Chapter 5

Engineering the Algorithms

Having presented the algorithms in a theoretical setting, it is now time to start looking at them from a practical perspective. In this chapter, we do a thorough investigation of what can be done to maximize performance of the algorithms presented in the previous chapter.

We will look into the design choices left open in the previous chapter and fill in the details while following a path leading to an implementation of high performance. In mergesort algorithms, in general we cannot avoid doing $N\log N$ comparisons and $4N/B$ I/Os. The mergesort algorithms we investigate also achieve this, so maximizing performance largely amounts to minimizing overhead. In this chapter, we will focus on possible approaches to minimizing overhead and evaluate these approaches through experimental analysis.

In the next chapter, we will compare our implementation to that of other algorithms. For that, it is important to ensure we have a reasonable efficient implementation [Joh01]. The results presented in this chapter provide knowledge of what combination of parameters and algorithmic details yield fast algorithms and data structures. This knowledge is then combined into optimized cache-oblivious sorting algorithms, the performance of which will be evaluated in the next chapter.

We begin this chapter with a section with general considerations on how we evaluate performance of algorithms. The remaining chapter is then dedicated to the implementation of the algorithms. In Section 5.2, we provide an overview of the structure of and the pieces that make up the implementation. In Section 5.3, we look into aspects of the funnel data structure. Of particular interest in this section is how to manage the data structure and how to implement a high performing fill algorithm. Section 5.3.4 focuses on the funnelsort algorithm. We provide a few optimizations and investigate good parameters for determining subproblem sizes and buffer sizes of the funnel. Section 5.3.4 provides a discussion on implementation details of the LOWSCOSA.

5.1 Measuring Performance

The focus of this and the next chapter will be on performance evaluation. Before presenting any benchmarks, we want to make clear exactly what it is we will be showing. The following is an overview of how the benchmarks, presented in both this and the next chapter, are conducted.

5.1.1 Programming Language

For portability and more importantly genericity, the algorithms are implemented in a generic high-level language. The language chosen is C++ [C++98]. The primary reason for this is that it allows for producing high performing code, while implementing generic algorithms. C++ was designed to have optimal run-time efficiency; depending on compiler quality, abstraction penalties are minimal. In addition, the accompanying library, the Standard Template Library (STL), contains a highly optimized sorting function, named `std::sort`, with which we may compare the algorithms developed here.

5.1.2 Benchmark Platforms

The underlying platforms for the benchmarks have been chosen based on diversity and availability. As discussed in Chapter 2, different processors and operating systems behave and perform different under certain circumstances. It is thus important to cover as many types of processors as possible, when arguing that the design choices made will be sound on not one but many different architectures. Our implementation would only be compelling to people using one particular platform, were we only able to show high performance on that type of platform.

We feel it is important to benchmark in real world scenarios and have thus chosen not to use simulation tools and to use the memory subsystem as is; we will not reduce the memory available to our algorithms artificially. The fact that modern computers now come with at least half a gigabyte RAM makes results obtained on machines artificially restricted to 32 or 64 MB of RAM of no practical relevance.

Hardware

Benchmarks made on three radically different architectures to ensure that we do not accidentally tune the algorithms for a specific architecture, thus defying one of the design goals of cache-oblivious algorithms. The architectures are Pentium 3, Pentium 4 and MIPS 10000 based. Their specifications can be found in Appendix B.

The Pentium 3 platform represents the traditional modern CISC. It has a pipelined, out-of-order, and super-scalar core. Its pipeline is as short (12 stages) as seem sensible when designing CISCs. The Pentium 2 and the AMD Athlon both have designs similar to the Pentium 3 and is thus expected to perform comparatively. The Pentium 4 computers represent a significant change in design philosophy. They signify a departure from the ideal of keeping pipelines short to minimize the cost of pipeline hazards and feature a 20-stage pipeline. This means that a branch miss-predict may waste as much as 20 clock cycles. The benefit of the long pipeline is very high clock rates. In applications such as sorting, where unpredictable conditional branches are commonplace, a 20-stage pipeline may well cause performance to degrade despite the high clock rate. To counteract the performance loss due to branch miss-predicts, the Pentium 4 employs the most sophisticated branch prediction logic of the three processors. Whether it will help it in the context of sorting remains to be seen.

To represent the RISC family of processors, we include a MIPS 10000 based computer. It has a traditional 6-stage pipeline and the simplicity of the core has made the inclusion of a large 1MB L2 cache possible. A notable feature of this processor is its ability to use an address space larger than 2^{32} bytes. Its word size is 64 bits both when used as address operation operands, and in the ALU. It is a relatively old processor and it is significantly slower than the Pentiums, so unfortunately due to time constraints, it could not participate in all benchmarks, though we have included it in all benchmarks presented in this chapter to guarantee that our implementation is not optimized for the Pentiums alone.

We feel that these three platforms are representative of most computers in use today in that most processors in use to day have a design resembling one of these three CPUs.

Software
On the software side, the Pentium computers are running the Linux operating system and the MIPS computers are running the IRIX operating system. The primary development platform, however, has been Windows. We feel that this has also contributed to diversifying the code.

The compilers used are listed in Appendix B. All executables used to generate benchmark results were compiled using the GNU Compiler Collection (GCC), which is the only one available on all platforms used. This was done to ensure that no algorithm had the benefit or detriment of good or poor code generation from the compiler, on any of the platforms. Say, for example, the MIPS Pro compiler is very good at generating code for funnelsort and not for `std::sort`. This would then put aspects of `std::sort` in a particular bad light, but only on the IRIX platform. We have found that for our experiments, the GCC generates code that is at least as good as any of the other compilers used generates. If anything, it generates very fast code for the quicksort implementation included in the standard library. The code generated by the MIPS Pro compiler was of equal quality, but both the Intel and Microsoft compilers generated significantly slower code.

5.1.3 Data Types

Sorting is used in a wide variety of applications. It is important that our benchmarks closely reflect as many applications as possible [Joh01]. Recent efforts in developing in particular cache-efficient sorting algorithms have opted to evaluate the performance sorting elements consisting only of a single integer key ([LL99], [XZK00], and [ACV$^+$00]). We feel, however, that sorting only integers is of limited applicability; some sort of information should be associated with the integers. At the very least, a pointer to some structure should accompany the integer. This may have an impact on algorithms that move elements a lot.

Inspired by the Datamation Benchmark, in turn inspired by sorting problems encountered in the database community, we have included a type of size 100 bytes [DB03]. The problem in the Datamation Benchmark originally consisted of sorting one million such records. This has since proved too easy and the total time became dominated by startup time. In response, the problem was changed to that of sorting as many records in one minute. This is known as the Minute Sort Benchmark. Since there are no restrictions on the platforms used when performing the benchmark, these benchmarks are largely a test of hardware and operating system I/O subsystems, rather than algorithm implementation. To allow contenders with limited finances to compete, the Penny Sort Benchmark was introduced. This benchmark is essentially the Minute Sort Benchmark with the result scaled by the price of the platform used in dollars. Algorithms competing in this benchmark are however still designed to be fast on one specific platform.

For our benchmarks, we choose to look at these three data types:

- Integers. This data type is simply a long.
- pairs. These represent key-value pairs and are implemented as class with data members of type long and void*. Their relative order is based on the value of the long.
- records. Represents database records or equivalents. They are implemented a class with a data member of type char[100]. Their relative order is determined by the strncmp function of the C standard library, such that the entire record is also the key.

Note that on the MIPS machine, both long and void* are 64-bit, while they are 32-bit on the Pentiums.

It would be infeasible to conduct the entire study in this chapter with several different data types. Thus in this chapter we will only use pairs. We then risk optimizing for relatively small data types. We will be weary of this when it comes to choosing between implementations that favor small elements, and then evaluate the performance of our implementation used on all three data types in the next chapter.

For the same reason, in this chapter, we limit the experiments to uniformly distributed random data. In addition, we do not want to optimize for any special case distribution, and since some results could be highly dependant on the distribution of elements and our algorithm implementation should not favor any distribution, we conduct the experiment on uniformly distributed pairs.

To generate random keys, we use the drand48 function available on both Linux and IRIX. In Windows, we use the rand function of the C standard library.

5.1.4 Performance Metrics

We may measure performance of our implementations by several ways. Here we bring an overview of the metrics used in this thesis.

Running Time

Of absolute primary concern is the total time spending solving the problem (sorting, merging, or other) measured on a physical clock. This measure is an indication of how long one would wait for the problem to be solved, which we believe to be of primary concern to the user of our algorithms.

As an alternative to the wall clock time, one may use the *CPU time*. That is the total time the algorithm is actually running on the processor. This measure is important if we were to estimate how much the processor would be occupied by the solving the problem. This could be of concern when other processes need access to the CPU. However, our implementation will not be designed with multiprocessing in mind. Furthermore, measuring the CPU time does not take the time the algorithm spends waiting for a page fault into account, because during this time, it is not scheduled on the CPU. Thus, we will not consider CPU time for our benchmarks.

The *wall clock time* is determined using the gettimeofday C library function. In Linux and IRIX, it appears to have a precision in the order of microseconds. In Windows, it appears to have a precision of milliseconds only, so we use the high-resolution performance counter available through the QueryPerformanceCounter API. This appears to have a precision of a couple of nanoseconds.

Page Faults
Albeit not of primary concern, the number of page faults incurred running the algorithm may provide us with important insights into the behavior of the total running time of the algorithm.

In the next chapter, when sorting large data sets, we will thus also present the number of page faults incurred by the algorithms. For this, we use the get_rusage system call in Linux and IRIX. This call provides both the number of minor and major page faults. Since only the major page faults have significant impact on performance, on that number will be reported. In Windows, we assign a job object to the process and use the QueryInformationJobObject API.

Cache and TLB misses
Aside from the number of page faults, the number of cache and TLB misses also influence performance. Performance Application Programming Interface (PAPI) allows for monitoring hardware counters [PAPI03]. Hardware counters can keep track of cache misses, TLB misses, and similar hardware events. PAPI is a cross-platform software library that provides access to these counters. Unfortunately, to make the Linux version work, a patch has to be applied to the kernel and we did not have that privilege for our test machines.

No patch was needed for the IRIX version, however, so we can use PAPI on the MIPS machine to show the cache behavior of the algorithms. We will be measuring the number of L2 cache misses (the PAPI_L2_DCM event) and TLB misses (PAPI_TLB_TL).

5.1.5 Validity

To provide valid and relevant experimental analysis, we should attempt to even out any disturbances in the results due to effects external to the algorithm, such as the scheduling of other processes running on the system. This may be achieved through more or less elaborate ways of averaging results from multiple runs of the same algorithm on the same problem.

In this thesis, however, we deal with such massive data sets, that individual benchmark runs take several minuets, sometimes even several hours. In comparison, any anomalies due to process scheduling or other operating system operations often cause no more than in the order of milliseconds of delays in the running time, so we do not expect this to influence our results greatly. Periodic scheduling of other processes may interfere significantly with the result of the benchmark. However, such interference is only normal in modern multiprocessing environments.

Primarily for time considerations, we choose to run each benchmark only once. This means that sudden “jumps” in measurements may be present in the results. However, we will attempt to run them with as many different parameters as feasible, to expose any systematic behavior of interest, and to expose what may be irregularities and what reflects actual algorithm performance.

5.1.6 Presenting the Results

A vast number of benchmarks have been run. Not all of them present new and relevant information. To avoid cluttering the discussion in the present chapter, the results of all benchmark are included in Appendix C and in electronic form as described in Appendix A, and only the ones that present relevant information will be included in the text. For the rest of the results, we thus refer to Appendix A. The results are presented in **Chart**s and their number in the appendix corresponds to their number in this chapter. The titles of the charts are consistently titled `<processor>, <cache size>/<RAM size>` with <…> substituted with values of the machine they were run.

The engineering effort carried out in this chapter is intended to compare different approaches to solving the same basic problems; they should not be viewed as performance evaluations of the individual implementations. Thus, when comparing an algorithm using method A with B and C, we prefer to show the performance of method B and C *relative* to A. This is done to emphasize what is the focus of this chapter, namely identifying approaches to implementing the algorithms that maximize performance. A more absolute performance analysis is carried out in the next chapter, when we have found the best way to implement the algorithms.

For each benchmark, we discuss exactly which part of the algorithm we will be analyzing. Each benchmark is accompanied by a discussion of the results, relating them to design choices made. Based on this we draw conclusion on what choices result in efficient implementation solving the problem.

5.1.7 Engineering Effort Evaluation

The engineering effort presented in this chapter seeks to find a good way to implement the algorithms. To do this, a series of questions need to be answered, such as

- How should the funnel be laid out in memory?
- How do we locate nodes and buffers in the funnel?
- How should we implement the merge functionality?
- What is a good value for z and how do we merge multiple streams efficiently?
- How do we reduce the overhead in the sorting algorithm?
- How do we sort at the base of the recursion?
- What are good functions for determining output buffer sizes and sizes of subproblems to recurse on in the sorting algorithm, i.e. what are good values for α and d?

All of these questions have multiple possible answers that will influence the performance of our implementation. The answer to one question does not necessarily influence the answer to another. Finding the answer to all questions that combine to yield an optimally performing implementation implies searching the entire space of possible combinations of answers. This space is so vast it would simply be infeasible. What we thus do in this chapter is examine one question at a time, first determining the best layout of the funnel, then the best way to locate nodes, and so on. We suspect that the result of this investigating the design options in this manner will bring us very close to an optimally performing implementation.

Since α and d influence both the funnel data structure and the funnelsort algorithm, we postpone the analysis of what constitute good values until the implementation details have been settled. Until then, we do not know what values will yield a fast sorting algorithm, so we choose by intuition. As a guideline, we choose large values when analyzing choices that influence merging, such as how to implement binary merging, and large values for choices that concern the tree structure, such as layout of the funnel. Since small values will yield small buffers and thus fewer elements merged per node we visit, we will expose aspects of the performance relating to operating the funnel. Conversely, large values will yield large buffers and likely more elements merged per node we visit, thus emphasizing the performance of the implementation of the merging algorithm. Regardless of the choice of values for the constants, for consistency we merge k streams of k^2 elements. This may not be the ideal for all values of the constants, but it is necessary to compare across different values of constants, since merging $k' < k$ streams is easier than merging k streams.

When measuring performance of the funnel (Section 5.3) we do not store the output of the funnel, we only check that the elements are output in sorted order. We do this to eliminate the overhead of writing and storing all the elements. Since this overhead is common for all implementations of the funnel, it does not influence a study comparing different implementations. As an added benefit, we automatically verify that the result of the algorithm is correct.

5.2 Implementation Structure

In this section, we give an overview of the pieces that make up the implementation, how they relate to each other, and what their roles are. We provide illustrative interfaces and defer the implementation details to the following sections.

5.2.1 Iterators

The concept of iterators is used extensively in the STL. An iterator has the functionality of a traditional pointer, in that it can be dereferenced to give the object it points to. As with pointers to elements in arrays, an iterator can also be incremented to point to the next element. However, any class with these properties is an iterator, so iterators serve as a generalization of the traditional pointer and its relationship with the array; they represent a general way of iterating through the elements of a data structure (*container* of elements) in essence, a way of flattening the structure.

Using iterators is the primary way of implementing generic algorithms in C++. The algorithm is designed without any knowledge of with what type of iterator it is used. By this token, we can implement any container of elements and have an algorithm work on it by implementing an iterator for it. In that sense, iterators are the glue that binds together algorithms and elements. By abstracting away the implementation of the iterator from both the containers and the algorithm, any algorithm can be made to work with any set of elements.

Some containers are not as easy to navigate as arrays. For instance, one cannot (at least in constant time) add, say, twenty to an iterator pointing to an element in a linked list and get an iterator pointing to the element twenty past the original. For this reason, the STL defines six categories of iterators, by what operations can be done on them in constant time:

- *Input iterator*. An iterator that can only be dereferenced, incremented and compared for equality.
- *Output iterator*. An iterator that can only be dereferenced and incremented. The result of a dereference must be assignable, that is, the expression `*x = t, ++x` must be valid for some object `t` if `x` is an output iterator. Equality comparisons are not requred by output iterators.
- *Forward iterator*. An iterator that can be dereferenced and incremented. Further, an iterator can be compared with other iterators to determine the relative positions of elements they point to. The result of a dereference should be a reference to an object, as opposed to output iterators that are allowed to return proxy object to which objects can be assigned.
- *Backward iterator*. Same as a forward iterator, except it can be decremented, not incremented.
- *Bidirectional iterator*. An iterator that is both a forward and a backward iterator.
- *Random access iterator*. An iterator with all the functionality of a traditional pointer; a distance between two elements can be computed and integer arithmetic can be done on it and iterators can be advanced a given distance. A random iterator is also a bidirectional iterator.

A goal when designing algorithms is to restrict the requirement of the iterators used, as much as possible.

We will be using the iterator abstraction throughout the implementation.

5.2.2 Streams

A stream is a sequence of elements. Its state consists of where to find the next element and how many remain. To this end, we simply represent streams as a pair of input iterators, one that points to the next element, and one that points one past the last element. A stream is constructed from two such iterators. The iterator pointing to the next element is returned by the member function `begin` and the iterator pointing the one past the last element is returned by `end`.

Most containers implemented in the STL, such as `std::vector`, `std::list`, and `std::set`, have member functions `begin` and `end` with the same semantics. Streams can thus be used as wrappers for any of these containers, as well as ordinary arrays. Streams may also be used to represent continuous (in the sense of the iterator) subsets of the containers, in essence slices of the flattened data structure.

