The Influence of Caches on the Performance of Heaps [*]

TR # UW-CSE-96-02-03

Anthony LaMarca
Xerox PARC

Richard E. Ladner
University of Washington
Seattle, WA 98195

January 6, 1997

Abstract

As memory access times grow larger relative to processor cycle times, the cache performance of algorithms has an increasingly large impact on overall performance. Unfortunately, most commonly used algorithms were not designed with cache performance in mind. This paper investigates the cache performance of implicit heaps. We present optimizations which significantly reduce the cache misses that heaps incur and improve their overall performance. We present an analytical model called *collective analysis* that allows cache performance to be predicted as a function of both cache configuration and algorithm configuration. As part of our investigation, we perform an approximate analysis of the cache performance of both traditional heaps and our improved heaps in our model. In addition empirical data is given for five architectures to show the impact our optimizations have on overall performance. We also revisit a priority queue study originally performed by Jones [25]. Due to the increases in cache miss penalties, the relative performance results we obtain on today's machines differ greatly from the machines of only ten years ago. We compare the performance of implicit heaps, skew heaps and splay trees and discuss the difference between our results and Jones's.

1 Introduction

The time to service a cache miss to memory has grown from 6 cycles for the Vax 11/780 to 120 for the AlphaServer 8400 [10, 18]. Cache miss penalties have grown to the point where good overall performance cannot be achieved without good cache performance. Unfortunately, many fundamental algorithms were developed without considering caching. In this paper, we investigate and optimize the cache performance of implicit heaps. As part of this study, we introduce *collective analysis*, a framework we have developed that allows an algorithm's cache performance to be predicted for direct mapped caches. Our hope is that algorithm designers will use models like collective analysis in conjunction with traditional analysis techniques in order to better understand how their algorithms perform on real machines.

1.1 Methodology

This paper is a study of the performance of algorithms. We examine the way programs are analyzed and optimized for performance. In this section we present the methodology we use in our performance study and contrast it with

[*] This paper appears in *The ACM Journal of Experimental Algorithmics*. The work of LaMarca was performed at the University of Washington and was partly supported by an AT&T fellowship.

Author's addresses: A. LaMarca, Xerox PARC, 3333 Coyote Hill Road, Palo Alto CA 94304; email: lamarca@parc.xerox.com; R. E. Ladner, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195; email: ladner@cs.washington.edu

approaches that have been used in the past.

The true test of performance is execution time and numerous researchers investigate algorithm performance by comparing execution times. The drawback of simply implementing an algorithm and measuring its execution time, however, is that very little intuition is provided as to why the performance was either good or bad. As we will show in this paper, it is possible for one algorithm to exhibit good cache locality with high instruction counts while a similar algorithm exhibits just the opposite. While execution time is the final measure of performance, it is too coarse-grained a metric to use in the intermediate stages of analysis and optimization of algorithms.

The majority of researchers in the algorithm community compare algorithm performance using analyses in a unit-cost model. The RAM model [4, 12] is used most commonly, and in this abstract architecture all basic operations, including reads and writes to memory, have unit cost. Unit-cost models have the advantage that they are simple, easy to use, and produce results that are easily compared. The big drawback is that unit-cost models do not reflect the memory hierarchies present in modern computers. In the past, they may have been fair indicators of performance, but that is no longer true.

It is also common for the analyses of algorithms in a specific area to only count particular expensive operations. Analyses of searching algorithms, for example, typically only count the number of comparisons performed. The motivation behind only counting expensive operations is a sound one. It simplifies the analysis yet retains accuracy since the bulk of the costs are captured. The problem with this approach is that shifts in technology can render the expensive operations inexpensive and vice versa. Such is the case with comparisons performed in searching algorithms. On modern machines comparing two registers is no more expensive than copying or adding them. In Section 4.1, we show how this type of analysis can lead to incorrect conclusions regarding performance.

In this paper, we employ an incremental approach to evaluating algorithm performance. Rather than discard existing analyses and perform them again in a new comprehensive model, we leverage as much as we can off existing analyses and fill in where they are weak. An evolutionary approach should involve less work than a complete re-evaluation and it offers an easy transition for those interested in a more accurate analysis.

One weakness of existing unit-cost analysis is that it fails to measure the cost of cache misses that algorithms incur. For this reason, we divide total execution time into two parts. The first part is the time the algorithm would spend executing in a system where cache misses do not occur. We refer to this as the *instruction cost* of the algorithm. The second part is the time spent servicing the cache misses that do occur and we call this the *memory overhead*.

We measure instruction cost with both analyses in a unit-cost model and with dynamic instruction counts from executions. In this paper we employ both existing unit-cost analyses and some simple analyses of our own. Details of our technique for instrumenting programs to produce dynamic instruction counts are described in Section 4.4. Neither of these techniques will model instruction cost perfectly. Execution delays such as branch delay stalls and Translation Lookaside Buffer (TLB) misses are not measured with either of these methods. Nor do they capture variance in the cycle times of different types of instructions. Despite these shortcomings, however, they both provide a good indication of relative execution time ignoring cache misses.

In this study we focus on the memory overhead caused by data references and we do not attempt to measure the overhead due to instruction cache misses. In this paper, our experimental cache performance data is gathered using trace-driven cache simulation. Cache simulations have the benefit that they are easy to run and the results are accurate. In addition, we use collective analyses to explore ways in which the memory behavior of an algorithm might be predicted without address traces or implementations. When both analytical predictions and simulation results are available, we compare them to validate the accuracy of the predictions.

1.2 Implicit Heaps

In this paper, we perform a study of the cache performance of implicit heaps [42]. The main goal of our study is to understand and improve the memory system performance of heaps using both experimental and analytical tools. We evaluate the performance of heaps as both a tool for sorting as well as the event queue in a discrete event simulation. We collect performance data from execution times on various machines, from trace-driven simulations, and from analysis in our new framework. Using this performance data as a guide, we develop an improved implicit heap which has better memory system and overall performance. In our experiments, our improved heap incurs as much as 65% fewer cache misses running in the hold model [25], a benchmark that measures how well a priority queue serves as the event set for a simulation. The reduction in cache misses translates to speedups as high as 75% for a heap running in the hold model, and performing heapsort with our optimized heaps improved performance by as much as a factor of two.

In this paper we also reproduce a subset of the experiments performed by Jones comparing the performance of various priority queues [25]. In our experiments, we compare the performance of traditional heaps, our improved heaps, top-down skew heaps and both top-down and bottom-up splay trees. Our comparison of priority queues yielded very different results from Jones's original experiments. Caches have changed architectures to the point that a bottom-up splay tree, the structure which performed best in Jones's experiments, now performs many times worse than a traditional heap.

1.3 Related Work

There has been a large amount of research on analyzing and improving cache performance. Most cache performance analysis is currently done with hardware monitoring [39] or with trace-driven simulation [17, 41]. Neither of these solutions offers the benefits of an analytical cache model, namely the ability to quickly obtain estimates of cache performance for varying cache and algorithm configurations. An analytical cache model also has the inherent advantage that it helps a designer understand the algorithm and helps uncover possible optimizations.

A number of researchers have employed hybrid modeling techniques in which a combination of trace-driven simulation and analytical models is used [1, 35]. These techniques compress an address trace into a few key parameters describing an application's behavior, and these are then used to drive an analytical cache model. The difference between the collective analysis framework we present and these techniques is that collective analysis does not involve any trace data. The behavior of an algorithm is explicitly stated, and this serves as input to our model. While this makes the analysis more difficult to perform, it offers the advantage that our model can provide fast predictions for a variety of algorithm configurations and cache configurations. Temam, Fricker and Jalby [38] provide a purely analytical approach for predicting conflict misses in loop nests of numerical codes. In their model, the memory reference patterns are examined, and cache performance is predicted by determining when each data item is reused and how often this reuse is disrupted. Our work differs from theirs in that they work with algorithms whose exact reference pattern is known, while ours is intended for algorithms whose general behavior is known but whose exact reference pattern is not predictable across iterations.

Martonosi *et al.* augment trace-driven cache simulation by categorizing the cache misses by the type of miss and by the name of the data structure incurring the miss [31]. Systems with similar features include include Cprof [30] and SHMAP [16].

A study by Rao [33] examines the performance of page replacement policies using the independent reference model [11] to predict miss behavior. In collective analysis, we make assumptions similar to Rao's about the distribution of references within cache regions. The result is that our formulas for cache performance are very similar to Rao's.

There are many examples of case studies of the cache performance of specific applications. In most cases, the memory system behavior of the piece of code is investigated using profiling tools. Typically, this investigation leads to optimizations that speed up the application by a factor of two or more. Agarwal *et al.* optimized a library of linear algebra routines by tuning its memory behavior to the Power2 architecture [2]. Lebeck and Wood improved the performance of the SPEC benchmarks with cache optimizations [30]. LaMarca and Ladner use empirical and analytical techniques to investigate and improve the performance of sorting algorithms [28, 29].

A number of abstract architectures for sequential machines have been introduced that take into account the memory hierarchies present in modern machines. The uniform memory hierarchy [5] models the memory system as a series of increasingly large memory modules. As well as the computation that is to occur, an algorithm for the uniform memory hierarchy describes when blocks of data should be moved between modules. Another example is the hierarchical memory model [3] with a single address space in which an access to location i takes time $\lceil \log i \rceil$. Efficient programming in this model requires commonly used data items to be copied from high addresses to low addresses during their periods of high use.

The compiler community has produced many optimizations for improving the cache locality of code [9, 6, 43] and algorithms for deciding when these optimizations should be applied [26, 8, 20]. The vast majority of these compiler optimizations focus on loop nests and offer little improvement to pointer-based structures and algorithms whose reference patterns are difficult to predict. The reference patterns of the priority queue algorithms we examine in the paper are complicated enough that these compiler optimizations offer no benefit.

Priority queues and heaps in particular have been well studied. Improved algorithms have been developed for building, adding and removing from heaps [27, 13, 7, 21, 19]. A number of pointer-based, self-balancing priority queues, including splay trees and skew heaps [36], have been developed since the heap was introduced. The main difference between the work in this paper and previous work is that we focus on cache performance as well as instruction cost. This different approach raises opportunities not previously considered and allows us to improve performance beyond existing results. Evaluations of priority queue performance include Jones's study using various architectures and data distributions [25], and a study by Naor *et al.* which examines instruction counts as well as page fault behavior [32].

1.4 Overview

In Section 2 we describe a typical memory system and motivate the analysis of the cache performance of algorithms. In Section 3 we introduce *collective analysis*, a framework for predicting the cache performance of algorithms. We provide a definition of our model as well as a simple example analysis. In Section 4 we describe the heap we use as the base of our performance study and we introduce two optimizations to this heap. We then perform a collective analysis of heaps and explore the impact our optimizations have on cache performance. In Section 5 we present data from trace-driven simulations and executions on various architectures to show the impact of our optimizations on both cache performance and overall performance of heaps. In Section 6 we revisit Jones's priority queue study [25]. We compare the performance of a traditional heap, an improved heap, skew heaps and splay trees in the hold model using exponentially distributed keys. In Section 7 we present our conclusions.

