation, including temporary objects. This is different from the technique of object modification based instrumentation, which is widely used in cache profiling. With object-based instrumentation, objects assigned to registers are not traced, and only accesses to the objects that are not assigned to registers are traced. Other differences stem from the fact that compiler optimizations that would ordinarily apply are often inapplicable to SMAT-adapted objects due to their side effects on the trace log. Consequently, the raw traces produced by SMAT contain far more memory accesses than those produced by object-based instrumentation. SMAT overcomes these difficulties and provides convincing cache profiling by its cache simulator that simulates registers as well as cache memory, along with its trace file optimizer that eliminates some of the inefficient memory accesses.

The cache performance analysis of STL sorting algorithms has shown that introsort is efficient in terms of cache behavior as well. When sorting a large sequence, the number of cache misses of introsort is kept quite low whereas the mergesort and heapsort incurs many cache misses. It is observed that cache performance varies greatly before and after the saturation point, where the sequence becomes the same size as the cache. Before the saturation point, the number of clock cycles is growing almost linearly.

Experiments that change cache parameters have shown interesting results. The increase of cache size just shifts the cache saturation point to the right. An experiment of changing associativity demonstrated that set associativities higher than 2-way are not so effective as might be intuitively expected. In experiments with larger block sizes, the number of cache misses decreased in inverse proportion to the block size with algorithms that makes sequential memory accesses, as with introsort and mergesort. However, it was not effective in heapsort. The number of registers did not decrease cache misses, but it did decrease cache hits. Increasing the registers up to 8 has shown clear improvement. Increasing main memory latency affected the total number of clock cycles in proportion to the number of cache misses.

Although the typical implementation of STL sorting algorithms is already efficient, there is still a chance of improvement in terms of cache performance. The tiled mergesort algorithm, which starts merging from much larger subarrays, has achieved a significant speedup, 35%, on the simulator and 20% on the Pentium 4. The already-efficient introsort algorithm has shown a speedup of 7% on the simulator by redistributing the insertion sort performed at the very end.

Not only the cache simulator, but also other tools such as the trace plotter and operation counter, help in understanding cache behavior and support cache performance analysis. SMAT is a simple, easy to use, and portable cache profiling tool. We expect that SMAT will be helpful in broader efforts toward development of cache-conscious algorithms.

6.1 Future Work

Every component of SMAT can be improved independently to provide more usability, accuracy, and efficiency.

The SMAT adaptor library could be extended to support other container iterator types, such as `std::list` and `std::map`. Supporting them would require a small number of implementation-dependent codes, in order to trace pointer dereferences performed inside the iterator.

The cache simulator already supports multi-level caches. In fact, users are already using its multi-level cache feature because registers are implemented as the level-zero cache. It is possible to add a second level cache to the cache simulator and study its behavior. Another possibility to extend the cache simulator is to support non-blocking caches, but that would be more challenging as it may entail substantial rewriting of the implementation.

Though it will be difficult, the trace file optimizer should be extended to make the source-level instrumentation technique more accurate. As seen in Section 4.2.2, the current SMAT has a weakness in it that prevents an important optimization normally done by the compiler, CSE (Common Subexpression Elimination). This problem could be overcome in the trace file optimizer, but it evidently will require reproduction of sophisticated data-flow analyses done in compilers.

At a more mundane level, the trace plotter currently generates a huge EPS file from a large trace, making it difficult or sometimes impossible to print the plot or even view it. The solution to this problem probably lies in introducing yet another utility to filter or aggregate trace data before attempting to plot it.

Bibliography

[1] Netpbm — graphics tools and converters. http://sourceforge.net/projects/netpbm/. 2.2

[2] Glenn Ammons, Tom Ball, Mark Hill, Babak Falsafi, Steve Huss-Lederman, James Larus, Alvin Lebeck, Mike Litzkow, Shubhendu Mukherjee, Steven Reinhardt, Madhusudhan Talluri, and David Wood. Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/~larus/warts.html. 1.3.2

[3] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986. 5.4.2

[4] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Phil Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000. http://icl.cs.utk.edu/projects/papi/. 1.3.1

[5] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996. http://www.simplescalar.com/. 1.3.2

[6] Martin Burtscher and Metha Jeeradit. Compressing Extended Program Traces Using Value Predictors. Cornell Computer Systems Lab Technical Report CSL-TR-2003-1029, Cornell University, January 2003. http://www.csl.cornell.edu/~burtscher/papers/TR1029.pdf. 4.3.1

[7] Intel Corporation. VTune Performance Analyzer. http://developer.intel.com/software/products/vtune/. 1.3.1

[8] Gillmer J. Derge and David R. Musser. The Operation Counting Adaptor Library. http://www.cs.rpi.edu/~musser/gp. 2.1

[9] Gnuplot Central. Gnuplot Central. http://www.gnuplot.info/. 2.2

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Third edition, May 2002. http://www.mkp.com/cod2e.htm. 4.1.1

[11] Charles Antony Richard Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, 1961. http://doi.acm.org/10.1145/366622.366642. 5.1.1

[12] Silicon Graphics, Inc. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/. 5.1

[13] 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. `http://www.lamarca.org/anthony/pubs/cachesort.pdf`. 1.2, 5

[14] Michael L. LaSpina and David R. Musser. Itrace, an Iterator Trace Plotting Tool. `http://www.cs.rpi.edu/~musser/gp`. 2.1, 1

[15] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998. `http://www.math.keio.ac.jp/matumoto/emt.html`. A.10, B.2.6

[16] David R. Musser. Introspective Sorting and Selection Algorithms. *Software — Practice and Experience*, 27(8):983–993, August 1997. `http://www.cs.rpi.edu/~musser/gp/introsort.ps`. 5.1.1

[17] Boost Organization. Boost Libraries. `http://www.boost.org/`. 2.2

[18] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. `http://rsim.cs.uiuc.edu/rsim/`. 1.3.2

[19] Mendel Rosenblum, Stephan A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SIMOS Approach. *IEEE Parallel and Distributed Technology*, 1995. `http://simos.stanford.edu/`. 1.3.2

[20] Maz Spork. Design and Analysis of Cache-Conscious Programs. Master's thesis, University of Copenhagen, Department of Computer Science, February 1999. `http://maz.spork.dk/spork99.pdf`. 1.3.2

[21] Brinkley Sprunt. Brink and Abyss: Pentium 4 Performance Counter Tools for Linux. `http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm`. 1.3.1

[22] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 196–205. ACM Press, 1994. `http://doi.acm.org/10.1145/178243.178260`. 1.3.2, 5

Appendix A

Source Code

A.1 Makefile

"Makefile" 111 ≡

```
     CXX = g++
     CXXFLAGS = -W -Wall -O3 -fomit-frame-pointer $(DEFS)
     #CXXFLAGS = -W -Wall -O0 $(DEFS)
     CXXCOMPILE = $(CXX) $(INCLUDES) $(CXXFLAGS)
     CXXLINK = $(CXX) $(CXXFLAGS) -o $@
     INCLUDES = -I/home/obata/build/boost_1_30_0

     # Cache parameters
     NREG = 8
     L1_SIZE = 8192
     L1_BLOCK = 32
     L1_ASSOC = 2
     L1_LATENCY = 1
     L1_WB = true
     MAIN_LATENCY = 32

     SOURCES = bestuse.cpp example1a.cpp example1b.cpp example2.cpp lackdtor.cpp \
             localaddr.cpp smat.bib smat.h smatcount.cpp smatnod.cpp smatopt.cpp \
             smatplot smatrange.cpp smatsim.cpp smatsim.h smatview.cpp \
             stl_tempbuf.h stlsort.cpp early.h lomuto.h tiled.h timer.h \
             test-smat.cpp test-readlog.cpp test-random.cpp test-exec.sh \
             test-sort.sh ans-random.txt ans-reg-count.txt ans-reg-opt.txt \
             ans-reg-plot.txt ans-reg-sim.txt ans-reg-simopt.txt test-reg.log

     .SUFFIXES:
     .SUFFIXES: .c .cpp .o .w .ps .dvi

     all: all-redirect
     all-redirect: all-am
     all-am: Makefile

     Makefile: smat.w
             pdfnuweb smat

     doc: smat.pdf
```

```
examples: example1a example1b example2
tools: smatopt smatsim smatview smatcount smatrange smatplot
        chmod +x smatplot

smat.pdf: smat.w
        pdfnuweb smat; pdflatex smat; bibtex smat; pdfnuweb smat
        pdflatex smat; pdfnuweb smat; pdflatex smat

smatsim.o: smatsim.cpp smatsim.h smat.h
        $(CXXCOMPILE) \
         -DNREG=$(NREG) -DL1_SIZE=$(L1_SIZE) -DL1_BLOCK=$(L1_BLOCK) \
         -DL1_ASSOC=$(L1_ASSOC) -DL1_LATENCY=$(L1_LATENCY) -DL1_WB=$(L1_WB) \
         -DMAIN_LATENCY=$(MAIN_LATENCY) -c smatsim.cpp -o smatsim.o

stlsort-timer.o: stlsort.cpp smat.h timer.h
        $(CXXCOMPILE) -DTIMER -DSMAT_DISABLE -c stlsort.cpp -o stlsort-timer.o

check: test-smat test-readlog test-random
        @chmod +x test-exec.sh test-sort.sh
        ./test-exec.sh
        ./test-sort.sh

clean:
        rm -f *.o smatopt smatsim smatview smatcount smatrange smatplot \
          stlsort stlsort-timer test-exec example1a example1b example2 \
          smat.aux smat.log smat.out smat.pdf smat.tex smat.toc \
          smat.brf smat.bib smat.bbl smat.blg smat.ps smat.dvi \
          $(SOURCES) *.tps *.dps

smatopt.o: smatopt.cpp smat.h
smatview.o: smatview.cpp smat.h
smatcount.o: smatcount.cpp smat.h
smatrange.o: smatrange.cpp smat.h smatsim.h
stlsort.o: stlsort.cpp smat.h early.h lomuto.h tiled.h
stlsort-timer.o: stlsort.cpp early.h lomuto.h tiled.h
example1a.o: example1a.cpp
example1b.o: example1b.cpp smat.h
example2.o: example2.cpp smat.h
test-smat.o: smat.h
test-readlog.o: test-readlog.cpp
test-random.o: test-random.cpp

%: %.o
        $(CXXLINK) $< $(LIBS)

%: %.cpp
        $(CXXCOMPILE) -o $@ $< $(LIBS)

%.o: %.cpp
        $(CXXCOMPILE) -c $<
```

A.2 Source Code for Examples

This section contains source code for the examples presented in Chapter 2.

A.2.1 Example 2

The example program `example2.cpp` implements four algorithms to find the minimum and maximum elements in the given sequence.

```
"example2.cpp" 113a ≡
    ⟨Include common header files. 115a⟩
    using namespace std;

    ⟨Minmax in two passes. 12a⟩
    ⟨Minmax in one pass. 12b⟩
    ⟨Minmax with fewer comparisons. 13⟩
    ⟨Minmax with fewer comparisons and CSE. 52⟩
    ⟨Convert a string to a value. 187b⟩
    ⟨Random number generator class. 187c⟩
    ⟨Usage for Example 2. 113b⟩
    ⟨Main function of Example 2. 114a⟩
```

The program takes two command line arguments, the name of algorithm and the number of elements. The name must be one of `twopass`, `onepass`, `fewercmp`, and `cse`. The second argument specifies the number of elements to generate. The option `-s` can specify a random seed.

```
⟨Usage for Example 2. 113b⟩ ≡
    static void usage(char *progname, int status)
    {
      if (status != 0)
        cerr << "Try `" << progname << " -h' for more information.\n";
      else
        {
          cout << "Usage: " << progname << " [OPTION]... ALGO NELEM\n\
    Generate NELEM random numbers and print the minimux and maximum elements.\n\
    ALGO must be one of `twopass', `onepass', `fewercmp', and `cse'.\n\
    \n\
      -s NUM    random seed\n\
      -h        display this help and exit\n";
        }

      exit(status);
    }
```
Used in part 113a.

The main function of `example2.cpp` is as follows. It interprets command line arguments, and then generates a random sequence. Finally, it invokes one of the four algorithms in which parameters are wrapped by SMAT adaptors. The results (the values of the minimum and maximum elements) are printed to the standard output.

⟨Main function of Example 2. 114a⟩ ≡

```
int main(int argc, char *argv[])
{
  unsigned seed = 3U, nelem;

  ios::sync_with_stdio(false);
  cin.tie(0);

  ⟨Parse command line arguments. 114b⟩

  vector<unsigned> v(nelem);
  generate(v.begin(), v.end(), genrand_limit<unsigned>(seed));

  typedef vector<unsigned>::iterator iter;
  pair<iter, iter> p;

  if (argv[optind][0] == 't')
    p = minmax_twopass(smat<iter>(v.begin()), smat<iter>(v.end()));
  else if (argv[optind][0] == 'o')
    p = minmax_onepass(smat<iter>(v.begin()), smat<iter>(v.end()));
  else if (argv[optind][0] == 'f')
    p = minmax_fewercmp(smat<iter>(v.begin()), smat<iter>(v.end()));
  else
    p = minmax_fewercmp_cse(smat<iter>(v.begin()), smat<iter>(v.end()));

  cerr << *p.first << ", " << *p.second << '\n';
  return EXIT_SUCCESS;
}
```

Used in part 113a.

The following code interprets the command line arguments. It sets up variables `seed` and `nelem`.

⟨Parse command line arguments. 114b⟩ ≡

```
{
  int optc;
  while ((optc = getopt(argc, argv, "hs:")) != -1)
    {
      switch (optc)
        {
        case 'h':
          usage(argv[0], EXIT_SUCCESS);
          break;

        case 's':
          if (! lazy_atoi(&seed, optarg))
            {
              cerr << argv[0] << ": invalid seed `" << optarg << "'\n";
              return EXIT_FAILURE;
            }
          break;

        default:
          usage(argv[0], EXIT_FAILURE);
          break;
        }
    }
}

if (optind != argc - 2)
  usage(argv[0], EXIT_FAILURE);

if (argv[optind][0] != 't' && argv[optind][0] != 'o'
    && argv[optind][0] != 'f' && argv[optind][0] != 'c')
  {
    cerr << argv[0] << ": invalid algorithm `" << argv[optind] << "'\n";
    return EXIT_FAILURE;
  }

if (! lazy_atoi(&nelem, argv[optind + 1]) || nelem == 0)
  {
    cerr << argv[0] << ": invalid number of elements `"
         << argv[optind + 1] << "'\n";
    return EXIT_FAILURE;
  }
```

Used in part 114a.

⟨Include common header files. 115a⟩ ≡

```
    #include "smat.h"
    #include <cstdlib>
    #include <algorithm>
    #include <iostream>
    #include <iterator>
    #include <map>
    #include <sstream>
    #include <utility>
    #include <vector>
    #ifdef linux
    #include <getopt.h>
    #endif
```

Used in parts 113a,115b,120b,153,167b,169a,174,185c,193,196a,199d.

A.2.2 Making the Best Use of SMAT

The program `bestuse.cpp` calls five algorithms presented in Section 2.5.

`"bestuse.cpp"` 115b ≡

⟨Include common header files. 115a⟩
⟨Accumulate stride algorithm (not generic). 18a⟩
⟨Accumulate stride algorithm (iterators). 18b⟩
⟨Accumulate stride algorithm (templates). 19a⟩
⟨Accumulate stride algorithm (traits types). 19b⟩
⟨Accumulate stride algorithm (final). 20⟩

```
int main()
{
  int x[5] = { 3, 6, 7, 2, 4 };
  float y[5] = { 3, 6, 7, 2, 3.5 };
  std::vector<int> v(&x[0], &x[5]);
  typedef std::vector<int>::iterator iter;

  std::cerr << accumulate_stride_1(v, 2) << '\n';
  std::cerr << accumulate_stride_2(v.begin(), v.end(), 2) << '\n';
  std::cerr << accumulate_stride_3(&x[0], &x[5], 2) << '\n';
  std::cerr << accumulate_stride_4(&y[0], &y[5], 2) << '\n';
  std::cerr << accumulate_stride_4(smat<iter>(v.begin()),
                                   smat<iter>(v.end()), 2) << '\n';
  std::cerr << accumulate_stride_final(smat<iter>(v.begin()),
                                       smat<iter>(v.end()), 2) << '\n';
}
```

A.3 Testing STL Sorting Algorithms

The test program used in Section 5 is presented in this section. The program `stlsort` takes two command line arguments, the name of the algorithm and the number of elements. The name must be one of `intro`, `merge`, `heap`, `early`, `lomuto`, and `tiled`. The last three names correspond to the improved sorting algorithms described in Sections 5.4.1, 5.4.2, and 5.5, respectively.

```
"stlsort.cpp" 117 ≡
    ⟨Header files for stlsort.cpp. 120b⟩

    #if __GNUC__ == 3 && 2 <= __GNUC_MINOR__
    #include <ext/algorithm>        // is_sorted
    using __gnu_cxx::is_sorted;

    ⟨Type traits for SMAT value type. 121a⟩

    #endif

    using namespace std;

    ⟨Random number generator class. 187c⟩

    ⟨Convert a string to a value. 187b⟩

    ⟨Display help for stlsort. 120a⟩

    int main(int argc, char *argv[])
    {
      int c;
      bool dump = false;
      unsigned tile_size = 4000U, seed = 7U, nelem;

      ios::sync_with_stdio(false);
      cin.tie(0);

      ⟨Parse command line arguments for stlsort. 118c⟩

      vector<unsigned> v(nelem);
      typedef smat<vector<unsigned>::iterator>::type Iter;

    ⟨Generate and sort a random sequence. 118a⟩

      if (dump)
        copy(v.begin(), v.end(), ostream_iterator<unsigned>(cerr, "\n"));

      if (! is_sorted(v.begin(), v.end()))
        return EXIT_FAILURE;

      return EXIT_SUCCESS;
    }
```

If macro `TIMER` is defined, the sorting algorithm is executed repeatedly until its running time exceeds 0.2 seconds. If `TIMER` is not defined, the algorithm is executed only once.

⟨Generate and sort a random sequence. 118a⟩ ≡

```
#ifdef TIMER

  timer t(1500000000.0); // 1.5GHz
  do {
    generate(v.begin(), v.end(), genrand_limit<unsigned>(seed));
    t.restart();
    ⟨Invoke the specified algorithm. 118b⟩
    t.pause();
  } while (t.elapsed() < 0.2);
  cerr << t.elapsed() * 1000.0 / t.npause() << '\n';

#else  // not TIMER

  generate(v.begin(), v.end(), genrand_limit<unsigned>(seed));
  ⟨Invoke the specified algorithm. 118b⟩

#endif // not TIMER
```

Used in part 117.

One of the six sorting algorithms is chosen, according to its first character.

⟨Invoke the specified algorithm. 118b⟩ ≡

```
if (argv[optind][0] == 'i')
  sort(Iter(v.begin()), Iter(v.end()));
else if (argv[optind][0] == 'e')
  early_introsort(Iter(v.begin()), Iter(v.end()));
else if (argv[optind][0] == 'l')
  lomuto_introsort(Iter(v.begin()), Iter(v.end()));
else if (argv[optind][0] == 'm')
  stable_sort(Iter(v.begin()), Iter(v.end()));
else if (argv[optind][0] == 't')
  {
    if (! tiled_mergesort(Iter(v.begin()), Iter(v.end()), tile_size))
      {
        cerr << argv[0] << ": cannot allocate memory\n";
        return EXIT_FAILURE;
      }
  }
else // if (argv[optind][0] == 'h')
  partial_sort(Iter(v.begin()), Iter(v.end()), Iter(v.end()));
```

Used in part 118a.

⟨Parse command line arguments for `stlsort`. 118c⟩ ≡

```
while ((c = getopt(argc, argv, "t:s:Sdh")) != -1)
  {
    switch (c)
      {
      case 't':
        if (! lazy_atoi(&tile_size, optarg) || tile_size < 16)
          {
            cerr << argv[0] << ": invalid tile size `" << optarg << "'\n";
```

```
              return EXIT_FAILURE;
            }
          break;

        case 's':
          if (! lazy_atoi(&seed, optarg))
            {
              cerr << argv[0] << ": invalid seed `" << optarg << "'\n";
              return EXIT_FAILURE;
            }
          break;

        case 'S':
          {
            struct timeval tv;
            gettimeofday(&tv, NULL);
            seed = (unsigned long int)getpid() * tv.tv_sec * tv.tv_usec;
            cerr << "seed: " << seed << '\n';
          }
          break;

        case 'd':
          dump = true;
          break;

        case 'h':
          usage(argv[0], EXIT_SUCCESS); break;

        default:
          usage(argv[0], EXIT_FAILURE); break;
        }
    }

  if (optind != argc - 2)
    usage(argv[0], EXIT_FAILURE);

  if (argv[optind][0] != 'i' && argv[optind][0] != 'e'
      && argv[optind][0] != 'l' && argv[optind][0] != 'm'
      && argv[optind][0] != 't' && argv[optind][0] != 'h')
    {
      cerr << argv[0] << ": invalid algorithm `" << argv[optind] << "'\n";
      return EXIT_FAILURE;
    }

  if (! lazy_atoi(&nelem, argv[optind + 1]))
    {
      cerr << argv[0] << ": invalid number of elements `"
           << argv[optind + 1] << "'\n";
      return EXIT_FAILURE;
    }
```
Used in part 117.

The function `usage` displays a help message. The option `-s` can be used to specify a random seed and option `-t` changes the tile size.