5.2.3 Mergers

The STL provides a binary merge algorithm. It is declared as

```
template<class InIt1, class InIt2, class OutIt>
OutIt merge(InIt1 begin1, InIt1 end1, InIt2 begin2, InIt2 end2, OutIt dest);
```

where the `InIt` name indicates that it only requires `begin1`, `end1`, `begin2`, and `end2` to be input iterators and the `OutIt` indicates that `dest` should at least be an output iterator. The precondition is that there are a sorted set of elements between `begin1` and `end1`, and between `begin2` and `end2`. When the function returns, all elements of these sets have been written consecutively to `dest`, by dereferencing, assigning, and incrementing.

For merging in our implementation, a somewhat different approach is needed. First, we may want to merge more than two streams at a time. Secondly, a different set of semantics is needed. We need two kinds of mergers: the general merger and the basic merger. Their semantics differ and their interfaces reflect it, yet they are similar.

To accommodate for more than two input streams, both are implemented as function objects rather than functions. Function objects are simply objects that can be used as functions. As any object, they maintain a *state*. The input streams of a merger are then a part of the state of the function object, allowing us to add input streams to the merge function. With the `merge` interface of the STL, we are restricted by the number of arguments we can provide; however, there is no language restriction on the number of times, we can alter the state of a function object.

The basic merger has a compiletime set limit on the number of streams it can merge. Between zero and that limit of streams can be associated with it. Empty streams cannot be associated with a basic merger. Attempting to do so will have no effect. The semantics is essentially that of the Algorithm 4-4 on page 48; it merges as long as there is room in the output and elements in all associated streams. The associated streams are updated to reflect that elements have been extracted. To pass on information about which stream caused the merger to stop by becoming empty, we use a concept of tokens. When a stream is associated with the basic merger, a token is in turn associated with the stream and when invoked, the basic merger is given the output and what token to associate with the output. When done merging, it simply returns the token associated with the stream that caused it to stop. The interface looks like this:

```
template<int Order, class InStream, class Token>
class basic_merger
{
public:
    typedef Token token;
    void add_stream(InStream *s, token t);
    template<class FwIt>
    token operator()(FwIt& dest, FwIt dest_end, token outtoken);
};
```

Note that tokens can be anything from a simple integer indicating the number of the stream or a pointer to a complex user defined object. They are expected to be small, however. Note also that the first argument of the `operator()` is passed by reference, so it too can be updated. We require forward iterators because we need to be able to compare them to see if we have hit the end of the input. `Order` is the order of the `basic_merger`, also denoted z. If `add_stream` is called more than `Order` times with non-empty streams the state of the merger becomes undefined, as is the state after a merger has been invoked. As a simple illustration, here is what Algorithm 4-4 could look like using a basic merger:

```
template<class Node>
void fill(Node *n)
{
    basic_merger<2,typename Node::stream,Node*> merger;
    merger.add_stream(n->left_input, n->left_child);
    merger.add_stream(n->right_input, n->right_child);
    n = merger(n->out_begin, n->out_end, NULL);
    if( n )
        fill(n);
}
```

In this example, we use Node* as tokens. The right input buffer is associated with the right child and the same with left. For the output token, we simply use NULL, so if the merger returns non-null, we call recursively on the node returned.

General mergers will be used on a larger scale and should thus provide for an arbitrary number of input streams. The semantics differ from the basic merger either in that it merges until the output is full or until *all* input streams are empty. This eliminates the need for tokens. The interface looks like this:

```
template<class InStream, class Refiller, class Allocator>
class general_merger
{
public:
    general_merger(int order);
    general_merger(int order, const Allocator& a);
    static typename Allocator::size_type size_of(int order);
    void add_stream(const InStream& s);
    template<class OutIt>
    OutIt operator()(OutIt dest, OutIt dest_end);
    template<class OutIt>
    OutIt empty(OutIt dest, OutIt dest_end);
    void reset();
    void set_refiller(const Refiller& r);
    const Refiller& get_refiller();
    stream_iterator begin();
    stream_iterator end();
};
```

Among the main differences are that streams are now copied and maintained internally; there is no obligation to maintain associated streams. It is still possible to see how far the streams have advanced, by running through them using the stream_iterators returned by begin and end. It is possible to invoke the merger repeatedly. The empty member function template is there in anticipation of the merger storing elements internally after they have been read from the input and before they are written to the output. empty then provides a way of retrieving these elements, in no particular order. The reset member function sets the begin iterator of each stream to the end, essentially marking them all empty, and resets the internal state of the merger. This has the effect of destroying the merger and creating a new one with the same order. The get_refiller and set_refiller provides the interface for adding a refiller as described in Section 4.2.1. The rest of the interface has to do with memory management, to which we will return.

5.3 Funnel

We choose to implement the two-phase funnel, not because it is the easiest to implement, but because the simplicity it brings to prior funnel variants will not make it perform any worse and likely make it perform better, as discussed in Section 4.1.1. The funnel is a k-merger and when input streams are added, its implementation as such follows the interface of a general_merger.

5.3.1 Merge Tree

We will denote the combined funnel and input streams a *merge tree*. A merge tree consists of nodes and buffers. Buffers can contain any number of elements. These elements can only exist contiguously in the buffer, but they need not be located at the tail or the end of the buffer. When calling fill on a node, we need to identify where the elements are in the buffer, so we can resume from where we left off the last time we were filling its output buffer. A minimal description of the state of the merger is thus a pair of iterators for each buffer. Conceptually, a node may also contain pointers to where the buffers begin and end, as well as its parent and its children.

While a node need not maintain pointers for both its input and output buffers, it should do so for either the inputs or the output. Which one is not clear; a natural one to one relation ship exist between a node and its output, however, if we have no information about the state of the input buffers of a node, we have to go to the child nodes to get it, when we first start filling. This can reduce locality of reference, since nodes are not generally located near their children. We consider that an important aspect, so in our implementation we choose to let a node be responsible for the state of its input buffers. Figure 5-1 illustrates a node (with $z = 2$), the triangle, before fill returns from its right child. Also depicted are four pointers per buffer. Head (h) and tail (t) indicate the beginning and end of the contiguous section of elements in the buffers and begin (b) and end (e) indicate the beginning and end of the entire buffer.

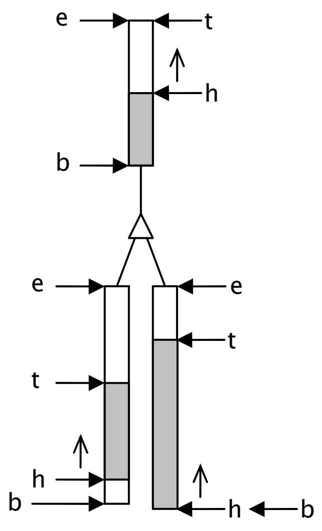

Figure 5-1. The pointers involved in a fill operation.

The situation in the figure is that prior to invoking its right child, fill called fill on the left child. That fill operation caused the subtree to become exhausted and thus the output was not completely filled. Some elements were then merged from the left input before the right got empty and fill was invoked on the right child. It, in turn, exhausted its subtree before returning.

Basic mergers are used to carry out the fill. Before invoking the basic merger, streams consisting of the head and tail of the input buffers are added to it, using add_stream. Then it is invoked with head and tail of the output as its arguments. This requires an invariant that *elements in input streams lie from head to tail and elements in the output stream lie between begin and head*. To maintain this invariant, we *flip* the buffers with a flip operation as we pass them when calling recursively on a child node or return from a recursive call. It consists of the double assignment (t = h, h = b). When returning from a fill, by induction, we know that the buffer we passed contain elements from b to h. After the flipping the buffer, we have h equal to the old b, the beginning of the elements, and t equal to h, the end of the elements, and thus a valid input buffer. Conversely, when calling recursively and passing an input buffer the flip operation turns the buffer into a valid output buffer.

As discussed, the gereral merger interface allows for arbitrary types of input streams, while the funnel maintains its own buffers. These buffers are elements allocated from the heap and the iterators used when merging them are simple pointers stored in the nodes. However, the input streams of the general merger cannot in general be represented by a pair of pointers. This presents two problems. First, we are wasting space storing pointers we are not using. Second, the leaves do not readily know from where to get the input. The first problem is easily solved by not actually allocating space for the leaf nodes.[1] We may then say that pointers the non-existing leaves in their parents are wasteful, however they are not, because we need some way to distinguish internal nodes from leaf nodes. The second problem is solved by storing the input streams in a separate array. When calling recursively, we keep track of the path we took and use it to locate the appropriate streams in the array. This in turn will give a minor overhead, however we consider it a small price to pay to get genericity.

5.3.2 Layout

As early as 1964, laying out trees in a particular way was known to be useful; careful layout of the tree used in the implementation of the heap, paved the way for the in-place heapsort [Wil64]. For our purpose, neither the analysis nor the correctness of the algorithms requires us to lay out the tree in any particular way. However, as we saw in Section 3.1.3, page 32, in case of binary search and as shown through experimental analysis in [BFJ02] and [LFN02], a well-chosen layout of the tree can yield a significant increase in performance.

As we have seen in Section 3.1.3, using the van Emde Boas layout for binary search trees gives an asymptotical reduction in the number of memory transfers incurred. That is not the case in when dealing with funnels. However, as argued in the proof of Theorem 4-3, fill does a number of tree operations, including recursive call invocations, flip operations, etc., proportional to the total number of comparisons and moves

[1] Our implementation does not current exploit this observation. It allocates a full tree, but never actually visits the leaf nodes.

performed, and thus visits a new node a significant number of times. Laying out nodes, so they are near each other, may then increase the algorithms overall locality significantly.

Aside from increasing locality, using controlled layout allows us to compute the position of the nodes relative to each other. This eliminates the need for accessing pointers to children stored in nodes and the potential data hazard in the pipeline. This latter aspect may well be as important as the first.

Implementation

In the implementation of the funnel, we use the STL concept of an allocator. The allocator is simply a class, through which algorithms can dynamically create objects. All containers in the STL provide a way for the user to supply an allocator. By abstracting away the allocation mechanism, the user of the containers is free to provide their own allocators and thereby control how objects are dynamically created. Such a mechanism is also provided through the new operator; however, the new operator is global and cannot be customized on a per algorithm or per container basis. If the user does not supply an allocator a default allocator, std::allocator, is used. std::allocator in turn uses the new operator.

The construction and destruction of funnels are done through a layout class template. Its interface is simple; the only reason for putting this functionality in a class is that we may parameterize funnels over different implementations.

```
template<class Navigator, class Splitter, class T, class Allocator>
class layout
{
public:
    typedef typename Navigator::node node;
    static node *do_layout(int order, Allocator& alloc);
    static void destroy(node *root, int order, Allocator& alloc);
};
```

The interface consists of two static member functions: do_layout and destroy. do_layout allocates and lays out a complete tree of a given order and returns a pointer to the root, and destroy tears down and deallocates the tree. It is parameterized by a navigator (see below), a splitter defining the size of the buffers, the type of elements in buffers and finally the allocator.

The Splitter plays the important role of deciding at what height we split the funnel when doing the van Emde Boas recursion (our implementation simply returns $\lfloor h/2 \rfloor$, with h being the height of the funnel being split) and what capacity the output buffer of a k-funnel should have. As such, it is used extensively throughout the implementation of both the funnel and funnelsort.

A given layout implementation ensures that allocating nodes and arrays of elements is done in a specific order. Achieving correct layout then relies on the allocator fulfilling memory requests in a contiguous manner. Our implementation includes an allocator, stack_allocator that does this by allocating a large chunk of memory once using new and move and return a pointer into this chunk, in response to memory requests. The amount of space needed for the initial allocation has to be determined at allocator construction time. For general mergers, the space needed is computed exactly by the static size_of member function.

Mixed and Pooled Layouts

Each layout comes in two variants. One that, as described below, allocates nodes and buffers intermixed, and one that allocates a pool of elements in which the buffers are placed. The first variant is called mixed, the latter pooled. The pooled layouts result in nodes being allocated together, much like the search trees of [BFJ02] followed by buffers laid out contiguously.

The van Emde Boas Layout

We have already discussed the van Emde Boas layout. In our implementation, it is realized by first recursively laying out the top tree then for each bottom tree from left to right, allocating its output buffer then recursively laying out the tree. Figure 5-2 shows a funnel laid out in the array below it. Note the `stack_allocator` allocates backwards.

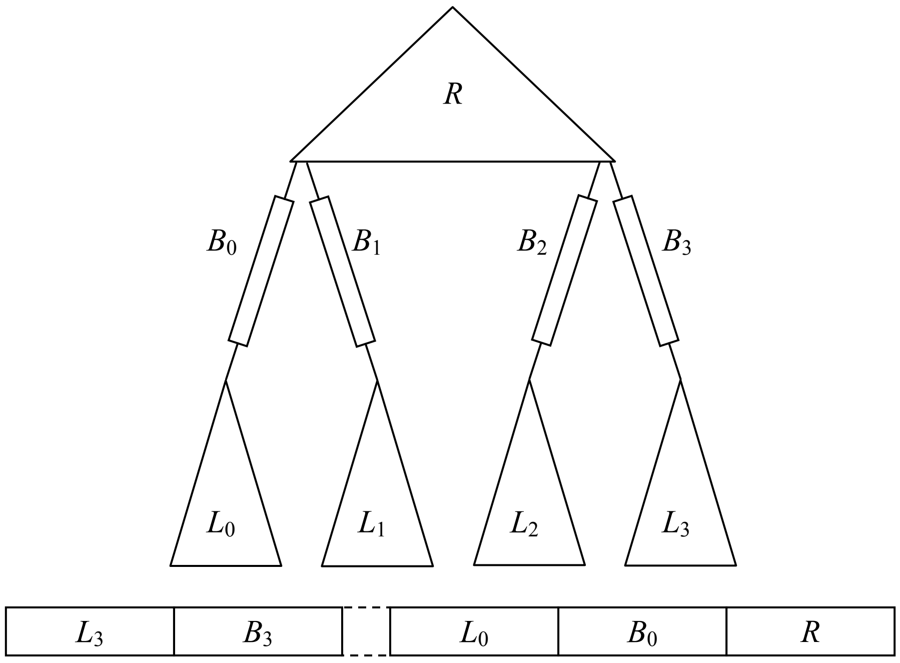

Figure 5-2. The van Emde Boas mixed layout.

Breadth-first Layout

The breadth first layout was the layout used in [Wil64] for implementing heaps. The nodes of the tree are allocated in the order they are visited by a left-to-right breadth first traversal of the tree. This is achieved by recursively allocating a funnel of height one smaller and then allocate the leaf nodes and their output buffers from left to right. The number by the nodes and buffers in Figure 5-4 show the order in which they are allocated. The numbers in Figure 5-3 shows the relative positions of nodes and buffers, when using pooled breadth-first layout.

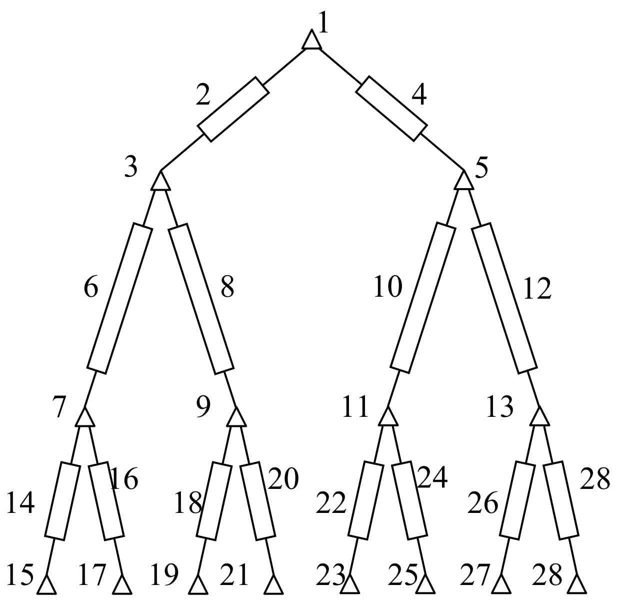

Figure 5-3. Breadth-first pooled layout.

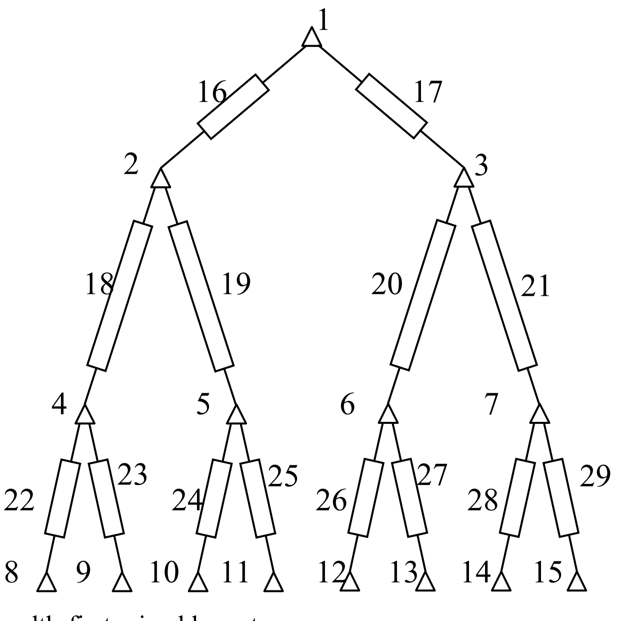

Figure 5-4. Breadth-first mixed layout.