2 Motivation

While machines have varying memory system configurations, almost all share some characteristics. As an example, one design characteristic of caches is the *associativity*, which indicates the number of different locations in the cache where a particular block can be loaded. While associativity might, in principle, vary widely from cache to cache, the current state of technology leads the vast majority of architects to implement *direct-mapped* caches which have an associativity of one, and can load a particular block only in a single location. A safe generalization is that most new machines have a multi-level cache architecture where the cache closest to memory is typically direct-mapped and has a high miss penalty [23]. The third-level cache in the DEC AlphaServer 8400, for example, has a four megabyte capacity and and a miss penalty of 120 cycles [18]. Accordingly, the techniques we present in this paper are primarily targeted for algorithms running on architectures with large direct-mapped caches.

2.1 Why Look at Cache Performance?

Traditional algorithm design and analysis has, for the most part, ignored caches. Unfortunately, as the following example about heaps will show, ignoring cache behavior can result in misleading conclusions regarding an algorithm's performance. Details about the definition and implementation of implicit heaps is given in Section 4, and can be consulted for those unfamiliar with heaps.

Building a heap from a set of unordered keys is typically done using one of two algorithms. The first algorithm is the obvious and naive way, namely to start with an empty heap and repeatedly perform *add*s until the heap is built. We call this the *Repeated-Adds* algorithm.

The second algorithm for building a heap is due to Floyd [19] and builds a heap in fewer instructions than Repeated-Adds. Floyd's method begins by treating the array of unordered keys as if it were a heap. It then starts half way into the heap and re-heapifies subtrees from the middle up until the entire heap is valid. The general consensus is that, due to its low operation count and linear worst case behavior, Floyd's method is the preferred algorithm for building a heap from a set of keys [34, 40, 12].

In our experiments, we executed both build-heap algorithms on a set of uniformly distributed keys. As the literature suggests, Floyd's method executes far fewer instructions per key than does Repeated-Adds. Our runs on a DEC Alphastation 250 showed that for uniformly distributed keys, both algorithms executed a number of instructions linear in the number of elements in the heap. Floyd's algorithm executed on average 22 instructions per element while Repeated-Adds averaged 33 instructions per element. This would seem to indicate that Floyd's method would be the algorithm of choice. When we consider cache performance, however, we see a very different picture.

First consider the locality of the Repeated-Adds algorithm. An *add* operation can only touch a chain of elements from the new node at the bottom up to the root. Given that we have just added an element, the expected number of uncached elements touched on the next *add* is likely to be very small. There is a 50% chance that the previously added

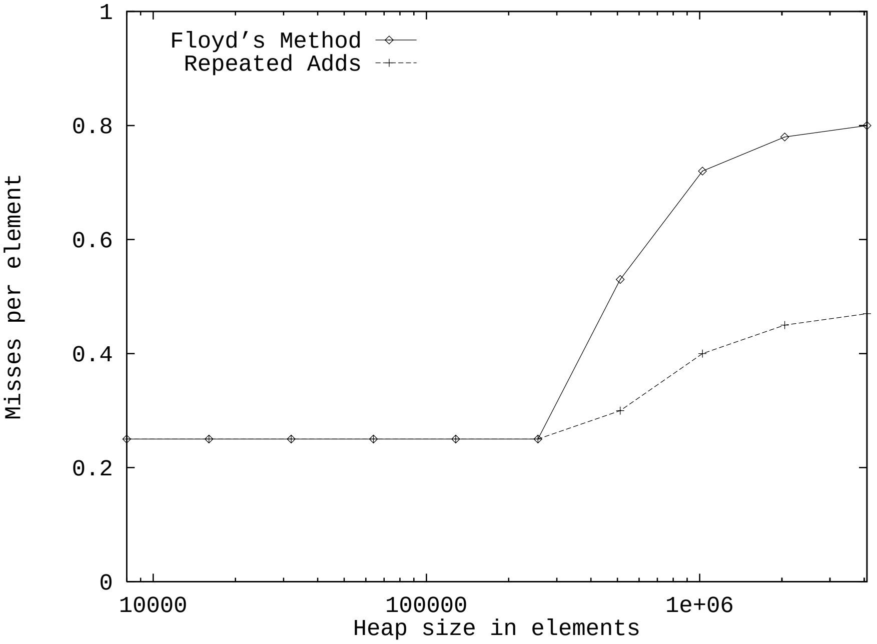

Figure 1: A comparison of the cache performance of Repeated-Adds and Floyd's method for building a heap. Simulated cache size is 2 megabytes, block size is 32 bytes and heap element size is 8 bytes.

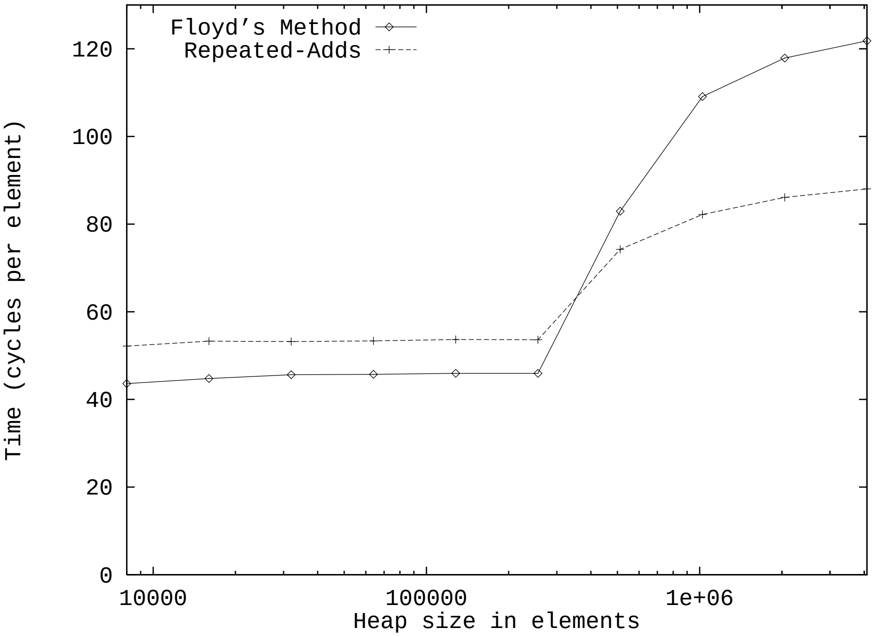

Figure 2: Execution time for Repeated-Adds and Floyd's method on a DEC Alphastation 250. Heap element size is 8 bytes.

element and the new element have the same parent, a 75% chance that they have the same grand-parent and so on. Thus, the number of new heap elements that needs to be brought into the cache for each *add* should be small on average.

Now consider the cache locality of Floyd's method. Recall, that Floyd's method works its way up the heap, re-heapifying subtrees until it reaches the root. For $i > 1$, the subtree rooted at i does not share a single heap element with the subtree rooted at $i - 1$. Thus, as the algorithm progresses re-heapifying successive subtrees, we expect the algorithm to exhibit poor temporal locality and to incur a large number of cache misses.

To test this, we performed a trace-driven cache simulation of these two algorithms for building a heap. Figure 1 shows a graph of the cache misses per element for a two megabyte direct-mapped cache with a 32 byte block size using 8 byte heap elements and varying the heap size from 8,000 to 4,096,000. This graph shows that up to a heap size of 262,144, the entire heap fits in the cache, and the only misses incurred are compulsory misses. Once the heap doesn't fit in the cache, however, we see that Floyd's method incurs significantly more misses per element than Repeated-Adds as our intuition suggested. We expect the execution time of an algorithm to represent a mix of the number of instructions executed and misses incurred. Figure 2 shows the execution times of these two algorithms on a DEC Alphastation 250 using 8 byte heap elements and varying the heap size from 8,000 to 4,096,000. When the heap fits in the cache, Floyd's method is the clear choice. Surprisingly, however, if the heap is larger than the cache, the difference in cache misses outweighs the difference in instruction count and the naive Repeated-Adds algorithm prevails.

This example serves as a strong indicator that more accurate analyses and better performing algorithms can be produced by design and analysis techniques that are conscious of the effects of caching.

3 Collective Analysis

While simple and accurate, trace-driven simulation does not offer the benefits of an analytical cache model, namely the ability to quickly obtain estimates of cache performance for varying cache and algorithm configurations. In this section, we present *collective analysis*, a framework for predicting cache performance of algorithms, as well as a sample analysis showing its use.

3.1 Our Memory Model

We assume that there is a single cache with a total capacity of C bytes, where C is a power of two. In our model, the cache is divided into blocks of size B bytes each, where $B \leq C$ and is also a power of two. In order to simplify analysis, we only model direct mapped caches [23], that is to say, we assume that any given block of memory can be loaded into only one location in the cache. Another simplification in our model is that we do not distinguish reads from writes. We also assume that items that are contiguous in the virtual address space map to contiguous cache locations, which means that we are modeling a virtually indexed cache [23]. Our model does not include a TLB, nor does it attempt to capture page faults due to physical memory limitations.

3.2 Applying the Model

The goal of collective analysis is to approximate the memory behavior of an algorithm and predict its cache performance characteristics from this approximation. The first step is to partition the cache into a set R of *regions*, where regions are non-overlapping, but not necessarily contiguous, sections in the cache. All cache blocks must belong to a single region, and a cache block cannot be split across regions. The intuition behind the decomposition is to group cache blocks into sets of blocks that are logically used in the same way. In collective analysis, the cache should be divided into regions in such a way that the accesses to a particular region are uniformly distributed across that region. If the accesses are not uniformly distributed, the region should be subdivided into multiple regions. Once the accesses within a region are uniformly distributed, further subdivision should be avoided in order to minimize the complexity of the analysis.

The next step is to break the system of algorithms to be analyzed into a set P of independent memoryless *processes*, where a process is intended to characterize the memory behavior of an algorithm or part of an algorithm from the system. The behavior of the system need not be represented exactly by the processes, and simplifying approximations are made at the expense of corresponding inaccuracies in the results. Areas of the virtual address space accessed in a uniform way should be represented with a single process. Collectively, all of the processes represent the accesses to the entire virtual address space and hence represent the system's overall memory behavior.

For purposes of the analysis we assume that the references to memory satisfy the *independent reference assumption* [11]. In this model each access is independent of all previous accesses, that is, the system is memoryless. Algorithms which exhibit very regular access patterns such as sequential traversals will not be accurately modeled in our framework because we make the independent reference assumption. A formal approach uses finite state Markov chains in the model. In the following we take an informal intuitive approach.

Define the *access intensity* of a process or set of processes to be the rate of accesses per instruction by the process or set of processes. Let λ_{ij} be the access intensity of process j in cache region i. Let λ_i be the access intensity of the set of all processes in cache region i. Let λ be the access intensity of all the processes in all the regions. Given our assumption about uniform access, if a cache block x is in region i and the region contains m cache blocks, then process j accesses block x with intensity λ_{ij}/m. Once the system has been decomposed into processes and the intensities have been expressed, the calculations to predict cache performance are fairly simple.