⟨Display help for `stlsort`. 120a⟩ ≡
```
static void usage(char *progname, int status)
{
  if (status != 0)
    cerr << "Try ‘" << progname << " -h’ for more information.\n";
  else
    {
      cout << "Usage: " << progname << " [OPTION]... ALGO NELEM\n\
Generate random numbers and sort them.  ALGO must be one of \"intro\",\n\
\"merge\", \"heap\", \"early\", \"lomuto\", and \"tiled\".\n\
\n\
  -s NUM    random seed\n\
  -S        choose random seed randomly\n\
  -t NUM    tile size for tiled mergesort (4000 by default)\n\
  -d        output sorted numbers to stderr\n\
\n\
  -h        display this help and exit\n";
    }

  exit(status);
}
```
Used in part 117.

As mentioned in Section 4.4, the temporary buffer allocation in `stl_tempbuf.h` has a problem. The following code contains a workaround to use modified version of `_Temporary_buffer` in local `stl_tempbuf.h`.

⟨Header files for `stlsort.cpp`. 120b⟩ ≡
```
#define __GLIBCPP_INTERNAL_TEMPBUF_H
namespace std {
template <class _ForwardIterator, class _Tp> class _Temporary_buffer;
}

⟨Include common header files. 115a⟩

#undef __GLIBCPP_INTERNAL_TEMPBUF_H
#include "stl_tempbuf.h"

#include <sys/time.h>
#include <unistd.h>

#ifdef TIMER
#include "timer.h"
#endif

#include "early.h"
#include "lomuto.h"
#include "tiled.h"
```
Used in part 117.

Next, we define type traits for the SMAT value type, which is used in `stl_tempbuf.h`. The following definition means that if type `T` has a trivial default constructor, then the type `smat_v<T>` also does. With this definition, unnecessary memory initialization of a temporary buffer can be eliminated.

⟨Type traits for SMAT value type. 121a⟩ ≡

```
template <typename T>
struct __type_traits<smat_v<T> > {
  typedef typename __type_traits<T>::has_trivial_default_constructor
    has_trivial_default_constructor;
  typedef __false_type has_trivial_copy_constructor;
  typedef __false_type has_trivial_assignment_operator;
  typedef __false_type has_trivial_destructor;
  typedef __false_type is_POD_type;
};
```

Used in part 117.

A.3.1 Early Finalization

"early.h" 121b ≡

```
#ifndef _EARLY_H_
#define _EARLY_H_

⟨Introsort loop with early finalization. 121d⟩
⟨Introsort with early finalization. 121c⟩

#endif // _EARLY_H_
```

Function `early_introsort` calls `early_introsort_loop` only if a sequence is not empty. This check is necessary because `__insertion_sort`, called from `early_introsort_loop`, does not work for an empty sequence and so the caller must guarantee that the given sequence is not empty.

⟨Introsort with early finalization. 121c⟩ ≡

```
template<typename Iter>
inline void
early_introsort(Iter first, Iter last)
{
  if (first != last)
    early_introsort_loop(first, last, __lg(last - first) * 2, first);
}
```

Used in part 121b.

Function `early_introsort_loop` is quite similar to `__introsort_loop` except that it performs insertion sort, by either `__insertion_sort` or `__unguarded_linear_insert`, at the end of the function. The latter function is slightly more efficient and thus preferred. However, it cannot be applied to the very first chunk because it assumes that there is a sentinel element `x` before `first` such that `x` $\leq$ `*first`, which is not true at the beginning of the sequence. Parameter `real_first` is used to tell whether `first` is *really* the beginning of the sequence or not.

⟨Introsort loop with early finalization. 121d⟩ ≡

```
template <typename Iter, typename Size>
void
early_introsort_loop(Iter first, Iter last, Size depth_limit,
                     const Iter& real_first)
{
  typedef typename std::iterator_traits<Iter>::value_type ValueType;

  while (last - first > _M_threshold) // 16
    {
      if (depth_limit == 0)
        return partial_sort(first, last, last);
      --depth_limit;
      Iter cut =
        __unguarded_partition(first, last,
                              ValueType(__median(*first,
                                                 *(first + (last - first)/2),
                                                 *(last - 1))));
      early_introsort_loop(cut, last, depth_limit, real_first);
      last = cut;
    }
  if (first == real_first)
    __insertion_sort(first, last);
  else
    __unguarded_insertion_sort(first, last);
}
```

Used in part 121b.

A.3.2 Lomuto's Partition

"lomuto.h" 122a ≡

```
#ifndef _LOMUTO_H_
#define _LOMUTO_H_

⟨Lomuto partition with pivot. 122b⟩
⟨Introsort loop with lomuto partition. 123b⟩
⟨Introsort with lomuto partition. 123a⟩

#endif // _LOMUTO_H_
```

Function `lomuto_partition_pivot` takes a pivot explicitly as an argument, unlike several partitioning algorithms available that assume the first or the last element is the pivot.

⟨Lomuto partition with pivot. 122b⟩ ≡

```
template <typename Iter, typename ValueType>
Iter
lomuto_partition_pivot(Iter first, Iter last, ValueType pivot)
{
  Iter lessthan = first - 1;
  for ( ; first != last; ++first)
    {
      if (*first < pivot)
        iter_swap(++lessthan, first);
    }
  return lessthan + 1;
}
```

Used in part 122a.

Functions `lomuto_introsort` and `lomuto_introsort_loop` are quite similar to functions `sort` and `__introsort_loop`, respectively, except that they use Lomuto's partition instead of Hoare's partition.

⟨Introsort with lomuto partition. 123a⟩ ≡
```
template <typename Iter>
inline void
lomuto_introsort(Iter first, Iter last)
{
  if (first != last)
    {
      lomuto_introsort_loop(first, last, __lg(last - first) * 2);
      __final_insertion_sort(first, last);
    }
}
```
Used in part 122a.

⟨Introsort loop with lomuto partition. 123b⟩ ≡
```
template <typename Iter, typename Size>
void
lomuto_introsort_loop(Iter first, Iter last, Size depth_limit)
{
  typedef typename std::iterator_traits<Iter>::value_type ValueType;

  while (last - first > _M_threshold)
    {
      if (depth_limit == 0)
        return partial_sort(first, last, last);
      --depth_limit;
      Iter cut =
        lomuto_partition_pivot(first, last,
                               ValueType(__median(*first,
                                                  *(first + (last - first)/2),
                                                  *(last - 1))));
      lomuto_introsort_loop(cut, last, depth_limit);
      last = cut;
    }
}
```
Used in part 122a.

A.3.3 Tiled Mergesort

"tiled.h" 123c ≡
```
#ifndef _TILED_H_
#define _TILED_H_

#ifndef TILED_SMALL_CHUNK
#define TILED_SMALL_CHUNK 16
#endif

using namespace std;
```

```
enum { small_elem = TILED_SMALL_CHUNK };

⟨Chunk mergesort. 124a⟩
⟨Tiled mergesort helper. 124b⟩
⟨Tiled mergesort. 125a⟩

#endif // _TILED_H_
```

The following three algorithms were explained in Section 5.5.

⟨Chunk mergesort. 124a⟩ ≡

```
template <typename Iter>
static void
chunk_mergesort(Iter first, Iter last,
  typename std::iterator_traits<Iter>::pointer tmpbuf,
  typename std::iterator_traits<Iter>::difference_type chunk_elem)
{
  while (first + chunk_elem <= last)
    {
      tiled_mergesort_internal(first, first + chunk_elem, tmpbuf, small_elem);
      first += chunk_elem;
    }

  if (first < last)
    tiled_mergesort_internal(first, last, tmpbuf, small_elem);
}
```

Used in part 123c.

⟨Tiled mergesort helper. 124b⟩ ≡

```
template <typename Iter>
static void
tiled_mergesort_internal(Iter first, Iter last,
  typename std::iterator_traits<Iter>::pointer tmpbuf,
  typename std::iterator_traits<Iter>::difference_type chunk_elem)
{
  typedef typename std::iterator_traits<Iter>::difference_type _Distance;
  typedef typename std::iterator_traits<Iter>::pointer _Pointer;

  _Distance nelement = last - first;
  _Pointer tmpbuf_last = tmpbuf + nelement;

  if (small_elem < chunk_elem)
    chunk_mergesort(first, last, tmpbuf, chunk_elem);
  else
    __chunk_insertion_sort(first, last, chunk_elem);

  _Distance nchunk = (nelement + chunk_elem - 1) / chunk_elem;

  while (nchunk > 1)
    {
      __merge_sort_loop(first, last, tmpbuf, chunk_elem);
```

```
            chunk_elem *= 2;
            __merge_sort_loop(tmpbuf, tmpbuf_last, first, chunk_elem);
            chunk_elem *= 2;
            nchunk = (nchunk + 3) / 4;
          }
      }
```

Used in part 123c.

⟨Tiled mergesort. 125a⟩ ≡

```
      template <typename Iter>
      inline bool
      tiled_mergesort(Iter first, Iter last,
        typename iterator_traits<Iter>::difference_type tile_size)
      {
        typedef typename iterator_traits<Iter>::value_type _Value;

        _Temporary_buffer<Iter, _Value> buf(first, last);
        if (buf.size() != buf.requested_size())
          return false;

        tiled_mergesort_internal(first, last, buf.begin(), tile_size / sizeof(_Value));

        return true;
      }
```

Used in part 123c.

A.3.4 High Resolution Timer

The file `timer.h` defines the high resolution timer class for `stlsort`. It supports Solaris on the UltraSparc and Linux on the x86. If the timer is used by a root user on Linux, it will be very accurate since it brings the priority of the process to the highest. See `stlsort.cpp` to see how the timer class is used.

"timer.h" 125b ≡

```
      #include <sys/time.h>
      #include <unistd.h>

      #ifndef _TIMER_H_
      #define _TIMER_H_

      #ifdef i386
      ⟨High resolution timer class for Linux on x86. 126⟩
      #endif // i386

      #ifdef sparc
      ⟨High resolution timer class for Solaris machine. 127a⟩
      #endif // sparc

      #endif // _TIMER_H_
```

On the Linux platform with an x86 architecture, the timer is based on RDTSC (ReaD Time Stamp Counter), the special register which is incremented at every CPU internal clock cycle. The default constructor performs a calibration to estimate the clock frequency, which takes about three seconds. To avoid this calibration, a user can use the constructor which takes an argument of type `double` to supply the CPU clock cycles explicitly (in clocks per second).

⟨High resolution timer class for Linux on x86. 126⟩ ≡

```
#include <sched.h>
class timer {
public:
  typedef unsigned long long int timer_t;

  timer() : total_time(0) {
    clock_per_second = calibrate();
  }

  timer(double cps) : total_time(0), clock_per_second(cps) {
    raise_priority();
  }

  void raise_priority() {
    struct sched_param sp;
    sp.sched_priority = 99;
    sched_setscheduler(0, SCHED_FIFO, &sp); // effective only for root user
  }

  void reset_priority() {
    struct sched_param sp;
    sp.sched_priority = 0;
    sched_setscheduler(0, SCHED_OTHER, &sp); // effective only for root user
  }

  double calibrate() {
    timer_t t1, t2;
    struct timeval tv1, tv2;
    double diff_gettimeofday, diff_clock;

    gettimeofday (&tv1, NULL);
    t1 = read_timer();
    sleep (3);
    gettimeofday (&tv2, NULL);
    t2 = read_timer();
    diff_gettimeofday = (((tv2.tv_sec - tv1.tv_sec) * 1000000.0
                          + tv2.tv_usec - tv1.tv_usec) / 1000000.0);
    diff_clock = (double)t2 - (double)t1;
    return diff_clock / diff_gettimeofday;
  }

  timer_t read_timer() {
    unsigned long long int x;
    __asm__ __volatile__ (".byte 0x0f, 0x31" : "=A" (x));
    return x;
  }
```

```
    ⟨Common functions for the high resolution timer. 127b⟩

private:
  unsigned running; // odd if running, even if not running.
  timer_t start_time, total_time;
  double clock_per_second;
};
```

Used in part 125b.

Here is the Solaris version, which is based on the `gethrvtime` system call. This works, but it does not seem to be very accurate. The constructor with one parameter of type `double` is provided just for compatibility with the Linux version and is ignored.

⟨High resolution timer class for Solaris machine. 127a⟩ ≡

```
class timer {
public:
  typedef hrtime_t timer_t;

  timer() : total_time(0), clock_per_second(1000000000.0) { }
  timer(double) : total_time(0), clock_per_second(1000000000.0) { }
  timer_t read_timer() { return gethrvtime(); }

  void raise_priority() { }
  void reset_priority() { }

    ⟨Common functions for the high resolution timer. 127b⟩

private:
  unsigned running; // odd if running, even if not running.
  timer_t start_time, total_time;
  double clock_per_second;
};
```

Used in part 125b.

⟨Common functions for the high resolution timer. 127b⟩ ≡

```
void reset() { running = 0; total_time = 0; }

void restart() {
  if (! (running & 1))
    {
      ++running;
      raise_priority();
      start_time = read_timer();
    }
}

void pause() {
  // reads timer only if running.
  if (running & 1)
    {
      total_time += read_timer() - start_time;
```

```
          reset_priority();
          ++running;
        }
    }

    unsigned npause() {
      return running / 2;
    }

    double elapsed() {
      pause();
      return static_cast<double>(total_time) / clock_per_second;
    }
```

Used in parts 126,127a.

A.4 STL Memory Access Tracer

File `smat.h` defines four adaptor classes `smat`, `smat_p`, `smat_v`, `smat_d`, and auxiliary classes `smat_singleton`, `smat_inst`, and `smat_io`. A `smat_inst` class object corresponds to one entry in a trace file. File I/O operations, both in binary and human-readable format, are provided.

"smat.h" 129 ≡

```
// -*- c++ -*-
#ifndef _SMAT_H_
#define _SMAT_H_

// forward declarations.
template <typename T> class smat_p;
template <typename T> class smat_v;
template <typename T> class smat_d;

⟨Define a singleton class. 148a⟩
⟨Define macros for SMAT. 130a⟩

#include <boost/static_assert.hpp>
#include <cstdio>                // sprintf
#include <cstdlib>               // EXIT_FAILURE
#include <fstream>
#include <iostream>
#include <iomanip>
#include <iterator>              // for std::iterator_traits
#include <netinet/in.h>          // htonl, ntohl

⟨Define trace instruction codes. 148b⟩
⟨Define trace instruction class. 148c⟩
⟨Define SMAT I/O class. 147d⟩
⟨Define SMAT iterator adaptor class. 131⟩
⟨Define SMAT pointer type adaptor class. 132⟩
⟨Define SMAT value type adaptor class. 133⟩
⟨Define SMAT difference type adaptor class. 134a⟩
⟨Public member functions for adaptors. 134b⟩
⟨Global definitions for adaptors. 135a⟩
⟨Disable SMAT macros. 130b⟩

#endif // _SMAT_H_
```

There are many member functions and global functions in SMAT whose definitions are quite similar. For this reason we define several macros to aid in expressing the definitions, both shortening the code and helping ensure consistency.

Macros related to trace generation are defined here. Macros `SMAT_LOG1`, `SMAT_LOG2`, `SMAT_LOG2B`, and `SMAT_LOG3` generate traces, using macros `SMAT_W` and `SMAT_A`. Macros `SMAT_SET_OSTREAM` and `SMAT_SET_ISTREAM` set the output and input streams used in SMAT (though the input stream is not used).

⟨Define macros for SMAT. 130a⟩ ≡

```
#define SMAT_A(ADDR) reinterpret_cast<unsigned>(&(ADDR))
//#define SMAT_A(ADDR) reinterpret_cast<unsigned>(boost::addressof(ADDR))

#define SMAT_W(C, ID, L, S1, S2, D) \
  smat_inst::write_binary(C, smat_inst::id2type(ID), L, S1, S2, D, \
                        (smat_singleton<smat_io>::ref().fout()))

#define SMAT_LOG1(C, ID, S1) do { \
  SMAT_W((C), (ID), sizeof(S1), SMAT_A(S1), 0U, 0U); } while (0)

#define SMAT_LOG2(C, ID, S1, D) do { \
  SMAT_W((C), (ID), sizeof(S1), SMAT_A(S1), 0U, SMAT_A(D)); } while (0)

#define SMAT_LOG2B(C, ID, S1, S2) do { \
  SMAT_W((C), (ID), sizeof(S1), SMAT_A(S1), SMAT_A(S2), 0U); } while (0)

#define SMAT_LOG3(C, ID, S1, S2, D) do { \
  SMAT_W((C), (ID), sizeof(S1), SMAT_A(S1), SMAT_A(S2), SMAT_A(D)); } while (0)

#define SMAT_MARK(M) do { SMAT_W(SMAT_FMARK, 0, 0, (M), 0, 0); } while (0)

#define SMAT_SET_OSTREAM(OS) \
  do { smat_singleton<smat_io>::ref().set_fout(OS); } while (0)
#define SMAT_SET_ISTREAM(IS) \
  do { smat_singleton<smat_io>::ref().set_fin(IS); } while (0)
```

Used in part 129.

The following definitions make the above macros do nothing if SMAT_DISABLE is defined.

⟨Disable SMAT macros. 130b⟩ ≡

```
#ifdef SMAT_DISABLE
#undef SMAT_A
#undef SMAT_W
#undef SMAT_SET_OSTREAM
#undef SMAT_SET_ISTREAM
#undef SMAT_LOG1
#undef SMAT_LOG2
#undef SMAT_LOG2B
#undef SMAT_LOG3
#undef SMAT_MASK
#define SMAT_SET_OSTREAM(OS)
#define SMAT_SET_ISTREAM(IS)
#define SMAT_LOG1(C, ID, S1)
#define SMAT_LOG2(C, ID, S1, D)
#define SMAT_LOG2B(C, ID, S1, S2)
#define SMAT_LOG3(C, ID, S1, S2, D)
#define SMAT_MASK(M)
#endif // SMAT_DISABLE
```

Used in part 129.

A.4.1 Adaptors

⟨Define SMAT iterator adaptor class. 131⟩ ≡

```
template <typename T>
class smat {
  T base_;

  typedef typename std::iterator_traits<T>::difference_type raw_difference_type;
  typedef typename std::iterator_traits<T>::value_type raw_value_type;
  typedef typename std::iterator_traits<T>::pointer raw_pointer;

public:
  typedef smat_v<raw_value_type> value_type;
  typedef smat_d<raw_difference_type> difference_type;
  typedef smat_p<raw_value_type*> pointer;
  typedef value_type& reference;
  typedef typename std::iterator_traits<T>::iterator_category iterator_category;
  ⟨Size check for the value type. 135b⟩

⟨Define type type for smat. 146b⟩

  smat();
  smat(const smat& x);
  smat(const T& x);
  smat(const T& x, int);
  ~smat();
  //explicit smat(value_type* x);

  operator T() const { SMAT_LOG1(SMAT_READ, 'i', base_); return base_; }
  const T& base() const { return base_; }

  smat& operator=(const smat& x);
  //bool operator!() const;
  T operator->() const;

  reference operator*() const;
  reference operator[](const difference_type& x) const;
  template <typename U> reference operator[](const U& x) const;

  smat& operator++();
  smat operator++(int);
  smat& operator--();
  smat operator--(int);

  smat& operator+=(const difference_type& x);
  smat& operator-=(const difference_type& x);
  template <typename U> smat& operator+=(const U& x);
  template <typename U> smat& operator-=(const U& x);
};
```

Used in part 129.

⟨Define SMAT pointer type adaptor class. 132⟩ ≡

```
template <typename T>
class smat_p {
  T base_;

  typedef typename std::iterator_traits<T>::difference_type raw_difference_type;
  typedef typename std::iterator_traits<T>::value_type raw_value_type;
  typedef typename std::iterator_traits<T>::pointer raw_pointer;

public:
  typedef smat_v<raw_value_type> value_type;
  typedef smat_d<raw_difference_type> difference_type;
  typedef smat_p<raw_value_type*> pointer;
  typedef value_type& reference;
  typedef std::random_access_iterator_tag iterator_category;

⟨Define type type for smat_p. 147a⟩

  smat_p();
  smat_p(const smat_p& x);
  smat_p(const T& x);
  smat_p(const T& x, int);
  ~smat_p();

  operator T() const { SMAT_LOG1(SMAT_READ, 'p', base_); return base_; }
  const T& base() const { return base_; }
  smat_p& operator=(const smat_p& x);
  smat_p& operator=(value_type* x);

  //bool operator!() const;
  T operator->() const;

  reference operator*() const;
  reference operator[](const difference_type& x) const;
  template <typename U> reference operator[](const U& x) const;

  smat_p& operator++();
  smat_p operator++(int);
  smat_p& operator--();
  smat_p operator--(int);

  smat_p& operator+=(const difference_type& x);
  smat_p& operator-=(const difference_type& x);
  template <typename U> smat_p& operator+=(const U& x);
  template <typename U> smat_p& operator-=(const U& x);
};
```

Used in part 129.