Depth-first Layout

In the depth-first layout, the nodes and their output buffers are allocated in the order they are visited in a left-to-right depth first traversal of the tree. This is achieved by allocating the root and recursively allocate the subtrees below it from left to right. Before allocating the subtrees, their output buffer is allocated. The order can be seen in Figure 5-5.

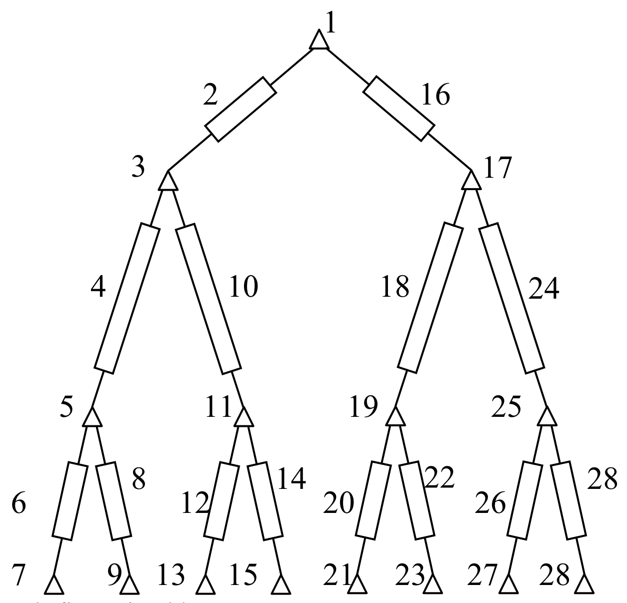

Figure 5-5. Depth-first mixed layout.

5.3.3 Navigation

A class known as a *navigator* is responsible for locating the parts of the funnel. Confining this functionality to a class allows us to experiment with different ways of traversing the funnel. Its interface is as follows:

```
template<class Node, class Splitter>
class navigator
{
public:
    typedef ... token;
    typedef ... bookmark;
    typedef Node node;
    typedef typename Node::stream buffer;
    token parent();
    token child(int i);
    navigator& operator+=(token t);
    navigator& next_dfs();
    template<class Functor>
    Functor enum_buffers(Functor f);
    level_iterator begin_level(int depth);
    level_iterator end_level(int depth)
    bookmark mark() const;
    bool operator==(bookmark m) const;
    bool operator!=(bookmark m) const;
    bool is_root() const;
    bool is_leaf() const;
    buffer *input();
    buffer *output();
};
```

Navigators resemble iterators in that they represent a single node in a data structure; however, they are capable of going in more directions than forward and backward. To support this in a generic way, we introduce a new token type, used to represent directions. Going to the parent is one direction and going to each of the children is another. The navigator is responsible for flipping buffers it passes.

The begin_level and end_level member functions provide a way to iterate through nodes on a specific level. level_iterators are bidirectional iterators dereferencing to pointers to nodes. This is used to tell the nodes where their children are placed during the construction and layout of the tree (hence the dependency of layout class on navigator classes). next_dfs moves the navigator to the next node in a search, where the nodes are enumerated in a way that when we visit a node, we have visited all nodes below it. This is used in the warm-up phase. enum_buffers provides for a way of enumerating buffers. This is used for resetting the merger for and emptying the buffers.

A simple implementation of a navigator is one that relies on the nodes to supply the address of their children and parents, but that requires the space in each node and the navigator to access these pointers. We define four categories of nodes, based on the information stored in them:

- *Simple node*. A node that stores nothing but the head and tail of its input streams.
- *Flip node*. A simple node that also stores the beginning and end of their input buffers. As the name implies, these nodes can flip their own input buffers.
- *Pointer node*. A simple node also storing the address of its children.
- *Pointer flip node*. A flip node also storing the address of its children.

Navigators that are more sophisticated will require less information of the nodes. Note that no category of nodes requires the node to store pointers to its parents. The reason for this is that the navigator can store pointers to nodes on the path to the current node, on a stack using much less space. On another stack, navigators keep information about output buffers on the path from the root to the current node. This is to avoid accessing data in the parent node.

A general pointer_flip_navigator has been implemented. It requires the funnel to be built using pointer flip nodes and all operations are implemented using the two mentioned stacks and the information stored in the nodes. Aside from the two stacks, a pointer to the current node is maintained and tokens are simply pointers to nodes.

If we choose to use the default allocator, we have no guarantee of where nodes and buffers are placed, so we are forced to use pointer flip nodes and the pointer_flip_navigator. When using stack_allocator and the mixed variants of the layouts, we know that the output buffer of a node lies immediately after the node itself. Using pointer nodes, the beginning of the i'th buffer can be obtained by adding the size of a node to the address of the i'th child, thus providing the information needed in a flip operation. When using the stack_allocator, we know where the nodes and buffers are placed. Our implementation includes navigators that exploit this for all pooled layouts and for the mixed variant of the van Emde Boas layout. For pooled layouts, we must use at least flip nodes, while the navigator for mixed van Emde Boas only requires simple nodes. We compute the address of the parents and children of each layout in the following way:

For the pooled breadth-first layout, the i'th child (counting from 0) of a node positioned at index j is located at index $(j\text{-}1)z+i+2$ and its parent is located at index $\lfloor(j\text{-}2)/z+1\rfloor$. For the pooled depth-first layout, we use a measure d that is the distance between the child nodes. When at the root, d is the number of nodes in a full tree of height one smaller than the funnel. The i'th child of a node positioned at index j is then located at address $j+id+1$. When going to a child we integer divide d with z. When going to the parent, we multiply and add one, and the index becomes j-id-1, where the node is the i'th child of the parent. i is computed as $(k+z\text{-}2) \bmod z$, with k being the breadth-first index. The result of these operations gives us the index of a node in the layout, with the root located at index 1. The final address is then computed by subtracting this index from the known location of the root. For this to work, we use perfect balanced trees as discussed below.

For the mixed van Emde Boas layout, we observe that when following the recursion until the node is the root of a bottom tree, it will be at an offset from the root of the top tree given by the size of the bottom trees and their output buffers times the number of bottom trees to the left of the child plus the size of the top tree. The number of bottom trees to the left is $k \bmod (n+1)$, with n being the number of nodes in the top tree and k being a breadth-first-like index that is updated with $kz+i$, when going to the i'th child and $\lfloor k/z \rfloor$ when going to a parent. The size of the bottom tree is kept is in a pre-computed table B as is the depth D of the root of the top and the number of nodes in the top tree N. These tables can be computed with one entry per level of the tree, since the recursion unfolds the same way for all nodes on the same level. The address of the nodes on the path from the root to the current node is kept in a table P, so address of the root of the top tree is $P[D[d]]$ with d being the depth of the current node. The last ingredients is a table T with the size of the top tree. The address of the i'th child then becomes

$$P[d] = P\big[D[d]\big] - \Big(\big(k \bmod \big(N[d]+1\big)\big)B[d] + T[d]\Big) \tag{5.1}$$

We know that $N[d] = (z^j\text{-}1)/(z\text{-}1)$ for some integer j. With $z = 2$, we can thus compute $k \bmod (N[d]+1)$ as $k \text{ and } N[d]$ which is likely to be faster. With the parameter z an integer template argument, we select the faster way through partial template specialization. For pooled layout, the modification lies in that buffer sizes should not be included in the offset from the root of the top tree. This in turn makes the table T identical to table N, so one of them can be discarded. For general z and $z = 2$ respectively, we get

$$P[d] = P\big[D[d]\big] - \Big(\big(k \bmod \big(T[d]+1\big)\big)B[d] + T[d]\Big) \tag{5.2}$$

$$P[d] = P\big[D[d]\big] - \big(\big(k \,\mathrm{and}\, T[d]\big)B[d] + T[d]\big) \tag{5.3}$$

The relation in (5.3) is in turn what was used in [BFJ02] for navigating optimal cache-oblivious binary search trees.

All implementations of navigators require storing multiple tables. This will make it, unlike iterators, infeasible to pass them as arguments to functions and in turn, unsuitable for use in recursive functions. Our implementation of fill is thus based on an unfolded recursion. Using the navigator abstraction and an unfolded recursion may increase the overall instruction count beyond what is possible using the simple recursive scheme of Algorithm 4-4. To clarify this, a recursive implementation was also made, however it requires the use of pointer flip nodes.

Non-power-of-*z*-funnels

Before proceeding with the experiments with layout and navigation, a point about unbalanced trees need to be made. The problem is that if we want to merge z^h+1 streams, we may have to lay out a z^{h+1}-funnel, which takes up a lot more space. It is indeed not necessary to lay out the entire z^{h+1}-funnel, since some of it will not be used. Figure 5-6 shows how to conserve space by not constructing an entire funnel.

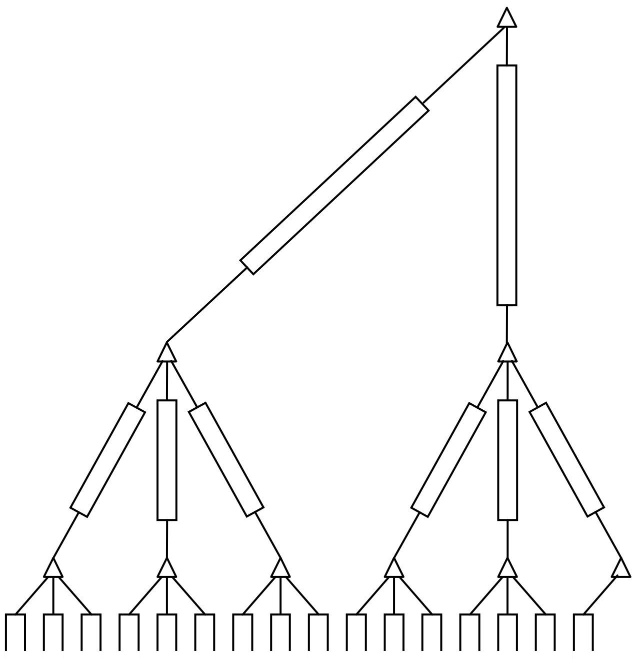

Figure 5-6. A merge tree of order 16 and $z = 3$.

However, not laying out a complete tree will foil all of the implicit navigation schemes described above, thus layout classes are asked to layout balanced trees when using implicit navigators. When using pooled layout, only the nodes of the balanced tree are laid out; the buffers are not. For the mixed van Emde Boas layout, however, we need to lay out a fully balanced funnel. When z becomes large, this may matter for certain values of k.

Test Results

For constructing funnels, we now have the choice between three layouts with two variants each, the choice of using the default allocator or using the `stack_allocator`, and the choice of using an implicit navigator or a general `pointer_navigator` or `pointer_flip_navigator`. This amounts to a total of $3{\cdot}2^3 = 24$ combinations, however, we cannot use implicit navigators with the default allocator, so a fourth of the combinations cannot be used. Furthermore, implicit navigators for mixed depth-first and mixed breadth-first have not been implemented. A total of $\frac{3}{4}{\cdot}24\text{-}2 = 16$ combinations remain. They are listed in Table 5-1.

Name	Layout	Allocator	Node	Navigator
pb_heap_mveb	Mixed van Emde Boas	std::allocator	Pointer flip	Pointer flip
pb_stack_mveb	Mixed van Emde Boas	stack_allocator	Pointer	Pointer
impl_mveb	Mixed van Emde Boas	stack_allocator	Simple	Implicit
pb_heap_veb	Pooled van Emde Boas	std::allocator	Pointer flip	Pointer flip
pb_stack_veb	Pooled van Emde Boas	stack_allocator	Pointer	Pointer
impl_veb	Pooled van Emde Boas	stack_allocator	Flip	Implicit
pb_heap_mbf	Mixed breadth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mbf	Mixed breadth-first	stack_allocator	Pointer	Pointer
pb_heap_bf	Pooled breadth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_bf	Pooled breadth-first	stack_allocator	Pointer	Pointer
impl_bf	Pooled breadth-first	stack_allocator	Flip	Implicit
pb_heap_mdf	Mixed depth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mdf	Mixed depth-first	stack_allocator	Pointer	Pointer
pb_heap_df	Pooled depth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mdf	Pooled depth-first	stack_allocator	Pointer	Pointer
impl_df	Pooled depth-first	stack_allocator	Flip	Implicit

Table 5-1. The possible combinations of layout and navigation available in our implementation

The 12 non-implicit combinations are also implemented with pointer flip nodes using a pure recursive fill function. Their names have “pb” exchanged with “rec”.

For the experiment, we have had each of the 24 funnels merge k streams of k^2 elements each using k-funnels with $z = 2$, $\alpha = 1$ and $d = 2$ and $k = 15, 25, \ldots 270$. The streams are formed by allocating an array of k^3 pseudorandom elements (pairs of long and void*) and sorting sections of size k^2 with std::sort. The funnel is constructed the streams attached, elements merged, and the merger reset $\lfloor 20{,}000{,}000/k^3 \rfloor$ times. The time measured is the time it takes to do this, save for the construction of the funnel. The output of the funnel is not stored anywhere; it is simply passed through an output iterator that checks whether the elements are sorted.

To avoid having to display 24 data series in the same chart, the result of the experiment is presented as a tournament with a group of implicits, a group of pointer navigators using default allocator, one using stack_allocator, one recursive using default allocator, and finally one using stack_allocator. From each group we choose a winner to appear in the final chart. The result is as follows. First, let us look at implicit navigation.

5.3.3 Navigation

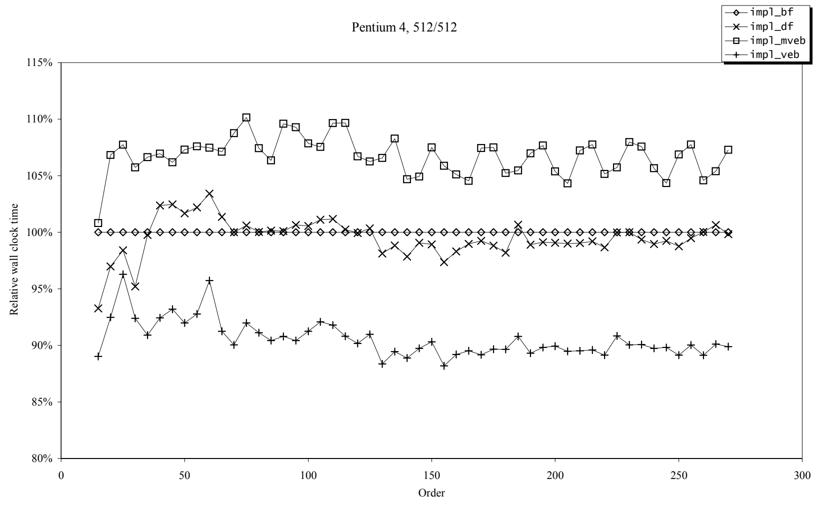

Chart C-1. Implicit layout on Pentium 4.

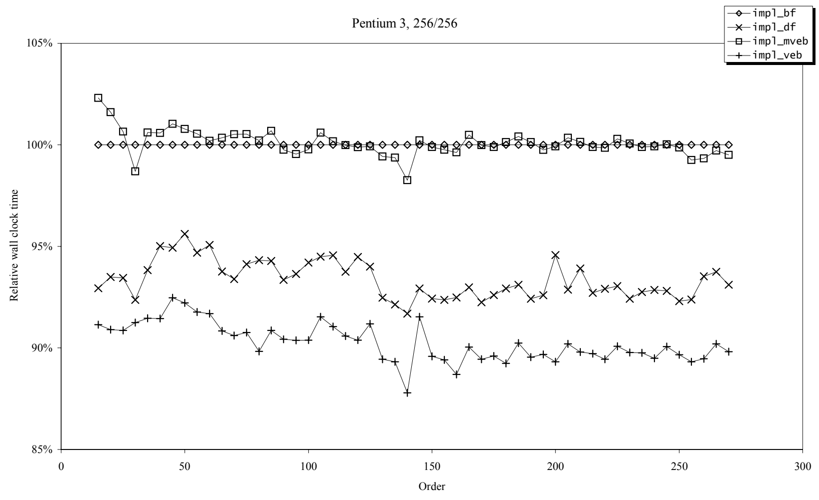

Chart C-2. Implicit layout on Pentium 3.

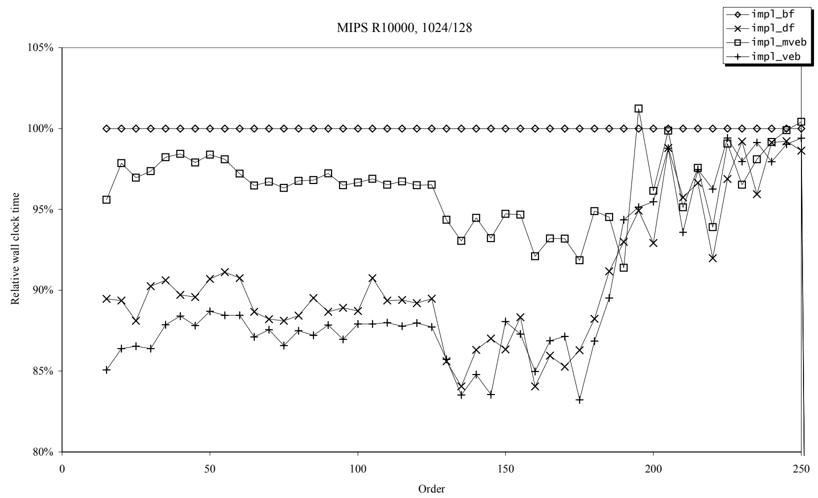

Chart C-3. Implicit layout on MIPS 10000.