The total access intensity by the system for cache region i is

$$\lambda_i = \sum_{j \in P} \lambda_{ij}.$$

The total access intensity of the system is

$$\lambda = \sum_{i \in R} \lambda_i = \sum_{i \in R} \sum_{j \in P} \lambda_{ij}.$$

In an execution of the system a *hit* is an access to a block by a process where the previous access to the block was by the same process. An access is a *miss* if it is not a hit. The following theorem is a reformulation of the results of Rao [33]. Our formulation differs from Rao's only in that we group together blocks into a region if the accesses are uniformly distributed in the region. This, however, simplifies the analysis considerably.

Theorem 1 *The expected total hit intensity of the system is*

$$\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2. \tag{1}$$

Proof: Define η to be the expected total hit intensity of the system and η_i to be the expected hit intensity for region i. The total hit intensity of the system is the sum of the hit intensities for each region. The hit intensity for a region is the sum across all processes of the hit intensity of each process in that region.

An access to a block in a direct mapped cache by process j will be a hit if no other process has accessed the block since the last access by process j. We say that a cache block is *owned* by process j at access t if process j performed the last access before access t on the block. Since accesses to region i by process j are uniform throughout the region, the fraction of accesses that process j owns of any block in region i is $\frac{\lambda_{ij}}{\lambda_i}$. Hence the expected hit intensity for region i by process j is $\frac{\lambda_{ij}^2}{\lambda_i}$. Hence, $\eta_i = \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2$ and $\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2$

■

Corollary 1 *The expected overall miss intensity is* $\lambda - \eta$.

Corollary 2 *The expected overall hit ratio is* $\frac{\eta}{\lambda}$.

3.3 A Simple Example

In order to illustrate the use of collective analysis, we introduce a simple algorithm which uses two data-structures: one array the size of the cache, and another array one-third the size of the cache. The algorithm loops forever, and each time through the loop it reads two values at random from the large array, adds them together and stores the result in a randomly selected element of the small array.

We begin by decomposing the cache into two regions. Region 0 is two-thirds the size of the cache and only holds blocks from the large array. Region 1 is one-third the size of the cache, and holds blocks from both the large and the small array.

We divide the system into two processes: process 0 for the reads from the large array and process 1 for the writes to the small array. First consider the accesses performed by process 1. Since process 1 never makes accesses outside of the small array, we know that λ_{01} is 0. Let μ be the loop's rate of execution. Process 1 accesses the small array once every time around the loop, so λ_{11} is μ. Since there are twice as many reads from the big array as writes to the small array, the total access intensity for process 0 is 2μ. Since the reads are evenly distributed across the big array, we expect that two-thirds of the reads go to region 0 and one-third go to region 1, thus λ_{00} is $\frac{4}{3}\mu$ and λ_{10} is $\frac{2}{3}\mu$. The overall hit ratio is easily computed:

$$\lambda_{00} = \tfrac{4}{3}\mu, \lambda_{01} = 0, \lambda_{10} = \tfrac{2}{3}\mu, \lambda_{11} = \mu$$

$$\lambda_0 = \tfrac{4}{3}\mu + 0 = \tfrac{4}{3}\mu$$

$$\lambda_1 = \tfrac{2}{3}\mu + \mu = \tfrac{5}{3}\mu$$

$$\lambda = \sum_{i \in R} \lambda_i = \frac{4}{3}\mu + \frac{5}{3}\mu = 3\mu$$

$$\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2 = \frac{(\frac{4}{3}\mu)^2}{\frac{4}{3}\mu} + \frac{0^2}{\frac{4}{3}\mu} + \frac{(\frac{2}{3}\mu)^2}{\frac{5}{3}\mu} + \frac{\mu^2}{\frac{5}{3}\mu} = \frac{33}{15}\mu$$

$$\text{Overall cache hit ratio} = \tfrac{\eta}{\lambda} = \frac{\frac{33}{15}\mu}{3\mu} = \tfrac{11}{15} \cong 73\%$$

While this example is simple, the next section shows how collective analysis can be used to predict the performance of more complex systems.

4 An Analytical Evaluation of Heaps

In this section, we first describe the basic implicit heap and our optimizations to this heap. We then present a collective analysis of both the unoptimized and optimized heaps. Since we are modeling the memory behavior of heaps using a set of independent processes, out collective analysis only approximates the heap's actual cache performance. Accordingly, we validate our analysis by comparing the predictions of collective analysis with trace-driven cache simulation results.

We use a implicit binary heap as our base. Throughout the paper we refer to this as a traditional heap. It is an array representation of a complete binary tree with N elements. All levels of the tree are completely full except for the lowest, which is filled from left to right. The tree has depth $\lceil \log_2(N+1) \rceil$. Each element i in the heap has a key value $Key[i]$ and optionally some associated data. The N heap elements are stored in array elements $0 \cdots N-1$, the root is in position 0, and the elements have the following relationships

$$Parent(i) = \lfloor \tfrac{i-1}{2} \rfloor, \text{ if } i > 0,$$

$$LeftChild(i) = 2i + 1,$$

$$RightChild(i) = 2i + 2.$$

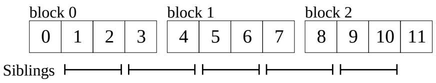

Figure 3: This typical layout of a 2-heap is not cache block aligned and siblings span cache blocks.

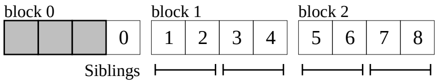

Figure 4: Starting this 2-heap on the last element of a cache block guarantees that siblings do not span cache blocks.

A heap must satisfy the *heap property*, which says that for all elements i except the root, $Key[Parent(i)] \leq Key[i]$. It follows that the minimum key in the data structure must be at the root.

In this paper, we use heaps as priority queues that provide the *add* and *remove-min* operations. In our implementation of *add*, the element is added to the end of the array and is then percolated up the tree until the heap property is restored. In our *remove-min*, the root is replaced with the last element in the array which is percolated down the tree by swapping it with the minimum of its children until the heap property is restored. In this paper, we do not consider the implication nor the optimization of other priority queue operations such as *reduce-min* or the merging of two priority queues. Heaps have been well studied and there are numerous extensions and more sophisticated algorithms for adding and removing [27, 13, 7, 21, 19].

4.1 Optimizing *Remove-min*

For a stream of operations on a heap where the number of adds and the number of removes are roughly equal, the work performed will be dominated by the cost of the removes. Doberkat [14] showed that independent of N, if the keys are uniformly distributed, *add* operations percolate the new item up only 1.6 levels on average. Doberkat [15] also studied the cost of the *remove-min* operation on a heap chosen at random from the set of legal heaps. He showed that after *remove-min* swaps the last element to the root, the swapped element is percolated down more that $(depth - 1)$ levels on average. Accordingly, we focus on reducing the cost of the *remove-min* operation.

Our first observation about the memory behavior of *remove-min* is that we do not want siblings to span cache blocks. Recall that *remove-min* moves down the tree by finding the minimum of a pair of unsorted siblings and swapping that child with the parent. Since both children need to be loaded into the cache, we would like to guarantee that both children are in the same cache block.

Figure 3 shows the way a heap will lay in the cache if four heap elements fit in a cache block and the heap starts at the beginning of a cache block. Unfortunately, in this configuration, half of the sets of siblings span a cache block. Similar behavior occurs for other cache configurations where the cache block size is an even multiple of heap element size and the memory allocated for the heap starts a cache block.

There is a straightforward solution to this problem, namely to pad the array so that the heap does not start a cache block. Figure 4 shows this padding and its effect on the layout of the heap in the cache. We expect this optimization to have its biggest impact when number of heap elements per cache line is two. As this represents a change in data layout only, the optimization will not change the number of instruction executed by the heap.

Our second observation about *remove-min* is that we want to fully utilize the cache blocks that are brought in from memory. Consider a *remove-min* operation that is at element 3 and is trying to decide between element 7 and element 8 as its minimum child. If a cache miss occurs while looking at element 7, we will have to bring the block containing elements 5-8 into the cache. Unfortunately, we will only look at two of the four elements in this cache block to find the minimum child. We are only using two elements even though we paid to bring in four.

As before, there is a straightforward solution to the problem, and the inefficiency can be eliminated by increasing the fanout of the heap so that a set of siblings fills a cache block. The *remove-min* operation will no longer load elements into the cache and not look at them. A d-heap is the generalization of a heap with fanout d rather than two. The d-heap was first introduced by Johnson as a way to reduce the cost of the *Reduce-Min* operation [24]. Naor *et al.* suggest using d-heaps with large d as a way to reduce the number of page faults that heaps incur in a virtual memory environment [32]. Figure 5 shows a 4-heap and the way it lays in the cache when four elements fit per cache block.

Unlike our previous optimization, this change will definitely have an impact on the dynamic instruction count of our heap operations. The *add* operation should strictly benefit from an increased fanout. *Add*s percolate an element from the bottom up and only look at one element per heap level. Therefore the shorter tree that results from a larger fanout will cause the *add* operation to execute fewer instructions. The instruction count of the *remove-min* operation, on the other hand, can be increased by this change. In the limit, as d grows large, the heap turns into an unsorted array that requires a linear number of comparisons. Recall that the *remove-min* operation moves the last element of the array to the root and then for each level finds the minimum of the d children and swaps this smallest element with its parent. Since the children are stored in an unsorted manner, d comparisons must be performed to find the minimum child and compare it to the parent. The cost of a swap can vary depending on how much data is stored with each element. We give the swap a cost of a relative to the cost of a comparison. Thus, the total cost at each level is $d + a$. We calculate the total cost as $d + a$ multiplied by the number of levels traversed. In our analysis, we assume that the tail element is always percolated back down to the lowest level in the heap. The total expected cost for *remove-min* counting swaps and comparisons is

$$(d+a)\log_d((d-1)N+1) \approx (d+a)\log_d(dN) =$$

$$(d+a)\log_d N + (d+a) = \frac{(d+a)}{\log_2 d}\log_2 N + d + a$$

We can see that for large N, the *remove-min* cost is proportional to $\log_2 N$ by a factor of $(d+a)/\log_2 d$. This expression shows that increasing d will increase the time spent searching for the minimum child ($d/\log_2 d$). Increasing d also reduces the total cost of the swaps ($a/\log_2 d$). Figure 6 shows a graph of $(d+a)/\log_2 d$ for various values of a. For *remove-min* operations with a swap cost of at least one comparison, we see that increasing fanout actually reduces the total cost initially. For a swap cost of two comparisons, the *remove-min* cost is reduced by a quarter by changing the fanout from two to four and does not grow to its initial level until the fanout is larger than twelve. Thus we expect that as long as fanouts are kept small, instruction counts should be the same or better than for heaps with fanout two.