⟨Define SMAT value type adaptor class. 133⟩ ≡

```
template <typename T>
class smat_v {
  T base_;

public:
  smat_v();
  smat_v(const smat_v& x);
  smat_v(const T& x);
  smat_v(const T& x, int);
  ~smat_v();

⟨Define type type for smat_v. 147b⟩

  operator T() const { SMAT_LOG1(SMAT_READ, 'v', base_); return base_; }
  const T& base() const { return base_; }
  smat_v& operator=(const smat_v& x);

  smat_v* operator&();
  const smat_v* operator&() const;
  smat_v operator+() const;
  smat_v operator-() const;
  smat_v operator~() const;

  smat_v& operator++();
  smat_v operator++(int);
  smat_v& operator--();
  smat_v operator--(int);

  smat_v& operator+=(const smat_v& x);
  smat_v& operator-=(const smat_v& x);
  smat_v& operator*=(const smat_v& x);
  smat_v& operator/=(const smat_v& x);
  smat_v& operator%=(const smat_v& x);
  smat_v& operator<<=(const smat_v& x);
  smat_v& operator>>=(const smat_v& x);
  smat_v& operator&=(const smat_v& x);
  smat_v& operator|=(const smat_v& x);
  smat_v& operator^=(const smat_v& x);
  template <typename U> smat_v& operator+=(const U& x);
  template <typename U> smat_v& operator-=(const U& x);
  template <typename U> smat_v& operator*=(const U& x);
  template <typename U> smat_v& operator/=(const U& x);
  template <typename U> smat_v& operator%=(const U& x);
  template <typename U> smat_v& operator<<=(const U& x);
  template <typename U> smat_v& operator>>=(const U& x);
  template <typename U> smat_v& operator&=(const U& x);
  template <typename U> smat_v& operator|=(const U& x);
  template <typename U> smat_v& operator^=(const U& x);
};
```

Used in part 129.

⟨Define SMAT difference type adaptor class. 134a⟩ ≡

```
template <typename T>
class smat_d {
  T base_;

public:
  smat_d();
  smat_d(const smat_d& x);
  smat_d(const T& x);
  smat_d(const T& x, int);
  ~smat_d();

⟨Define type type for smat_d. 147c⟩

  operator T() const { SMAT_LOG1(SMAT_READ, 'd', base_); return base_; }
  const T& base() const { return base_; }
  smat_d& operator=(const smat_d& x);

  smat_d* operator&();
  const smat_d* operator&() const;
  smat_d operator+() const;
  smat_d operator-() const;
  smat_d operator~() const;

  smat_d& operator++();
  smat_d operator++(int);
  smat_d& operator--();
  smat_d operator--(int);

  smat_d& operator+=(const smat_d& x);
  smat_d& operator-=(const smat_d& x);
  smat_d& operator*=(const smat_d& x);
  smat_d& operator/=(const smat_d& x);
  smat_d& operator%=(const smat_d& x);
  smat_d& operator<<=(const smat_d& x);
  smat_d& operator>>=(const smat_d& x);
  smat_d& operator&=(const smat_d& x);
  smat_d& operator|=(const smat_d& x);
  smat_d& operator^=(const smat_d& x);
  template <typename U> smat_d& operator+=(const U& x);
  template <typename U> smat_d& operator-=(const U& x);
  template <typename U> smat_d& operator*=(const U& x);
  template <typename U> smat_d& operator/=(const U& x);
  template <typename U> smat_d& operator%=(const U& x);
  template <typename U> smat_d& operator<<=(const U& x);
  template <typename U> smat_d& operator>>=(const U& x);
  template <typename U> smat_d& operator&=(const U& x);
  template <typename U> smat_d& operator|=(const U& x);
  template <typename U> smat_d& operator^=(const U& x);
};
```

Used in part 129.

⟨Public member functions for adaptors. 134b⟩ ≡

    ⟨Increment and decrement operators. 136b⟩
    ⟨Constructors and destructors. 135c⟩
    ⟨Arithmetic assignment operators. 137⟩
    ⟨Arithmetic assignment operators with immediate value. 138⟩
    ⟨Arrow operators. 140a⟩
    ⟨Assignment operators. 136a⟩
    ⟨Dereference operators. 140b⟩
    ⟨Unary plus, minus, complement, and address-of operators. 139⟩

Used in part 129.

⟨Global definitions for adaptors. 135a⟩ ≡

    ⟨Binary operators. 141⟩
    ⟨Binary operators with immediate values. 143⟩
    ⟨Stream operators. 144⟩
    ⟨Comparison operators. 145⟩
    ⟨Comparison operators with immediate values. 146a⟩

Used in part 129.

Constraints on the Value Type

There is an important constraint on the size of the value type adaptor, that the adapted object must have the same size as the base object. The constraint is imposed because the base object is "reinterpret-casted" to the adapted object type in iterator dereferences. Another constraint is that the size of value type must not exceed 255 bytes, which is another restriction coming from the trace file format (size field has only 8 bits).

⟨Size check for the value type. 135b⟩ ≡

```
BOOST_STATIC_ASSERT(sizeof(value_type) == sizeof(raw_value_type));
BOOST_STATIC_ASSERT(sizeof(value_type) < 256);
```

Used in part 131.

Constructors, Destructors and Related Functions

⟨Constructors and destructors. 135c⟩ ≡

```
#define SMAT_MDEF_CTORDTOR(CLASS, ID)                              \
template <typename T>                                              \
inline CLASS<T>::CLASS() : base_() {                               \
  SMAT_LOG1(SMAT_CTOR, ID, base_);                                 \
}                                                                  \
template <typename T>                                              \
inline CLASS<T>::CLASS(const CLASS<T>& x) : base_(x.base_) {       \
  SMAT_LOG2(SMAT_CCTOR, ID, x.base_, base_);                       \
}                                                                  \
template <typename T>                                                 \
inline CLASS<T>::CLASS(const T& x) : base_(x) {                       \
  SMAT_LOG2(SMAT_BCTOR, ID, x, base_);                                \
}                                                                     \
template <typename T>                                                 \
inline CLASS<T>::CLASS(const T& x, int) : base_(x) {                  \
  SMAT_LOG1(SMAT_CTOR, ID, base_);                                    \
}                                                                     \
template <typename T>                                                 \
inline CLASS<T>::~CLASS() {                                           \
  SMAT_LOG1(SMAT_DTOR, ID, base_);                                    \
}

SMAT_MDEF_CTORDTOR(smat, 'i')
SMAT_MDEF_CTORDTOR(smat_p, 'p')
SMAT_MDEF_CTORDTOR(smat_v, 'v')
SMAT_MDEF_CTORDTOR(smat_d, 'd')

#undef SMAT_MDEF_CTORDTOR
```

Used in part 134b.

⟨Assignment operators. 136a⟩ ≡

```
template <typename T>
inline smat<T>& smat<T>::operator=(const smat<T>& x) {
  SMAT_LOG2(SMAT_MOV, 'i', x.base_, base_); base_ = x.base_; return *this;
}

template <typename T>
inline smat_p<T>& smat_p<T>::operator=(const smat_p<T>& x) {
  SMAT_LOG2(SMAT_MOV, 'p', x.base_, base_); base_ = x.base_; return *this;
}

template <typename T>
inline smat_p<T>& smat_p<T>::operator=(
  smat_v<typename std::iterator_traits<T>::value_type>* x) {
  SMAT_LOG2(SMAT_MOV, 'p', x, base_);
  base_ = reinterpret_cast<T>(x); return *this;
}

template <typename T>
inline smat_v<T>& smat_v<T>::operator=(const smat_v<T>& x) {
  SMAT_LOG2(SMAT_MOV, 'v', x.base_, base_); base_ = x.base_; return *this;
}

template <typename T>
inline smat_d<T>& smat_d<T>::operator=(const smat_d<T>& x) {
  SMAT_LOG2(SMAT_MOV, 'd', x.base_, base_); base_ = x.base_; return *this;
}
```

Used in part 134b.

Increment and Decrement Operators

⟨Increment and decrement operators. 136b⟩ ≡

```
#define SMAT_MDEF_INCDEC(CLASS, ID)                                \
template <typename T>                                              \
inline CLASS<T>& CLASS<T>::operator++() {                          \
  SMAT_LOG1(SMAT_INC, ID, base_); ++base_; return *this;           \
}                                                                  \
template <typename T>                                              \
inline CLASS<T> CLASS<T>::operator++(int) {                        \
  SMAT_LOG1(SMAT_INC, ID, base_);                                  \
  CLASS<T> x(*this); ++base_; return x;                            \
}                                                                  \
template <typename T>                                              \
inline CLASS<T>& CLASS<T>::operator--() {                          \
  SMAT_LOG1(SMAT_DEC, ID, base_); --base_; return *this;           \
}                                                                  \
template <typename T>                                              \
inline CLASS<T> CLASS<T>::operator--(int) {                        \
  SMAT_LOG1(SMAT_DEC, ID, base_);                                  \
  CLASS<T> x(*this); --base_; return x;                            \
}

SMAT_MDEF_INCDEC(smat, 'i')
SMAT_MDEF_INCDEC(smat_p, 'p')
SMAT_MDEF_INCDEC(smat_v, 'v')
SMAT_MDEF_INCDEC(smat_d, 'd')

#undef SMAT_MDEF_INCDEC
```

Used in part 134b.

Arithmetic Assignment Operators

⟨Arithmetic assignment operators. 137⟩ ≡

```
#define SMAT_MDEF_ASSIGNOP(CLASS, ID, OP, TCODE)         \
template <typename T>                                    \
inline CLASS<T>&                                         \
CLASS<T>::operator OP(const CLASS<T>& x) {               \
  SMAT_LOG3(TCODE, ID, base_, x.base(), base_);          \
  base_ OP x.base(); return *this;                       \
}

SMAT_MDEF_ASSIGNOP(smat_v, 'v', +=, SMAT_ADD)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', -=, SMAT_SUB)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', *=, SMAT_MUL)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', /=, SMAT_DIV)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', %=, SMAT_MOD)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', <<=, SMAT_LSHIFT)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', >>=, SMAT_RSHIFT)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', &=, SMAT_AND)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', |=, SMAT_OR)
SMAT_MDEF_ASSIGNOP(smat_v, 'v', ^=, SMAT_XOR)

SMAT_MDEF_ASSIGNOP(smat_d, 'd', +=, SMAT_ADD)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', -=, SMAT_SUB)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', *=, SMAT_MUL)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', /=, SMAT_DIV)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', %=, SMAT_MOD)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', <<=, SMAT_LSHIFT)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', >>=, SMAT_RSHIFT)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', &=, SMAT_AND)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', |=, SMAT_OR)
SMAT_MDEF_ASSIGNOP(smat_d, 'd', ^=, SMAT_XOR)

#undef SMAT_MDEF_ASSIGNOP

template <typename T>
inline smat<T>&
smat<T>::operator+=(const smat_d<typename
                    std::iterator_traits<T>::difference_type>& x) {
  SMAT_LOG3(SMAT_ADD, 'i', base_, x.base(), base_);
  base_ += x.base(); return *this;
}

template <typename T>
inline smat<T>&
smat<T>::operator-=(const smat_d<typename
                    std::iterator_traits<T>::difference_type>& x) {
  SMAT_LOG3(SMAT_SUB, 'i', base_, x.base(), base_);
  base_ -= x.base(); return *this;
}

template <typename T>
inline smat_p<T>&
smat_p<T>::operator+=(const smat_d<typename
                      std::iterator_traits<T>::difference_type>& x) {
  SMAT_LOG3(SMAT_ADD, 'p', base_, x.base(), base_);
  base_ += x.base(); return *this;
}

template <typename T>
inline smat_p<T>&
smat_p<T>::operator-=(const smat_d<typename
                      std::iterator_traits<T>::difference_type>& x) {
  SMAT_LOG3(SMAT_SUB, 'p', base_, x.base(), base_);
  base_ -= x.base(); return *this;
}
```

Used in part 134b.

Arithmetic Assignment Operators with Immediate Value

Macro `SMAT_MDEF_ASSIGNOP_IMMEDIATE` defines arithmetic assignment operators with immediate values, such as an operator used in expression `i += 5`. For adaptors `smat` and `smat_p`, only assign addition and subtraction operators are defined while all possible operators are defined for `smat_v` and `smat_d`.

⟨Arithmetic assignment operators with immediate value. 138⟩ ≡

```
#define SMAT_MDEF_ASSIGNOP_IMMEDIATE(CLASS, ID, OP, TCODE)      \
template <typename T>                                               \
  template <typename U>                                             \
inline CLASS<T>&                                                    \
CLASS<T>::operator OP(const U& x) {                                 \
  SMAT_LOG2(TCODE, ID, base_, base_);                               \
  base_ OP x; return *this;                                         \
}

SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat, 'i', +=, SMAT_IADD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat, 'i', -=, SMAT_ISUB)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_p, 'p', +=, SMAT_IADD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_p, 'p', -=, SMAT_ISUB)

SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', +=, SMAT_IADD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', -=, SMAT_ISUB)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', *=, SMAT_IMUL)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', /=, SMAT_IDIV)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', %=, SMAT_IMOD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', <<=, SMAT_ILSHIFT)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', >>=, SMAT_IRSHIFT)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', &=, SMAT_IAND)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', |=, SMAT_IOR)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_v, 'v', ^=, SMAT_IXOR)

SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', +=, SMAT_IADD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', -=, SMAT_ISUB)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', *=, SMAT_IMUL)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', /=, SMAT_IDIV)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', %=, SMAT_IMOD)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', <<=, SMAT_ILSHIFT)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', >>=, SMAT_IRSHIFT)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', &=, SMAT_IAND)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', |=, SMAT_IOR)
SMAT_MDEF_ASSIGNOP_IMMEDIATE(smat_d, 'd', ^=, SMAT_IXOR)

#undef SMAT_MDEF_ASSIGNOP_IMMEDIATE
```

Used in part 134b.

Unary Operators

⟨Unary plus, minus, complement, and address-of operators. 139⟩ ≡

```
template <typename T>
smat_v<T> smat_v<T>::operator+() const {
  smat_v<T> _tmp(base_, 0);
  SMAT_LOG1(SMAT_READ, 'v', base_); return _tmp;
}
template <typename T>
smat_v<T> smat_v<T>::operator-() const {
  smat_v<T> _tmp(-base_, 0);
  SMAT_LOG2(SMAT_MINUS, 'v', base_, _tmp); return _tmp;
}
template <typename T>
smat_v<T> smat_v<T>::operator~() const {
  smat_v<T> _tmp(~base_, 0);
  SMAT_LOG2(SMAT_CMPL, 'v', base_, _tmp); return _tmp;
}
template <typename T>
smat_v<T>* smat_v<T>::operator&() {
  return reinterpret_cast<smat_v<T>*>(&base_);
}
template <typename T>
const smat_v<T>* smat_v<T>::operator&() const {
  return reinterpret_cast<const smat_v<T>*>(&base_);
}

template <typename T>
smat_d<T> smat_d<T>::operator+() const {
  smat_d<T> _tmp(base_, 0);
  SMAT_LOG1(SMAT_READ, 'd', base_); return _tmp;
}
template <typename T>
smat_d<T> smat_d<T>::operator-() const {
  smat_d<T> _tmp(-base_, 0);
  SMAT_LOG2(SMAT_MINUS, 'd', base_, _tmp); return _tmp;
}
template <typename T>
smat_d<T> smat_d<T>::operator~() const {
  smat_d<T> _tmp(~base_, 0);
  SMAT_LOG2(SMAT_CMPL, 'd', base_, _tmp); return _tmp;
}
template <typename T>
smat_d<T>* smat_d<T>::operator&() {
  return reinterpret_cast<smat_d<T>*>(&base_);
}
template <typename T>
const smat_d<T>* smat_d<T>::operator&() const {
  return reinterpret_cast<const smat_d<T>*>(&base_);
}
```

Used in part 134b.

⟨Arrow operators. 140a⟩ ≡

```
  template <typename T>
  inline T smat<T>::operator->() const {
    SMAT_LOG1(SMAT_READ, 'i', base_); return base_;
  }

  template <typename T>
  inline T smat_p<T>::operator->() const {
    SMAT_LOG1(SMAT_READ, 'p', base_); return base_;
  }
```

Used in part 134b.

⟨Dereference operators. 140b⟩ ≡

```
  template <typename T>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat<T>::operator*() const {
    SMAT_LOG1(SMAT_READ, 'i', base_);
    return reinterpret_cast<smat_v<
      typename std::iterator_traits<T>::value_type>&>(*base_);
  }

  template <typename T>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat<T>::operator[](const
    smat_d<typename std::iterator_traits<T>::difference_type>& x) const {
    return *(*this + x);
  }

  template <typename T>
    template <typename U>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat<T>::operator[](const U& x) const {
    return *(*this + x);
  }

  template <typename T>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat_p<T>::operator*() const {
    SMAT_LOG1(SMAT_READ, 'p', base_);
    return reinterpret_cast<smat_v<
      typename std::iterator_traits<T>::value_type>&>(*base_);
  }

  template <typename T>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat_p<T>::operator[](const
    smat_d<typename std::iterator_traits<T>::difference_type>& x) const {
    return *(*this + x);
  }

  template <typename T>
    template <typename U>
  inline smat_v<typename std::iterator_traits<T>::value_type>&
  smat_p<T>::operator[](const U& x) const {
    return *(*this + x);
  }
```

Used in part 134b.

Binary Operators

⟨Binary operators. 141⟩ ≡

```
  #define SMAT_GDEF_BINARYOP(CLASS, ID, OP, TCODE)                    \
  template <typename T>                                               \
  inline const CLASS<T>                                               \
  operator OP(const CLASS<T>& lhs, const CLASS<T>& rhs) {             \
    CLASS<T> _tmp(lhs.base() OP rhs.base(), 0);                       \
    SMAT_LOG3(TCODE, ID, lhs.base(), rhs.base(),_tmp); return _tmp;   \
  }
```

```
SMAT_GDEF_BINARYOP(smat_v, 'v', +, SMAT_IADD)
SMAT_GDEF_BINARYOP(smat_v, 'v', -, SMAT_ISUB)
SMAT_GDEF_BINARYOP(smat_v, 'v', *, SMAT_IMUL)
SMAT_GDEF_BINARYOP(smat_v, 'v', /, SMAT_IDIV)
SMAT_GDEF_BINARYOP(smat_v, 'v', %, SMAT_IMOD)
SMAT_GDEF_BINARYOP(smat_v, 'v', <<, SMAT_ILSHIFT)
SMAT_GDEF_BINARYOP(smat_v, 'v', >>, SMAT_IRSHIFT)
SMAT_GDEF_BINARYOP(smat_v, 'v', &, SMAT_IAND)
SMAT_GDEF_BINARYOP(smat_v, 'v', |, SMAT_IOR)
SMAT_GDEF_BINARYOP(smat_v, 'v', ^, SMAT_IXOR)

SMAT_GDEF_BINARYOP(smat_d, 'd', +, SMAT_IADD)
SMAT_GDEF_BINARYOP(smat_d, 'd', -, SMAT_ISUB)
SMAT_GDEF_BINARYOP(smat_d, 'd', *, SMAT_IMUL)
SMAT_GDEF_BINARYOP(smat_d, 'd', /, SMAT_IDIV)
SMAT_GDEF_BINARYOP(smat_d, 'd', %, SMAT_IMOD)
SMAT_GDEF_BINARYOP(smat_d, 'd', <<, SMAT_ILSHIFT)
SMAT_GDEF_BINARYOP(smat_d, 'd', >>, SMAT_IRSHIFT)
SMAT_GDEF_BINARYOP(smat_d, 'd', &, SMAT_IAND)
SMAT_GDEF_BINARYOP(smat_d, 'd', |, SMAT_IOR)
SMAT_GDEF_BINARYOP(smat_d, 'd', ^, SMAT_IXOR)

#undef SMAT_GDEF_BINARYOP

template <typename T>
const smat<T>
operator+(const smat<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs) {
  smat<T> _tmp(lhs.base() + rhs.base(), 0);
  SMAT_LOG3(SMAT_ADD, 'i', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat<T>
operator+(const smat_d<typename std::iterator_traits<T>::difference_type>& lhs,
          const smat<T>& rhs) {
  smat<T> _tmp(lhs.base() + rhs.base(), 0);
  SMAT_LOG3(SMAT_ADD, 'i', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat<T>
operator-(const smat<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs) {
  smat<T> _tmp(lhs.base() - rhs.base(), 0);
  SMAT_LOG3(SMAT_SUB, 'i', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat_d<typename std::iterator_traits<T>::difference_type>
operator-(const smat<T>& lhs, const smat<T>& rhs) {
  smat_d<typename std::iterator_traits<T>::difference_type>
    _tmp(lhs.base() - rhs.base(), 0);
  SMAT_LOG3(SMAT_SUB, 'i', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat_p<T>
operator+(const smat_p<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs) {
  smat_p<T> _tmp(lhs.base() + rhs.base(), 0);
  SMAT_LOG3(SMAT_ADD, 'p', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat_p<T>
operator+(const smat_d<typename std::iterator_traits<T>::difference_type>& lhs,
          const smat_p<T>& rhs) {
  smat_p<T> _tmp(lhs.base() + rhs.base(), 0);
  SMAT_LOG3(SMAT_ADD, 'p', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat_p<T>
operator-(const smat_p<T>& lhs,
          const smat_d<typename std::iterator_traits<T>::difference_type>& rhs) {
  smat_p<T> _tmp(lhs.base() - rhs.base(), 0);
  SMAT_LOG3(SMAT_SUB, 'p', lhs.base(), rhs.base(), _tmp); return _tmp;
}

template <typename T>
const smat_d<typename std::iterator_traits<T>::difference_type>
operator-(const smat_p<T>& lhs, const smat_p<T>& rhs) {
  smat_d<typename std::iterator_traits<T>::difference_type>
    _tmp(lhs.base() - rhs.base(), 0);
  SMAT_LOG3(SMAT_SUB, 'p', lhs.base(), rhs.base(), _tmp); return _tmp;
}
```

Used in part 135a.