The charts are normalized to breadth-first layout, which on both the Pentium 3 and the MIPS architectures are clearly the worst performers, even though it has the smallest instruction count. The reason for this must be effects in the memory system. The measurements made by PAPI on the MIPS, indicates that it is not as much the L2 cache rather the TLB that makes the difference:

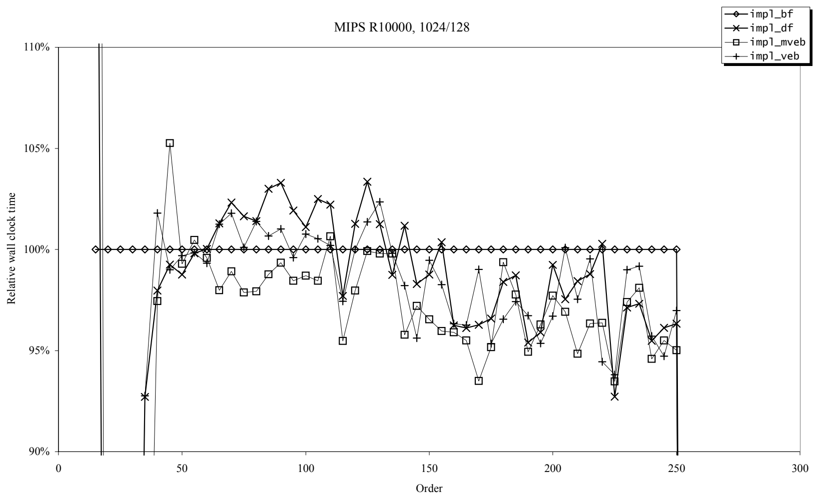

Chart C-4. Implicit layout on MIPS 10000, relative L2 cache misses.

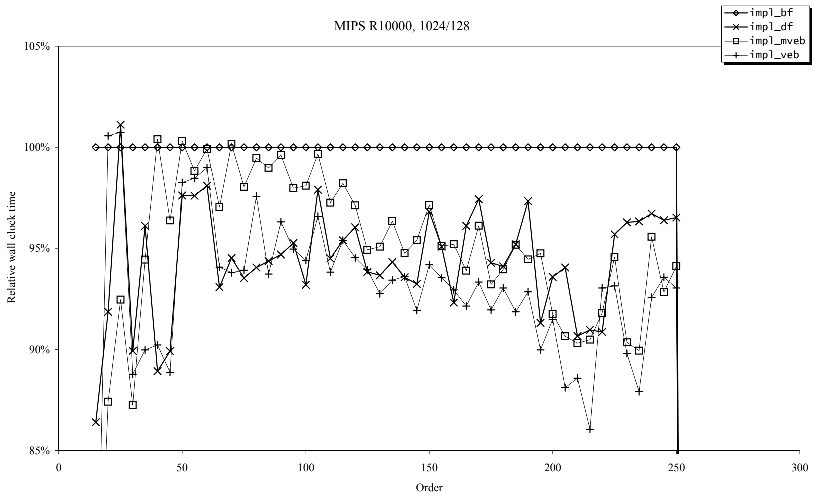

Chart C-5. Implicit layout on MIPS 10000, relative TLB misses.

The L2 cache incurs about the same number of misses regardless of layout, perhaps with the breadth-first incurring more misses; however, for large funnels (height at least six), we can see that some layouts are more "TLB friendly" than others are. TLB misses are handled in software on the MIPS so the performance penalty is greater. The breadth-first layout exhibits least locality as observed in [BFJ02]. The reason for this is that the parent is, except at the top, always located far from its children. All but the left most child are also placed far from the parent in depth-first layouts as well; however, for $z = 2$, that is half the children of a node, so it is not such a significant effect.

Navigating a pooled depth-first funnel requires us to update three variables when going from node to node. All three architectures are super-scalar, so these updates can occur in parallel and need thus not take any longer than just updating the breadth-first index.

That the mixed van Emde Boas layout does not suffer from being forced to lay out balanced trees is also noteworthy. The fact that the buffers are not touched during the fill phase of the merge contributes to this. The best combination seems to be the pooled van Emde Boas Layout on all architectures. It has good locality and the navigation is not as complex as the mixed van Emde Boas layout, which may have even higher locality. Hence, we choose the pooled van Emde Boas Layout as the winner of this group.

Let us now turn to the pointer-based navigators. When using the default allocator, none of the architectures seems to prefer any of the layouts particularly. As an example, the result from the Pentium 3 can be seen here:

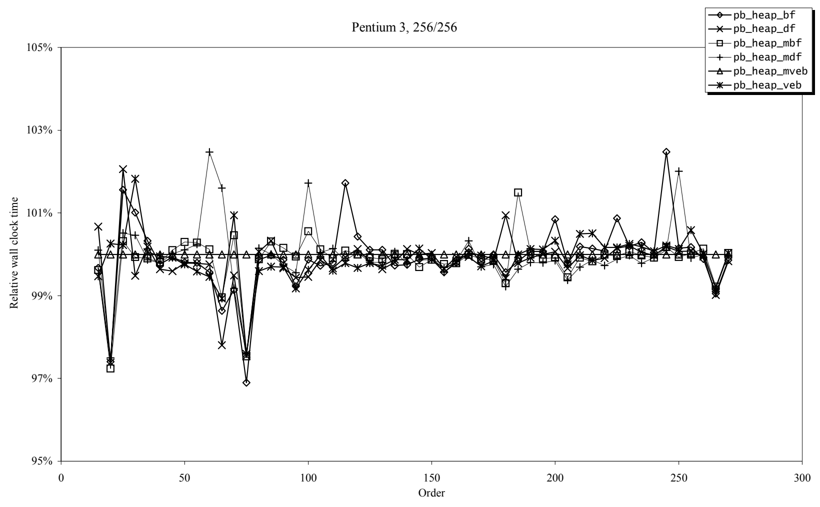

Chart C-7. Layout using std::allocator on Pentium 3.

We choose the depth-first layout for the final, because the Pentium 4 shows a slight (<1%) shift in its favor. The picture changes when using the stack_allocator:

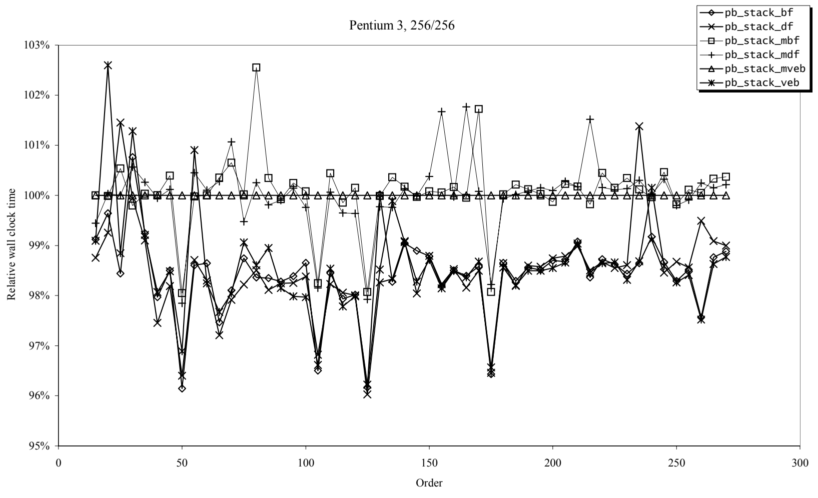

Chart C-12. Layout using stack_allocator on Pentium 3.

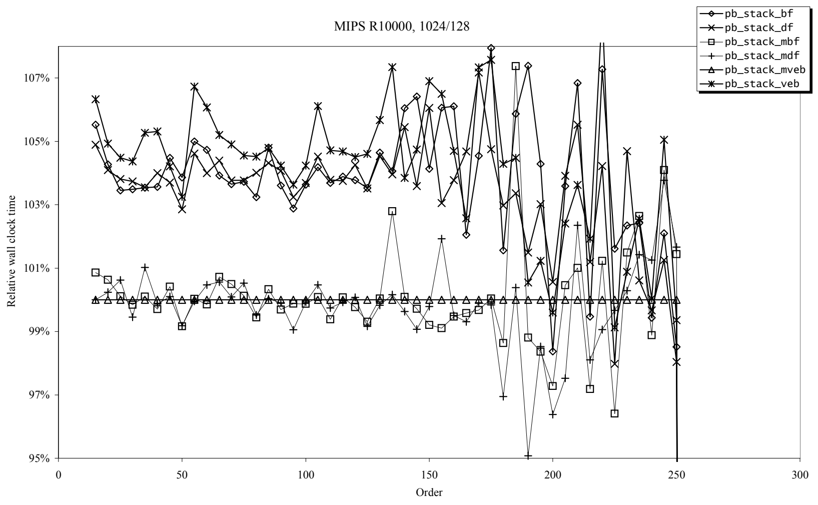

Chart C-13. Layout using stack_allocator on MIPS 10000.

The Pentium 3 (as well as the Pentium 4) clearly favors the pooled layouts, while the MIPS favors the mixed. There does not seem to be any special preference in the PAPI results. One explanation could lie in the lower level caches; when using pooled layouts, all the nodes can fit in L1 cache and if the associativity of the L1 cache is sufficiently high, they will likely stay there. However, if the cache has low associativity, it is likely that it cache lines with nodes on them will be evicted due to conflict misses during operations elsewhere in the funnel. In the latter case, it is probably best to store the nodes near the action. As it is, the Pentium 4 has a four-way set associative L1 cache while that of the MIPS is only two-way set associative. We choose in favor of the Pentiums and send the pooled depth-first layout to the final.

When using the recursive implementation of fill, the Pentium 4 again has no preferences with less than 2% difference in performance. The Pentium 3, however, seem to prefer the pooled layouts not only with the stack_allocator, but also when using the default allocator:

5.3.3 Navigation

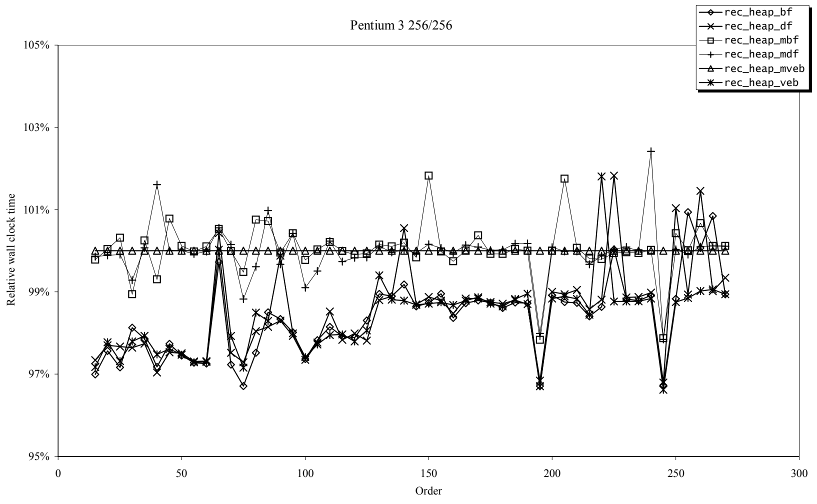

Chart C-17. Recursive fill, heap, Pentium 3.

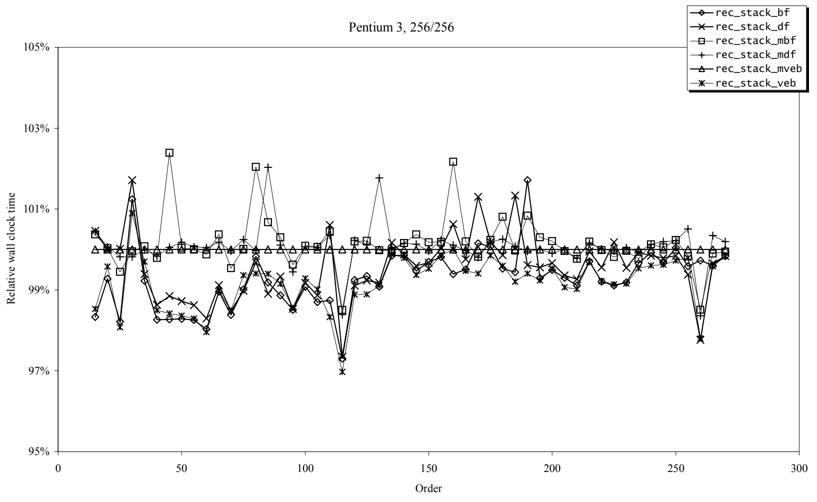

Chart C-22. Recursive fill, stack, Pentium 3.

The MIPS fortunately seem to have lost its interest in the mixed layouts:

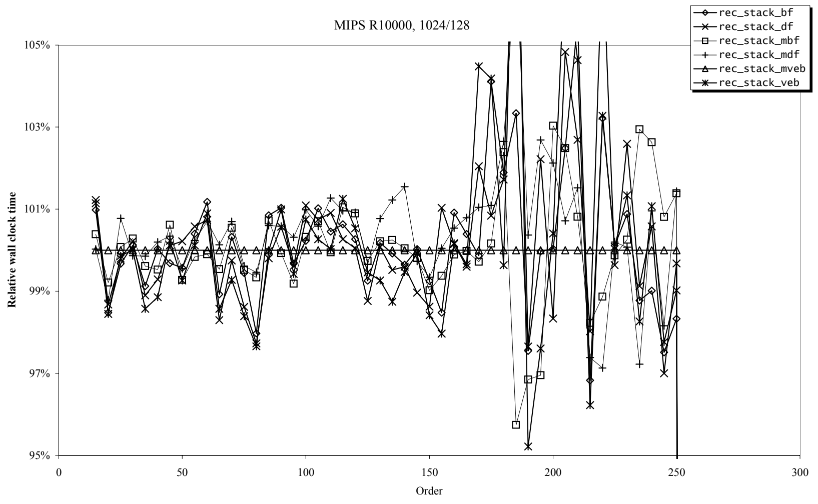

Chart C-23. Recursive fill, stack, MIPS 10000.

We choose the pooled van Emde Boas layouts for the final. Now that we have chosen a good layout from each of the groups, it is time to compare them to each other, now normalized to stack based layout with recursive navigation implementation.

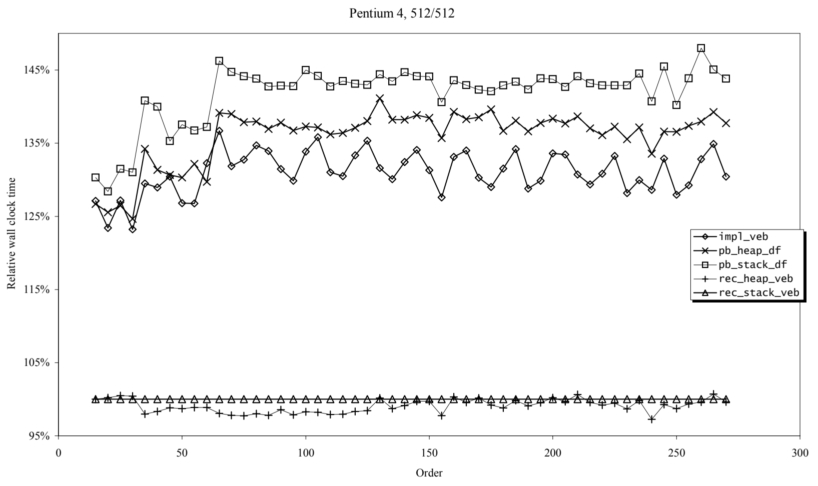

Chart C-26. Final, Pentium 4.

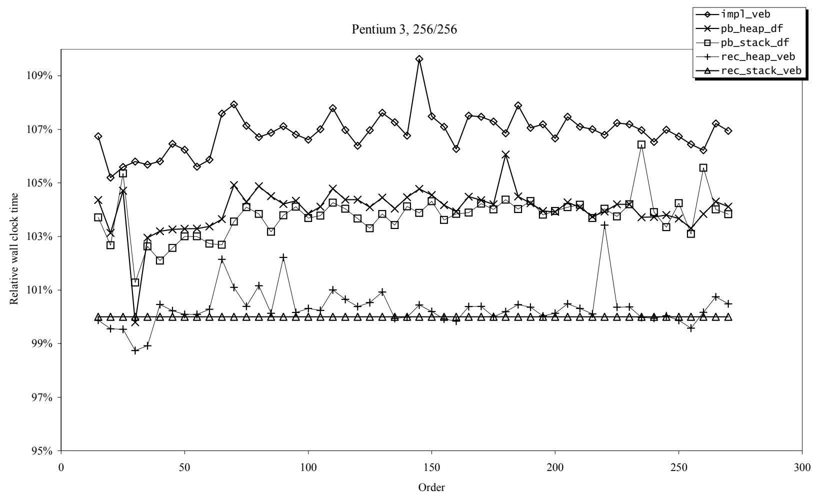

Chart C-27. Final, Pentium 3.