This graph also helps to point out the dangers of an analysis that only counts one type of operation. This graph clearly shows that even if we do not consider caching, a heap with fanout four should perform better than a heap with fanout two. This would not be evident however, if we were only to consider the number of comparisons performed, as is commonly done. The curve on the graph which has swap cost of zero is the equivalent of only counting comparisons. This curve does not show amortization of swap costs and suggests the misleading conclusion that larger fanouts are not beneficial [1].

4.2 Collective Analysis of Cache-Aligned d-heaps

We first perform collective analysis on d-heaps whose sets of siblings are cache aligned. A d-heap with N elements has depth $\lceil \log_d((d-1)N+1) \rceil$. $Parent(i) = \lfloor \frac{i-1}{d} \rfloor$ and $Children(i) = di+1, di+2, \cdots, di+d$. Let e be the size in bytes of each heap element.

In this analysis, we restrict fanout d to be a positive power of 2 and element size e to be a power of 2. We also restrict our heap configurations to those in which all of a parent's children fit in a single cache block (where $de \leq B$). This limits the values of d that we are looking at; for a typical cache block size of 32 bytes, fanout is limited to 4 for 8 byte heap elements, and fanout is limited to 8 for 4 byte heap elements. We also restrict our analysis to heap configurations in which the bottom layer of the heap is completely full (where $\lceil \log_d((d-1)N+1) \rceil = \log_d((d-1)N+1)$).

Heaps are often used in discrete event simulations as a priority queue to store the events. In order to measure the performance of heaps operating as an event queue, we analyze our heaps in the hold model [25]. In the hold model,

[1] It has been noticed previously that increasing a heap's fanout can reduce the instruction count of its operations [27, Ex. 28 Pg. 158][12, Ex. 7-2 Pg. 152].

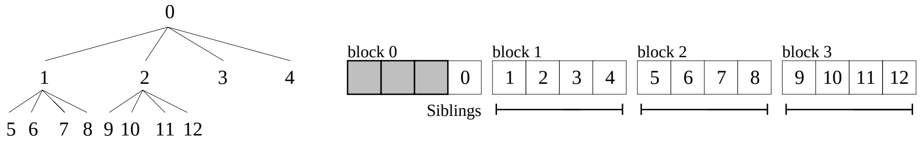

Figure 5: The layout of a 4-heap when four elements fit per cache line and the array is padded to cache-align the heap.

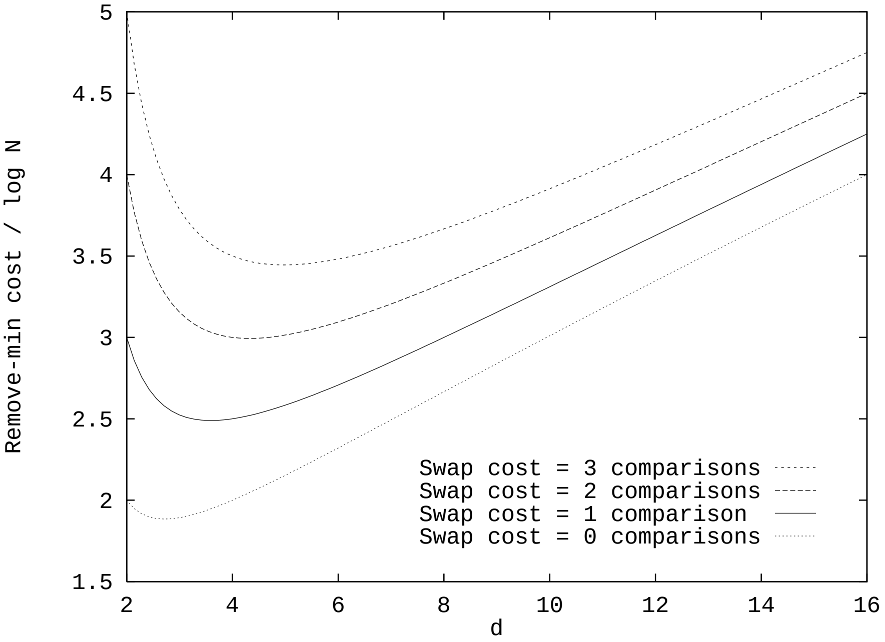

Figure 6: The cost of *remove-min* as a function of d for a number of comparison-to-swap-cost ratios.

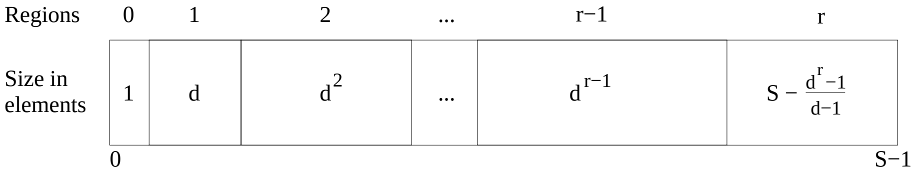

Figure 7: The division of the cache into regions for the d-heap.

the heap is initially seeded with some number of keys. The system then loops repeatedly, each time removing the minimum key from the heap, optionally performing some other work, and finally adding a random positive value to the element's key and inserting it back into to the heap. A random increment chosen from a distribution such as the uniform or exponential assures a proper mixing of the elements in the heap. This use of heaps is called the hold model because the size of the heap holds constant over the course of the run. In this analysis, the work performed between the *remove-min* and the *add* consists of w random uniformly distributed accesses to an array of size C. While this probably does not model the typical simulation event handler, it is a reasonable approximation of a work process and helps to show the cache interference between independent algorithms.

The first step of our analysis is to divide the cache into regions based on the heap's structure. Recall that we have a cache with a size of C bytes, a cache block size of B bytes, and a heap with N elements, fanout d and element size e. Let $S = \frac{C}{e}$ be the size of the cache in heap elements.

Given the heap's structure, $r = \lfloor \log_d((d-1)S+1) \rfloor$ whole levels of the heap will fit in the cache. That is to say, heap levels $0 \cdots r-1$ fit in the cache and heap level r is the first level to spill over and wrap around in the cache. We divide the cache into $r+1$ regions, where regions $0 \cdots r-1$ are the same size as the first r heap levels and region r takes up the remaining space in the cache. Region i, $0 \leq i < r$, is of size d^i and region r is of size $S - \frac{d^r-1}{d-1}$. In our analysis of d-heaps, the regions are all contiguous; Figure 7 shows the division. Let S_i be the size of cache region i.

The next step is to partition the system into processes which approximate its memory access behavior. It is at this point that we make three simplifying assumptions about the behavior of the *remove-min* operation. We first simplify the percolating down of the tail element by assuming that all levels of the heap are accessed independently, once per *remove-min* on average. We also assume that when a heap level is accessed by *remove-min*, all sets of siblings are equally likely to be searched through for the minimum. These simplifications enable a collective analysis to be performed on a data-structure with a behavior considerable more complex than our simple example. At the same time, these simplifications were carefully chosen to retain accuracy. While much of the *remove-min* algorithm's behavior has been lost, the rates of accesses and overall access distributions should still be reasonably accurate.

To further simplify the heap's behavior, we make a final assumption regarding the caching of elements. In the event that a set of siblings are brought into the cache on a miss, other sets of siblings may be read in at the same time (if $de < B$). The result is that a reference to a set of siblings may not incur a fault even though the set has never before been accessed. In our analysis, we ignore this effect and assume that neighboring sets of siblings are never faulted in. We again believe that this assumption will not significantly affect the accuracy of the analysis in practice.

The basic structure of our decomposition is to create one or more processes for each level in the heap. Given a total of N elements, there are $t = \log_d((d-1)N+1)$ levels in the heap. Let N_i be the size in elements of heap level i. We begin by dividing the t heap levels into two groups. The first group contains the first r levels $0 \cdots r-1$ of the heap that will fit in the cache without any overlap. The second group contains the remaining levels of the heap $r \cdots t-1$. For $0 \leq i < t$, $N_i = d^i$.

Lemma 1 *The size of heap level i, $r \leq i < t$, is a multiple of S.*

Proof: Since $N_i = d^i$, it is sufficient to prove that $d^i \bmod S = 0$. Since d is a positive power of 2, and C and e are powers of 2, and $e < B \leq C$, both d^i and S are also powers of 2. It is sufficient to show that $d^i \geq S$. Let $d = 2^x$ where $x > 0$ and $S = 2^y$.

$$d^i = d^{i-r}d^r \geq d^r = d^{\lfloor \log_d((d-1)S+1) \rfloor} \geq d^{\lfloor \log_d(\frac{dS}{2}) \rfloor} =$$

$$2^{x \lfloor \log_{2^x}(2^{x+y-1}) \rfloor} = 2^{x \lfloor \frac{x+y-1}{x} \rfloor} \geq 2^{x(\frac{x+y-1}{x} - \frac{x-1}{x})} = 2^y = S.$$

■

For each heap level i in the first group, we create a process i, giving us processes $0 \cdots r-1$. For the second group we create a family of processes for each heap level, and each family will have one process for each cache-sized piece of the heap level. For heap level i, $r \leq i < t$, we create $\frac{N_i}{S}$ processes called $(i, 0) \cdots (i, \frac{N_i}{S} - 1)$.

Finally, we create a process to model the random reads that occur between the *remove-min* and the *add*. We call this process o.

If *remove-min* is performed on the heap at an intensity of μ our access intensities are

$$\lambda_{ij} = \begin{cases} \mu & \text{if } 0 \leq j < r \text{ and } i = j, & (2) \\ \frac{S_i}{N_x}\mu & \text{if } j = (x, y) \text{ and } r \leq x < t \text{ and } 0 \leq y < \frac{N_x}{S}, & (3) \\ \frac{S_i}{S}w\mu & \text{if } j = o \text{ and } 0 \leq i \leq r, & (4) \\ 0 & \text{otherwise.} & (5) \end{cases}$$

In our simplified *remove-min* operation, heap levels are accessed once per *remove-min* on average, and each access to a heap level touches one cache block. An access by process j, $0 \leq j < r$, represents an access to heap level j. We know that all accesses by process j will be in cache region j, and process j makes no accesses outside of cache region j. Thus, with a *remove-min* intensity of μ, process j, $0 \leq j < r$, will access region i with an intensity of μ if $i = j$ and 0 otherwise (Equations 2 and 5).

Next, consider a process (x, y) where $r \leq x < t$. This process represents one of $\frac{N_x}{S}$ cache-sized pieces from heap level x. We expect that one cache block will be accessed from heap level x per *remove-min*. The chance that the block accessed belongs to the process in question is $\frac{S}{N_x}$. The total access intensity of the process is the access intensity of the level multiplied by the chance that an access belongs to the process, or $\frac{S}{N_x}\mu$. Since the process maps a piece of the heap exactly the size of the cache, it is easy to calculate the access intensities for each region. Since the accesses of process (x, y) are uniformly distributed across the cache, the access intensities for each region will be proportional to its size. Given that the process's access intensity is $\frac{S}{N_x}\mu$, the access intensity of process (x, y) where $r \leq x < t$ in cache region i is $\frac{S}{N_x}\mu\frac{S_i}{S} = \frac{S_i}{N_x}\mu$ (Equation 3).