Binary Operators with Immediate Values

⟨Binary operators with immediate values. 143⟩ ≡

```
#define SMAT_GDEF_BINARYOP_IMMEDIATE(CLASS, ID, OP, TCODE)      \
template <typename T, typename U>                               \
inline const CLASS<T>                                           \
operator OP(const CLASS<T>& lhs, const U& rhs) {                \
  CLASS<T> _tmp(lhs.base() OP rhs, 0);                          \
  SMAT_LOG2(TCODE, ID, lhs.base(), _tmp); return _tmp;          \
}                                                               \
template <typename T, typename U>                               \
inline const CLASS<T>                                           \
operator OP(const U& lhs, const CLASS<T>& rhs) {                \
  CLASS<T> _tmp(lhs OP rhs.base(), 0);                          \
  SMAT_LOG2(TCODE, ID, rhs.base(), _tmp); return _tmp;          \
}
```

```
SMAT_GDEF_BINARYOP_IMMEDIATE(smat, 'i', +, SMAT_IADD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat, 'i', -, SMAT_ISUB)

SMAT_GDEF_BINARYOP_IMMEDIATE(smat_p, 'p', +, SMAT_IADD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_p, 'p', -, SMAT_ISUB)

SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', +, SMAT_IADD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', -, SMAT_ISUB)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', *, SMAT_IMUL)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', /, SMAT_IDIV)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', %, SMAT_IMOD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', <<, SMAT_ILSHIFT)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', >>, SMAT_IRSHIFT)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', &, SMAT_IAND)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', |, SMAT_IOR)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_v, 'v', ^, SMAT_IXOR)

SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', +, SMAT_IADD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', -, SMAT_ISUB)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', *, SMAT_IMUL)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', /, SMAT_IDIV)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', %, SMAT_IMOD)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', <<, SMAT_ILSHIFT)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', >>, SMAT_IRSHIFT)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', &, SMAT_IAND)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', |, SMAT_IOR)
SMAT_GDEF_BINARYOP_IMMEDIATE(smat_d, 'd', ^, SMAT_IXOR)

#undef SMAT_GDEF_BINARYOP_IMMEDIATE
```
Used in part 135a.

Stream Operators

Stream operators must be defined explicitly so that the iostream naturally works. (Otherwise, `operator<<()` with two parameters defined just above would be used, which is incorrect.)

⟨Stream operators. 144⟩ ≡
```
template <typename T>
inline std::ostream&
operator<<(std::ostream& os, const smat<T>& x) { return os << x.base(); }

template <typename T>
inline std::ostream&
operator<<(std::ostream& os, const smat_d<T>& x) { return os << x.base(); }

template <typename T>
inline std::ostream&
operator<<(std::ostream& os, const smat_v<T>& x) { return os << x.base(); }

template <typename T>
inline std::ostream&
operator<<(std::ostream& os, const smat_p<T>& x) { return os << x.base(); }
```

```
template <typename T>
inline std::istream&
operator>>(std::istream& os, smat<T>& x) { return is >> x.base(); }

template <typename T>
inline std::istream&
operator>>(std::istream& os, smat_d<T>& x) { return is >> x.base(); }

template <typename T>
inline std::istream&
operator>>(std::istream& os, smat_v<T>& x) { return is >> x.base(); }

template <typename T>
inline std::istream&
operator>>(std::istream& os, smat_p<T>& x) { return is >> x.base(); }
```

Used in part 135a.

Comparison Operators

⟨Comparison operators. 145⟩ ≡

```
#define SMAT_GDEF_CMP(CLASS, ID, OP, TCODE)                                \
template <typename T>                                                       \
inline bool operator OP(const CLASS<T>& lhs, const CLASS<T>& rhs) {        \
  SMAT_LOG2B(TCODE, ID, lhs.base(), rhs.base());                            \
  return lhs.base() OP rhs.base();                                          \
}

SMAT_GDEF_CMP(smat, 'i', ==, SMAT_EQ)
SMAT_GDEF_CMP(smat, 'i', !=, SMAT_NEQ)
SMAT_GDEF_CMP(smat, 'i', <, SMAT_LT)
SMAT_GDEF_CMP(smat, 'i', >, SMAT_GT)
SMAT_GDEF_CMP(smat, 'i', <=, SMAT_LEQ)
SMAT_GDEF_CMP(smat, 'i', >=, SMAT_GEQ)

SMAT_GDEF_CMP(smat_p, 'p', ==, SMAT_EQ)
SMAT_GDEF_CMP(smat_p, 'p', !=, SMAT_NEQ)
SMAT_GDEF_CMP(smat_p, 'p', <, SMAT_LT)
SMAT_GDEF_CMP(smat_p, 'p', >, SMAT_GT)
SMAT_GDEF_CMP(smat_p, 'p', <=, SMAT_LEQ)
SMAT_GDEF_CMP(smat_p, 'p', >=, SMAT_GEQ)

SMAT_GDEF_CMP(smat_v, 'v', ==, SMAT_EQ)
SMAT_GDEF_CMP(smat_v, 'v', !=, SMAT_NEQ)
SMAT_GDEF_CMP(smat_v, 'v', <, SMAT_LT)
SMAT_GDEF_CMP(smat_v, 'v', >, SMAT_GT)
SMAT_GDEF_CMP(smat_v, 'v', <=, SMAT_LEQ)
SMAT_GDEF_CMP(smat_v, 'v', >=, SMAT_GEQ)

SMAT_GDEF_CMP(smat_d, 'd', ==, SMAT_EQ)
SMAT_GDEF_CMP(smat_d, 'd', !=, SMAT_NEQ)
SMAT_GDEF_CMP(smat_d, 'd', <, SMAT_LT)
SMAT_GDEF_CMP(smat_d, 'd', >, SMAT_GT)
SMAT_GDEF_CMP(smat_d, 'd', <=, SMAT_LEQ)
SMAT_GDEF_CMP(smat_d, 'd', >=, SMAT_GEQ)

#undef SMAT_GDEF_CMP
```

Used in part 135a.

Comparison Operators with Immediate Values

⟨Comparison operators with immediate values. 146a⟩ ≡

```
#define SMAT_GDEF_CMP_IMMEDIATE(CLASS, ID, OP, TCODE)           \
template <typename T, typename U>                               \
inline bool operator OP(const CLASS<T>& lhs, const U& rhs) {    \
  SMAT_LOG1(TCODE, ID, lhs.base());                             \
  return lhs.base() OP rhs;                                     \
}                                                               \
template <typename T, typename U>                               \
inline bool operator OP(const U& lhs, const CLASS<T>& rhs) {    \
  SMAT_LOG1(TCODE, ID, rhs.base());                             \
  return lhs OP rhs.base();                                     \
}

SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', ==, SMAT_IEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', !=, SMAT_INEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', <, SMAT_ILT)
SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', >, SMAT_IGT)
SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', <=, SMAT_ILEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_v, 'v', >=, SMAT_IGEQ)

SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', ==, SMAT_IEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', !=, SMAT_INEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', <, SMAT_ILT)
SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', >, SMAT_IGT)
SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', <=, SMAT_ILEQ)
SMAT_GDEF_CMP_IMMEDIATE(smat_d, 'd', >=, SMAT_IGEQ)

#undef SMAT_GDEF_CMP_IMMEDIATE
```

Used in part 135a.

Disabling SMAT

The type `type`, defined for every adaptor class, yields the base type `T` if macro `SMAT_DISABLE` is defined, and yields the adapted type otherwise. This type can be used to disable memory tracing feature without modifying the source. See Section 2.6.1 for more information.

⟨Define type `type` for `smat`. 146b⟩ ≡

```
#ifdef SMAT_DISABLE
  typedef T type;
#else
  typedef smat<T> type;
#endif
```

Used in part 131.

⟨Define type type for smat_p. 147a⟩ ≡
```
#ifdef SMAT_DISABLE
  typedef T type;
#else
  typedef smat_p<T> type;
#endif
```
Used in part 132.

⟨Define type type for smat_v. 147b⟩ ≡
```
#ifdef SMAT_DISABLE
  typedef T type;
#else
  typedef smat_v<T> type;
#endif
```
Used in part 133.

⟨Define type type for smat_d. 147c⟩ ≡
```
#ifdef SMAT_DISABLE
  typedef T type;
#else
  typedef smat_d<T> type;
#endif
```
Used in part 134a.

A.4.2 Trace Generation

SMAT I/O class

Class `smat_io` holds output and input streams, and provides access functions to them.

⟨Define SMAT I/O class. 147d⟩ ≡
```
class smat_io {
  std::ostream* fout_;
  std::istream* fin_;

public:
  smat_io(std::ostream& os = std::cout, std::istream& is = std::cin)
    : fout_(&os), fin_(&is) { }
  std::ostream& fout() { return *fout_; }
  std::istream& fin() { return *fin_; }
  void set_fout(std::ostream& os) { fout_ = &os; }
  void set_fin(std::istream& is) { fin_ = &is; }
};
```
Used in part 129.

Singleton

The `smat_singleton` class guarantees that class `T` has only one instance, and provides global access to it. When a program is compiled with the SMAT library, an instance of `smat_io` is created to hold the output stream for the trace file.

⟨Define a singleton class. 148a⟩ ≡

```
template <typename T>
class smat_singleton {
  smat_singleton();
  static T* p;

public:
  static T& ref() {
    static T obj;
    if (!p)
      p = &obj;
    return *p;
  }
};

template <typename T>
T* smat_singleton<T>::p;
```

Used in part 129.

A.4.3 Trace Instruction Codes

Trace instruction codes are defined here. Use caution when modifying this because some functions, for example `smat_inst::name()`, depend on the order of the identifiers in the definition.

⟨Define trace instruction codes. 148b⟩ ≡

```
enum smat_code {
  SMAT_NOP, SMAT_MOV, SMAT_CMPL, SMAT_AND, SMAT_OR, SMAT_XOR, SMAT_ADD,
  SMAT_SUB, SMAT_MUL, SMAT_DIV, SMAT_MOD, SMAT_LSHIFT, SMAT_RSHIFT,
  SMAT_INC, SMAT_DEC, SMAT_MINUS, SMAT_EQ, SMAT_NEQ, SMAT_GT, SMAT_GEQ,
  SMAT_LT, SMAT_LEQ, SMAT_IAND, SMAT_IOR, SMAT_IXOR, SMAT_IADD,
  SMAT_ISUB, SMAT_IMUL, SMAT_IDIV, SMAT_IMOD, SMAT_ILSHIFT,
  SMAT_IRSHIFT, SMAT_IEQ, SMAT_INEQ, SMAT_IGT, SMAT_IGEQ, SMAT_ILT,
  SMAT_ILEQ, SMAT_READ, SMAT_WRITE, SMAT_DTOR, SMAT_FMARK,
  SMAT_CTOR, SMAT_CCTOR, SMAT_BCTOR, SMAT_END
};
```

Used in part 129.

A.4.4 Trace Instruction Class

⟨Define trace instruction class. 148c⟩ ≡

```
struct smat_inst {
  typedef unsigned long addr_t;

  int code, type;
  size_t len;
  addr_t src1, src2, dst;

  smat_inst(int c = SMAT_NOP, int t = 0, size_t l = 0, addr_t s1 = 0,
            addr_t s2 = 0, addr_t d = 0)
    : code(c), type(t), len(l), src1(s1), src2(s2), dst(d) {}
  smat_inst& operator=(const smat_inst& x) {
    code = x.code; type = x.type; len = x.len; src1 = x.src1;
    src2 = x.src2; dst = x.dst; return *this; }

  static std::istream& read_binary(int& c, int &t, size_t& l,
    addr_t& s1, addr_t& s2, addr_t& d, std::istream& is = std::cin);

  static std::istream& read_readable(int& c, int &t, size_t& l,
    addr_t& s1, addr_t& s2, addr_t& d, std::istream& is = std::cin);

  static std::ostream& write_readable(int c, int t, size_t l,
    addr_t s1, addr_t s2, addr_t d, std::ostream& os = std::cout);

  static std::ostream& write_binary(int c, int t, int l,
    addr_t s1, addr_t s2, addr_t d, std::ostream& os = std::cout);

  std::ostream& write_readable(std::ostream& os = std::cout) const
  { return write_readable(code, type, len, src1, src2, dst, os); }

  std::istream& read_readable(std::istream& is = std::cin)
  { return read_readable(code, type, len, src1, src2, dst, is); }

  std::ostream& write(std::ostream& os = std::cout) const
  { return write_binary(code, type, len, src1, src2, dst, os); }

  std::istream& read(std::istream& is = std::cin)
  { return read_binary(code, type, len, src1, src2, dst, is); }

  static const char* name(int c);
  static int str2code(const std::string& s);
  static inline int id2type(char id);
};
```

⟨Binary read and write functions for `smat_inst`. 149⟩

⟨Human-readable write functions for `smat_inst`. 150⟩

⟨Stream operators for `smat_inst`. 152a⟩

⟨Auxiliary functions for `smat_inst`. 152b⟩

Used in part 129.

Functions **`read_binary`** and **`write_binary`** read and write trace files in binary format. Section 4.3 describes the format-related issues. Note that every 32-bit word is converted to the network byte order on write.

⟨Binary read and write functions for `smat_inst`. 149⟩ ≡

```
  inline std::istream&
  smat_inst::read_binary(int& c, int &t, size_t& l, addr_t& s1, addr_t& s2,
                         addr_t& d, std::istream& is) {
    addr_t v[4], v0;
    is.read(reinterpret_cast<char *>(v), sizeof(v));
    v0 = ntohl(v[0]);
    c = ((v0 >> 24) & 0x3f); t = (v0 >> 30); l = v0 & 0xff;
    s1 = ntohl(v[1]); s2 = ntohl(v[2]); d = ntohl(v[3]);
    return is;
  }

  inline std::ostream&
  smat_inst::write_binary(int c, int t, int l, addr_t s1, addr_t s2, addr_t d,
                          std::ostream& os) {
    addr_t v[4];
    v[0] = htonl((t << 30) | (c << 24) | (l & 0xff));
    v[1] = htonl(s1); v[2] = htonl(s2); v[3] = htonl(d);
    return os.write(reinterpret_cast<char *>(v), sizeof(v));
  }
```
Used in part 148c.

Functions read_readable and write_readable input and output a trace instruction in human-readable format. In write_readable, the old C-style sprintf is used instead of C++ iostream for efficiency reasons.

⟨Human-readable write functions for smat_inst. 150⟩ ≡
```
inline std::ostream&
smat_inst::write_readable(int c, int t, size_t l, addr_t s1, addr_t s2,
                          addr_t d, std::ostream& os) {
  char buf[80];
  const char *opname = name(c);
  char id[] = { 'i', 'v', 'd', 'p' };

  switch (c)
    {
    case SMAT_MOV: case SMAT_CMPL: case SMAT_MINUS: case SMAT_IAND:
    case SMAT_IOR: case SMAT_IXOR: case SMAT_IADD: case SMAT_ISUB:
    case SMAT_IMUL: case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
    case SMAT_IRSHIFT: case SMAT_CCTOR: case SMAT_BCTOR:
      sprintf(buf, "\t%s%c\t%d, (0x%08lx), (0x%08lx)\n", opname, id[t], l, s1, d);
      os << buf;
      break;

    case SMAT_INC: case SMAT_DEC: case SMAT_READ: case SMAT_WRITE:
    case SMAT_DTOR: case SMAT_IEQ: case SMAT_INEQ: case SMAT_IGT:
    case SMAT_IGEQ: case SMAT_ILT: case SMAT_ILEQ: case SMAT_CTOR:
      sprintf(buf, "\t%s%c\t%d, (0x%08lx)\n", opname, id[t], l, s1);
      os << buf;
      break;

    case SMAT_AND: case SMAT_OR: case SMAT_XOR: case SMAT_ADD: case SMAT_SUB:
    case SMAT_MUL: case SMAT_DIV: case SMAT_MOD: case SMAT_LSHIFT:
    case SMAT_RSHIFT:
      sprintf(buf, "\t%s%c\t%d, (0x%08lx), (0x%08lx), (0x%08lx)\n",
              opname, id[t], l, s1, s2, d);
      os << buf;
      break;

    case SMAT_EQ: case SMAT_NEQ: case SMAT_GT: case SMAT_GEQ:
    case SMAT_LT: case SMAT_LEQ:
      sprintf(buf, "\t%s%c\t%d, (0x%08lx), (0x%08lx)\n", opname, id[t], l, s1, s2);
      os << buf;
      break;

    case SMAT_FMARK:
      os << '\t' << opname << '\t' << s1 << '\n';
      break;

    case SMAT_NOP:
      os << '\t' << opname << '\n';
      break;

    default:
      os << '\t' << opname << " (code = " <<  c << ")" << '\n';
      break;
    }

  return os;
}

inline std::istream&
smat_inst::read_readable(int& c, int &t, size_t& l, addr_t& s1, addr_t& s2,
                         addr_t& d, std::istream& is) {
  std::string mnemonic;
  char comma, paren_open, paren_close;
  is >> mnemonic;
  t = id2type(mnemonic[mnemonic.size() - 1]);
  mnemonic.erase(mnemonic.size() - 1, 1);
  c = str2code(mnemonic);
  l = s1 = s2 = d = 0;
  switch (c)
    {
    case SMAT_MOV: case SMAT_CMPL: case SMAT_MINUS: case SMAT_IAND:
    case SMAT_IOR: case SMAT_IXOR: case SMAT_IADD: case SMAT_ISUB:
    case SMAT_IMUL: case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
    case SMAT_IRSHIFT: case SMAT_CCTOR: case SMAT_BCTOR:
      is >> std::dec >> l >> std::hex;
      is >> comma >> paren_open >> s1 >> paren_close;
      is >> comma >> paren_open >> d  >> paren_close;
      break;

    case SMAT_INC: case SMAT_DEC: case SMAT_READ: case SMAT_WRITE:
    case SMAT_DTOR: case SMAT_IEQ: case SMAT_INEQ: case SMAT_IGT:
    case SMAT_IGEQ: case SMAT_ILT: case SMAT_ILEQ: case SMAT_CTOR:
      is >> std::dec >> l >> std::hex;
      is >> comma >> paren_open >> s1 >> paren_close;
      break;

    case SMAT_AND: case SMAT_OR: case SMAT_XOR: case SMAT_ADD: case SMAT_SUB:
    case SMAT_MUL: case SMAT_DIV: case SMAT_MOD: case SMAT_LSHIFT:
    case SMAT_RSHIFT:
      is >> std::dec >> l >> std::hex;
      is >> comma >> paren_open >> s1 >> paren_close;
      is >> comma >> paren_open >> s2 >> paren_close;
      is >> comma >> paren_open >> d  >> paren_close;
      break;

    case SMAT_EQ: case SMAT_NEQ: case SMAT_GT: case SMAT_GEQ:
    case SMAT_LT: case SMAT_LEQ:
      is >> std::dec >> l >> std::hex;
      is >> comma >> paren_open >> s1 >> paren_close;
      is >> comma >> paren_open >> s2 >> paren_close;
      break;

    case SMAT_FMARK:
      is >> std::dec >> s1;
      break;

    default:
      break;
    }

  return is;
}
```

Used in part 148c.

⟨Stream operators for `smat_inst`. 152a⟩ ≡

```
inline std::istream&
operator>>(std::istream& is, smat_inst& s)
{ return smat_inst::read_binary(s.code, s.type, s.len, s.src1, s.src2, s.dst, is); }

inline std::ostream&
operator<<(std::ostream& os, const smat_inst& s)
{ return s.write_binary(s.code, s.type, s.len, s.src1, s.src2, s.dst, os); }
```

Used in part 148c.

The following part defines auxiliary functions. Function `name` returns a mnemonic of the given instruction code, in a C-style string. Conversely, function `str2code` returns the instruction code of given mnemonic. Function `id2type` maps "id," which is represented by a character, one of `i`, `v`, `d`, or `p`, to numbers 0–3, respectively.

⟨Auxiliary functions for `smat_inst`. 152b⟩ ≡

```
inline const char*
smat_inst::name(int c) {
  static char* const opname[] = {
    "nop", "mov", "cmpl", "and", "or", "xor", "add", "sub",
    "mul", "div", "mod", "sl", "sr", "inc", "dec", "minus", "eq", "neq",
    "gt", "geq", "lt", "leq",
    "iand", "ior", "ixor", "iadd", "isub",
    "imul", "idiv", "imod", "isl", "isr", "ieq", "ineq",
    "igt", "igeq", "ilt", "ileq",
    "read", "write", "dtor", "mark",
    "ctor", "cctor", "bctor"
  };
  if (SMAT_NOP <= c && c < SMAT_END)
    return opname[c];
  return "???";
}

inline int
smat_inst::str2code(const std::string& s) {
  static char* const opname[] = {
    "nop", "mov", "cmpl", "and", "or", "xor", "add", "sub",
    "mul", "div", "mod", "sl", "sr", "inc", "dec", "minus", "eq", "neq",
    "gt", "geq", "lt", "leq",
    "iand", "ior", "ixor", "iadd", "isub",
    "imul", "idiv", "imod", "isl", "isr", "ieq", "ineq",
    "igt", "igeq", "ilt", "ileq",
    "read", "write", "dtor", "mark",
    "ctor", "cctor", "bctor", 0
  };

  for (int i = SMAT_NOP; i < SMAT_END; ++i)
    {
      if (s.compare(opname[i]) == 0)
        return i;
    }
  return SMAT_END;
}

inline int
smat_inst::id2type(char id) {
  static int tbl[] = {
    3, 3, 3, 2, 3, 3, 3, 3, 0, 3, 3, 3, 3, // 'a' - 'm'
    3, 3, 3, 3, 3, 3, 3, 3, 1, 3, 3, 3, 3  // 'n' - 'z'
  };
  return tbl[id - 'a'];
}
```

Used in part 148c.