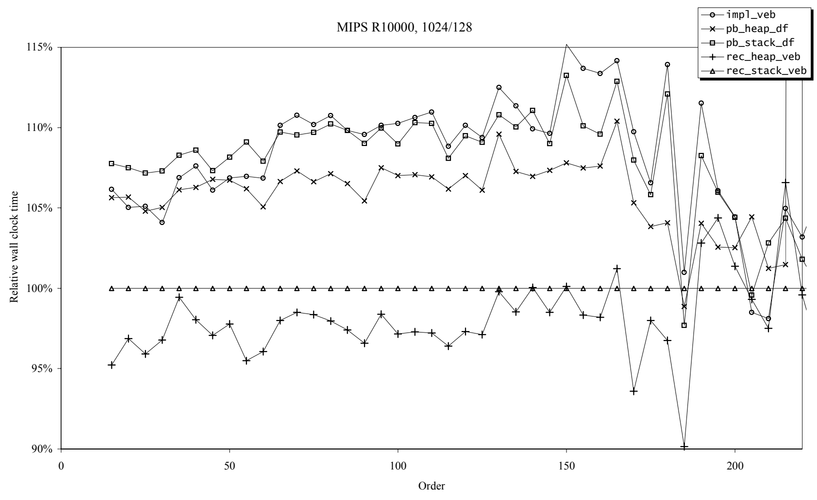

Chart C-28. Final, MIPS 10000.

We see that the Pentium 4 gains tremendously from using a recursive implementation of fill. This can be contributed to the number of transistors dedicated to avoiding control hazards. The Pentium 4 has a special return address stack, used by the fetch unit when returning from a function call. The stack contains the address of the next instruction to be fetched, which will then be ready immediately. When recursions are not too deep (as is the case here), this approach is far better than using conditional branches in the loops of the unrolled recursion. The effect is far from as pronounced on the Pentium 3 and the MIPS, where the effect is more likely due to overall lower instruction count.

We can also see that the implicit navigation is competitive only when on equal terms, comparing to the pointer-based navigators. When comparing implicit with the recursive algorithm, the simple recursive approach performs much better. Moreover, it turns out that whether using controlled layout through stack_allocator or leaving it to the heap allocator does not make a significant difference. Indeed, MIPS tend to favor memory delivered directly by the heap allocator. A reason for this is that the heap allocator is system specific and thus has detailed knowledge of the system parameters. This in turn allows it to allocate memory that is e.g. aligned on cache line boundaries.

Conclusion

In all, we conclude that the effects of the layout, and in turn the effects of cache, are dwarfed by other aspects. The key to achieving high performance in funnel implementations is through simplicity, rather than complex layouts. However, a good layout, such as depth-first or the van Emde Boas, seems to give a couple of percent on the performance scale.

5.3.4 Basic Mergers

By far the most time in a good funnel implementation should be spend merging elements. In our implementation, this means the basic mergers. Making sure they are performing optimally is thus important to achieving overall high performance.

The body of the fill algorithm (page 48) essentially implements the basic_merger application operator. When calling add_stream on a basic_merger, if the stream is not empty a counter, named active, is incremented and the stream and the associated token is stored. If it is empty, it is simply ignored. Upon invocation, the basic merger will check active. If it is zero, it returns immediately. If it is one, the contents of the only stream are copied to the output. If the input got empty, we return its token; otherwise, we return the output token. If active is greater than one, the actual merging begins. The implementation of the merging is put in a member function named invoke.

Binary Mergers

There are a couple of subtleties concerning the use of basic mergers, which we will now discuss. The streams added to the basic merger is a part of the object state and are as such accessed through the this pointer. In general, this will cause a slight overhead every time we access them, which is a couple per element merged. However, basic_mergers are stack objects of the fill function, so in fill, the this pointer is a compiletime computable constant offset from the stack pointer. Provided the operator() is inlined into the fill function, this will also merely be a constant offset from the stack pointer, thus the member variables will act as if they are normal stack variables and can as such be accessed without having to dereference the this pointer. Nonetheless, even though we insist that the compiler should inline the functions, the speed is increased if we make local copies of the member variables. This must be contributed to poor code generation on behalf of the compiler.

Another subtle issue that cannot be attributed to the compiler is the aliasing problem, that arises from passing the begin iterator of the output by reference. In such situations, the compiler cannot in general be certain that the iterator (which is often just a pointer) does not reference another iterator, in particular one of the input iterators. This in turn means it has to generate code that updates the referenced iterator and not just a local copy, each time the output iterator is updated. This turns writing elements to the output into a double dereferencing and incrementing the output begin iterator a load-increment-store instead of just an increment.

Since we know the output begin pointer is unique, and a reference to it does not reference any other pointer, we can solve this problem by explicitly making a local copy of it, use that for merging, and write it to the referenced iterator before returning. The complete merging code implementing Algorithm 4-4 looks like this:

```
template<class FwIt, class T, class Comp>
inline Token invoke(FwIt& b, FwIt e, Token outtoken, Comp& comp)
{
    typename Stream::pointer head[2] =
        { stream[0]->begin(), stream[1]->begin() };
    typename Stream::pointer tail[2] =
        { stream[0]->end(), stream[1]->end() };
    FwIt p = b;
    while( p != e )
    {
        if( comp(*head[0],*head[1]) )
        {
            *p = *head[0], ++head[0], ++p;
            if( head[0] == tail[0] )
            {
                outtoken = token[0];
                break;
            }
        }
        else
        {
            *p = *head[1], ++head[1], ++p;
            if( head[1] == tail[1] )
            {
                outtoken = token[1];
                break;
            }
        }
    }
    *stream[0] = Stream(head[0],stream[0]->end());
    *stream[1] = Stream(head[1],stream[1]->end());
    b = p;
    return outtoken;
}
```

This basic merger implementation is called simple_merger. We see that each time a single element is merged in the funnel at least three conditional branches have to be evaluated, namely the branch in the while loop, the branch on which head is smaller, and the branch on whether the input got empty. This could be a major overhead. However, due to sophisticated branch prediction techniques, predictable branches need not cause any performance penalty. The test that branches on which head element is smaller is inherently unpredictable; however, we expect the loop branch and the branch on empty input to be more predictable.

Consider a funnel with height power-of-two. No rounding is necessary when following the van Emde Boas recursion, so between every other level, there is a buffer of size αz^d. With $\alpha = 1$, $z = 2$, and $d = 3$, these buffers can contain eight elements. This means that at most eight elements can be merged before one of the two branches something different from the last time and cause a pipeline flush. This is not a lot. A quick fix would be to increase α, but this will not make the per merged element branches go away. We could also look at the problem more intelligently; since we know these branches will not fail (in the sense that they cause the loop to break) until enough elements have been moved to either make the output full or on of the inputs empty. We can see that the number of elements will be at least the minimum of the number of elements in the input streams and the space available in the output. The adapted loop then looks like this:

```
Diff min = e-p;
if( tail[0]-head[0] < min )
    min = tail[0]-head[0];
if( tail[1]-head[1] < min )
    min = tail[1]-head[1];
do
{
    assert( min );
    for( ; min; --min )
        if( comp(*head[0],*head[1]) )
            *p = *head[0], ++head[0], ++p;
        else
            *p = *head[1], ++head[1], ++p;
    min = e-p;
    if( tail[0]-head[0] < min )
        min = tail[0]-head[0];
    if( tail[1]-head[1] < min )
        min = tail[1]-head[1];
}
while( min );
```

which we denote the `two_merger`. The benefit of this approach is that we have eliminated one of the branches from the core merge loop, but at the price of having to compute the minimum now and again. However, the minimum can be computed entirely without using branches, namely by using conditional move instructions, so the overhead should be small. A worst-case scenario would be an input buffer consisting of a single large element, the other input of many small elements, and plenty of space in the output. The single element would cause the minimum to be one and thus the minimum to be recomputed every time one of the small elements is moved to the output.

To get a feel for how often such asymmetrical stream sizes occur, we counted the number of times the smallest input stream was a given fraction of the size of the largest stream. The resulting distribution can be seen in Figure 5-7. This was obtained through a full run of funnelsort on 0.7 million, 7 million, and 16.3 million uniformly distributed elements with $\alpha = 16$ and $d = 2.5$.

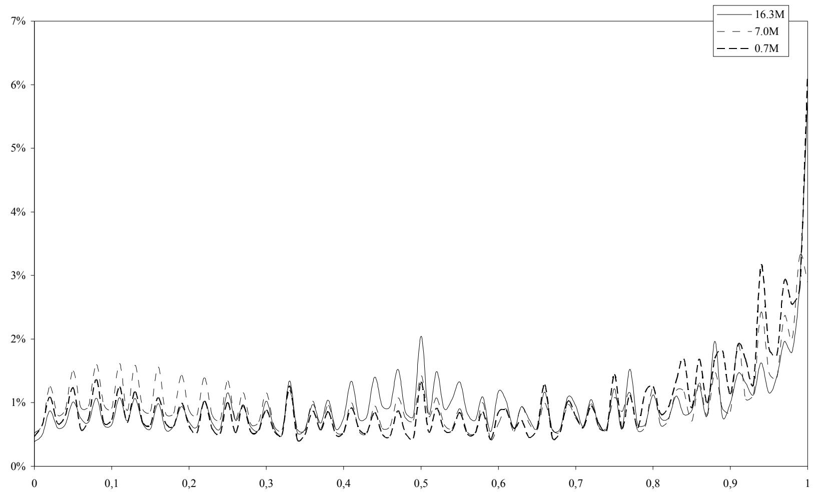

Figure 5-7. The distribution of relative sizes of input streams of basic mergers.

We can see that there is a slight tendency for the input streams to be of equal size; however, in general the small input stream can have a size any fraction of the size of the large input stream. Thus, we do not expect extremely small streams to be merged with very large streams with any significant frequency.

Still, perhaps we can gain further performance if we used a merge function that took into account the fact that sometimes we need to merge smaller streams with large streams and do that more efficiently. [Knu98] includes a description of an algorithm (Algorithm H, Section 5.3.2) originally due to F. K. Hwang and S. Lin that achieves near-optimal number of comparisons on inputs of this type. The adaptation of it to the basic merger setting is slightly tricky so we leave it out and refer to the accompanying source, where it is implemented as the hl_merger. It has a significant overhead but it may be that it is outweighed by the frequency of asymmetrical stream sizes. From a theoretical perspective, this merger can decrease the total number of comparisons performed in the funnel.

Realizing that the overhead of these more clever mergers may hamper their performance, we could also employ hybrid mergers; mergers that only use clever tricks under certain conditions. The hyb3 merger checks the relative size of the input streams. If the size of one stream is more than four times the size of the other, the hl_merger is used. Otherwise, the two_merger is used, but only as long as minimum is at least eight. From then on, it uses simple_merger. The hyb merger is a hybrid of only two_merger and simple_merger also with a cutoff at minimum of eight. The hyb0 only computes minimum once does one iteration of two_merger and proceeds with simple_merger. The reason this makes sense is that about half the times a basic merger is invoked, the minimum will be determined by the space available in the output, since on every other level of the funnel, the output buffer has a larger capacity than the input buffers. If that is the case, the minimum computed will be the exact number of elements moved during the entire merge. If it is not the case, we continue with simple_merger to minimize overhead.

We performed the same benchmark as with the analysis of layout and navigation. Here we used the rec_heap_mveb and realizing that the choice of constants α and d can be significant we ran the test for $(\alpha,d) = (1.0,\ 3.0)$, $(4.0,\ 2.5)$, and $(16.0,\ 1.5)$. With these parameters, the smallest buffers are of size 8, 23, and 45, respectively. The results for $(\alpha,d) = (1.0,\ 3.0)$ can be seen here:

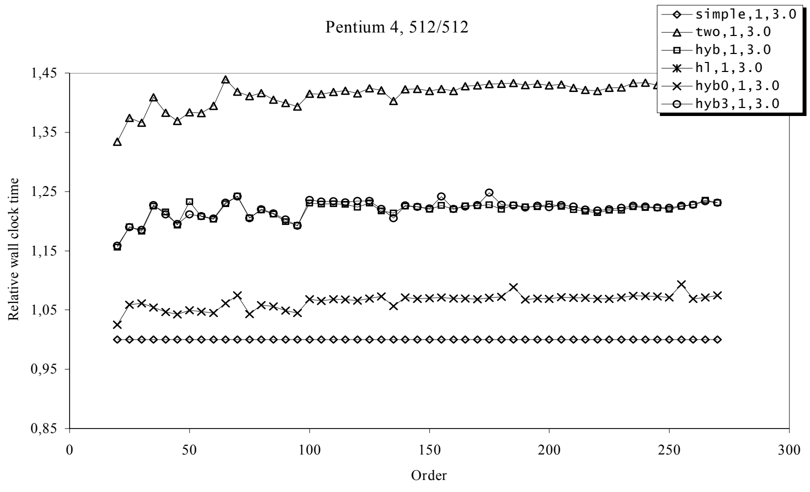

Chart C-31. Basic mergers, $(\alpha,d) = (1{,}3)$, Pentium 4.

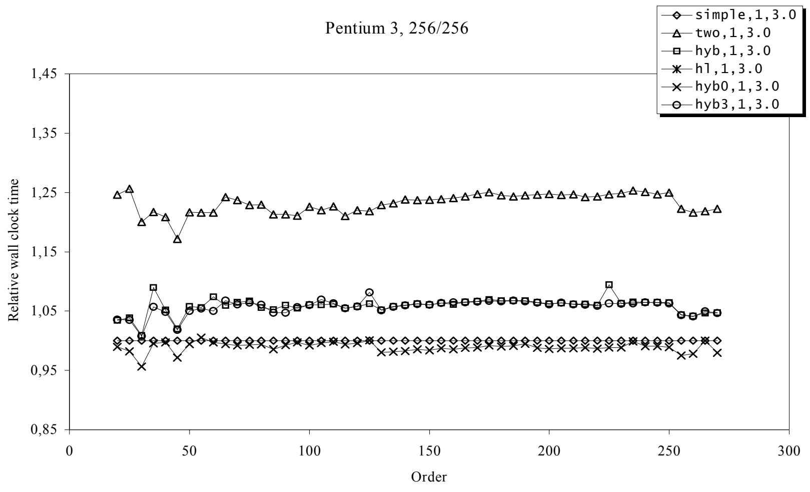

Chart C-32. Basic mergers, $(\alpha,d) = (1{,}3)$, Pentium 3.

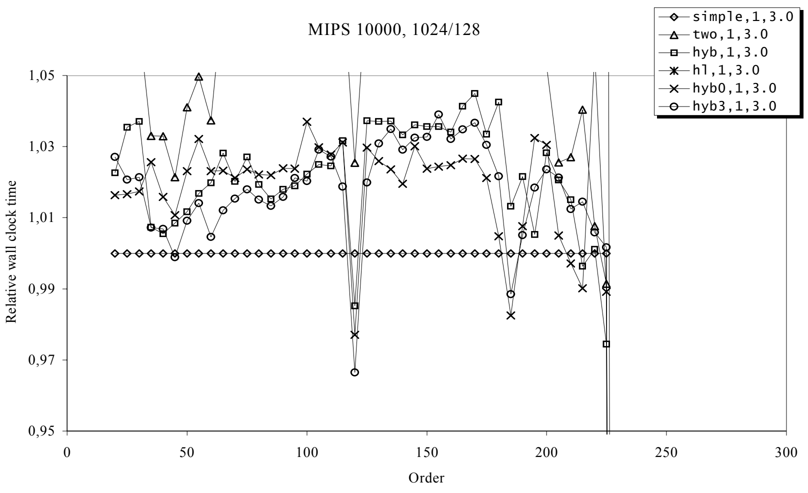

Chart C-28. Basic mergers, $(\alpha,d) = (1,3)$, MIPS 10000.

The MIPS produces a lot of noise (note the scale); however, it is clear that with a minimum buffer size of eight, not enough elements are merged per basic merger invocation to warrant the use of any method that has an overhead associated with it on any of the architectures. The overhead of the hl_merger made the entire merge take at least three times longer and is thus far off scale. In addition, the difference in all benchmarks between the hyb_merger and the hyb3_merger is minimal, implying that cases where the smaller stream has less than a forth the number of elements of the large stream are rare and that in those cases using the hl_merger neither improves nor worsens the performance.

Going from $(\alpha,d) = (1.0,\ 3.0)$ to $(4.0,\ 2.5)$ the two_merger does not gain much, but the hybrids start to get competitive at least on Pentium 3 and MIPS. Going to $(16.0,\ 1.5)$, the Pentium 4 finally seems to benefit from the tighter inner loops of the hyb0_merger; however, using the pure two_merger still incurs a 35% running time increase.

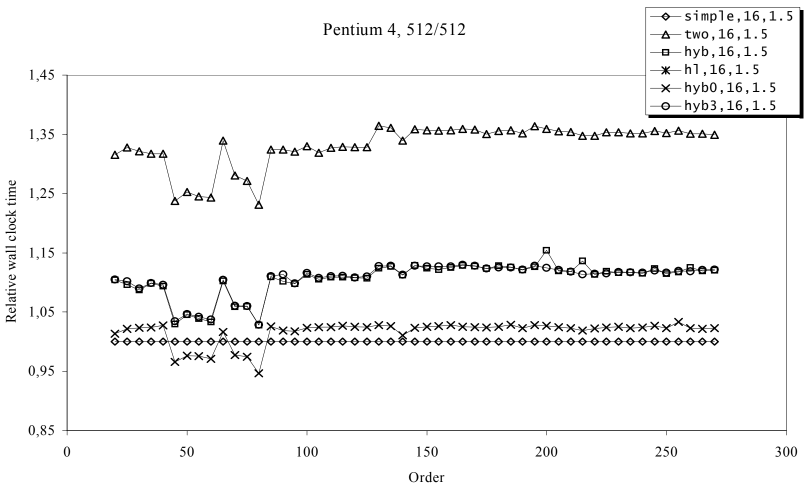

Chart C-37. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 4.