Given that the system iterates at a rate of μ, the total access intensity of process o is $w\mu$. Since process o accesses an array of size C uniformly, we expect the accesses to spread over each cache region proportional to its size. Thus, the access intensity of process o in cache region i is $\frac{S_i}{S}w\mu$ (Equation 4).

The region intensities λ_i and the overall intensity are easily calculated:

$$\lambda_i = \begin{cases} \mu + \frac{S_i}{S}(t - r + w)\mu & \text{if } 0 \leq i < r, \\ \\ \frac{S_r}{S}(t - r + w)\mu & \text{if } i = r, \end{cases}$$

$$\lambda = (t + w)\mu.$$

An expression for the hit intensity η can be derived from Equation 1. The sum across regions is broken into $0 \cdots r-1$ and r, and the sum across processes is broken up based on the two groups of heap levels. Plugging in our access intensities and reducing gives us

$$\begin{aligned}
\eta &= \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2 \\
&= \sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(\sum_{j=0}^{r-1} \lambda_{ij}^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \lambda_{i(x,y)}^2 + \lambda_{io}^2 \right) + \frac{1}{\lambda_r} \left(\sum_{j=0}^{r-1} \lambda_{rj}^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \lambda_{r(x,y)}^2 + \lambda_{ro}^2 \right) \\
&= \sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(\mu^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} (\frac{S_i}{N_x}\mu)^2 + (\frac{S_i}{S}w\mu)^2 \right) + \frac{1}{\lambda_r} \left(\sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} (\frac{S^r}{N_x}\mu)^2 + (\frac{S_r}{S}w\mu)^2 \right) \\
&= \left(\sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(1 + \frac{S_i^2}{S}(\frac{d^{1-t} - d^{1-r}}{1-d}) + (\frac{S_i}{S}w)^2 \right) + \frac{1}{\lambda_r} \left(\frac{S_r^2}{S}(\frac{d^{1-t} - d^{1-r}}{1-d}) + (\frac{S_r}{S}w)^2 \right) \right) \mu^2 \\
&= \left(\sum_{i=0}^{r-1} \frac{1 + \frac{S_i^2}{S}(\frac{d^{1-t} - d^{1-r}}{1-d} + \frac{w^2}{S})}{1 + \frac{S_i}{S}(t - r + w)} + \frac{S_r(\frac{d^{1-t} - d^{1-r}}{1-d} + \frac{w^2}{S})}{t - r + w} \right) \mu \qquad (6)
\end{aligned}$$

4.3 Collective Analysis of Unaligned d-heaps

The cache performance of a d-heap in which sets of siblings are not cache block aligned can be predicted with a simple change to the cache-aligned analysis. In our cache-aligned analysis, we correctly assume that examining a set of siblings will touch one cache block. In the unaligned case this is not necessarily true. A set of siblings uses de bytes of memory. On average, the chance that a set of siblings spans a cache block is $\frac{de}{B}$. In the event that the siblings do span a cache block, a second block will need to be touched. Thus on average, we expect $1 + \frac{de}{B}$ cache blocks to be touched examining a set of siblings. This simple change yields the following new intensities

$$\lambda_{ij} = \begin{cases} (1+\frac{de}{B})\mu & \text{if } 0 \le j < r \text{ and } i = j, \\ \frac{S_i}{N_x}(1+\frac{de}{B})\mu & \text{if } j = (x,y) \text{ and } r \le x < t \text{ and } 0 \le y < \frac{N_x}{S}, \\ \frac{S_i}{S} w\mu & \text{if } j = o, \\ 0 & \text{otherwise.} \end{cases}$$

$$\lambda_i = \begin{cases} (1+\frac{de}{B})\mu + \frac{S_i}{S}(t-r)(1+\frac{de}{B})\mu + \frac{S_i}{S} w\mu & \text{if } 0 \le i < r, \\ \frac{S_r}{S}(t-r)(1+\frac{de}{B})\mu + \frac{S_r}{S} w\mu & \text{if } i = r. \end{cases}$$

$$\lambda = t(1+\tfrac{de}{B})\mu + w\mu$$

Similar to the derivation of Equation 6, an expression for η can be derived by substituting these intensities into Equation 1 and reducing.

4.4 Experimental Apparatus

Throughout this paper, both our dynamic instruction counts and our cache simulation results were measured using Atom [37]. Atom is a flexible toolkit developed by DEC for instrumenting codes on Alpha workstations. Dynamic instruction counts were obtained by using Atom to insert an increment to an instruction counter after each instruction executed by the algorithm. We also used Atom to generate address traces and simulate the cache performance of our test programs. Cache performance was measured by using Atom to insert calls after every load and store to compute statistics and maintain the state of the simulated cache. The execution times for our test programs were measured using the Alpha processor's cycle timer. For all of the execution times reported in the paper, data points represent the median of 15 executions.

4.5 Validation of Collective Analysis

In order to validate our analysis, we compared the analytical results with the output of cache simulations of programs that exercise the appropriate heap implementations. When testing heaps in the hold model we used a uniform random increment and varying amounts of outside work. In each run, the mean of the uniform increment distribution was 40 times the number of elements in the heap. Choosing a mean proportional to the number of events in the queue caused the simulation time to flow at the same expected rate in all runs, regardless of the number of events. This both kept events sufficiently spaced in simulations with a large number of events and prevented time-code overflow in simulations with a small number of event. Our simulations were run for 1,000,000 iterations and were then measured for the 200,000 iterations in order to measure the steady state behavior. In both the analysis and the traced executions, we set the cache size equal to 2 megabytes, the cache block size to 32 bytes and the heap element size to 4 bytes. In this configuration, a cache block holds 8 heap elements.

The quantity we compare is miss intensity ($\lambda - \eta$). Miss intensity is an interesting measure, as it tell us how many times an operation must service cache misses. We compare the model's predictions with the number of misses per iteration observed by the cache simulator. Figure 8 shows this comparison for an unaligned 2-heap and an aligned 2, 4 and 8-heap with $w = 0$ and a range of N between 1,000 and 8,192,000. Since we do not consider startup costs and since no work is performed between the *remove-min* and the *add*, we do not predict or measure any cache misses until the size of the heap grows larger than the cache. This graph shows that the miss intensity predicted by the collective analysis agrees with the results of the cache simulations surprisingly well considering the simplicity of our model. For

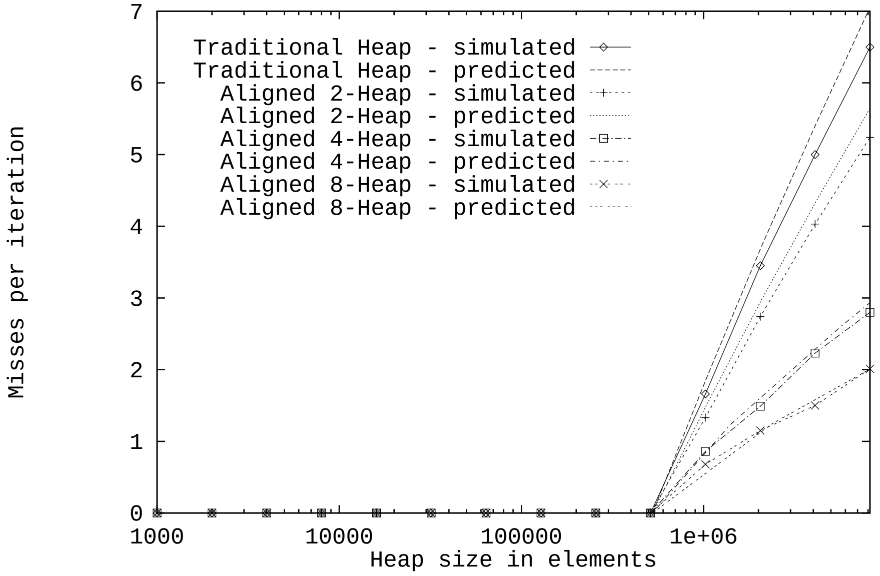

Figure 8: Cache misses per iteration for the hold model with $w = 0$ predicted by collective analysis and measured with trace-driven simulation. Caches are direct-mapped with 2048K capacity and a 32 byte block size. Heap element size is 4 bytes.

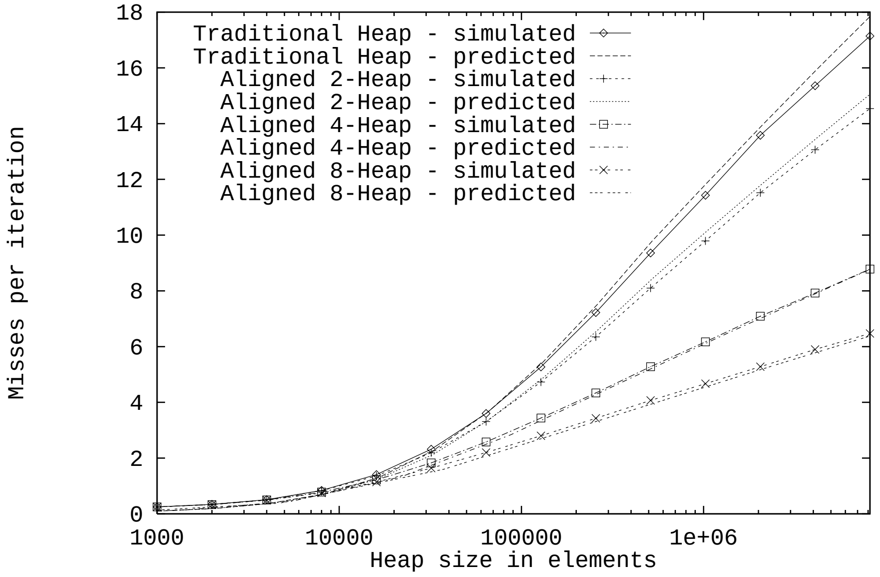

Figure 9: Cache misses per iteration for the hold model with $w = 25$ predicted by collective analysis and measured with trace-driven simulation. Caches are direct-mapped with 2048K capacity and a 32 byte block size. Heap element size is 4 bytes.