A.5 Cache Simulator

"smatsim.cpp" 153 ≡

```
⟨Include common header files. 115a⟩
#include "smatsim.h"
using namespace std;

static int verbose, types, dump;

⟨Define cache objects. 154a⟩
⟨Print cache configuration. 154b⟩
⟨Cache simulation loop. 156⟩
⟨Convert a string to a value. 187b⟩
⟨Usage information for smatsim. 155a⟩
```

```
int main(int argc, char *argv[])
{
  int c;
  ofstream graph_os;

  ios::sync_with_stdio(false);
  cin.tie(0);

  ⟨Interpret command line arguments for smatsim. 155b⟩

  if (graph_os.is_open())
    sim(cache_g);
  else
    sim(cache);

  return 0;
}
```

Two cache objects, with and without plot-helper capability (outputs hit-miss information for all memory accesses), are defined. The object with plot-helper capability (`cache_g`) is used only when specified by a user since that capability slows the simulation down significantly.

⟨Define cache objects. 154a⟩ ≡

```
smat_cache<'0', NREG * 4, 4, NREG, 0, true, false,
  smat_cache<'1', L1_SIZE, L1_BLOCK, L1_ASSOC, L1_LATENCY, L1_WB, true,
  smat_cache_main<MAIN_LATENCY> > > cache_g;

smat_cache<'0', NREG * 4, 4, NREG, 0, true, false,
  smat_cache<'1', L1_SIZE, L1_BLOCK, L1_ASSOC, L1_LATENCY, L1_WB, false,
  smat_cache_main<MAIN_LATENCY> > > cache;
```

Used in part 153.

Function `config` prints the compiled-in cache parameters.

⟨Print cache configuration. 154b⟩ ≡

```
static void config()
{
  cout << "Number of registers\t" << NREG << " (32-bit)\n";

  char assoc[64];
  if (L1_ASSOC == L1_SIZE / L1_BLOCK)
    sprintf(assoc, "Fully associative");
  else if (L1_ASSOC == 1)
    sprintf(assoc, "Direct mapped");
  else
    sprintf(assoc, "%d-way set associative", L1_ASSOC);

  cout << "L1 cache block size\t" << L1_BLOCK << " bytes\n"
       << "L1 cache cache size\t" << L1_SIZE << " bytes ("
       << (L1_SIZE / L1_BLOCK) << " blocks)\n"
```

```
          << "L1 cache associativity\t" << assoc << '\n'
          << "L1 cache write policy\t"
          << (L1_WB ? "write back" : "write through") << '\n'
          << "L1 cache hit time\t" << L1_LATENCY << " clock cycle(s)\n";

    cout << "Main memory hit time\t" << MAIN_LATENCY << " clock cycle(s)\n";
  }
```

Used in part 153.

⟨Usage information for **smatsim**. 155a⟩ ≡

```
  static void usage(char *progname, int status)
  {
    if (status != 0)
      std::cerr << "Try `" << progname << " -h' for more information.\n";
    else
      {
        std::cout << "Usage: " << progname << " [OPTION]... < TRACEFILE\n\
  Simulate cache behavior caused by the given TRACEFILE.\n\
  \n\
    -c        display cache parameters and exit\n\
    -d        debug mode (-dd gives you more)\n\
    -g FILE   output time-position information to FILE (for internal use)\n\
    -t        type analysis\n\
    -v        generate detailed statistics\n\
  \n\
    -h        display this help and exit\n";
      }

    exit(status);
  }
```

Used in part 153.

⟨Interpret command line arguments for **smatsim**. 155b⟩ ≡

```
  while ((c = getopt(argc, argv, "cdg:hvt")) != -1)
    {
      switch (c)
        {
        case 'c': config(); return EXIT_SUCCESS; break;
        case 'd': ++dump; break;
        case 'g':
          graph_os.open(optarg);
          if (!graph_os)
            {
              cerr << argv[0] << ": " << optarg << ": " << strerror(errno) << '\n';
              return EXIT_FAILURE;
            }
          SMAT_SET_OSTREAM(graph_os);
          break;
        case 'h': usage(argv[0], EXIT_SUCCESS); break;
        case 't': ++types; break;
        case 'v': ++verbose; break;
        default: usage(argv[0], EXIT_FAILURE); break;
        }
    }

  if (optind != argc)
    usage(argv[0], EXIT_FAILURE);
```

Used in part 153.

A.5.1 Cache Simulation Loop

A cache simulation is performed in the following function, `sim`. It reads trace instructions one by one and calls simulator functions `c.read`, `c.write`, and `c.dtor`. These functions update the internal states of cache object `c`.

The current time (clock counts) in the simulator can be obtained by `c.time('0')`. The number of cache misses and cache hits can be accessed by `c.miss('1')` and `c.hit('1')`, respectively.

⟨Cache simulation loop. 156⟩ ≡

```
template <typename Cache>
int sim(Cache& c)
{
  istream_iterator<smat_inst> first(cin), last;
  vector<unsigned> clocks(4), hits(4), misses(4); // i, v, d, p
  int t = 0;
  size_t hit = 0, miss = 0;

  for (; first != last; ++first)
    {
      if (0 < dump)
        {
          cout << c.time('0');
          (*first).write_readable();
        }

      switch ((*first).code)
        {
        case SMAT_MOV: case SMAT_IADD: case SMAT_ISUB: case SMAT_IMUL:
        case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
        case SMAT_IRSHIFT: case SMAT_IAND: case SMAT_IOR: case SMAT_IXOR:
        case SMAT_MINUS: case SMAT_CMPL:
        case SMAT_CCTOR: case SMAT_BCTOR:
          c.read((*first).src1, (*first).len);
          c.write((*first).dst, (*first).len);
          break;

        case SMAT_INC: case SMAT_DEC:
          c.read((*first).src1, (*first).len);
          c.write((*first).src1, (*first).len);
          break;

        case SMAT_READ: case SMAT_IEQ: case SMAT_INEQ: case SMAT_IGT:
        case SMAT_IGEQ: case SMAT_ILT: case SMAT_ILEQ:
          c.read((*first).src1, (*first).len);
          break;
        case SMAT_WRITE:
          c.write((*first).src1, (*first).len);
          break;

        case SMAT_DTOR:
          c.dtor((*first).src1, (*first).len);
          break;

        case SMAT_AND: case SMAT_OR: case SMAT_XOR: case SMAT_ADD:
        case SMAT_SUB: case SMAT_MUL: case SMAT_DIV: case SMAT_MOD:
        case SMAT_LSHIFT: case SMAT_RSHIFT:
          c.read((*first).src1, (*first).len);
          c.read((*first).src2, (*first).len);
          c.write((*first).dst, (*first).len);
          break;

        case SMAT_EQ: case SMAT_NEQ: case SMAT_GT: case SMAT_GEQ:
        case SMAT_LT: case SMAT_LEQ:
          c.read((*first).src1, (*first).len);
          c.read((*first).src2, (*first).len);
          break;

        case SMAT_FMARK:
          cout << (*first).src1 << '\t';
          // c.report_simple();
          break;

        case SMAT_NOP: case SMAT_CTOR:
          break;

        default:
          return 1;
        }

      if (types)
        {
          int new_t = c.time('0');
          size_t new_hit = c.hit('1'), new_miss = c.miss('1');

          clocks[(*first).type] += new_t - t;
          hits[(*first).type] += new_hit - hit;
          misses[(*first).type] += new_miss - miss;

          t = new_t;
          hit = new_hit;
          miss = new_miss;
        }

      if (1 < dump)
        c.dump();
    }
```

```
    ⟨Output cache statistics. 158a⟩

    return 0;
  }
```

Used in part 153.

Cache statistics are printed at end of simulation. The detailed statistics will be generated with option `-v`, and statistics are presented separately for types if option `-t` is specified.

⟨Output cache statistics. 158a⟩ ≡

```
  if (verbose)
    {
      ⟨Output detailed cache statistics. 158b⟩
    }
  else if (! types)
    {
      cout << c.hit('1') << " hits, " << c.miss('1') << " misses, "
           << c.time('0') << " clocks\n";
    }

  ⟨Output cache statistics separately for types. 159a⟩
```

Used in part 156.

⟨Output detailed cache statistics. 158b⟩ ≡

```
  cout.precision(3);
  cout << "Register hits\t\t" << c.hit('0') << " hits ("
       << c.hit('0') - c.whit('0') << "r + "
       << c.whit('0') << "w)\n";
  cout << "Register misses\t\t" << c.miss('0') << " misses ("
       << c.miss('0') - c.wmiss('0') << "r + "
       << c.wmiss('0') << "w, "
       << (double)c.miss('0') / (c.hit('0') + c.miss('0')) * 100 << "%)\n";
  cout << "Register writebacks\t" << c.wb('0')
       << " (" << (double)c.wb('0') / c.miss('0') * 100
       << "% of all misses)\n";
  cout << "Register reads\t\t" << c.r('0') << " bytes\n";
  cout << "Register writes\t\t" << c.w('0') << " bytes\n\n";

  cout << "L1 cache hits\t\t" << c.hit('1') << " hits ("
       << c.hit('1') - c.whit('1') << "r + "
       << c.whit('1') << "w)\n";
  cout << "L1 cache misses\t\t" << c.miss('1') << " misses ("
       << c.miss('1') - c.wmiss('1') << "r + "
       << c.wmiss('1') << "w, "
       << (double)c.miss('1') / (c.hit('1') + c.miss('1')) * 100 << "%)\n";
  cout << "L1 cache writebacks\t" << c.wb('1')
       << " (" << (double)c.wb('1') / c.miss('1') * 100
       << "% of all misses)\n";
  cout << "L1 cache reads\t\t" << c.r('1') << " bytes\n";
  cout << "L1 cache writes\t\t" << c.w('1') << " bytes\n\n";
```

```
cout << "Main memory reads\t" << c.r('m') << " bytes\n";
cout << "Main memory writes\t" << c.w('m') << " bytes\n\n";

cout << "Total clock cycles\t" << c.time('0') << " clocks\n";
```

Used in part 158a.

⟨Output cache statistics separately for types. 159a⟩ ≡

```
if (types)
  {
    char *title[] = { "Iterator type\t\t", "Value type\t\t",
                      "Difference type\t\t", "Pointer type\t\t" };

    cout << "Total\t\t\t" << c.hit('1') << " hits, " << c.miss('1')
         << " misses, " << c.time('0') << " cycles\n";

    for (int i = 0; i < 4; ++i)
      {
        cout << title[i] << hits[i] << " hits, "
             << misses[i] << " misses, " << clocks[i] << " cycles\n";
      }

  }
```

Used in part 158a.

A.5.2 Cache Simulator Class

"smatsim.h" 159b ≡

⟨Numeric traits class. 187a⟩

⟨Plot instruction class. 167a⟩

⟨Cache tag class. 160a⟩

⟨Cache set class. 160b⟩

⟨Cache simulator class. 159c⟩

⟨Simulator class for main memory. 166a⟩

Cache class

⟨Cache simulator class. 159c⟩ ≡

```
template <char Id, int Capacity, int Blocksize, int Assoc, int Hittime,
          bool Writeback, bool Plot, typename Lower>
class smat_cache {
  typedef unsigned time_type;
  typedef unsigned addr_type;
  typedef smat_set<Assoc, addr_type, time_type> set_type;
  typedef typename set_type::iterator iterator_type;

  BOOST_STATIC_ASSERT(smat_numeric<Capacity>::is_power_of_two);
  BOOST_STATIC_ASSERT(smat_numeric<Blocksize>::is_power_of_two);
  BOOST_STATIC_ASSERT(smat_numeric<Assoc>::is_power_of_two);
  static const int capacity_exp = smat_numeric<Capacity>::log2;
  static const int blocksize_exp = smat_numeric<Blocksize>::log2;
  static const int nway_exp = smat_numeric<Assoc>::log2;
  static const int index_exp = capacity_exp - blocksize_exp - nway_exp;
  static const int nset = (1 << index_exp);
  static const addr_type offset_mask = Blocksize - 1;
  static const addr_type index_mask = (1 << index_exp) - 1;
  static const addr_type tag_mask = (1 << (32 - blocksize_exp - index_exp)) - 1;

  Lower lower;
  std::vector<set_type> setv;
  size_t total_read, total_write, hit_, miss_, wb_, wmiss_, whit_;
  time_type time_;

  ⟨Cache access functions (private). 162⟩

public:
  smat_cache() : setv(nset), total_read(0), total_write(0), hit_(0), miss_(0),
                 wb_(0), wmiss_(0), whit_(0), time_(0) {}
  ⟨Register invalidation. 164b⟩
  ⟨Cache statistics inquiry functions. 165⟩
  ⟨Cache access functions (public). 164a⟩
  ⟨Cache dump function. 166b⟩
};
```

Used in part 159b.

Blocks and Sets

A cache consists of sets (`smat_set`) and a set consists of blocks (`smat_block`). A struct `smat_block` has a tag field, dirty flag, and last referenced counter.

⟨Cache tag class. 160a⟩ ≡

```
template <typename Addr = unsigned, typename Time = unsigned>
struct smat_block {
  Addr tag;
  bool dirty;
  Time last_referred;
  smat_block(Addr t = 0) : tag(t), dirty(false), last_referred(0) {}
};
```

Used in part 159b.

A set is implemented as a vector of blocks. This design decision is made because the set associativities higher than 2-way are not effective, and so it is assumed that the 2-way cache will be used mostly, for which a vector is efficient enough. For caches with higher associativities than 2-way, a container with $O(\lg n)$ search time might be a better choice.

Member function `find_tag` returns an iterator for the block containing a given tag. If not found, `end()` is returned. Function `oldest` returns an iterator for the least recently used block. Function `is_empty` returns true if all blocks in a set are not assigned addresses.

⟨Cache set class. 160b⟩ ≡

```
template <int Assoc, typename Addr = unsigned, typename Time = unsigned>
class smat_set {
  std::vector<smat_block<Time> > lru;
  typedef typename std::vector<smat_block<Time> >::iterator iterator;

  bool is_empty()
  {
    for (iterator i = begin(); i != end(); ++i)
      {
        if ((*i).tag != 0 || (*i).last_referred != 0)
          return false;
      }
    return true;
  }

public:
  smat_set() : lru(Assoc) {}

  iterator begin() { return lru.begin(); }
  iterator end() { return lru.end(); }

  iterator find_tag(unsigned int tag)
  {
    for (iterator i = begin(); i != end(); ++i)
      {
        if ((*i).tag == tag)
          return i;
      }
    return end();
  }

  iterator oldest()
  {
    iterator i, o;
    for (i = o = begin(); i != end(); ++i)
      {
        if ((*i).last_referred < (*o).last_referred)
          o = i;
      }
    return o;
  }

  void dump(int index, int index_exp, int blocksize_exp, const char *tab)
  {
    char buf[80];

    if (is_empty())
      return;

    iterator o = oldest();

    for (iterator i = begin(); i != end(); ++i)
      {
        if ((*i).tag != 0 || (*i).last_referred != 0)
          {
            Addr addr = (((*i).tag << (index_exp + blocksize_exp))
                         | (index << blocksize_exp));
            sprintf(buf, "%s%c%08x-%02x:%d:%d%s\n", tab,
                    ((*i).dirty ? '*' : ' '), addr,
                    ((addr + (1 << blocksize_exp) - 1) & 0xff), index,
                    (*i).last_referred, (i == o ? " <--" : " "));
            std::cout << buf;
          }
      }
  }
};
```

Used in part 159b.

Cache Access

Given an address, access length, and operation type (read or write), function `access` updates the internal state of the cache. The access range may span blocks. Such an access is split by this function so that every access does not span blocks.

The cache simulation core is function `access_nosplit`. It first identifies the set and block where the given address may reside, and then updates block information and statistics. A member variable `time_` holds the current time of the cache level. The current time in different cache levels may differ slightly because the time is updated only when the access function is called.

⟨Cache access functions (private). 162⟩ ≡

```
  void access(addr_type addr, size_t size, char op)
  {
    addr_type next_block_top = (addr + Blocksize) & ~offset_mask;

    while (next_block_top - addr < size)
      {
        access_nosplit(addr, next_block_top - addr, op);
        size -= (next_block_top - addr);
        addr = next_block_top;
        next_block_top += Blocksize;
      }

    access_nosplit(addr, size, op);
  }

  time_type access_nosplit(addr_type addr, size_t size, char op)
  {
    const addr_type index = (addr >> blocksize_exp) & index_mask;
    const addr_type tag = (addr >> (blocksize_exp + index_exp)) & tag_mask;
    set_type& s = setv[index];
    iterator_type i = s.find_tag(tag);

    // Block boundary check.  The first byte and the last byte of ADDR
    // must be in the same block.
    //assert((addr & ~offset_mask) == ((addr + size - 1) & ~offset_mask));
```

```
if (op == 'w')
  total_write += size;
else
  total_read += size;

if (Plot)
  {
    smat_plot::write_binary(time_, addr, size, (i != s.end()), (op == 'w'),
                            (smat_singleton<smat_io>::ref().fout()));
  }

if (i != s.end())
  {
    // Hit.
    ++hit_;

    if (op == 'w')
      {
        ++whit_;
        if (! Writeback)
          time_ = lower.write_nosplit(addr, size, time_);
        (*i).dirty = true;
      }

    time_ += Hittime;
    (*i).last_referred = time_;
    return time_;
  }

++miss_;

i = s.oldest();
if (Writeback && (*i).dirty)
  {
    addr_type block_top = (((*i).tag << (index_exp + blocksize_exp))
                          | (index << blocksize_exp));
    time_ = lower.write_nosplit(block_top, Blocksize, time_);
    ++wb_;
  }

// Fill the cache line.
if (op == 'w')
  {
    ++wmiss_;
    //if (size < Blocksize)
    time_ = lower.read_nosplit(addr & ~offset_mask, Blocksize, time_);
    if (! Writeback)
      time_ = lower.write_nosplit(addr & ~offset_mask, Blocksize, time_);
  }
else
  time_ = lower.read_nosplit(addr & ~offset_mask, Blocksize, time_);

time_ += Hittime;
```

```
    (*i).tag = tag;
    (*i).dirty = (op == 'w' ? true : false);
    (*i).last_referred = time_;
    return time_;
  }
```

Used in part 159c.

The following public functions provide interfaces to the internal access functions. The nosplit versions are called from the upper level cache, with the current time argument which is used to update the time of this level. Functions **read** and **write** are called from user's code.

⟨Cache access functions (public). 164a⟩ ≡

```
  time_type read_nosplit(addr_type addr, size_t size, time_type current_time)
  { time_ = current_time; return access_nosplit(addr, size, 'r'); }

  time_type write_nosplit(addr_type addr, size_t size, time_type current_time)
  { time_ = current_time; return access_nosplit(addr, size, 'w'); }

  void read(addr_type addr, size_t size)
  { access(addr, size, 'r'); }

  void write(addr_type addr, size_t size)
  { access(addr, size, 'w'); }
```

Used in part 159c.

Register Invalidation

Fuction **dtor** invalidates the block containing address **addr**. If **size** is larger than the block size of cache, **size** is split so that each part does not span blocks (the last else clause). The invalidation is done by setting the last referenced time to zero and the dirty flag to false. Note that the tag field is not cleared. That is, the contents of **addr** are still valid in a register set, in spite of its name "register invalidation." This is because it is likely that **addr** is reused soon, especially in a small loop. Keeping **addr** in the register set improves performance in such a situation.

⟨Register invalidation. 164b⟩ ≡

```
  void dtor(addr_type addr, size_t size)
  {
    if ((addr & offset_mask) == 0 && size == Blocksize)
      {
        const unsigned index = (addr >> blocksize_exp) & index_mask;
        set_type& s = setv[index];
        const unsigned tag = (addr >> (blocksize_exp + index_exp)) & tag_mask;
        iterator_type i = s.find_tag(tag);
        if (i != s.end())
          {
            (*i).last_referred = 0;
            (*i).dirty = false;
          }
      }
    else if (size < Blocksize)
      return;
```

```
        else
          {
            addr_type next_block_top = (addr + Blocksize) & ~offset_mask;

            while (next_block_top - addr < size)
              {
                dtor(addr, next_block_top - addr);
                size -= (next_block_top - addr);
                addr = next_block_top;
                next_block_top += Blocksize;
              }

            dtor(addr, next_block_top - addr);
          }
      }
```

Used in part 159c.

Statistics Inquiry

Various statistics inquiry functions are provided. They take `char id` as an argument which is the first template parameter of the `smat_cache` class.

⟨Cache statistics inquiry functions. 165⟩ ≡

```
      size_t hit(const char id)
      { return id == Id ? hit_ : lower.hit(id); }

      size_t miss(const char id)
      { return id == Id ? miss_ : lower.miss(id); }

      size_t whit(const char id)
      { return id == Id ? whit_ : lower.whit(id); }

      size_t wmiss(const char id)
      { return id == Id ? wmiss_ : lower.wmiss(id); }

      size_t wb(const char id)
      { return id == Id ? wb_ : lower.wb(id); }

      size_t r(const char id)
      { return id == Id ? total_read : lower.r(id); }

      size_t w(const char id)
      { return id == Id ? total_write : lower.w(id); }

      time_type time(const char id)
      { return id == Id ? time_ : lower.time(id); }
```

Used in part 159c.