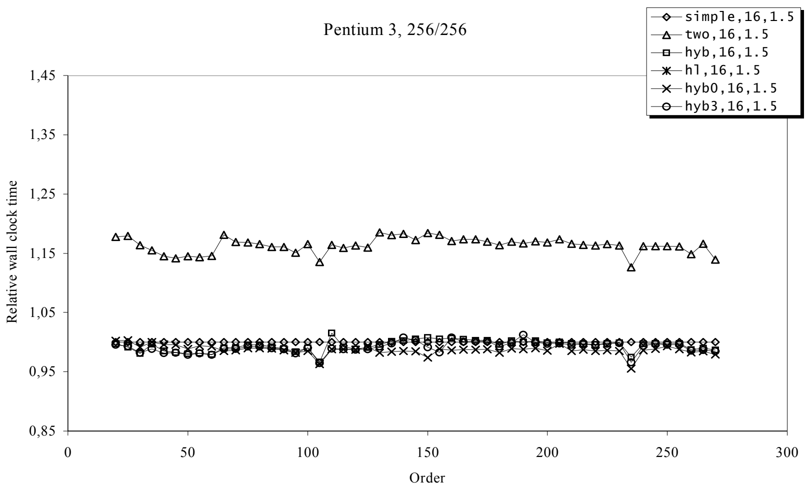

Chart C-38. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 3.

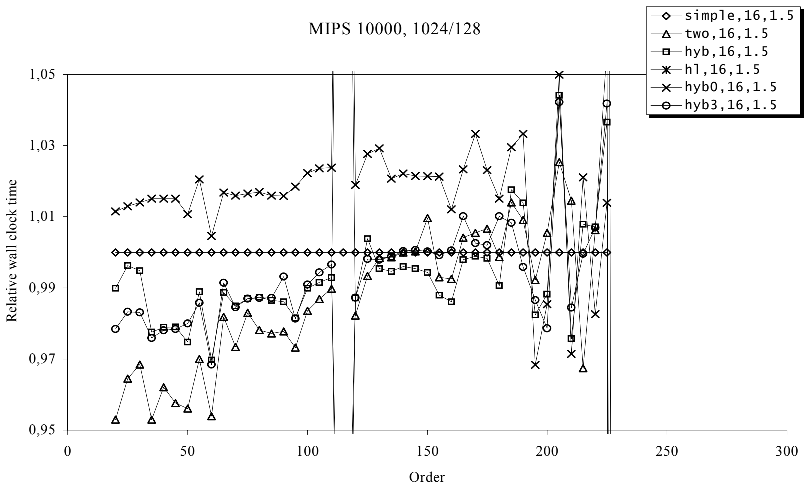

Chart C-39. Basic mergers, $(\alpha,d) = (16,1.5)$, MIPS 10000.

We conclude that the branch prediction unit of the Pentium 4 is very effective and that using any explicit intelligence to aid in avoiding slightly unpredictable branches will only hurt performance. The MIPS only has a six-stage pipeline, so any unpredictability in branches will not influence performance much. Still, it benefits from the tighter loop. Handling cases where the output buffer sets the limit on the number of elements merged in a special tight loop will improve average performance 3-5% percent on Pentium 3 and MIPS. Any more overhead and performance will get poorer.

Higher Order Mergers

Having established good ways to merge two streams, we are interested in extending the capability to merging of higher order and establish how that affects performance. [ACV$^+$00] provides compelling evidence that merging with low orders can significantly increase performance; instead of using a traditional multiway mergesort, they restrict the sort to only use mergers of order no higher than some constant, instead of allowing the order to grow to M/B. This in turn will give them more passes, but the benefit of using small mergers outweighs that cost. We are looking at the other end of the scale, comparing binary merge to higher order; never the less, we should see that performance increases, when increasing the order to a certain number.

There can be at least two reasons for any increase in performance. One is, as [ACV⁺00] argues, that merging e.g. four or six streams can be done with all stream pointers stored in registers. The same is the case with merging two streams but with more streams, the registers are better utilized. This will not be the case on the Pentium machines, where only eight general purpose registers are available, barely enough to hold the pointers involved in merging two streams, however spilling the pointers to fast L1 cache may not be a performance problem. The second reason that performance would benefit is that we skip potentially expensive tree navigation operations; using four-way basic mergers is like using two-way basic mergers, except the edges containing the smallest buffers have collapsed.

On the other hand, leaving two-way basic mergers also means leaving a compiletime knowledge of how many input streams a basic merger can have; using z-way basic mergers means we have to be able to handle merging of any number between two and z streams, since any of the z input streams may have become exhausted.

Let us examine the ways in which we can implement z-way basic mergers. Recall that a basic_merger implementation keeps a member variable active counting the number of non-empty input streams. A simple for-loop based extension of the simple_merger could then look like this:

```
template<class FwIt, class T, class Comp>
inline Token invoke(FwIt& b, FwIt e, Token outtoken, Comp& comp)
{
    struct ht { typename Stream::pointer h, t; } s[order];
    for( int i=0; i!=active; ++i )
        s[i].h = stream[i]->begin(), s[i].t = stream[i]->end();
    FwIt p = b;
    assert( active > 1 );
    while( p != e )
    {
        for( ht *m=s, *q=s+1; q!=s+active; ++q )
            if( comp(*q->h,*m->h) )
                m = q;
        *p = *m->h, ++(m->h), ++p;
        if( m->h == m->h ) // the input became empty
        {
            outtoken = token[m-s];
            break;
        }
    }
    for( int i=0; i!=active; ++i )
        stream[i]->begin() = s[i].h, stream[i]->end() = s[i].t;
    b = p;
    return outtoken;
}
```

Pairs of head and tail pointers are kept in an array on the stack. A for-loop finds the pair m with the head pointing the smallest element. The element is moved to the output and the head pointer incremented. This implementation is called simple_for_merger. An implementation that, like the two_merger, uses a tight loop merging a minimum number of elements before recomputing the minimum is also implemented as the for_merger.

In the implementation above, each time we compare to find the smallest head, we do a double dereference. This can be alleviated by maintaining the value of the head along with the pair of pointers of that stream. However, this in turn means moving all elements to a temporary local variable, doubling the total number of elements moves. This could potentially be expensive when merging larger elements. for_val_merger and simple_for_val_merger has been implemented that are like for_merger and simple_for_merger, except they maintain a local copy of the head element.

A problem with all of these solutions is the overhead of the for-loop. While the stream with the smallest head can be isolated using conditional moves, neither the compiler nor the processor at runtime have any idea of how many streams we need to consider. Instead, we could do a switch on active out side the loop. In the switch, we now know what active is. The implementation simple_comp_merger uses this information as a template argument that then picks out the smallest head. The templates are illustrated here:

```
template<int active>
inline bool move_min(It *head, It *tail, Token *tokens, It out)
{
    if( *head[0] < *head[active-1] )
        return move_min<active-1>(head,tail,token);
    else
        return move_min<active-1>(head+1,tail+1,token+1);
}
template<>
inline bool move_min<2>(It *head, It *tail, Token *tokens, It out)
{
    if( *head[0] < *head[1] )
    {
        *out = *head[0], ++head[0];
        return head[0] == tail[0];
    }
    else
    {
        *out = *head[0], ++head[0];
        return head[0] == tail[0];
    }
}
```

and used like this:

```
switch( active )
{
    case 2:
        ...
    case 4:
        for( p!=e; ++p )
            if( move_min<4>(head,tail,token,p) )
                break;
        break;
    case 5:
        ...
}
```

In move_min<k> a comparison is made to see which of the first or the k-1st stream contains the larger element. That stream cannot contain the smallest element, so move_min<k> calls recursively on all but that particular stream. Provided the compiler inlines the entire recursion, this implementation will do exactly z comparisons per element merged in a z-way basic merger, and when they are done we know exactly where on the stack the pointer to the smallest element is. The problem is that the code is exponential in size and that none of the outcomes of the z comparisons are predictable nor can they be replaced by conditional moves. A version using minimum determination like the two_merger has also been implemented and is called comp_merger

Instead of using sequential comparisons, we can also use optimal data structures such as heaps. The looser_merger is based on a looser tree that only does $\log z$ comparisons and moves [Knu98]. It too has been implemented using templates; when the looser has been located, we switch on the number of its associated stream. In this switch, we call a function specialized for that particular stream which then updates the looser tree.

For the evaluation of the different implementations, we use them in a 120-funnel with $(\alpha,d) = (16.0,\ 2.0)$ to merge 1,728,000 elements. We do this eight times and measure the total time on a physical clock. For reference, we also include the binary basic mergers from the previous section. Here is the result:

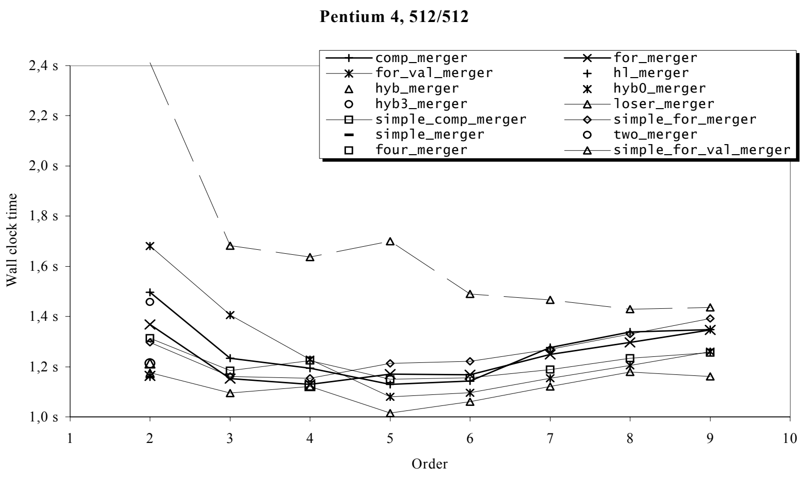

Chart C-40. Basic mergers, Pentium 4.

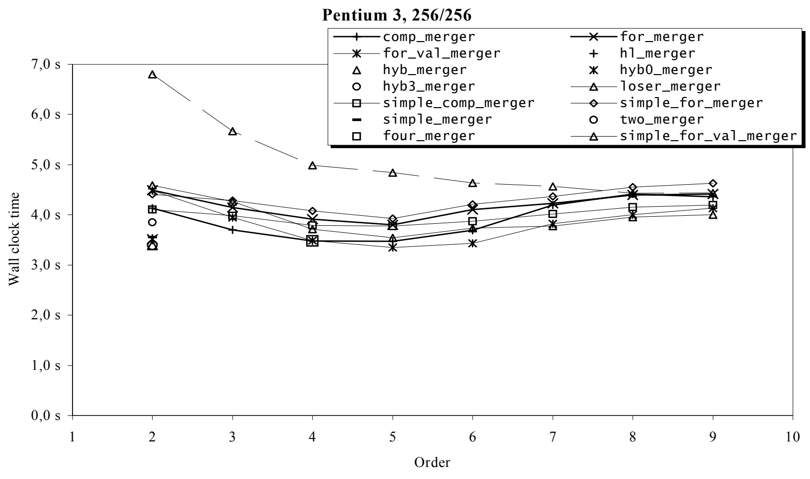

Chart C-41. Basic mergers, Pentium 3.

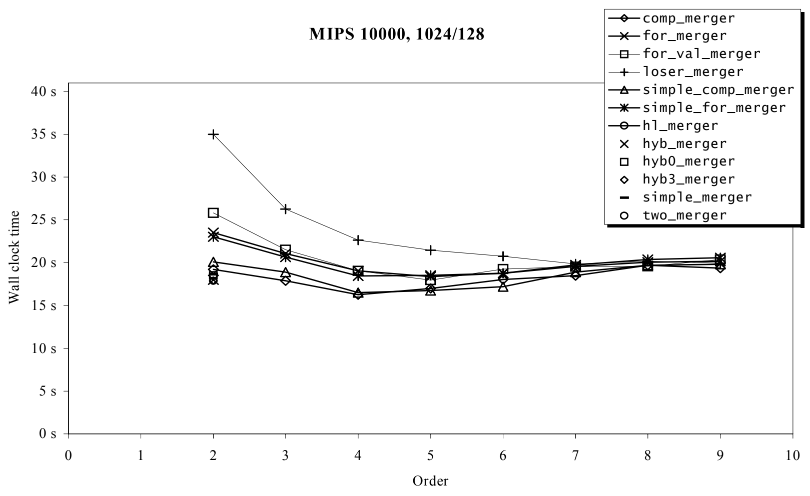

Chart C-42. Basic mergers, MIPS 10000.

Realizing compilers are not always eager to inline functions to the extend we need in the comp_merger and simple_comp_merger, we manually inlined a simple_comp_merger with $z = 4$. This is the four_merger. Since there is no discernible difference in performance between it and the simple_comp_merger, we conclude that the compiler does complete the inlining, at least for $z = 4$.

The charts clearly show there is performance to be gained from increasing z; however, at some point the performance begins to deteriorate. The optimum value seems to be either 4 or 5. The overhead of using the optimal loser_merger is too great to use on these orders. For sufficiently large z, determining the minimum number of elements merged and merging them in a tight loop is faster than the naïve approach. This could indicate that it is the small buffers in the tree that largely contributes to the overhead of this approach.

To some extent on MIPS but in particular on the Pentiums, it is hard to beat the handcrafted binary mergers. The reason for this is most likely the increased overhead of making local copies of streams and iterating through them. Why the generalized simple_comp_merger takes such a performance hit when $z = 2$, compared to the simple_merger is not clear; the complier should inline the move_min<2> function and thus get a merge function identical to that of simple_comp_merger. As with the previous experiments, here too we must conclude that the simplest implementations are very good candidates to being the highest performer.

5.4 Funnelsort

Now that we have a high performing funnel in place, we will look into applying it in the algorithm for which it was designed. The algorithm as it is described in Algorithm 4-5, page 55, does not leave as many options open to the implementation. The analysis requires it to be recursive so we cannot experiment with the structure of the algorithm. However, there is a base case for which we need to decide how to sort and there is the matter of how the output of the merging should be handled. Finally, there is the matter of the values of α and d. We will first look at how to handle the output and memory management, introduce two final optimizations, then look at buffer sizes, and finally settle on the base sorting algorithm.

5.4.1 Workspace Recycling

In multiway mergesort (Algorithm 3-3, page 39), runs were merged using complete scans; the entire file of elements were read in and a file containing the merged runs was written to disk. The subproblems are solved in a level-wise order, allowing the reading and writing of all elements from and to disk at each level. The reason this is optimal is that the number of levels in the recursion exactly fits with what is possible with the block and memory sizes, namely $\mathcal{O}(\log_{M/B}(N/B))$.

The number of recursions in funnelsort will be higher ($\mathcal{O}(\log\log N)$) so we cannot merge by scanning all elements. We have to follow the recursion and store the output of one recursive call before we recurse to the bottom of the next problem and we cannot simply keep a file for each level in the recursion. One simple solution would be to, for each recursive call, allocate a buffer the size of the subproblems in that call, around $\alpha^{1/d}N^{d-1/d}$ elements. Each recursive sort would then put their output into that buffer and when the recursive sort was done, the elements of the buffer would be copied back into the original array. In this approach, providing the output space for the mergesort is left to the caller, making the interface look essentially like the std::copy STL function:

```
template<class Merger, class Splitter, class RanIt, class OutIt>
OutIt mergesort(RanIt begin, RanIt end, OutIt out);
```

The body would consist of allocating the temporary buffer and a number of recursive calls, each followed by a call to std::copy, to free the temporary buffer.

The problem with this approach is that all elements are merged to a buffer and copied back. That is one more move per element than need be made. We can do better than that, observing that when we have made the first recursive call, all the elements from that subproblem are now in the temporary buffer. That leaves a “hole” in the original array just big enough to hold the output of the next recursive call. When all the recursive calls have completed, the hole have moved to the end of the array. We then do the only move of elements, namely from the buffer to the end of the array. The process is illustrated in Figure 5-8. Using this procedure saves us a considerable $N-\alpha^{1/d}N^{d-1/d}$ element moves.

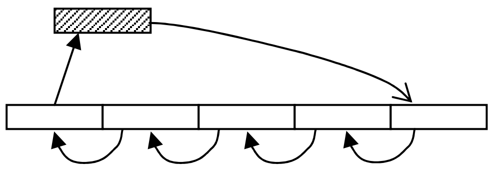

Figure 5-8. The merge procedure used in funnelsort. The thick arrows indicate sorting output while the thin arrow indicates a move.

In each recursive call, we need the temporary buffer and a k-funnel, but not both at the same time. Using the stack_allocator, described in Section 5.3.2, we can first compute which of the two takes up most space, construct a stack_allocator large enough to hold either of them, allocate the buffer, sort recursively, move the buffer elements back into the array, deallocate the buffer, and then layout the funnel using that allocator. This way, the funnel is laid out in exactly the memory locations the temporary buffer occupied. Recycling the workspace like this, will likely mean that the funnel is already in cache when it is needed.

5.4.2 Merger Caching

As with any function, at each recursive call a new set of local variables are allocated and constructed on the execution stack. This is normally not much of a performance issue, but if one of those variables is a funnel, having to allocate and construct it at each recursive call may soon become a performance issue.

In fact, constructing a new funnel at each recursive call is far from necessary. In all calls at the same level of recursion, we use a funnel of the same order, so instead of using a funnel local to the merge_sort function, we start out by simulating the recursive calls of the merge_sort function and at each level noting the order of the funnel needed. A funnel is then allocated for each level and they are in turn used in the recursive calls. For the simulation, we are only interested in the levels of the recursion and so could do with a single tail-recursive call, easily converted to a loop.