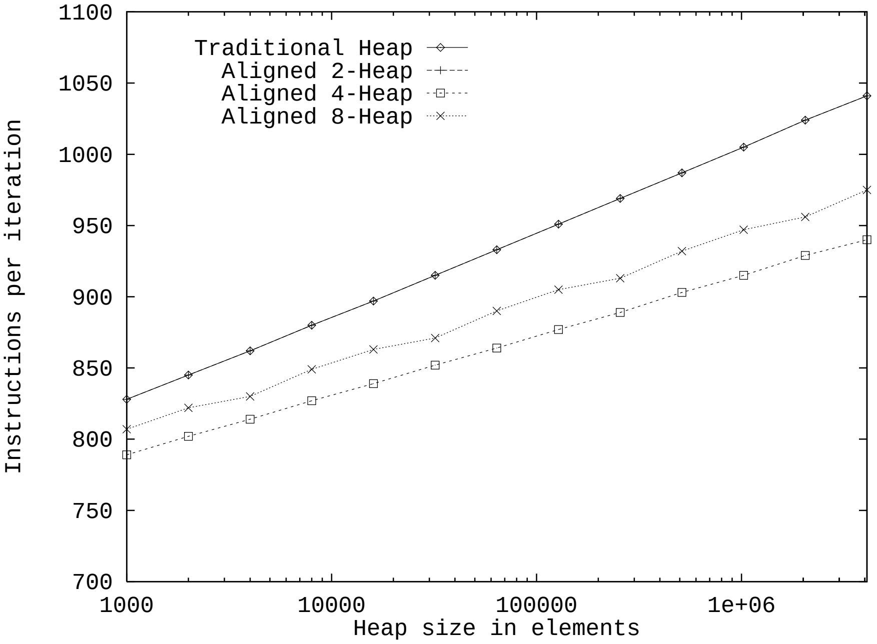

Figure 10: Dynamic instruction count for a variety of heaps running in the hold model ($w = 25$).

now we focus on the accuracy of our predictions and comments regarding the effects our optimizations have on cache misses are reserved for the next section.

We next compare our model predictions against simulations for a system in which 25 random memory accesses are performed each iteration. Figure 9 shows the miss intensities for an unaligned 2-heap and an aligned 2, 4 and 8-heap with $w = 25$ and a range of N between 1,000 and 8,192,000. Again, we see that the predictions of the collective analysis closely match the simulation results. Unlike the heap operating alone, this system incurs cache misses even when the heap is smaller than the cache due to interactions with the work process.

5 An Experimental Evaluation of Heaps

In this section we present performance data collected for our improved heaps. Our executions were run on a DEC Alphastation 250. Dynamic instruction counts and cache performance data was collected using Atom [37] (see Section 4.4). In all cases, the simulated cache was configured as direct-mapped with a total capacity of two megabytes with a 32 byte block size, the same as the second-level cache of the Alphastation 250.

5.1 Heaps in the Hold Model

Figures 10-12 show the dynamic instruction count, cache misses, and execution time (in cycles) for an unaligned 2-heap and aligned 2,4 and 8-heap running in the hold model with uniform random increment and outside work consisting of 25 random accesses to memory ($w = 25$). Our simulations were run for 1,000,000 iterations and were then measured for the 200,000 iterations. In these experiments, the heap size is varied from 1,000 to 8,192,000, and eight heap elements fit per cache block.

Figure 10 shows that the traditional heap and aligned 2-heap execute the same number of instructions, since the only difference is the layout of data. Changing the fanout of the heap from two to four provides a sizable reduction in instruction count as Figure 6 predicted. Also as predicted, changing the fanout from four to eight increased the instruction count, but not higher than the heaps with fanout two.

Machine	Processor	Clock Rate	L1 Cache Size	L2 Cache Size
Alphastation 250	Alpha 21064A	266 Mhz	8K/8K	2048K
Pentium 90	Pentium	90 Mhz	8K/8K	256K
DEC 3000/400	Alpha 21064	133 Mhz	8K/8K	512K
Power PC	MPC601	80 Mhz	32K	512K
Sparc 20	SuperSparc	60 Mhz	20K/16K	1024K

Table 1: Clock rate and cache sizes of the machines on which we tested our heaps.

Figure 11 shows the number of cache misses incurred per iteration in the hold model for our heaps when eight heap elements fit in a cache block. This graph shows that cache aligning the traditional heap reduces the cache misses by around 15% for this cache configuration. Increasing the heap fanout from two to four provides a large reduction in cache misses, and increasing from four to eight reduces the misses further still. This graph serves as a good indication that our optimizations provide a significant improvement in the memory system performance of heaps.

Figure 12 shows the execution time for the heaps in the hold model when eight heap elements fit in a cache block and $w = 25$. This graph looks very much like a simple combination of the previous two graphs. Aligning the traditional heap provides a small reduction in execution time. Increasing the fanout from two to four provides a large reduction in execution time due to both lower instruction count and fewer cache misses. The 8-heap executes more instructions that the 4-heap and consequently executes slower initially. Eventually, however, the reduction in cache misses overcomes the difference in instruction count, and the 8-heap performs best for large data sets.

5.2 Heapsort

We next examine the effects our improvements have on heapsort [42]. We compare heapsort built from traditional heaps and our aligned d-heaps. The traditional heaps were built using Floyd's method. Our aligned heaps are built using either Floyd's method or Repeated-Adds; the choice is made dynamically at runtime depending on the heap size and fanout in order to maximize performance.

Figure 13 shows the execution time of our four heapsorts running on an DEC Alphastation 250 sorting uniformly distributed 32 bit integers. In our heapsort executions, all three of the cache-aligned heaps outperform the traditional heap and the majority of the speedup occurs in the *remove-min* operation. As before, we see that increasing fanout from 2 to 4 provides a large performance gain. Again we see that the instruction count overhead of the 8-heap is overcome by the reduced cache misses and the 8-heap performs best for larger heap sizes. The 8-heap sorting 8,192,000 numbers nearly doubles the performance of the traditional binary heap.

5.3 Generality of Results

So far all our data has been presented for the DEC Alphastation 250, and all of our simulations have been run for two megabyte caches. In order to demonstrate the general applicability of our optimizations across modern architectures, we now present data for our heaps running in the hold model on four machines other than the Alphastation. As before, our heaps were warmed up for 1,000,000 iterations and were measured for 200,000. Figure 14 shows the speedup of our 4-heap over the traditional heap running in the hold model with $w = 0$ with 8 byte heap elements. As our four additional machines we chose the Sparc 20, the IBM Power PC, the DEC Alpha 3000/400 and a Pentium 90 PC. The clock rates of the machines tested ranged from 60 to 266 megahertz, and the cache sizes ranged from 256K to 2024K. The second-level caches on all of these machines are direct mapped, as are most of the first-level caches. Table 1 shows the processor, clock rate and cache sizes for the machines we tested. Figure 14 shows that despite the differences in architecture, our 4-heaps consistently achieve good speedups that increase with the size of the heap.

Both the Power PC and the Pentium speedup curves begin to climb earlier than we might expect. Since no work is performed between iterations, we expect few second-level cache misses to occur until the heap is larger than the cache. Recall, however, that this expectation is based on the assumption that data contiguous in the virtual address space is

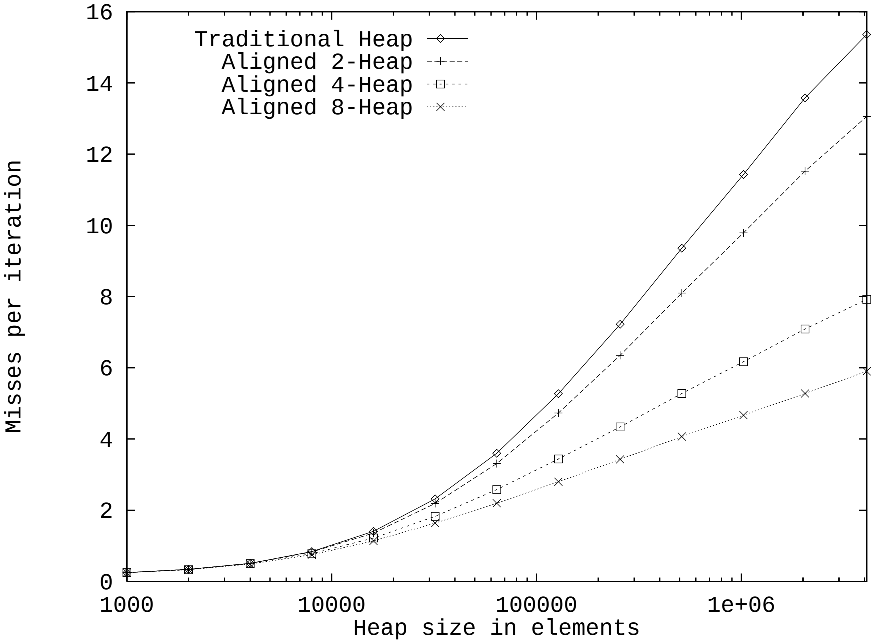

Figure 11: Cache misses per iteration for a variety of heaps running in the hold model ($w = 25$). Simulated cache was 2048K in capacity with a 32 byte block size. Heap element size is 4 bytes.

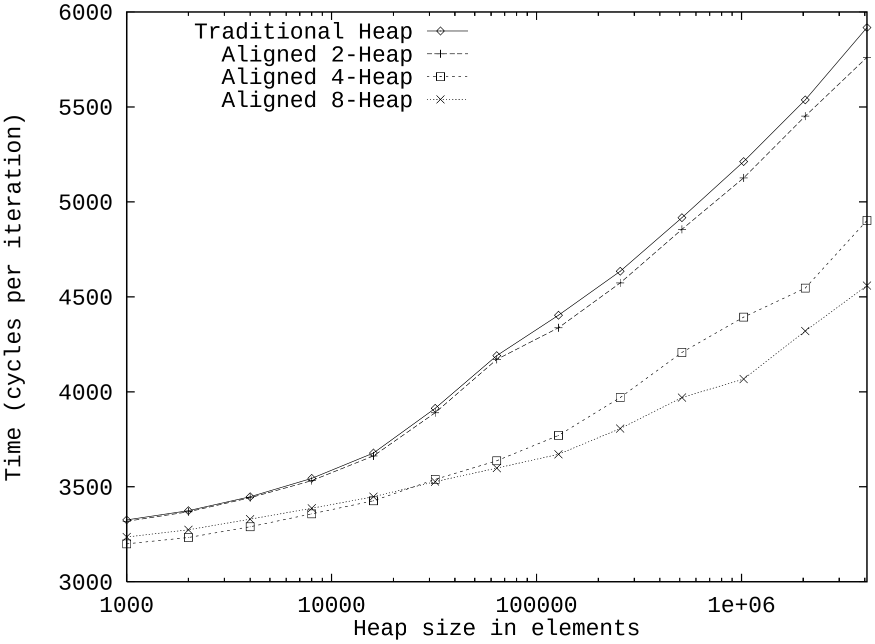

Figure 12: Cycles per iteration on a DEC Alphastation 250 for a variety of heaps running in the hold model ($w = 25$). Heap element size is 4 bytes.