Main Memory

Class `smat_cache_main` only counts the total number of read and write, returning zeros for other inquiries.

⟨Simulator class for main memory. 166a⟩ ≡

```
template <int Hittime>
class smat_cache_main {
  size_t total_read, total_write;

 public:
  smat_cache_main() : total_read(0), total_write(0) {}

  size_t hit(const char) { return 0; }
  size_t miss(const char) { return 0; }
  size_t whit(const char) { return 0; }
  size_t wmiss(const char) { return 0; }
  size_t wb(const char) { return 0; }
  size_t r(const char) { return total_read; }
  size_t w(const char) { return total_write; }
  unsigned time(const char) { return 0; }

  unsigned read_nosplit(unsigned, size_t size, unsigned current_time)
  {
    total_read += size;
    return current_time + Hittime;
  }

  unsigned write_nosplit(unsigned, size_t size, unsigned current_time)
  {
    total_write += size;
    return current_time + Hittime;
  }

  void dump() { }
};
```

Used in part 159b.

Cache Dump

⟨Cache dump function. 166b⟩ ≡

```
void dump()
{
  int i = 0;
  for (typename std::vector<set_type>::iterator vi = setv.begin();
       vi != setv.end(); ++vi, ++i)
    (*vi).dump(i, index_exp, blocksize_exp, "\t    ");
  std::cout << '\n';
  lower.dump();
}
```

Used in part 159c.

Plot-Helper Class

The relationship between classes `smat_cache` and `smat_plot` is very much like the relationship between `smat` and `smat_inst`. Member functions of `smat_plot` are called inside `smat_cache` functions, and output data in a binary format (for efficiency in reading).

⟨Plot instruction class. 167a⟩ ≡

```
struct smat_plot {
  int atime;
  unsigned addr;
  size_t size;
  bool hit, wr;

  smat_plot() : atime(0), addr(0), size(0), hit(false), wr(false) {}

  std::ostream& write_readable(std::ostream& os = std::cout) const {
    static char buf[80];
    if (hit)
      sprintf(buf, "%u %u.0 ? %u\n", atime, addr + size / 2, size / 2);
    else
      sprintf(buf, "%u ? %u.0 %u\n", atime, addr + size / 2, size / 2);
    os << buf;
    return os;
  }

  static std::ostream& write_binary(const int& t, const unsigned &a,
    const size_t& s, const bool& h, const bool& w, std::ostream& os = std::cout) {
    unsigned v[3];
    v[0] = htonl(t); v[1] = htonl(a);
    v[2] = htonl((w ? 0x40000000 : 0) | (h ? 0x80000000U : 0) | s & 0x3fffffff);
    return os.write(reinterpret_cast<char *>(v), sizeof(v));
  }

  static std::istream& read_binary(int& t, unsigned &a, size_t& s,
    bool& h, bool&w, std::istream& is = std::cin) {
    unsigned v[3], v2;
    is.read(reinterpret_cast<char *>(v), sizeof(v));
    t = ntohl(v[0]); a = ntohl(v[1]); v2 = ntohl(v[2]); s = (v2 & 0x3fffffff);
    h = ((v2 & 0x80000000U) != 0); w = ((v2 & 0x40000000) != 0);
    return is;
  }
};

inline std::istream& operator>>(std::istream& is, smat_plot& p) {
  return smat_plot::read_binary(p.atime, p.addr, p.size, p.hit, p.wr, is);
}

inline std::ostream& operator<<(std::ostream& os, const smat_plot& p) {
  return p.write_binary(p.atime, p.addr, p.size, p.hit, p.wr, os);
}
```

Used in part 159b.

A.6 Trace File Viewer

The viewer `smatview` is quite simple. It just reads every instruction using `istream_iterator` and outputs one by one with `write_readable` function.

"smatview.cpp" 167b ≡

```
⟨Include common header files. 115a⟩

⟨Usage information for smatview. 168a⟩

int main(int argc, char *argv[])
{
  int c, verbose = 0;

  std::ios::sync_with_stdio(false);
  std::cin.tie(0);

  ⟨Interpret command line arguments for smatview. 168b⟩

  for (std::istream_iterator<smat_inst> i(std::cin), e; i != e; ++i)
    {
      if (verbose || (i->code != SMAT_NOP && i->code != SMAT_CTOR))
        (*i).write_readable();
    }

  return 0;
}
```

⟨Usage information for smatview. 168a⟩ ≡

```
static void usage(char *progname, int status)
{
  if (status != 0)
    std::cerr << "Try `" << progname << " -h' for more information.\n";
  else
    {
      std::cout << "Usage: " << progname << " [OPTION]... < TRACEFILE\n\
Convert a trace file into a human-readable format.\n\
\n\
  -v    output SMAT_NOP and SMAT_CTOR as well\n\
  -h    display this help and exit\n";
    }

  exit(status);
}
```

Used in part 167b.

⟨Interpret command line arguments for smatview. 168b⟩ ≡

```
while ((c = getopt(argc, argv, "hv")) != -1)
  {
    switch (c)
      {
      case 'h': usage(argv[0], EXIT_SUCCESS); break;
      case 'v': ++verbose; break;
      default: usage(argv[0], EXIT_FAILURE); break;
      }
  }

if (optind != argc)
  usage(argv[0], EXIT_FAILURE);
```

Used in part 167b.

A.7 Operation Countor

"smatcount.cpp" 169a ≡

```
⟨Include common header files. 115a⟩
⟨Output simple report. 170⟩
⟨Output detailed report. 171a⟩
⟨Usage information for smatcount. 171b⟩

int main(int argc, char *argv[])
{
  int c, verbose = 0;
  bool show[5] = { false, false, false, false, false };
  char *title[] = { "Iterator", "Value type", "Difference type",
                    "Pointer type", "Total" };

  std::ios::sync_with_stdio(false);
  std::cin.tie(0);

  ⟨Interpret command line arguments for smatcount. 172⟩
  ⟨Count operations. 169b⟩
  ⟨Call report functions. 169c⟩

  return 0;
}
```

Operation counts are held in five map containers. The first four maps count operations for the iterator type and its associated types. The last map counts all operations.

⟨Count operations. 169b⟩ ≡

```
std::map<int, unsigned> cnt[5]; // i, v, d, p, all

for (std::istream_iterator<smat_inst> i(std::cin), e; i != e; ++i)
  {
    ++cnt[(*i).type][(*i).code];
    ++cnt[4][(*i).code];
  }
```

Used in part 169a.

In the following code, the first if-clause corresponds to when one of options `IDVPT` is specified (selective mode), in which only specified types are reported. The else-clause reports counts only if the size of the map container is non-zero, i.e., at least one operation is counted for that type.

⟨Call report functions. 169c⟩ ≡

```
if (show[0] || show[1] || show[2] || show[3] || show[4])
  {
    for (int i = 0; i < 5; ++i)
```

```
          {
            if (show[i])
              {
                if (verbose)
                  report(title[i], cnt[i], verbose);
                else
                  report_simple(title[i], cnt[i]);
              }
          }
      }
    else
      {
        for (int i = 0; i < 5; ++i)
          {
            if (cnt[i].size() == 0)
              continue;

            if (verbose)
              report(title[i], cnt[i], verbose);
            else
              report_simple(title[i], cnt[i]);
          }
      }
```

Used in part 169a.

In the simple report, all operations are classified into three categories: assignment, comparison, and arithmetic operations. Each of them is computed by just summing up corresponding operation counts.

⟨Output simple report. 170⟩ ≡

```
    static void report_simple(char *name, std::map<int, unsigned>& m)
    {
      unsigned mov, cmp, arith;

      mov = m[SMAT_MOV] + m[SMAT_CCTOR] + m[SMAT_BCTOR];
      cmp = (m[SMAT_EQ] + m[SMAT_NEQ] + m[SMAT_GT]
             + m[SMAT_GEQ] + m[SMAT_LT] + m[SMAT_LEQ]
             + m[SMAT_IEQ] + m[SMAT_INEQ] + m[SMAT_IGT]
             + m[SMAT_IGEQ] + m[SMAT_ILT] + m[SMAT_ILEQ]);
      arith = (m[SMAT_ADD] + m[SMAT_SUB] + m[SMAT_MUL]
               + m[SMAT_DIV] + m[SMAT_MOD] + m[SMAT_LSHIFT]
               + m[SMAT_RSHIFT] + m[SMAT_INC] + m[SMAT_DEC]
               + m[SMAT_MINUS]
               + m[SMAT_IADD] + m[SMAT_ISUB] + m[SMAT_IMUL]
               + m[SMAT_IDIV] + m[SMAT_IMOD] + m[SMAT_ILSHIFT]
               + m[SMAT_IRSHIFT]
               + m[SMAT_AND] + m[SMAT_OR] + m[SMAT_XOR]
               + m[SMAT_IAND] + m[SMAT_IOR] + m[SMAT_IXOR]
               + m[SMAT_CMPL]);

      std::cout << name << ":\n  Assignment\t    " << mov
                << "\n  Comparison\t    " << cmp
                << "\n  Arithmetic\t    " << arith << "\n\n";
    }
```
Used in part 169a.

The detailed report is generated operator-wise. The struct `tbl` has a heading string and two trace instruction codes, the sum of whose operation counts is displayed.

⟨Output detailed report. 171a⟩ ≡
```
    static void report(char *name, std::map<int, unsigned>& m, int verbose)
    {
      struct tbl {
        char *heading;
        int i1, i2;
      };

      static tbl t[] = {
        { "Ctor (default)    ", SMAT_CTOR, SMAT_END },
        { "Ctor (copy)\t    ", SMAT_CCTOR, SMAT_END },
        { "Ctor (base type)  ", SMAT_BCTOR, SMAT_END },
        { "Dtor\t\t    ", SMAT_DTOR, SMAT_END },
        { "=\t\t    ", SMAT_MOV, SMAT_END },
        { "+\t\t    ", SMAT_ADD, SMAT_IADD },
        { "-\t\t    ", SMAT_SUB, SMAT_ISUB },
        { "*\t\t    ", SMAT_MUL, SMAT_IMUL },
        { "/\t\t    ", SMAT_DIV, SMAT_IDIV },
        { "%\t\t    ", SMAT_MOD, SMAT_IMOD },
        { "<<\t\t    ", SMAT_LSHIFT, SMAT_ILSHIFT },
        { ">>\t\t    ", SMAT_RSHIFT, SMAT_IRSHIFT },
        { "==\t\t    ", SMAT_EQ, SMAT_IEQ },
        { "!=\t\t    ", SMAT_NEQ, SMAT_INEQ },
        { "<\t\t    ", SMAT_LT, SMAT_ILT },
        { ">\t\t    ", SMAT_GT, SMAT_IGT },
        { "<=\t\t    ", SMAT_LEQ, SMAT_ILEQ },
        { ">=\t\t    ", SMAT_GEQ, SMAT_IGEQ },
        { "&\t\t    ", SMAT_AND, SMAT_IAND },
        { "|\t\t    ", SMAT_OR, SMAT_IOR },
        { "^\t\t    ", SMAT_XOR, SMAT_IXOR },
        { "++\t\t    ", SMAT_INC, SMAT_END },
        { "--\t\t    ", SMAT_DEC, SMAT_END },
        { "~\t\t    ", SMAT_CMPL, SMAT_END },
        { "- (unary)\t    ", SMAT_MINUS, SMAT_END }
      };

      std::cout << name << ":\n";
      for (unsigned i = 0; i < sizeof(t)/sizeof(t[0]); ++i)
        {
          if (verbose == 2 || m[t[i].i1] + m[t[i].i2] > 0)
            std::cout << "  " << t[i].heading << (m[t[i].i1] + m[t[i].i2]) << '\n';
        }
      std::cout << '\n';
    }
```
Used in part 169a.

⟨Usage information for `smatcount`. 171b⟩ ≡

```
static void usage(char *progname, int status)
{
  if (status != 0)
    std::cerr << "Try ‘" << progname << " -h’ for more information.\n";
  else
    {
      std::cout << "Usage: " << progname << " [OPTION]... < TRACEFILE\n\
Count and report the number of operations in a trace file.\n\
\n\
If one of options I, D, V, P, and T is specified, the program enters\n\
a selective mode, in which the only specified types are reported.\n\
Otherwise, types with non-zero operation counts are reported.\n\
\n\
  -I    output iterator type operations\n\
  -D    output difference type operations\n\
  -V    output value type operations\n\
  -P    output pointer type operations\n\
  -T    output total operations\n\
\n\
  -v    report verbosely (-vv gives you more)\n\
\n\
  -h    display this help and exit\n";
    }

  exit(status);
}
```

Used in part 169a.

⟨Interpret command line arguments for `smatcount`. 172⟩ ≡

```
while ((c = getopt(argc, argv, "DhIPTVv")) != -1)
  {
    switch (c)
      {
      case 'I': show[0] = true; break;
      case 'V': show[1] = true; break;
      case 'D': show[2] = true; break;
      case 'P': show[3] = true; break;
      case 'T': show[4] = true; break;
      case 'v': ++verbose; break;
      case 'h': usage(argv[0], EXIT_SUCCESS); break;
      default: usage(argv[0], EXIT_FAILURE); break;
      }
  }

if (optind != argc)
  usage(argv[0], EXIT_FAILURE);
```

Used in part 169a.

A.8 Trace File Optimizer

A.8.1 Sliding Window

Class `swin` provides a simple sliding window for a trace instruction sequence, of size `wsize` (must be the power of 2). All trace instructions are identified by a sequence number starting from 0, and instructions in the window can be accessed by `operator[]`. The window can slide only forward, and so the window only holds the last `wsize` elements.

Member function `seqno` returns the sequence number of the first element of the window. Function `back` returns a reference to the last element in the window. Functions `is_full` and `is_empty` return true if the window is full and empty, respectively. Function `read` reads one instruction from a trace file, and stores it at the end of the window. A user must make sure that there is an empty slot. Function `write` outputs the first instruction of the window and slides the window forward.

⟨Sliding window class. 173⟩ ≡

```
template <size_t wsize>
class swin {

  BOOST_STATIC_ASSERT(smat_numeric<wsize>::is_power_of_two);
  BOOST_STATIC_ASSERT(wsize >= 2);

  static const unsigned mask = wsize - 1;

  smat_inst v_[wsize];
  size_t start_;               // index of the window-top in v_.
  size_t end_;                 // index of the window-bottom in v_.
                               // window is empty if start_ == end_.
  size_t last_;                // index of the last element if any

  size_t seqno_;               // sequence number at v_[start_].

  // slide the window forward by one element.
  void slide() { ++seqno_; ++start_; start_ &= mask; }

public:
  swin() : start_(0), end_(0), seqno_(0) {}

  size_t seqno() { return seqno_; }
  smat_inst& back() { return v_[last_]; }

  bool is_full() { return (start_ & mask) == ((end_ + 1) & mask); }
  bool is_empty() { return start_ == end_; }

  template <typename T>
  smat_inst& operator[](const T& s)
  { return v_[(s - seqno_ + start_) & mask]; }

  int read() {
    if (! v_[end_].read())
      return -1;
    last_ = end_;
    ++end_;
```

```
      end_ &= mask;
      return 0;
    }

    void write() {
      v_[start_].write();
      slide();
    }
  };
```

Used in part 174.

A.8.2 Optimizer

The optimizer uses a sliding window of 1024 elements to access trace instructions, and it cannot refer to instructions outside the window.

"smatopt.cpp" 174 ≡

```
⟨Include common header files. 115a⟩
using namespace std;

⟨Numeric traits class. 187a⟩
⟨Sliding window class. 173⟩
⟨Copy propagation. 179⟩
⟨Address database. 177c⟩
⟨Usage information for smatopt. 177a⟩

struct addr_len {
  unsigned addr, len;
  int type;
  addr_len(unsigned addr_, unsigned len_, int type_)
    : addr(addr_), len(len_), type(type_) {}
};

int main(int argc, char *argv[])
{
  int c;
  addr_history ah;              // just a counter
  //smat_inst dtor_writer(SMAT_DTOR);
  swin<1024> w;
  multimap<unsigned, addr_len> dtor;

  ios::sync_with_stdio(false);
  cin.tie(0);

  ⟨Interpret command line arguments for smatopt. 177b⟩

  // sentinel
  dtor.insert(pair<unsigned, addr_len>(0xffffffffU, addr_len(0, 0, 0)));

  for (;; ++ah.seqno_born)
    {
      if (w.is_full())
        ⟨Slide the window and output the scheduled dtor. 176a⟩

      if (w.read() < 0)
        break;

      if (w.back().code != SMAT_DTOR)
        {
          addr_update(w.back(), ah);
          continue;
        }

      // Destructing unregistered address?
      Iter i = m.find(w.back().src1);
      if (i == m.end())
        continue;

      addr_history& s = i->second;

      ⟨Optimize trace instructions. 175⟩

      m.erase(i);
    }

  while (! w.is_empty())
    ⟨Slide the window and output the scheduled dtor. 176a⟩

  return EXIT_SUCCESS;
}
```

All optimization is performed in the following if statement. Note this code is executed only when the current instruction is a destructor (SMAT_DTOR). If the current address is constructed inside the window and the address is written only once (at a construction), advanced optimizations, such as copy propagation and temporary object elimination, are considered. Even if the address is not eligible to have advanced optimizations, early destruction is performed. If any optimization is ever performed, the current trace instruction is removed (overwritten by SMAT_NOP).

⟨Optimize trace instructions. 175⟩ ≡

```
    if (w.seqno() <= s.seqno_born && s.nwrite == 1)
      {
        if (s.nread > 0 && (w[s.seqno_born].code == SMAT_CCTOR
                            || w[s.seqno_born].code == SMAT_MOV
                            || w[s.seqno_born].code == SMAT_BCTOR))
          {
            // copy propagation
            if (copy_propagation(w, s.seqno_born + 1, ah.seqno_born - 1,
                  s.nread, w[s.seqno_born].dst, w[s.seqno_born].src1))
              w[s.seqno_born].code = w.back().code = SMAT_NOP;
          }
        else if (s.nread == 1 && (w[s.seqno_last].code == SMAT_CCTOR
                                  || w[s.seqno_last].code == SMAT_MOV
                                  || w[s.seqno_last].code == SMAT_BCTOR))
          {
            // temporary object elimination

            // w[s.seqno_last].dst must be born at s.seqno_last.
            Iter j = m.find(w[s.seqno_last].dst);
            if (j != m.end() && j->second.seqno_born == s.seqno_last)
              {
                w[s.seqno_born].dst = w[s.seqno_last].dst;
                w[s.seqno_last].code = w.back().code = SMAT_NOP;
                j->second.seqno_born = s.seqno_born;
              }
          }
        else
          ⟨Early destruction. 176b⟩
      }
    else
      ⟨Early destruction. 176b⟩
```

Used in part 174.

The following code outputs the oldest instruction in the window, and then outputs `dtors` that are scheduled to be written just after that instruction.

⟨Slide the window and output the scheduled dtor. 176a⟩ ≡

```
    {
      w.write();
      while (dtor.begin()->first <= w.seqno() - 1)
        {
          smat_inst::write_binary(SMAT_DTOR, dtor.begin()->second.type,
            dtor.begin()->second.len, dtor.begin()->second.addr, 0, 0);
          dtor.erase(dtor.begin());
        }
    }
```

Used in part 174.

The early destruction schedules the destructor to be written just after the instruction in which the address is last used.

⟨Early destruction. 176b⟩ ≡

```
    {
      if (s.seqno_last < ah.seqno_born - 1)
        {
          dtor.insert(pair<unsigned, addr_len>
                      (max(s.seqno_last, w.seqno()),
                       addr_len(w.back().src1, w.back().len, w.back().type)));
          w.back().code = SMAT_NOP;
        }
    }
```

Used in part 175.

The optimizer does not take any command line arguments, and so the `usage` function and `getopt` loop are simple.

⟨Usage information for **smatopt**. 177a⟩ ≡

```
static void usage(char *progname, int status)
{
  if (status != 0)
    std::cerr << "Try ‘" << progname << " -h’ for more information.\n";
  else
    {
      std::cout << "Usage: " << progname << " [OPTION]... < TRACEFILE\n\
Optimize a trace file.\n\
\n\
  -h         display this help and exit\n";
    }

  exit(status);
}
```

Used in part 174.

⟨Interpret command line arguments for **smatopt**. 177b⟩ ≡

```
while ((c = getopt(argc, argv, "h")) != -1)
  {
    switch (c)
      {
      case ’h’: usage(argv[0], EXIT_SUCCESS); break;
      default: usage(argv[0], EXIT_FAILURE); break;
      }
  }

if (optind != argc)
  usage(argv[0], EXIT_FAILURE);
```

Used in part 174.

A.8.3 Address Database

The optimizer stores all address information into database, using `hash_map` if the compiler is GCC version 3.2 or newer, or `map` otherwise. The entry recorded for an address is a struct `addr_history`. Member `seqno_born` holds the sequence number of construction, and `seqno_born` holds the sequence number in which the address is last used. Members `nwrite` and `nread` are counters for write access and read access, respectively.

Functions `addr_read` and `addr_write` update the database, and function `addr_update` calls them appropriately.