Using this scheme, we can only apply workspace recycling at the root of the recursion, but since that will dominate the rest of the recursion, both in workspace consumption and memory transfers, this will also be where we gain the most.

We have implemented the funnelsort algorithm both with and without merger caching to asses whether pre-computing the total space needed throughout the algorithm will be a considerable overhead, or constructing a new funnel in each call will hurt performance. The premise of using workspace recycling was that the funnel used the same stack based allocator as used to allocate the temporary buffer. However, since we deallocate the buffer just before we start allocating the funnel, using a heap allocator could achieve the same effect, if the allocator chooses to allocate from the newly freed area. At the same time, using a heap allocator may be slower than the stack_allocator, due to the complexity of managing a general heap, thus shifting the performance in favor of using merger caching.

We have tested the two versions of funnelsort with both a stack_allocator and a heap allocator. For this test we use $\alpha = 4$, $d = 2.5$, and the simple_merger basic merger ($z = 2$). We use the std::sort provided with STL to sort subarrays smaller than $\alpha z^d = 23$. We sort uniformly distributed pairs. The result is as follows:

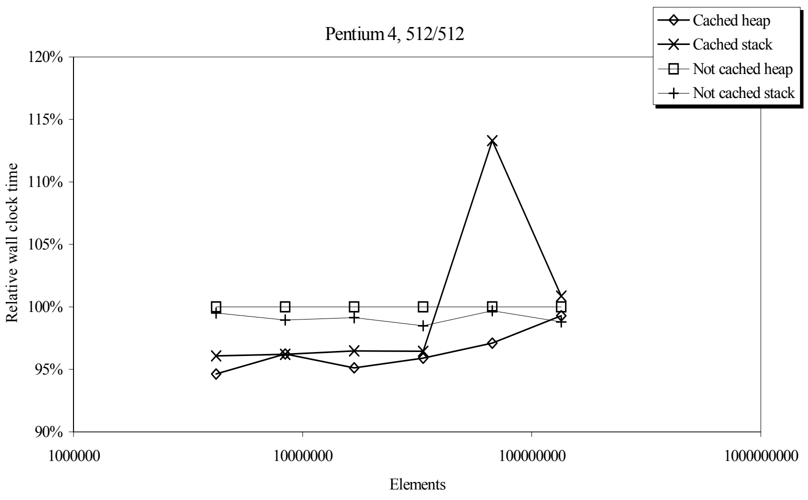

Chart C-43. Effects of merger caching, Pentium 4.

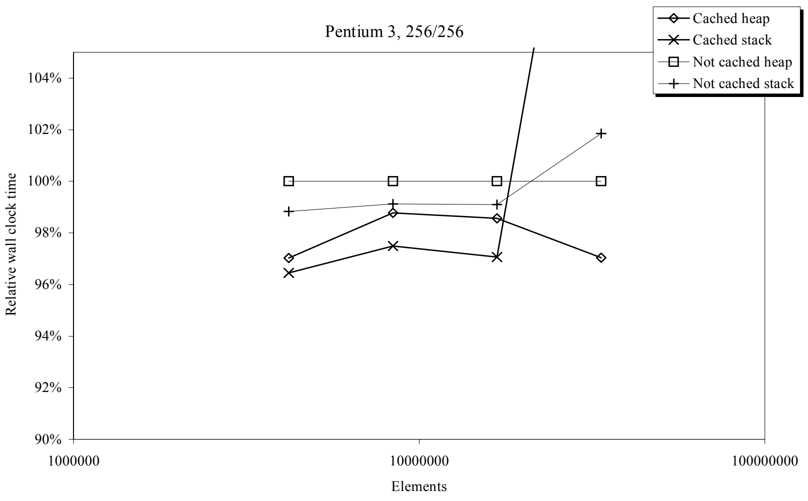

Chart C-44. Effects of merger caching, Pentium 3.

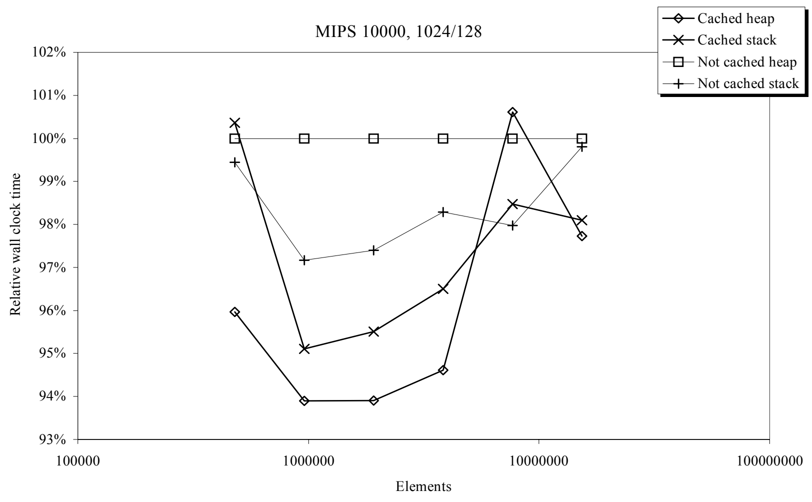

Chart C-45. Effects of merger caching, MIPS 10000.

We saw in Sections 5.3.2 and 0 that the different architectures preferred different allocators. We see the same picture here. We do however see a more consistent picture here; all architectures clearly prefer the mergers to be cached. We suspect that this is mostly due to avoiding the computational overhead of constructing mergers in each recursive call. There is only slight evidence of savings due to increased locality, by recycling workspace and using std::allocator, as can be seen here in the number of TLB misses:

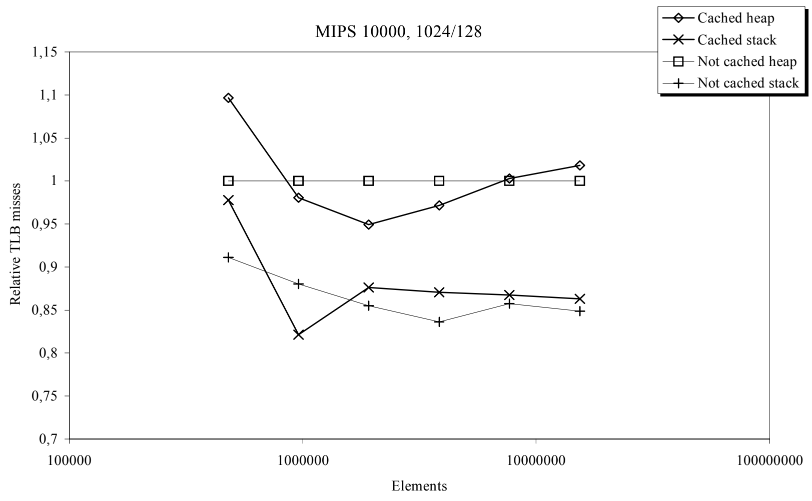

Chart C-47. Effects of merger caching, MIPS 10000, TLB misses.

The effect of reusing mergers is dwarfed by the effect of using stack_allocator. As discussed, using the stack_allocator allows us to reuse the temporary buffer for laying out the funnel. When we recycle the workspace like this, we effectively recycle virtual memory addresses, in turn keeping the translation look-aside buffer entries alive longer. This reduction in TLB misses does not affect the overall execution time significantly, however. We should use both merger caching and controlled allocation to reduce the construction overhead and increase overall locality.

5.4.3 Base Sorting Algorithms

As an improvement to quicksort, Sedgewick introduced the idea of not completing the quicksort recursion, but stop before problem sizes got too small [Sed78]. This would leave the elements only partially sorted. To sort it fully, insertion sort was used in a final pass. What made it efficient was the special property of insertion sort, that if no element is more than c places from where is should be in the sorted sequence, insertion sort can sort all n element using no more than $\mathcal{O}(cn)$ moves and comparisons [Knu98]. Ladner and LaMarca have since proposed that the insertion sort should be done at the bottom of the recursion rather than as a final pass, since a final pass would incur N/B additional memory transfers [LL99]. As a side effect, the special property of insertion sort is no longer needed; any low instruction count sorting algorithm can be used.

With funnelsort, we are faced with a similar situation – below a certain problem size, we have to switch to a different sorting algorithm, simply because no funnel can merge such small streams. We choose to switch to another algorithm when problem sizes becomes smaller than αz^d, because that in turn will make funnelsort choose at least a $z+1$-funnel, that is a funnel of greater than one height, on all inputs sorted by funnelsort. This avoids the need to handle the special case, where the root of a funnel is also a leaf.

The choice of sorting algorithm for the base is not clear. Insertion sort as proposed by Sedgewick performs $\mathcal{O}(n^2)$ moves in the worst case; however, it performs much better when applied to data that is almost sorted. Indeed, it naturally detects completely sorted sequences with only $\mathcal{O}(n)$ comparisons and uses no moves at all. A very low-overhead alternative to insertion sort is selection sort [Knu98, Algorithm S]. A compelling feature of selection sort is that for each position in the sequence, the correct element is located and then moved there; it only moves an element once. However, it does $\mathcal{O}(n^2)$ comparisons even in the best case.

The limitation of insertion sort is that most elements are never moved more than one position. Shell sort attempts to remedy this by doing several passes of insertion sort, first only on elements far apart, then on elements closer and closer to each other [Knu98, Algorithm D]. It has a higher overhead but will asymptotically perform fewer operations per element. Considering that modern processors are super-scalar and capable of executing several instructions in parallel, it is only natural to investigate sorting algorithms that are not inherently sequential. One such algorithm is Batcher's merge sort [Knu98, Algorithm M]. Similar to Shell sort, Batcher's sort uses several passes, each sorting elements closer and closer together. The difference is that the sequence of comparisons in Batcher's sort is such that they can be executed in parallel. Modern processors may be able to detect and exploit this. The downside is that computing the sequence gives this algorithm a considerable overhead. Heapsort is a special kind of selection sort, where each element is selected in $\mathcal{O}(\log n)$ moves and comparisons, making it an asymptotically optimal sorting algorithm.

These algorithms were implemented and run on small arrays of uniformly distributed random pairs. We measure the wall clock time it takes to sort a total of 4,096 such pairs. For this test, we had the unique opportunity to run on an Intel Itanium 2-based computer. The Itanium class of processors uses so-called explicit parallelism. This means that when the compiler issues instructions, it will bundle them in instructions capable of being executed in parallel. This is opposed to RISC and CISC architectures, where instructions are emitted by the compiler as sequential as they should be executed and the compiler is not concerned with what instructions can be executed in parallel. It will then attempt to extract any parallelism. Another side of the Itanium architecture is the heavy use of conditional execution; all instructions can be executed conditionally and on any of 128 predication bits. The results are as follows:

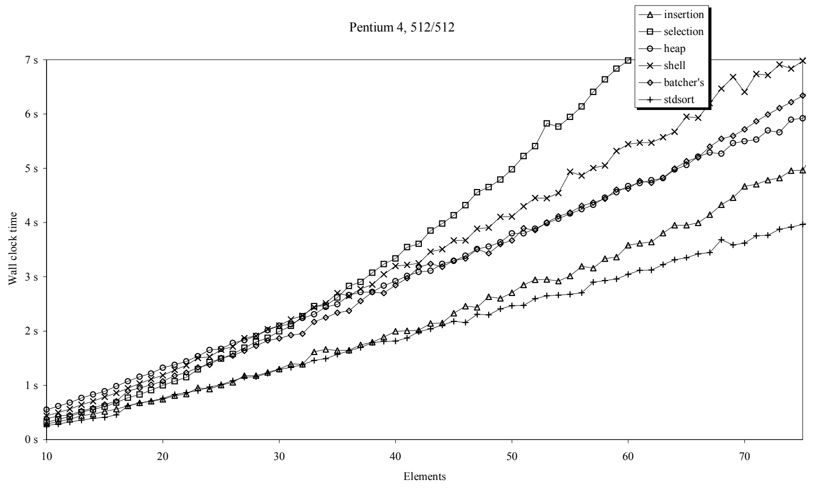

Chart C-48. Base sorting algorithms, Pentium 4.

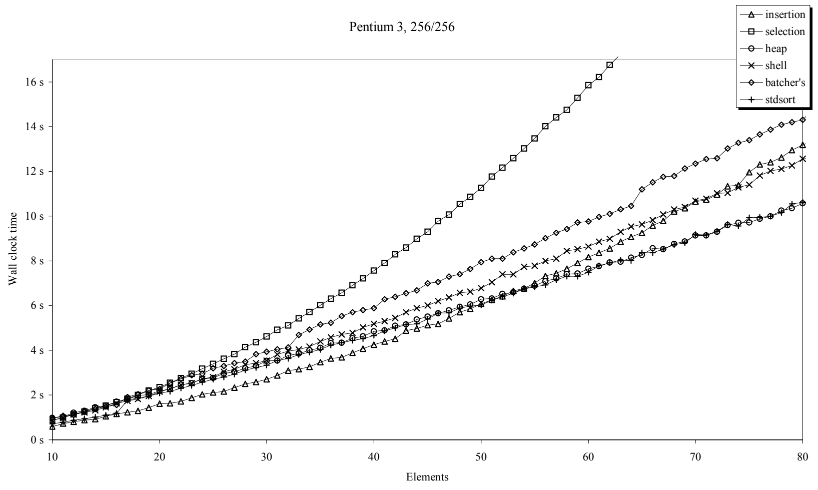

Chart C-49. Base sorting algorithms, Pentium 3.

5.4.3 Base Sorting Algorithms

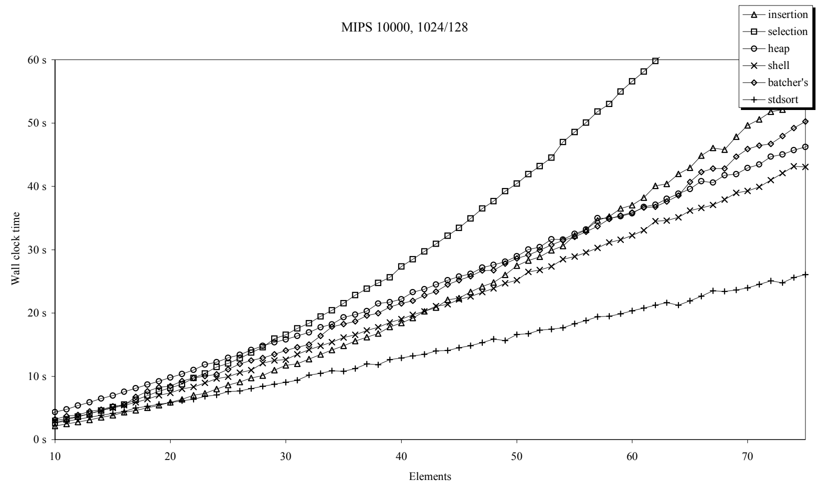

Chart C-50. Base sorting algorithms, MIPS 10000.

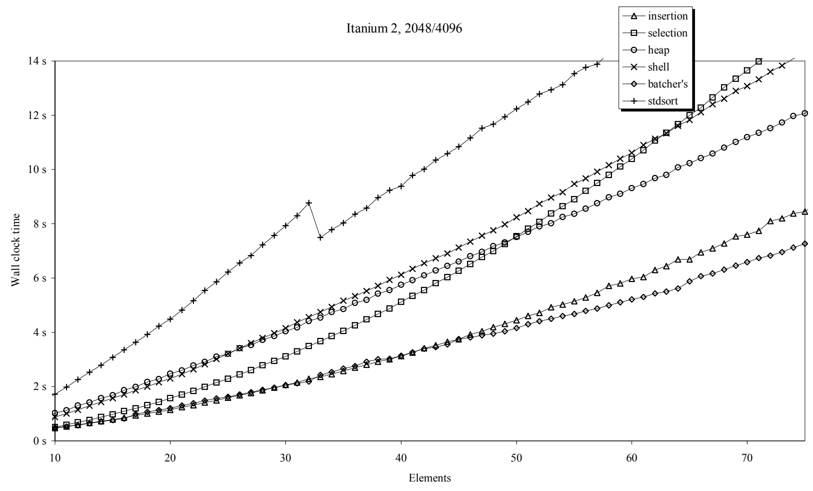

Chart C-51. Base sorting algorithms, Itanium 2.

For the test on the Itanium, we used the Intel C++ compiler version 7. This compiler comes with the Dinkumware implementation of the STL. This particular implementation features an `std::sort` function that like the SGI implementation is based on introsort. However, for the partitioning, a more robust function is used than in the SGI implementation. This function does a so-called Dutch flag partition, collecting elements that are equal to the partition element between the two partitions. Furthermore, it uses a sophisticated rotate function in the implementation of insertion sort used in the bottom of introsort. In all, while it makes the implementation faster on certain inputs, it clearly makes it slower on the sets we tested. The switch to insertion sort is `std::sort` can clearly be seen. In the SGI implementation (perhaps most clear on the Pentium 3 results) the switch happens at 16 elements, while in Dinkumware, it happens at 32 elements. Sedgewick originally suggested a switch around 9 or 10 elements; however, we see here that insertion sort remains competitive at least up in the 20's, even 40's on the Pentium 4, at least when sorting uniformly distributed `pairs`.

Selection sort is apparently too hampered by its best-case $\mathcal{O}(n^2)$ comparison count to be competitive. Comparing selection sort to insertion sort, we can see that insertion sort is indeed significantly faster than its $\mathcal{O}(n^2)$ worst-case time. Eventually, however, insertion sort will loose to all but selection sort. The optimal heapsort is quite competitive on all architectures and most problem sizes, while some architectures prefer Shell sort more than others.