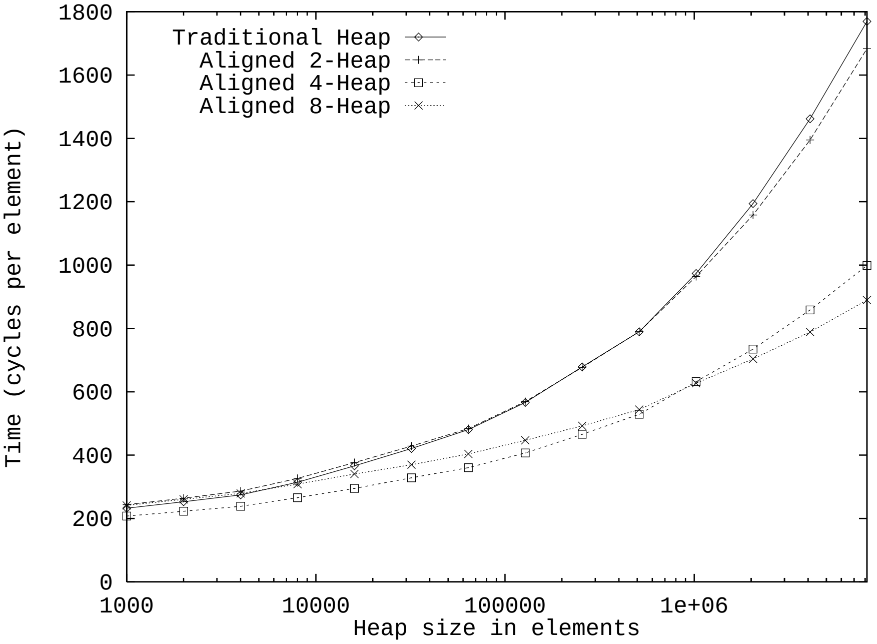

Figure 13: Cycles per iteration on a DEC Alphastation 250 for a variety of heapsort algorithms sorting uniformly distributed 32 bit integers.

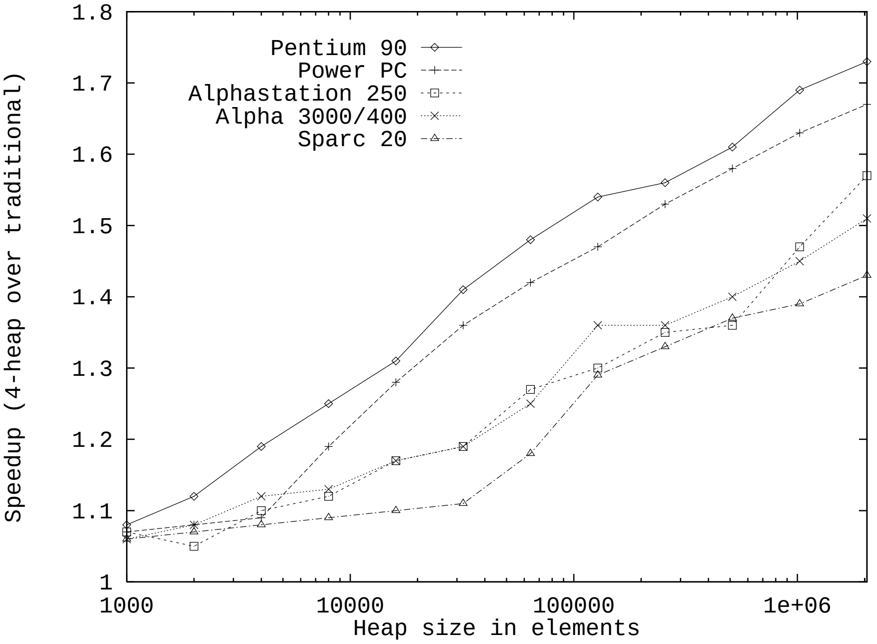

Figure 14: The speedup of an aligned 4-heap over the traditional heap running in the hold model with $w = 0$ and 8 byte heap elements for a number of architectures.

always contiguous in the cache. This assumption is not true for any of these five machines, all of whose second-level caches are physically indexed. It is likely that for the Power PC and the Pentium, the operating systems (AIX and Linux respectively) are allocating pages that conflict in the cache and that this is causing second-level cache misses to occur earlier than we expect. These cache misses provide an earlier opportunity for the 4-heap to reduce misses and account for the increased speedup. These differences might also be due in part to differences in the number of first-level cache misses. This would not explain all of the difference however, as the first level miss penalties in these cases are small.

6 A Comparison of Priority Queues

In this section, we compare a number of priority queues running in the hold model. Our experiments are intended to be a reproduction of a subset of the priority queue study comparison performed by Jones in 1986 [25]. The results of Jones's experiments indicated that for the architecture of that time, pointer-based, self-balancing priority queues such as splay trees and skew heaps performed better than the simple implicit heaps. The changes that have occurred in architecture since then, however, have shifted the priorities in algorithm design, and our conclusions are somewhat different than those drawn by Jones.

6.1 The Experiment

We examined the performance of five priority queue implementations operating in the hold model. In these experiments we use an exponential random increment rather than a uniform increment in order to match the distribution used by Jones [25]. As before, the mean of the increment distribution was set to 40 times the number of events in the queue (see Section 4.5). In our experiments we compared a traditional heap, a cache-aligned 4-heap, a top-down skew heap and both a top-down and a bottom-up splay tree. The implementations of the traditional heap and the aligned 4-heap were those described earlier in this paper. The implementations of the skew heap and the bottom-up splay tree were taken from an archive of the code used in Jones's study. The top-down splay tree implementation is an adoption of Sleator and Tarjan's code [36]. As Jones did in his study, the queues were run in the hold model, and no work was performed between the *remove-min* and the *add* of each iteration ($w = 0$). In our experiments, a queue element consisted of an 8 byte key and no data. This varies slightly from Jones's experiments where he used 4 byte keys. A larger key value was chosen to allow extended runs without the keys overflowing.

In our runs, the priority queues were initially seeded with exponentially distributed keys and the priority queues were run in the hold model for 3,000,000 iterations to allow the queue to reach steady state. The queue was then measured for 200,000 iterations. As before, executions were run on a DEC Alphastation 250, and cache simulations were configured for a two megabyte direct-mapped cache and a 32 byte block size. Dynamic instruction counts and cache performance data was collected using Atom [37] (see Section 4.4). Execution times were collected with the Alpha's cycle timer and represent the median of 15 executions.

In our experiments, the queue size was varied from 1,000 to 1,024,000. (The bottom-up splay tree was only run up to a size of 512,000 elements due to physical memory pressure.) This differs from Jones study where he used considerably smaller queues, ranging from size 1 to 11,000.

6.2 Results

Figure 15 shows a graph of the dynamic instruction count of the five priority queues. As we expect, the number of instructions executed per iteration grows with the log of the number of elements for all five of the priority queues. The 4-heap executed around 25% fewer instructions than the traditional heap as Figure 6 predicts for a swap cost of two comparisons. The bottom-up splay tree performed almost exactly the same number of instructions as the traditional heap. This somewhat contradicts Jones's findings in which the splay tree far outperformed the heap. This difference is not surprising since Jones's experiments were run on a CISC machine offering a host of memory-memory instructions while ours were run on a load-store RISC machine. Splay trees and skew heaps are extremely memory intensive. The bottom-up splay tree and the traditional heap executed roughly the same number of instructions in our study, yet the splay tree executed almost three times as many loads and stores as the heap. The pointer manipulations performed by the self-balancing queues were fairly inexpensive on the architectures that Jones used. On our load-store RISC machine, however, these pointer manipulations translate into a larger instruction count, slowing down the bottom-up splay tree relative to the heaps.

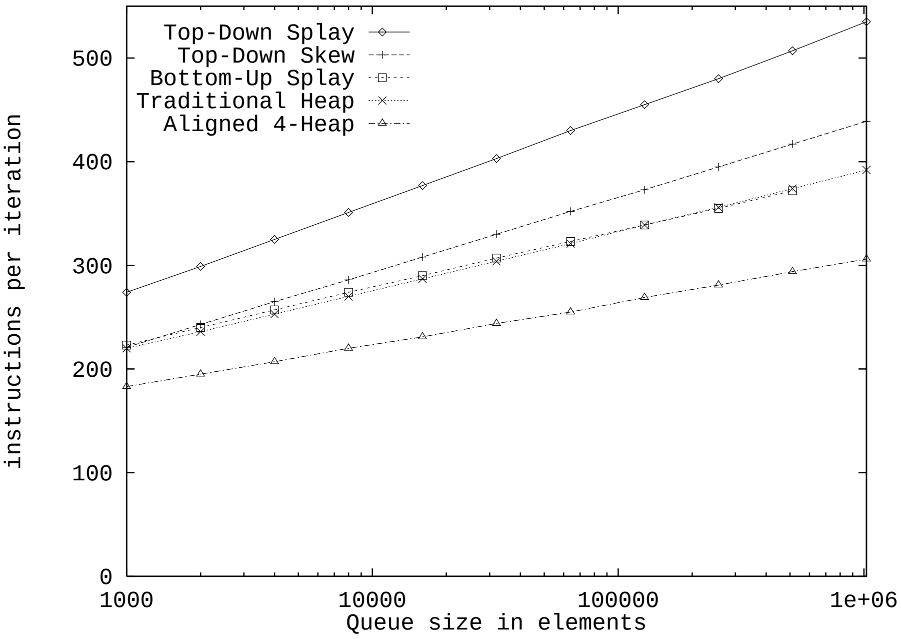

Figure 15: Instruction count per iteration of five priority queues running in the hold model with $w = 0$ and 8 byte keys.

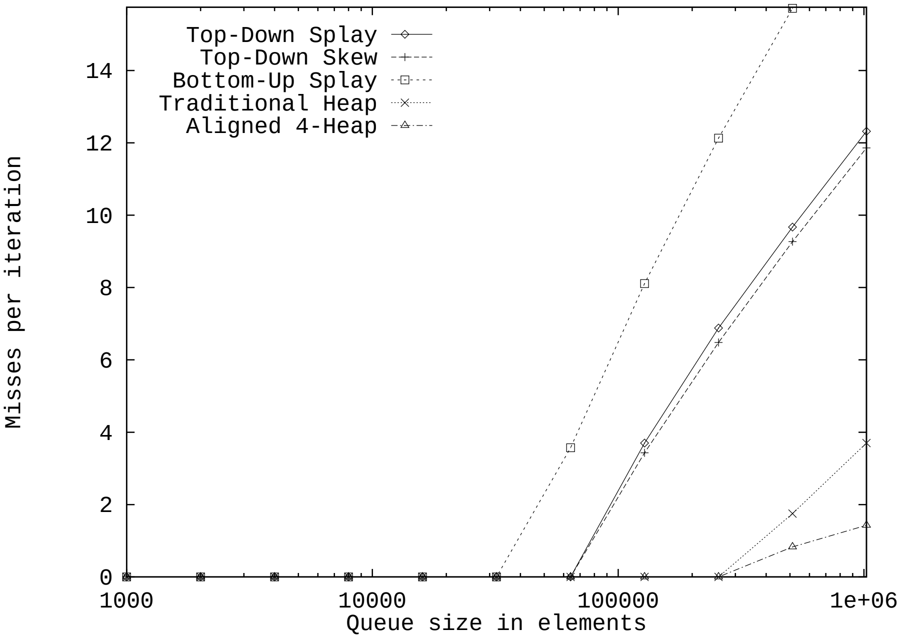

Figure 16: Cache misses per iteration of five priority queues running in the hold model with $w = 0$ and 8 byte keys. Simulated cache is 2048K in capacity with a 32 byte block size.