⟨Address database. 177c⟩ ≡

```
struct addr_history {
  size_t seqno_born, seqno_last, nwrite, nread;
  addr_history() : seqno_born(0), seqno_last(0), nwrite(0), nread(0) {}
};

#if __GNUC__ == 3 && 2 <= __GNUC_MINOR__
#include <ext/hash_map>

struct addr_hash {
  size_t operator()(unsigned a) const { return a / 4; }
};

typedef unsigned _Key;
typedef __gnu_cxx::hash_map<_Key, addr_history, addr_hash> Map;
typedef Map::iterator Iter;
#else
typedef unsigned _Key;
typedef map<_Key, addr_history> Map;
typedef Map::iterator Iter;
#endif

Map m;

static inline void addr_read(const _Key& addr, const addr_history& ah)
{
  pair<Iter, bool> p;
  p = m.insert(pair<_Key, addr_history>(addr, ah));
  p.first->second.seqno_last = ah.seqno_born;
  ++(p.first->second.nread);
}

static inline void addr_write(const _Key& addr, const addr_history& ah)
{
  pair<Iter, bool> p;
  p = m.insert(pair<_Key, addr_history>(addr, ah));
  p.first->second.seqno_last = ah.seqno_born;
  ++(p.first->second.nwrite);
}

static void addr_update(smat_inst& s, const addr_history& ah)
{
  switch (s.code)
    {
    case SMAT_MOV: case SMAT_IADD: case SMAT_ISUB: case SMAT_IMUL:
    case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
    case SMAT_IRSHIFT: case SMAT_IAND: case SMAT_IOR: case SMAT_IXOR:
    case SMAT_MINUS: case SMAT_CMPL: case SMAT_CCTOR: case SMAT_BCTOR:
      addr_read(s.src1, ah);
      addr_write(s.dst, ah);
      break;

    case SMAT_INC: case SMAT_DEC:
      addr_read(s.src1, ah);
      addr_write(s.src1, ah);
      break;

    case SMAT_READ: case SMAT_IEQ: case SMAT_INEQ: case SMAT_IGT:
    case SMAT_IGEQ: case SMAT_ILT: case SMAT_ILEQ:
      addr_read(s.src1, ah);
      break;

    case SMAT_WRITE:
      addr_write(s.src1, ah);
      break;

    case SMAT_AND: case SMAT_OR: case SMAT_XOR: case SMAT_ADD:
    case SMAT_SUB: case SMAT_MUL: case SMAT_DIV: case SMAT_MOD:
    case SMAT_LSHIFT: case SMAT_RSHIFT:
      addr_read(s.src1, ah);
      addr_read(s.src2, ah);
      addr_write(s.dst, ah);
      break;

    case SMAT_EQ: case SMAT_NEQ: case SMAT_GT: case SMAT_GEQ:
    case SMAT_LT: case SMAT_LEQ:
      addr_read(s.src1, ah);
      addr_read(s.src2, ah);
      break;

    case SMAT_DTOR: case SMAT_CTOR: case SMAT_NOP: case SMAT_FMARK:
      break;

    default:
      exit(EXIT_FAILURE);
      break;
    }
}
```

Used in part 174.

A.8.4 Copy Propagation

Function `copy_propagation` consists of two loops. The first loop checks whether address `addr_dst` is used as the destination address in the sequence range [`seq_from`, `seq_to`] or not. If used, the function returns false and the copy propagation is not taken place. The second loop replaces all occurrences of `addr_src` to `addr_dst`. (For `addr_src`, it is already guaranteed at the caller that the contents of `addr_src` are not changed.) The number of replacements is at most `nread` and so the second loop exits as soon as `nread` replacements.

⟨Copy propagation. 179⟩ ≡

```
  static bool copy_propagation(swin<1024>& w,
                               const size_t& seq_from, const size_t& seq_to,
                               size_t nread, const unsigned& addr_src,
                               const unsigned& addr_dst)
  {
    // make sure that addr_dst does not appear on dst.
    for (size_t s = seq_from + 1; s <= seq_to; ++s)
      {
        switch (w[s].code)
          {
          case SMAT_MOV: case SMAT_CMPL: case SMAT_MINUS: case SMAT_IAND:
          case SMAT_IOR: case SMAT_IXOR: case SMAT_IADD: case SMAT_ISUB:
          case SMAT_IMUL: case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
          case SMAT_IRSHIFT: case SMAT_CCTOR: case SMAT_BCTOR:
          case SMAT_INC: case SMAT_DEC: case SMAT_WRITE:
```

```
          case SMAT_AND: case SMAT_OR: case SMAT_XOR:
          case SMAT_ADD: case SMAT_SUB: case SMAT_MUL:
          case SMAT_DIV: case SMAT_MOD: case SMAT_LSHIFT: case SMAT_RSHIFT:
            if (w[s].dst == addr_dst)
              return false;
            break;
          }
      }

    for (size_t s = seq_from; s <= seq_to; ++s)
      {
        switch (w[s].code)
          {
          case SMAT_AND: case SMAT_OR: case SMAT_XOR: case SMAT_ADD:
          case SMAT_SUB: case SMAT_MUL: case SMAT_DIV: case SMAT_MOD:
          case SMAT_LSHIFT: case SMAT_RSHIFT:
          case SMAT_EQ: case SMAT_NEQ: case SMAT_GT: case SMAT_GEQ:
          case SMAT_LT: case SMAT_LEQ:
            if (w[s].src2 == addr_src)
              {
                w[s].src2 = addr_dst;
                if (--nread == 0)
                  return true;
              }
            // fall through

          case SMAT_MOV: case SMAT_IADD: case SMAT_ISUB: case SMAT_IMUL:
          case SMAT_IDIV: case SMAT_IMOD: case SMAT_ILSHIFT:
          case SMAT_IRSHIFT: case SMAT_IAND: case SMAT_IOR: case SMAT_IXOR:
          case SMAT_MINUS: case SMAT_CMPL: case SMAT_INC: case SMAT_DEC:
          case SMAT_READ: case SMAT_IEQ: case SMAT_INEQ: case SMAT_IGT:
          case SMAT_IGEQ: case SMAT_ILT: case SMAT_ILEQ:
          case SMAT_CCTOR: case SMAT_BCTOR:
            if (w[s].src1 == addr_src)
              {
                w[s].src1 = addr_dst;
                if (--nread == 0)
                  return true;
              }
            break;

          default:
            break;
          }
      }

    return true;
  }
```
Used in part 174.

A.9 Trace File Plotter (smatplot)

"smatplot" 180 ≡

```
#! /bin/sh

⟨Define shell functions. 184⟩

⟨Interpret command line arguments for smatplot. 181⟩

⟨Version dependent hack. 182a⟩

EXT=‘echo "$OUTFILE" | sed -ne 's!^.*\.!!p'‘
BASE=‘echo "$OUTFILE" | sed -e 's!\.[^.]*$!!'‘

⟨Define line styles. 182b⟩

if [ x"$XTICS" != x\# ]; then XTICS="set xtics $XTICS"; fi

TMPDIR=${TMP-/tmp}/smat$$
rm -rf $TMPDIR
mkdir $TMPDIR || exit 1
trap 'rm -rf $TMPDIR' 0 1 2 15

⟨Run simulator and helper command. 183a⟩

⟨Output range information. 183b⟩

⟨Determine yrange and xrange. 183c⟩

if [ x"$CMDFILE" != x ]; then
    cp $TMPDIR/a.out $CMDFILE.dat || exit 1; DATFILE=$CMDFILE.dat
fi

cat <<EOF > $TMPDIR/cmd
⟨Template of gnuplot command file. 185a⟩
EOF

if [ x"$CMDFILE" != x ]; then
    cp $TMPDIR/cmd $CMDFILE.cmd || exit 1; exit 0;
fi

gnuplot $TMPDIR/cmd

⟨Convert to PNG format if necessary. 185b⟩
```

Command line arguments are processed and shell variables are initialized.

```
⟨Interpret command line arguments for smatplot. 181⟩ ≡
    PREFIX=
    OUTFILE='tmp.eps'
    XRANGE=\#
    YRANGE=\#
    SIM=smatsim
    XTICS=\#
    LINESTYLE=linestyle
```

```
while [ x"$1" != x ]; do
    case $1 in
    -A) PREFIX=$2; shift; shift; continue;;
    -a) RANGEONLY=y; shift; continue;;
    -c) CMDFILE=$2; shift; shift; continue;;
    -h) usage; exit 0;;
    -o) OUTFILE=$2; shift; shift; continue;;
    -x) X=`echo $2 | sed -e 's/^[0-9][0-9]*:[0-9][0-9]*$//g'`
        if [ x"$X" != x ]; then #if
            echo $0: invalid clock range $2 1>&2; exit 1
        fi
        XRANGE="[$2]"; shift; shift; continue;;
    -y) Y=`echo $2 | sed -e 's/^0x[a-f0-9][a-f0-9]*:0x[a-f0-9][a-f0-9]*$//g'`
        if [ x"$Y" != x ]; then
            echo $0: invalid address range $2 1>&2; exit 1
        fi
        YRANGEHEX="$2"; YRANGE=`convyrange $2`; shift; shift; continue;;
    -X) XTICS=$2; shift; shift; continue;;

    -m) MONO=y; shift; continue;;
    -s) SIM=$2; shift; shift; continue;;

    -*) echo $0: Try \`$0 -h\' for more information. 1>&2; exit 1;;
    *) break;;
    esac
done

if [ 0 -ne $# ]; then usage; exit 0; fi
```

Used in part 180.

Some versions of `gnuplot` seem to prefer `set style line` instead of `set linestyle`, to specify a line style. The following code determines which syntax to use.

⟨Version dependent hack. 182a⟩ ≡

```
# adhoc version check to distinguish "set linestyle" and "set style line"
if [ ! -x `which gnuplot 2>/dev/null` ]; then
    echo cannot find gnuplot 1>&2; exit 1
fi
BINARY=`which gnuplot`
if strings $BINARY | grep version\.c | grep 'v 1.4' >/dev/null; then
    LINESTYLE="style line"
fi
```

Used in part 180.

Line styles are defined in variables `STYLE_A` (for cache misses) and `STYLE_B` (for cache hits).

⟨Define line styles. 182b⟩ ≡

```
if [ x"$EXT" = xeps ]; then
    TERMINAL="postscript eps color \"Helvetica\" 20"
    EXT_=eps
    STYLE_A="1 lt 1 pt 5 ps .8"
    STYLE_B="2 lt 9 pt 5 ps .2"
```

```
        if [ x"$MONO" = xy ]; then
            TERMINAL="postscript eps color \"Helvetica\" 20"
            EXT_=eps
            STYLE_A="1 lt 5 pt 5 ps .8"
            STYLE_B="2 lt 7 pt 5 ps .2"
        fi
    elif [ x"$EXT" = xpbm -o x"$EXT" = xpng ]; then
        TERMINAL="pbm small color"
        EXT_=pbm
        STYLE_A="1 lt 1 pt 3 ps .5"
        STYLE_B="2 lt 0 pt 0 ps .2"
    fi
```

Used in part 180.

Now run a cache simulator `SIM`, with option `-g` which directs the simulator to generate hit-miss information for all memory accesses. A helper command **`smatrange`** reads that file to convert it into a gnuplot-readable format and to determine address ranges the trace contains.

⟨Run simulator and helper command. 183a⟩ ≡

```
    $SIM -g $TMPDIR/log >/dev/null || exit 1

    DATFILE=$TMPDIR/a.out
    smatrange <$TMPDIR/log >$DATFILE 2>$TMPDIR/range || exit 1
```

Used in part 180.

If option `-a` is specified, it only prints the range information and exits.

⟨Output range information. 183b⟩ ≡

```
    if [ x"$RANGEONLY" = xy ]; then
        cat <<EOF
    The trace file contains the following address ranges.  To plot the
    address range other than the first one, specify the first two digits
    shown at the top of line to option -A, for example, "-A bf".

    EOF
        awk '{ printf "     %s: %s-%s\n", $1, $2, $3;}' $TMPDIR/range
        exit 0
    fi
```

Used in part 180.

The y-range and x-range are determined if not given as the command line option. It seems that **`gnuplot`** does not accept y-range larger than 0x8000000 if specified with integers, and so real number representation is used in `YRANGE`. Variable `YRANGEHEX` holds its hexadecimal notation just for a commentary in the command file.

⟨Determine yrange and xrange. 183c⟩ ≡

```
    if [ x"$YRANGE" = x\# ]; then
        if [ x"$PREFIX" = x ]; then
            YRANGE=`sed -n 1p $TMPDIR/range \
                | awk '{printf "[%u.0:%u.0]", $4, $5}'`
            YRANGEHEX=`sed -n 1p $TMPDIR/range \
                | awk '{printf "[%s:%s]", $2, $3}'`
            if [ x"$YRANGE" = x ]; then
                echo "$0: no traces to plot" 1>&2; exit 1
            fi
        else
            YRANGE=`grep -i 0x$PREFIX $TMPDIR/range \
                | awk '{printf "[%u.0:%u.0]", $4, $5}'`
            YRANGEHEX=`grep -i 0x$PREFIX $TMPDIR/range \
                | awk '{printf "[%s:%s]", $2, $3}'`
            if [ x"$YRANGE" = x ]; then
                echo "$0: no traces for address prefix $PREFIX" 1>&2; exit 1
            fi
        fi
    fi

    if [ x"$XRANGE" = x\# ]; then
        X1=`tail -1 $DATFILE | sed -e 's/ .*//'`
        XRANGE="[0:$X1]"
    fi
```
Used in part 180.

Function **hextodec** converts a number in hexadecimal notation into the decimal notation. Function **convyrange** takes a string like `0x08082290:0x08084000` into `[134750864.0:134758400.0]`, using **hextodec**. Function **usage** displays the command line help.

⟨Define shell functions. 184⟩ ≡
```
    hextodec () {
        ( echo ibase=16; \
          ( echo $1 | sed -e s/0x//g -e y/abcdef/ABCDEF/ ) ) | bc
    }

    convyrange () {
        FROM=`echo $1 | sed -e 's/^\(.*\):.*/\1/'`
        FROM=`hextodec $FROM`
        TO=`echo $1 | sed -e 's/^[^:]*:\(.*\)/\1/'`
        TO=`hextodec $TO`
        echo "[$FROM.0:$TO.0]"
    }

    usage () {
        cat <<EOF
    Usage: $0 [OPTION]... < TRACEFILE

      -A PREFIX    process the address range which has PREFIX
      -a           just print the range information and exit
      -c NAME      generate command file NAME.cmd and data file NAME.dat
                     for gnuplot
      -o FILE.EXT  output file (EXT should be one of: eps, png, pbm)
                     'tmp.eps' if not given
      -x FROM:TO   clock range
      -y FROM:TO   address range in hex (0x08082290:0x08083a00, for example)
      -X NUM       change x-tics
```

```
  -h           display this help and exit
EOF
}
```

Used in part 180.

Here is a template of the gnuplot command file.

⟨Template of gnuplot command file. 185a⟩ ≡

```
set term $TERMINAL
set output "$BASE.$EXT_"
$XTICS
# Note YRANGE does not accept hexadecimal range nor even integral range
# for high address range.  So we use real number as YRANGE.
set xrange $XRANGE
set yrange $YRANGE # $YRANGEHEX
set nokey
set xlabel "Clock Cycles" 0,0.15
set ylabel "Address" 0.5,0
set format y ""
set rmargin 4
set $LINESTYLE $STYLE_A
set $LINESTYLE $STYLE_B
plot "$DATFILE" using 1:3 title "Misses" with points ls 1,\
     "$DATFILE" using 1:2 title "Hits" with points ls 2
```

Used in part 180.

At last, the output file `$BASE.$EXT_` is converted to PNG format using `pnmtopng` if specified so.

⟨Convert to PNG format if necessary. 185b⟩ ≡

```
if [ x"$EXT" = xpng ]; then
    pnmtopng -compress 9 < $BASE.$EXT_ > $BASE.png 2>/dev/null \
        && ( echo generated $BASE.png; rm -f $BASE.$EXT_ ) \
        || ( echo $BASE.$EXT_: cannot convert to PNG format 1>&2; \
             rm -f $BASE.png )
else
    echo generated $BASE.$EXT_
fi
```

Used in part 180.

A.9.1 Helper Command for Plotter

Command `smatrange` inputs a file generated by the cache simulator with option `-g`, and outputs hit-miss information for every access in gnuplot-readable format. Besides, it generates address range information.

"smatrange.cpp" 185c ≡

```
⟨Include common header files. 115a⟩
#include "smatsim.h"

using namespace std;
```

```
template <typename T>
static void update_range(T& lm, T& um, unsigned lower, unsigned upper)
{
  unsigned ltag = (lower >> 24), utag = (upper >> 24);
  if (lm[(ltag) * 256] == 0)
    {
      lm[(ltag) * 256] = 1;
      lm[ltag] = lower;
    }
  else if (lower < lm[ltag])
    lm[ltag] = lower;

  if (um[(utag) * 256] == 0)
    {
      um[(utag) * 256] = 1;
      um[utag] = upper;
    }
  else if (um[utag] < upper)
    um[utag] = upper;
}

int main()
{
  map<unsigned, unsigned> lm, um;

  ios::sync_with_stdio(false);
  cin.tie(0);

  for (istream_iterator<smat_plot> i(cin), e; i != e; ++i)
    {
      (*i).write_readable();
      update_range(lm, um, (*i).addr, (*i).addr + (*i).size);
    }

  map<unsigned, unsigned>::iterator mi;
  for (mi = lm.begin(); mi != lm.end(); ++mi)
    {
      if ((*mi).first <= 0xff)
        {
          cerr << hex << setw(2) << setfill('0') << ((*mi).second >> 24)
               << " 0x" << hex << setw(8) << setfill('0')
               << (*mi).second << " 0x" << hex << setw(8) << setfill('0')
               << um[(*mi).first] << dec << ' ' << (*mi).second
               << ' ' << um[(*mi).first] << '\n';
        }
    }

  return EXIT_SUCCESS;
}
```

A.10 Miscellaneous

The SMAT numeric traits class can compute the logarithm of base 2, and check whether an interger is a power of 2, at compile-time.

⟨Numeric traits class. 187a⟩ ≡

```
template <int n>
struct smat_numeric {
  enum {
    log2 = 1 + smat_numeric<n / 2>::log2,
    on_bit = (n & 1) + smat_numeric<n / 2>::on_bit,
    is_power_of_two = (on_bit == 1)
  };
};
template <> struct smat_numeric<2>
{ enum { log2 = 1, on_bit = 1, is_power_of_two = true }; };
template <> struct smat_numeric<1>
{ enum { log2 = 0, on_bit = 1, is_power_of_two = true }; };
template <> struct smat_numeric<0>
{ enum { on_bit = 0, is_power_of_two = false }; };
```

Used in parts 159b,174.

Numbers specified in the command line are converted to values by `lazy_atoi` funtion. Although its name says `atoi`, `lazy_atoi` can be actually used to convert a string into any type `T` if the operator `istream& operator>>(istream&, T&)` is defined.

⟨Convert a string to a value. 187b⟩ ≡

```
template <typename T>
static bool lazy_atoi(T* dst, const char *s)
{
  std::istringstream ist(s);
  return (!(ist >> *dst).fail() && (ist >> std::ws).eof()) ? true : false;
}
```

Used in parts 113a,117,153.

The template class `genrand_limit` is a random number generator based on Mersenne Twister by Matsumoto and Nishimura [15]. The value type of the random numbers must be given by the template parameter. The constructor takes two optional arguments, a random seed and the upper limit of a random number sequence. If the upper limit `u` is given, random numbers are in a range $[0, \mathtt{u}]$ (note `u` is included); otherwise, the maximum value of the value type is used as the upper limit. The result is undefined if `u` is negative.

⟨Random number generator class. 187c⟩ ≡

```
#include <boost/random.hpp>

template <typename Elem>
class genrand_limit {
  ⟨New initializer for boost::mt19937. 188a⟩
  mt19937_new_initializer inimt;
  boost::mt19937 genmt;
  double divisor;
public:
  genrand_limit(unsigned seed = 7U,
                Elem u = std::numeric_limits<Elem>::max())
    : inimt(seed), genmt(inimt),
      divisor((std::numeric_limits<unsigned>::max() + 1.0) / (u + 1.0)) {}
  Elem operator()() { return static_cast<Elem>(genmt() / divisor); }
};
```

Used in parts 113a,117,199d.

The initializer implemented in `boost::mt19937` is known to have a small problem that the most significant bit of the seed does not contribute much to the randomization. A new initializer recommended by Matsumoto and Nishimura [15] fixes this problem.

⟨New initializer for `boost::mt19937`. 188a⟩ ≡

```
class mt19937_new_initializer {
  unsigned x, c;
public:
  explicit mt19937_new_initializer(unsigned i = 1) : x(i), c(0) { }
  unsigned operator()()
  { return ++c == 1 ? x : x = (1812433253U * (x ^ (x >> 30)) + c - 1); }
};
```

Used in part 187c.

A.11 STL Temporary Buffer

The following file, `stl_tempbuf.h`, is the modified version of the same file included in STL header files. This file uses the correct types that are traceable by SMAT. See `stlsort.cpp` for an example of how to use this version instead of the original one.