Most interesting is perhaps Batcher's sort. On Pentium 3 and MIPS, its performance is in the mid-range, for the most part performing worse than Shell sort does. However, on Pentium 4, it performs better than Shell sort performs and is even able to keep up with heapsort. On the Itanium, however, it outperforms all other algorithms, being almost twice as fast as heapsort. This indicates that as processor performance get more and more dependant on instruction level parallelism, more and more performance can be gained when using sorting algorithm that allow for such parallelism.

As suspected, at least on the more traditional architectures, no algorithm can beat the hybrid and highly optimized approach of introsort.

5.4.4 Buffer Sizes

Finally, the implementation details of the complete funnelsort are in place. Without further ado, here are the results of sorting using funnelsort with different values of α and d:

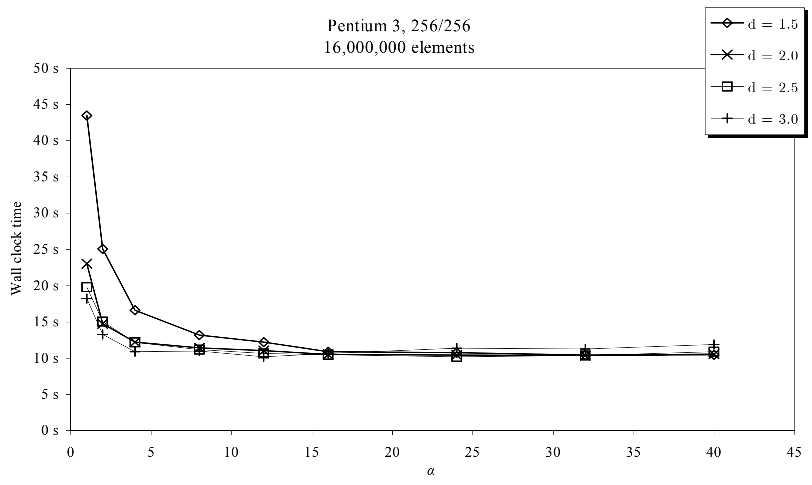

Chart C-55. Buffer parameters, sorting 16,000,000 elements on Pentium 3.

5.4.4 Buffer Sizes

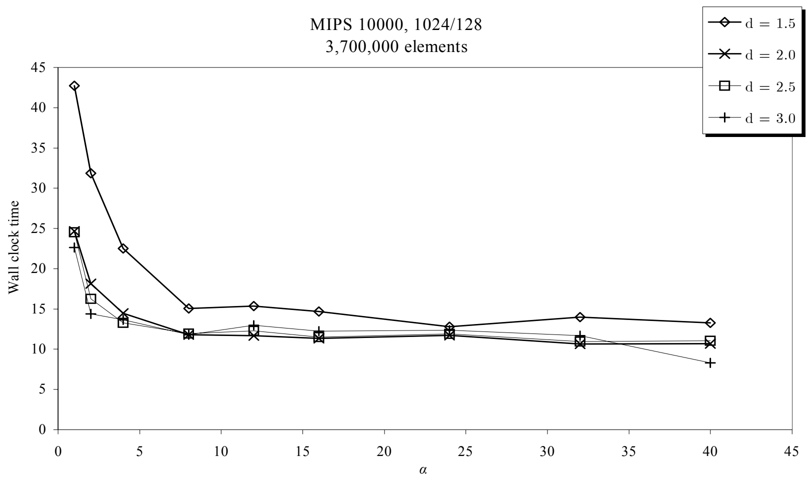

Chart C-58. Buffer parameters, sorting 3,700,000 elements on MIPS.

The test was conducted for three different array sizes on each machine, all of which fit in main memory. As suspected, when decreasing the values of α and d, fewer elements are merged per call to fill, and the overhead of navigating the tree and managing buffers become significant. With $\alpha > 4$ and $d \geq 2$, we can see that this overhead is virtually gone. Maximal performance is reached around $\alpha = 16$ and $d = 2.5$.

Choosing α and d is not as simple as the above two charts imply, however. The choice of values influences both the order of the funnel used and the space needed to hold it. To expose these effects, one of the array sizes were chosen close to what can fit in RAM. The results are as follows:

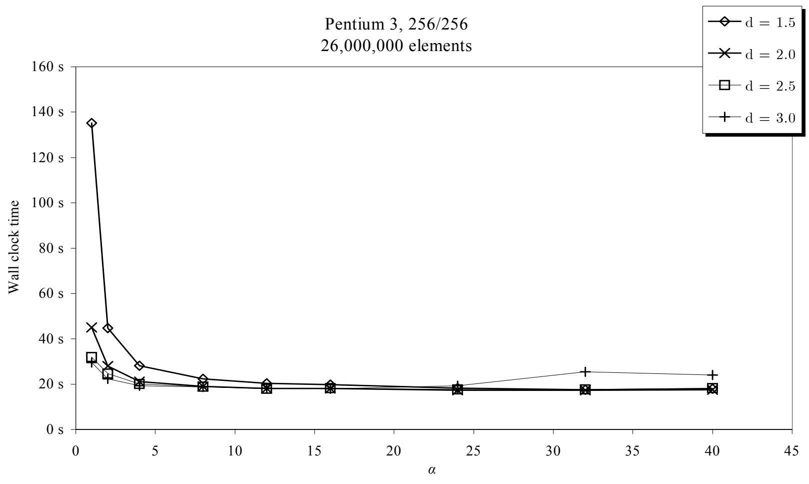

Chart C-57. Buffer parameters, sorting 26,000,000 elements on Pentium 3.

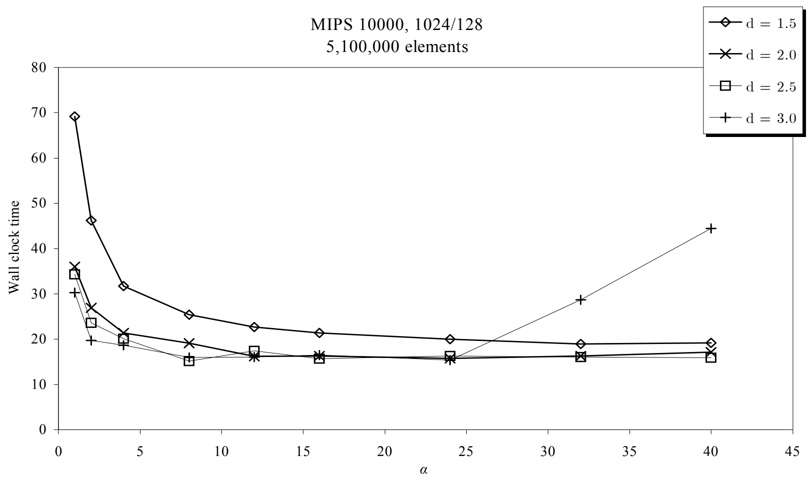

Chart C-59. Buffer parameters, sorting 5,100,000 elements on MIPS.

Two effects are dominating. One, if we choose d small, we will need a very high order funnel, and since $d > 1$, the total space consumed by its buffers are super linear. The total space needed for the algorithm then becomes too much to fit in memory. On the other hand, when the values of α and d are increased, the funnel it self will require more space and even though a lower order funnel is used, the size of the funnel is again too much for it to fit in RAM.

For any choice of values of α and d, the algorithm will require space for the funnel and for some array size this particular choice will make the total space requirements of the algorithm too high for it to fit in cache. The point is that we should avoid extreme values of α and d, since it will cause extreme space requirements of the funnel; it may be tempting to choose high values of α and d to minimize the overhead; however, doing so may cause the algorithm to incur memory transfers on smaller arrays than had we chosen more sensible α and d.

5.5 LOWSCOSA

The primary components of the LOWSCOSA are partitioning and merging with funnels. With a high performance funnel, already in place this leaves partitioning, which we will look at in this section. At the end of the section, we will briefly discuss what performance to expect from the LOWSOSA.

5.5.1 Partitioning

Partitioning elements of an array consists of two phases: median finding and partitioning. The partitioning phase uses the median as a pivot element and during a single scan moves elements that are larger than the pivot to one side and elements that are smaller to the other side. The exact median can also be found in linear time [BFP+73].

For algorithms like quicksort, we are not required to partition into two equally large partitions. For those algorithms, we thus do not have to use the exact median as the pivot; we can make due with an approximate median. Such a median can be computed as the exact median of a small sample of elements instead of all elements. Popular sample sizes for quicksort are three and nine [Knu98], even $\sqrt{n}$ [MR01]. The effects of this is that virtually no time is spend finding the median and thus only the partitioning contributes to the linear term in the complexity. A downside is that we risk making uneven partitions where one part not much smaller than the original array. This can mean that the time spent partitioning is largely wasted. In quicksort, however, even with a sample size of one, that is we use a predetermined element as the approximate median, on uniformly distributed elements the expected running time is only a small constant larger than what could be achieved if we new the exact median in advance [Knu98].

In the interest of performance, we would like to use an approximate median for the LOWSCOSA also. The consequences of the resulting uneven partitions are however not as trivial as in the case of quicksort. If we partition such that there are more small elements than large elements, the output of the merger cannot fit in the space originally occupied by the large elements (see Figure 4-3, page 58). This is a design problem in the algorithm that needs to either be solved or avoided. This means that we cannot hope to generate more sorted elements than there are elements in the smaller of the two partitions at each iteration of the LOWSCOSA. Furthermore, if we go ahead, sort the large number of small elements for the input to the funnel, and only output a small number, we have wasted a considerable time sorting them.

5.5.2 Strategy for Handling Uneven Partitions

Before we look at how to handle the case of an uneven partition, we make the following observation. It is possible to combine the partition phase with the sorting of the subarrays to be merged in the current call; during the partitioning, when we have moved a sufficiently number of small elements to the end of the array, we put the partition on hold and sort them. This way, when these subarrays are small enough to fit in cache, we can complete the partition phase *and* the following sorting phase incurring only N/B memory transfers instead of up to $3N/(2B)$. We consider this an important optimization in the interest of increasing locality and cache usage.

Repartition
The simplest strategy is perhaps to perform the partition using the approximate median. When that is done, if more than half the elements we partitioned as smaller than the median, we simply pick a new median and partition again. An improvement is to only partition the small elements. This will take less time and more likely generate a partitioning with fewer small elements than large.

However, this approach makes it infeasible to sort streams while partitioning, because we risk having to repartition and thus make the sorting a wasted effort. In addition, we would expect every other partition to generate more small elements than large, so repeating the partitioning every time that happens will generate a considerable overhead.

Abort on Empty Refiller
Instead of repartitioning, thus avoiding the problem of outputting too many elements, we can continue with the uneven partition and handle the problem explicitly. The problem can be solved by giving the refiller a way to abort the merging. It would then

do so just before it begins reading into the part containing small elements. The situation is depicted in Figure 5-9. This will leave a "hole" in the input between the output and the refiller. The hole is patched with the elements contained in the funnel and included in the recursive call.

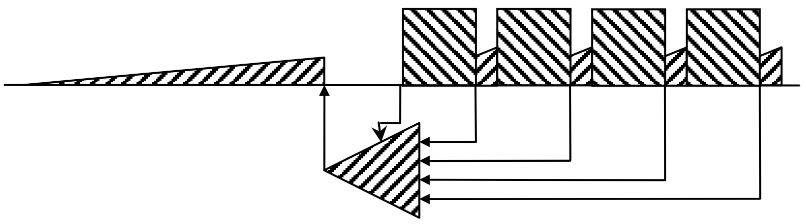

Figure 5-9. The refiller has no more large elements.

This scheme avoids a second partitioning and allows us to sort the input streams during the partitioning; however, it is flawed in the case of extremely few large elements. In these cases, the buffers in the funnel will not be filled and not a single element output. The refiller reads in all large elements before a single element is output from the funnel. In these extreme cases, we would have to fall back on the repartitioning scheme or employ some other special-case handling scheme.

Abort on Full Output
To remedy the fault, we may continue the merging until we have filled the left side of the array with small elements. To avoid the refiller starting to read in parts of the sorted streams, potentially duplicating elements, we need to detach it from the funnel. We have to keep the input streams attached so the funnel keeps reading the small elements. When the output has filled the left side, some elements are both in the input streams (the space they occupied was not refilled) and in either the funnel or in the output. In essence, the hole from the previous scheme is now scattered in all the input streams. Like before, we fill these holes with the elements remaining in the funnel.

Merge Big Elements
Perhaps the most elegant approach is to make input streams of the elements in then smallest of the partitions, not necessarily of the small elements. If the smallest of the partitions contain large elements, we use a funnel that outputs large elements first and writes them from the end of the array to the beginning. The process is illustrated in Figure 5-10. Note that the output and input of the funnel is now written and read in reverse direction.

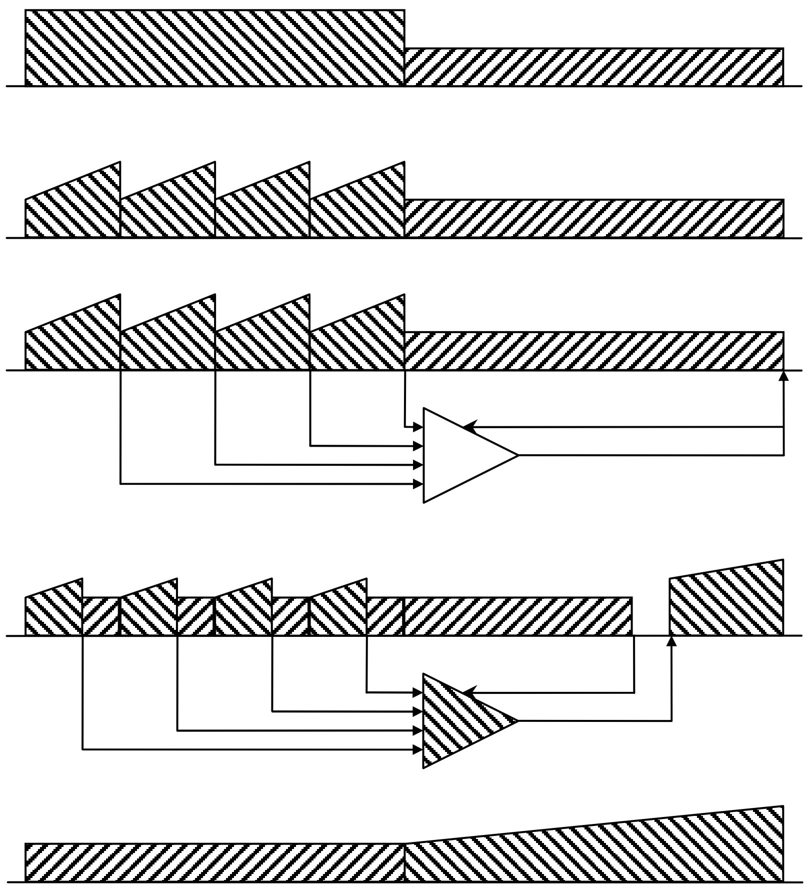

Figure 5-10. The process of multiway merging when there are more small elements than large element after the partition.

In our implementation, we have chosen to detach the refiller when it hits the small elements and to sort the streams while partitioning. This means that if we do an uneven partition, we may have sorted streams containing many more elements than the number of elements sorted by the end of an iteration. To reduce the risk of that happening, we have increased the sample size to 31 elements. We sort this sample and use the 17th smallest as the partition. Using a larger element than the 15th smallest reduces the risk of partitioning more small elements than large elements.

5.5.3 Performance Expectation.

When looking at the virtual memory level of the memory hierarchy, we have argued that for all sensible input sizes neither multiway mergesort nor funnelsort will do more than $4N/B$ memory transfers. The LOWSCOSA will unfortunately do more than that.

Under the assumption that an entire input stream of the funnel can fit in memory, the partitioning and sorting of input streams can cause up to $2N/B$. The funnel will read in and write out half the elements for a total of N/B memory transfers. However, by now only half the elements are sorted, assuming partition into equally many large and small elements. The algorithm will continue on arrays of geometrically decreasing sizes, effectively doubling the the number of memory transfers. The total number of memory transfers will be largely dominated by those incurred at the first iterations, so when N is significantly larger than M, the fact that the last $\log M$ iterations can be performed without incurring memory transfers has little influence.

Thus, the total number of memory transfers incurred under these assumptions could be as high as $6N/B$. With an input occupying 2GB and half a gigabyte of RAM, the first two iterations incur the full number of memory transfers, while after the partitioning phase of the third iteration the rest of the algorithm activity is within RAM. This gives a total of $(3+(3/2)+2/4)N/B = 5N/B$ memory transfers, only slightly less than quicksort on the same input size and under more realistic assumptions (see Section 3.2.4).

In addition to added memory transfers, the LOWSCOSA also has an increased instruction count; each time an element is read into the funnel, another element is moved in its place. This will, in turn, almost double the total number of element moves performed. The number of comparisons is also increased, since before an element is in its right place, it has participated in a partitioning and a merge, as well as the recursive sorts. Furthermore, the LOWSCOSA requires $\mathcal{O}(\log n)$ funnels to be constructed as opposed to funnelsort requiring $\mathcal{O}(\log\log n)$ funnels.

In all, the expectation of the performance of the LOWSCOSA does not look good. It is however optimal in the cache-oblivious model and the fact that it requires low order working space is for us reason enough to investigate its performance.