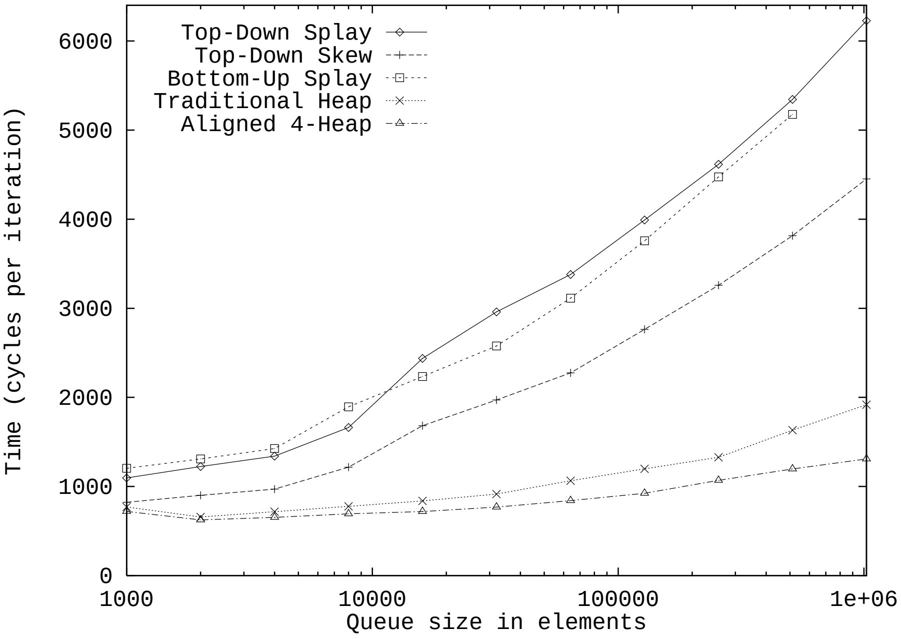

Figure 17: Cycles per iteration of five priority queues running in the hold model with $w = 0$ and 8 byte keys.

The top-down splay tree performed the most instructions by a wide margin. To be fair, we must say that while the top-down splay trees were coded for efficiency, they did not receive the heavy optimization that Jones gave his codes or that we gave our heaps.

The cache performance of the priority queues is compared in Figure 16. As the experiments were run in the hold model with no work between iterations, cache misses do not occur after warmup unless the queue is larger than the cache. It is at this point that an important difference in the priority queues is revealed. Depending on their algorithms for adding and removing elements, the queues have different sized elements and this has a huge impact on cache performance. Adding pointers to a queue element increases its size reducing the number of elements that fit in the cache which in turn reduces locality.

The implicit heap elements require no pointers and the elements only contain the keys. The top-down skew heap and top-down splay tree, on the other hand, both require a left and right pointer per element, adding 16 bytes of overhead to each queue element. In addition, queue elements for the pointer-based queues are allocated from the system memory pool, and 8 bytes of overhead are incurred each time an element is allocated[2]. The pointer overhead combined with the overhead of the system memory pool results in an element size of 8 (key) + 8 (left) + 8 (right) + 8 (memory pool) = 32 bytes, four times larger than a heap element. As a result, the top-down skew heaps and top-down splay trees start incurring cache misses at one-quarter the queue size of the heaps. The queue requiring the most overhead is the bottom-up splay tree with three pointers per element (left, right and parent). The result is that the splay tree has the largest footprint in memory and is the first to incur cache misses as the queue size is increased.

The execution times for the five queues executing on an Alphastation 250 are shown in Figure 17. Due to their low instruction count and small cache miss count, our aligned 4-heap performs best, with the traditional heap finishing second. This varies significantly from Jones's findings in which the implicit heaps finished worse than splay trees and skew heaps. While there are many architectural differences between the Vax used by Jones and the Alphastation we used, our results strongly suggest that cache performance is responsible for the bulk of the difference in execution times. Figure 17 initially shows a small performance spread across queues when they all fit in the cache, as well as showing that when cache misses increase significantly execution times also follow suit.

[2] There are memory pool implementations that do not incur per-allocation overhead for small objects [22]. In OSF/1, however, the default `malloc` incurs an 8 byte overhead per allocation.

Another interesting difference between our results and Jones's is that in our study, the skew heap outperformed the bottom-up splay tree. With only slightly higher instruction count than the traditional heap and moderate cache performance, the skew heap outperforms both of the splay trees and finishes third overall in our comparison.

Our two splay tree implementations turned in similar execution times with the higher instruction count for the top-down splay tree offsetting the higher cache miss count for the bottom-up splay tree. Despite the locality benefits splay trees afford, their large footprint in memory caused them to perform poorly in our study.

6.3 Impressions

These experiments illustrate the importance of a design that is conscious of memory overhead and the effects of caching, and strongly suggest that future designs will need to pay close attention to memory performance if good overall performance is to be achieved. The goal of this section is not to convince the reader that heaps are the best priority queue. There are obvious optimizations that could be applied to the splay trees and skew heaps which would reduce their cache misses and improve their performance. One good starting point would be to allocate queue nodes in large bundles to minimize the overhead incurred in the system memory pool. By allocating elements 1000 at a time, overhead could be reduced from 8 bytes per queue element to 8 bytes per 1000 queue elements.

An additional optimization would be to store queue nodes in an array and represent references to the left and right elements as offsets in the array rather than using 64 bit pointers. If the queue size was limited to 2^{32}, the 8 byte pointers could be replaced with 4 byte integers instead. These two optimizations alone would reduce the total overhead for the top-down skew heap to 4 (left) + 4 (right) = 8 bytes rather than 8 (left) + 8 (right) + 8 (memory pool) = 24 bytes. These simple changes would have a significant impact on the performance of all of the pointer-based priority queues used in this study. It is unlikely, however, that the splay trees or skew heaps could be made to perform as well as the implicit heaps for the applications we looked at, since the heaps have such low instruction counts and have no memory overhead due to their implicit structure.

7 Conclusions

The main conclusion of our work is that the effects of caching need to be taken into account in the design and analysis of algorithms. We presented examples that show how algorithm analysis that does not include memory system performance can lead to incorrect conclusions about performance.

We presented a study of the cache behavior of heaps and showed simple optimizations that significantly reduce cache misses, which in turn improve overall performance. For the DEC Alphastation 250, our optimizations sped up heaps running in the hold model by as much as 75% and heapsort by a factor of two. In order to demonstrate the generality of our results, we also provided empirical performance data for four additional architectures showing the impact our optimizations have on overall performance.

We also introduced collective analysis, an analytical model for predicting the cache performance of a system of algorithms. Collective analysis is intended for applications whose general behavior is known, but whose exact reference pattern is not. As the model does not perform analysis on address traces, it offers the advantage that algorithm configuration can be varied as well as cache configuration. In our framework, we performed a cache analysis of d-heaps, and our performance predictions closely match the results of a trace-driven simulation.

Finally, we reproduced a subset of Jones's experiments [25] examining the performance of a number of priority queues. Our experiments showed that the low memory overhead of implicit heaps makes them an excellent choice as a priority queue, somewhat contradicting Jones's results. We observed that the high memory overhead of the pointer-based, self-balancing queues translated into poor memory system and overall performance. We also discussed potential optimizations to these pointer-based queues to reduce their memory overhead and reduce cache misses.

Acknowledgments
We thank Ted Romer for helping us with the Atom toolkit. We also thank Mike Salisbury and Jim Fix for helpful comments on drafts of this paper. Finally, we would like to thank the anonymous referees for their suggestions for improvement.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7:2:184–215, 1989.

[2] R. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38:5:563–576, Sep 1994.

[3] A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *19th Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[4] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:2-3:72–109, 1994.

[6] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the 1993 ACM Symposium on Programming Languages Design and Implementation*, pages 112–125. ACM, 1993.

[7] S. Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters*, 37:2:117–120, 1991.

[8] S. Carr, K. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, 1994.

[9] M. Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the 1995 ACM Symposium on Programming Languages Design and Implementation*, pages 205–217. ACM, 1995.

[10] D. Clark. Cache performance of the VAX-11/780. *ACM Transactions on Computer Systems*, 1:1:24–37, 1983.

[11] E. Coffman and P. Denning. *Operating Systems Theory*. Prentice–Hall, Englewood Cliffs, NJ, 1973.

[12] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[13] J. De Graffe and W. Kosters. Expected heights in heaps. *BIT*, 32:4:570–579, 1992.

[14] E. Doberkat. Inserting a new element into a heap. *BIT*, 21:225–269, 1981.

[15] E. Doberkat. Deleting the root of a heap. *Acta Informatica*, 17:245–265, 1982.

[16] J. Dongarra, O. Brewer, J. Kohl, and S. Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9:2:185–202, June 1990.

[17] M. Farrens, G. Tyson, and A. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 338–347, 1994.

[18] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7:1:43–65, 1995.

[19] Robert W. Floyd. Treesort 3. *Communications of the ACM*, 7:12:701, 1964.

[20] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:5:587–616, Oct 1988.

[21] G. Gonnet and J. Munro. Heaps on heaps. *SIAM Journal of Computing*, 15:4:964–971, 1986.

[22] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 ACM Symposium on Programming Languages Design and Implementation*, pages 177–186. ACM, 1993.

[23] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[24] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4, 1975.

[25] D. Jones. An emperical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:4:300–311, 1986.

[26] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 323–334, 1992.

[27] D. E. Knuth. *The Art of Computer Programming, vol III – Sorting and Searching*. Addison–Wesely, Reading, MA, 1973.

[28] A. LaMarca. Caches and algorithms. Ph.D. Dissertation, University of Washington, May 1996.

[29] A. LaMarca and R.E. Ladner. The influence of caches on the performance of sorting. Technical Report 96-10-01, University of Washington, Department of Computer Science and Engineering, 1992. Also appears in the Proceedings of the *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1997*.

[30] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27:10:15–26, Oct 1994.

[31] M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1992.

[32] D. Naor, C. Martel, and N. Matloff. Performance of priority queue structures in a virtual memory environment. *Computer Journal*, 34:5:428–437, Oct 1991.

[33] G. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25:3:378–395, 1978.

[34] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1988.

[35] J.P. Singh, H.S. Stone, and D.F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41:7:811–825, 1992.

[36] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:3:652–686, 1985.

[37] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, 1994.

[38] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

[39] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12:3:175–205, 1994.

[40] M. Weiss. *Data structures and algorithm analysis*. Benjamin/Cummings Pub. Co., Redwood City, CA, 1995.

[41] H. Wen and J. L. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9:3:222–241, 1991.

[42] J. W. Williams. Heapsort. *Communications of the ACM*, 7:6:347–348, 1964.

[43] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM Symposium on Programming Languages Design and Implementation*, pages 30–44. ACM, 1991.