"stl_tempbuf.h" 188b ≡

```
// Temporary buffer implementation -*- C++ -*-

// Copyright (C) 2001, 2002 Free Software Foundation, Inc.
//
// This file is part of the GNU ISO C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
// Free Software Foundation; either version 2, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License along
// with this library; see the file COPYING.  If not, write to the Free
// Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307,
// USA.

// As a special exception, you may use this file as part of a free software
// library without restriction.  Specifically, if other files instantiate
// templates or use macros or inline functions from this file, or you compile
// this file and link it with other files to produce an executable, this
// file does not by itself cause the resulting executable to be covered by
// the GNU General Public License.  This exception does not however
// invalidate any other reasons why the executable file might be covered by
// the GNU General Public License.

/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

/** @file stl_tempbuf.h
 *  This is an internal header file, included by other library headers.
 *  You should not attempt to use it directly.
 */

#ifndef __GLIBCPP_INTERNAL_TEMPBUF_H
#define __GLIBCPP_INTERNAL_TEMPBUF_H

namespace std
{

/**
 *  @if maint
 *  This class is used in two places:  stl_algo.h and ext/memory, where it
 *  is wrapped as the temporary_buffer class.  See temporary_buffer docs for
 *  more notes.
 *  @endif
*/
template <class _ForwardIterator, class _Tp>
```

```
  class _Temporary_buffer
{
  // concept requirements
  __glibcpp_class_requires(_ForwardIterator, _ForwardIteratorConcept)

  typedef typename iterator_traits<_ForwardIterator>::difference_type _Distance;
  typedef typename iterator_traits<_ForwardIterator>::pointer _Pointer;

  _Distance  _M_original_len;
  _Distance  _M_len;
  _Pointer   _M_buffer;

  // this is basically get_temporary_buffer() all over again
  void _M_allocate_buffer() {
    _M_original_len = _M_len;
    _M_buffer = 0;

    if (_M_len > (ptrdiff_t)(INT_MAX / sizeof(_Tp)))
      _M_len = INT_MAX / sizeof(_Tp);

    while (_M_len > 0) {
      _M_buffer = (_Tp*) malloc(_M_len * sizeof(_Tp));
      if (_M_buffer)
        break;
      _M_len /= 2;
    }
  }

  void _M_initialize_buffer(const _Tp&, __true_type) {}
  void _M_initialize_buffer(const _Tp& val, __false_type) {
    uninitialized_fill_n(_M_buffer, _M_len, val);
  }

public:
  /// As per Table mumble.
  _Distance size() const { return _M_len; }
  /// Returns the size requested by the constructor; may be >size().
  _Distance requested_size() const { return _M_original_len; }
  /// As per Table mumble.
  _Pointer begin() { return _M_buffer; }
  /// As per Table mumble.
  _Pointer end() { return _M_buffer + _M_len; }

  _Temporary_buffer(_ForwardIterator __first, _ForwardIterator __last) {
    // Workaround for a __type_traits bug in the pre-7.3 compiler.
    typedef typename __type_traits<_Tp>::has_trivial_default_constructor
            _Trivial;

    try {
      _M_len = distance(__first, __last);
      _M_allocate_buffer();
      if (_M_len > 0)
        _M_initialize_buffer(*__first, _Trivial());
    }
    catch(...)
      {
        free(_M_buffer);
        _M_buffer = 0;
        _M_len = 0;
        __throw_exception_again;
      }
  }

  ~_Temporary_buffer() {
    _Destroy(_M_buffer, _M_buffer + _M_len);
    free(_M_buffer);
  }

private:
  // Disable copy constructor and assignment operator.
  _Temporary_buffer(const _Temporary_buffer&) {}
  void operator=(const _Temporary_buffer&) {}
};

} // namespace std

#endif /* __GLIBCPP_INTERNAL_TEMPBUF_H */
```

Appendix B

Test Plan and Results

Four SMAT adaptors, six utility programs, and three sorting algorithms are tested. Section B.1 tests SMAT adaptors by executing all overloaded functions. The compilation of the test program itself is also a test. Utility programs are tested in Section B.2 using regression-style tests, in which every program is given some inputs and the output is compared with known answers. Section B.3 tests sorting algorithms, introsort with early finalization, introsort with Lomuto's partition, and tiled mergesort. All tests presented in this chapter have been confirmed to run successfully.

A shell script, `test-exec.sh`, invokes tests described in Section B.1 and B.2. It executes test programs or test shell functions listed in variable `TESTS` one by one and reports the result.

"test-exec.sh" 192 ≡

```
#! /bin/sh

TESTS="test-smat sh_reg_sim sh_reg_opt sh_reg_count sh_reg_view  \
       sh_reg_plot sh_random"

TMPDIR=${TMP-/tmp}/smat$$; rm -rf $TMPDIR; mkdir $TMPDIR || exit 1
trap 'rm -rf $TMPDIR' 0 1 2 15

⟨Regression test for cache simulator. 196b⟩
⟨Regression test for trace file optimizer. 197b⟩
⟨Regression test for operation counter. 198a⟩
⟨Regression test for trace file viewer. 199a⟩
⟨Regression test for trace file plotter. 199b⟩
⟨Regression test for random number generator. 200a⟩

failed=0; all=0; for tst in $TESTS; do
  if test -f ./$tst; then tst=./$tst; fi
  if $tst > /dev/null; then all=`expr $all + 1`; echo "PASS: $tst";
  elif test $? -ne 77; then all=`expr $all + 1`; failed=`expr $failed + 1`;
  echo "FAIL: $tst"; fi;
done;
if test "$failed" -eq 0; then banner="All $all tests passed";
else banner="$failed of $all tests failed"; fi;
dashes=`echo "$banner" | sed s/./=/g`;
echo "$dashes"; echo "$banner"; echo "$dashes"; test "$failed" -eq 0
```

B.1 SMAT Adaptors

Test program `test-smat.cpp` invokes all operators that SMAT adaptors can overload. The results of each operation are confirmed one by one, using macro `EXPECT_TRUE`, to make sure that the operations yield correct answers.

This test succeeds if the program exits with status 0. The trace file is *not* examined here because the answer file would be more than ten pages.

"test-smat.cpp" 193 ≡

```
    ⟨Include common header files. 115a⟩

    #define EXPECT_TRUE(EXPR) \
      do { if (EXPR) ; else exit(EXIT_FAILURE); } while (0)

    short a[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

    template <typename T>
    static void cmp()
    {
      T i1(&a[0]), i2 = &a[2], i3;
      i3 = &a[2];

      EXPECT_TRUE(i2 == i3 && i1 != i2 && !(i2 == i1 || i2 != i3));
      EXPECT_TRUE(i1 < i2 && !(i2 < i1 || i2 < i3));
      EXPECT_TRUE(i2 > i1 && !(i1 > i2 || i3 > i2));
      EXPECT_TRUE(i1 <= i2 && i2 <= i3 && !(i2 <= i1));
      EXPECT_TRUE(i2 >= i1 && i3 >= i2 && !(i1 >= i2));
    }

    template <typename T>
    static void deref()
    {
      T i1(&a[0]), i2(&a[1]), i3 = i1;
      typename T::value_type v1 = a[0], v2 = a[1];
      typename T::difference_type d = 1;

      EXPECT_TRUE(*i1 == v1 && !(*i1 == v2));
      EXPECT_TRUE(*i1++ == v1 && i1 == i2 && *i1-- == v2 && i1 == i3);
      EXPECT_TRUE(*++i1 == v2 && i1 == i2 && *--i1 == v1 && i1 == i3);
      *i1 = v1; EXPECT_TRUE(*i1 == v1);
      EXPECT_TRUE(i1[d] == *i2 && i1[1] == *i2); i1[1] = v2;
    }

    template <typename T>
    static void plusminus()
    {
      T i1(&a[0]), i2(&a[5]), i3(i1);
      typename T::difference_type d = 5;

      EXPECT_TRUE(i2 == i1 + 5 && i1 + 5 == i2 && i2 == 5 + i1 && 5 + i1 == i2);
      EXPECT_TRUE(i2 == i1 + d && i1 + d == i2 && i2 == d + i1 && d + i1 == i2);

      EXPECT_TRUE(i1 == i2 - 5 && i2 - 5 == i1);
```

```
  EXPECT_TRUE(i1 == i2 - d && i2 - d == i1);
  EXPECT_TRUE(i1 == i2 + (-d) && i2 + (-d) == i1);

  i1 += d; EXPECT_TRUE(i1 == i2);
  i1 -= d; EXPECT_TRUE(i1 == i3);
  i1 += 5; EXPECT_TRUE(i1 == i2);
  i1 -= 5; EXPECT_TRUE(i1 == i3);

  EXPECT_TRUE(i1 && !!i1);
}

struct foo { int x, y; };

static void arrow()
{
  foo z = { 1, 2 };
  smat<foo*> i = &z;
  EXPECT_TRUE(i->x == 1 && i->y == 2);
}

template <typename T>
static void arith()
{
  T x(0),y,z(x);

  x = y = z = 5;
  x += 3; EXPECT_TRUE(x + 7 == 15 && 7 + x == 15); // x == 8
  x -= 2; EXPECT_TRUE(x - 2 == 4 && 8 - x == 2); // x == 6
  x *= -2; EXPECT_TRUE(x * 4 == -48 && 4 * x == -48); x *= 4; // x == -48
  x /= 3; EXPECT_TRUE(x / (-2) == 8 && 100 / x == -6); x /= -2;

  EXPECT_TRUE(++x == 9 && x++ == 9 && x == 10);
  EXPECT_TRUE(--x == 9 && x-- == 9 && x == 8);

  x %= 5; EXPECT_TRUE(x == 3 && x % 7 == 3 && (10 % (x = 3)) == 1);
  x <<= 3; EXPECT_TRUE(x == 24 && (x << 2) == 96 && ((1 << (x = 6)) == 64));
  x = 24;
  x >>= 1; EXPECT_TRUE(x == 12 && (x >> 1) == 6 && ((128 >> (x = 2)) == 32));

  x = 12;
  x &= 0x1c; EXPECT_TRUE(x == 0xc && (x & 0x1c) == 0xc && (0x1c & x) == 0xc);
  x |= 0x11; EXPECT_TRUE(x == 0x1d && (x | 0x20) == 0x3d && (0x20|x) == 0x3d);
  x ^= 0xff; EXPECT_TRUE(x == 0xe2 && (x ^ 0xff) == 0x1d && (0xff^x) == 0x1d);

  EXPECT_TRUE(+y == 5 && -y == -5 && ~y == -6 && y == z && y != x);
  EXPECT_TRUE(x > y && y < x && x >= y && y <= x && y >= z && y <= z);
}

int main()
{
  cmp<smat<short *> >();
  deref<smat<short *> >();
  plusminus<smat<short *> >();
```

```
    arrow();

    cmp<smat_p<short *> >();
    deref<smat_p<short *> >();
    plusminus<smat_p<short *> >();

    arith<smat_d<int> >();
    arith<smat_v<int> >();
  }
```

B.2 Regression Tests

All six utility programs are executed to process the following trace file, which is generated by the `std::copy_backward` algorithm. Long sequences of constructors and destructors in the trace are due to five levels of parameter passing. In fact, observing these long sequences was the motivation to develop the trace file optimizer.

"test-reg.log" 195 ≡

```
             bctori  4, (0xbffff71c), (0xbffff834)
             bctori  4, (0xbffff718), (0xbffff824)
             bctori  4, (0xbffff714), (0xbffff814)
             cctori  4, (0xbffff834), (0xbffff804)
             cctori  4, (0xbffff824), (0xbffff7f4)
             cctori  4, (0xbffff814), (0xbffff7e4)
             cctori  4, (0xbffff804), (0xbffff7d4)
             cctori  4, (0xbffff7f4), (0xbffff7c4)
             cctori  4, (0xbffff7e4), (0xbffff7b4)
             cctori  4, (0xbffff7d4), (0xbffff7a4)
             cctori  4, (0xbffff7c4), (0xbffff794)
             cctori  4, (0xbffff7b4), (0xbffff784)
             cctori  4, (0xbffff7a4), (0xbffff774)
             cctori  4, (0xbffff794), (0xbffff764)
             cctori  4, (0xbffff784), (0xbffff754)
             cctori  4, (0xbffff774), (0xbffff744)
             cctori  4, (0xbffff764), (0xbffff734)
             cctori  4, (0xbffff754), (0xbffff724)
             ctord   4, (0xbffff6c0)
             ctord   4, (0xbffff6b0)
             subi    4, (0xbffff734), (0xbffff744), (0xbffff6b0)
             movd    4, (0xbffff6b0), (0xbffff6c0)
             dtord   4, (0xbffff6b0)
             igtd    4, (0xbffff6c0)
             deci    4, (0xbffff724)
             readi   4, (0xbffff724)
             deci    4, (0xbffff734)
             readi   4, (0xbffff734)
             movv    4, (0x0804a68c), (0x0804a690)
             decd    4, (0xbffff6c0)
             igtd    4, (0xbffff6c0)
             deci    4, (0xbffff724)
             readi   4, (0xbffff724)
```

```
        deci    4, (0xbffff734)
        readi   4, (0xbffff734)
        movv    4, (0x0804a688), (0x0804a68c)
        decd    4, (0xbffff6c0)
        igtd    4, (0xbffff6c0)
        cctori  4, (0xbffff724), (0xbffff844)
        dtord   4, (0xbffff6c0)
        dtori   4, (0xbffff724)
        dtori   4, (0xbffff734)
        dtori   4, (0xbffff744)
        dtori   4, (0xbffff754)
        dtori   4, (0xbffff764)
        dtori   4, (0xbffff774)
        dtori   4, (0xbffff784)
        dtori   4, (0xbffff794)
        dtori   4, (0xbffff7a4)
        dtori   4, (0xbffff7b4)
        dtori   4, (0xbffff7c4)
        dtori   4, (0xbffff7d4)
        dtori   4, (0xbffff7e4)
        dtori   4, (0xbffff7f4)
        dtori   4, (0xbffff804)
        dtori   4, (0xbffff844)
        dtori   4, (0xbffff814)
        dtori   4, (0xbffff824)
        dtori   4, (0xbffff834)
```

All utility programs are only able to input trace files in binary format, and so `test-reg.log` must be converted into binary format. The following helper program reads a human-readable trace file and outputs in binary format, just the reverse of `smatview`,

"`test-readlog.cpp`" 196a ≡

⟨Include common header files. 115a⟩

```
int main()
{
  smat_inst i;
  for (;;)
    {
      if (i.read_readable())
        std::cout << i;
      else
        break;
    }
  return EXIT_SUCCESS;
}
```

B.2.1 Cache Simulator

The output with `-v` and `-t` options are generated and compared.

⟨Regression test for cache simulator. 196b⟩ ≡

```
sh_reg_sim () {
    ./test-readlog < test-reg.log | ./smatsim -vt > $TMPDIR/out || return 1
    cmp $TMPDIR/out ans-reg-sim.txt || return 1
    return 0
}
```

Used in part 192.

"ans-reg-sim.txt" 197a ≡

```
Register hits           39 hits (32r + 7w)
Register misses         27 misses (5r + 22w, 40.9%)
Register writebacks     15 (55.6% of all misses)
Register reads          148 bytes
Register writes         116 bytes

L1 cache hits           28 hits (13r + 15w)
L1 cache misses         14 misses (14r + 0w, 33.3%)
L1 cache writebacks     0 (0% of all misses)
L1 cache reads          108 bytes
L1 cache writes         60 bytes

Main memory reads       448 bytes
Main memory writes      0 bytes

Total clock cycles      490 clocks
Total                   28 hits, 14 misses, 490 cycles
Iterator type           23 hits, 12 misses, 419 cycles
Value type              4 hits, 1 misses, 37 cycles
Difference type         1 hits, 1 misses, 34 cycles
Pointer type            0 hits, 0 misses, 0 cycles
```

B.2.2 Trace File Optimizer

The long sequence of constructors and destructors in the input file is eliminated by the trace file optimizer. The following test compares the output with a precomputed answer file.

⟨Regression test for trace file optimizer. 197b⟩ ≡

```
sh_reg_opt () {
    ./test-readlog < test-reg.log | ./smatopt \
        | ./smatview > $TMPDIR/out || return 1
    cmp $TMPDIR/out ans-reg-opt.txt || return 1
    return 0
}
```

Used in part 192.

"ans-reg-opt.txt" 197c ≡

```
        cctori  4, (0xbffff718), (0xbffff734)
        cctori  4, (0xbffff714), (0xbffff724)
        subi    4, (0xbffff734), (0xbffff71c), (0xbffff6c0)
        igtd    4, (0xbffff6c0)
        deci    4, (0xbffff724)
        readi   4, (0xbffff724)
```

```
deci    4, (0xbffff734)
readi   4, (0xbffff734)
movv    4, (0x0804a68c), (0x0804a690)
decd    4, (0xbffff6c0)
igtd    4, (0xbffff6c0)
deci    4, (0xbffff724)
readi   4, (0xbffff724)
deci    4, (0xbffff734)
readi   4, (0xbffff734)
dtori   4, (0xbffff734)
movv    4, (0x0804a688), (0x0804a68c)
decd    4, (0xbffff6c0)
igtd    4, (0xbffff6c0)
dtord   4, (0xbffff6c0)
cctori  4, (0xbffff724), (0xbffff844)
dtori   4, (0xbffff724)
dtori   4, (0xbffff844)
```

B.2.3 Operation Counter

The output with `-v` option is generated and compared for the operation counter.

⟨Regression test for operation counter. 198a⟩ ≡

```
sh_reg_count () {
    ./test-readlog < test-reg.log | ./smatcount -v > $TMPDIR/out || return 1
    cmp $TMPDIR/out ans-reg-count.txt || return 1
    return 0
}
```

Used in part 192.

`"ans-reg-count.txt"` 198b ≡

```
Iterator:
  Ctor (copy)       16
  Ctor (base type)  3
  Dtor              19
  -                 1
  --                4

Value type:
  =                 2

Difference type:
  Ctor (default)    2
  Dtor              2
  =                 1
  >                 3
  --                2

Total:
  Ctor (default)    2
  Ctor (copy)       16
  Ctor (base type)  3
```

```
Dtor              21
=                 3
-                 1
>                 3
--                6
```

B.2.4 Trace File Viewer

Since `test-reg.log` is originally generated by `smatview`, processing the same trace instructions must generate the same output. In the following test, the output of `smatview` is compared with `test-reg.log`.

⟨Regression test for trace file viewer. 199a⟩ ≡

```
sh_reg_view () {
    ./test-readlog < test-reg.log | ./smatview -v > $TMPDIR/out || return 1
    cmp $TMPDIR/out test-reg.log || return 1
    return 0
}
```

Used in part 192.

B.2.5 Trace File Plotter

The trace file plotter is tested by comparing the output with option `-a`. The cache simulator and the helper command (`smatrange`) should work correctly to have this output, and so the test also includes testing for these two components.

⟨Regression test for trace file plotter. 199b⟩ ≡

```
sh_reg_plot () {
    ./test-readlog < test-reg.log | ./smatplot -a > $TMPDIR/out || return 1
    cmp $TMPDIR/out ans-reg-plot.txt || return 1
    return 0
}
```

Used in part 192.

"ans-reg-plot.txt" 199c ≡

```
The trace file contains the following address ranges.  To plot the
address range other than the first one, specify the first two digits
shown at the top of line to option -A, for example, "-A bf".

     08: 0x0804a688-0x0804a694
     bf: 0xbffff6b0-0xbffff848
```

B.2.6 Random Number Genrator

The random number generator is tested by `test-random.cpp`, which outputs the first 25 random numbers seeded by 4357. The output is compared with the answer in `ans-random.txt`. The answer can be obtained by the inventors' original implementation [15].

"test-random.cpp" 199d ≡

```
⟨Include common header files. 115a⟩
⟨Random number generator class. 187c⟩

int main()
{
  std::vector<unsigned> v(25);
  generate(v.begin(), v.end(), genrand_limit<unsigned>(4357U));
  for (std::vector<unsigned>::iterator i = v.begin(); i != v.end(); ++i)
    std::cout << *i << ((i - v.begin()) % 5 == 4 ? '\n' : ' ');
  return EXIT_SUCCESS;
}
```

⟨Regression test for random number generator. 200a⟩ ≡

```
sh_random () {
  ./test-random > $TMPDIR/out || return 1
  cmp $TMPDIR/out ans-random.txt || return 1
  return 0
}
```

Used in part 192.

"ans-random.txt" 200b ≡

```
4293858116 699692587 1213834231 4068197670 994957275
2082945813 4112332215 3196767107 2319469851 3178073856
3263933760 2828822719 1355653262 3454884641 2231974739
724013039 2042384640 1684975774 952055467 915646049
130288710 1432540170 833862964 4053232710 2490787578
```

B.3 Sorting Algorithms

Test script `test-sort.sh` sorts random sequences of random length with three sorting algorithms. Each algorithm sorts 20 sequences. Since random seeds are chosen randomly (from the current clock and process id), repeating this script will test over different random sequences.

"test-sort.sh" 200c ≡

```
#! /bin/sh

echo_n='echo -n'
if [ -x /usr/ucb/echo ]; then echo_n='/usr/ucb/echo -n'; fi

TMPDIR=${TMP-/tmp}/smat$$; rm -rf $TMPDIR; mkdir $TMPDIR || exit 1
trap 'rm -rf $TMPDIR' 0 1 2 15

failed () {
    echo;
    if [ -f $TMPDIR/out ]; then
        SEED=`cat $TMPDIR/out | awk '{print $2}'`
        echo "error found when executing \"./stlsort -s $SEED $algo $NELEM\""
        exit 1
    fi
    exit 0
}

for algo in early lomuto tiled; do
    $echo_n "Testing $algo"
    for N in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20; do
        NELEM=`./stlsort -S i 0 2>&1 >/dev/null | awk '{printf "%d\n", $2 % 5000}'`
        ./stlsort -S $algo $NELEM 2>$TMPDIR/out 1>/dev/null || failed
         $echo_n .
    done
    echo ' ok'
done

exit 0
```