Sorting in the Presence of Branch Prediction and Caches

Fast Sorting on Modern Computers

Paul Biggar David Gregg

Technical Report TCD-CS-2005-57
Department of Computer Science,
University of Dublin, Trinity College,
Dublin 2, Ireland.

August, 2005

Revision 1 - January, 2007

Abstract

Sorting is one of the most important and studied problems in computer science. Many good algorithms exist which offer various trade-offs in efficiency, simplicity and memory use. However most of these algorithms were discovered decades ago at a time when computer architectures were much simpler than today. Branch prediction and cache memories are two developments in computer architecture that have a particularly large impact on the performance of sorting algorithms.

This report describes a study of the behaviour of sorting algorithms on branch predictors and caches. Our work on branch prediction is almost entirely new, and finds a number of important results. In particular we show that insertion sort causes the fewest branch mispredictions of any comparison-based algorithm, that optimizations - such as the choice of the pivot in quicksort - can have a large impact on the predictability of branches, and that advanced two-level branch predictors are usually worse at predicting branches in sorting algorithms than simpler branch predictors. In many cases it is possible to draw links between classical theoretical analyses of algorithms and their branch prediction behaviour.

The other main work described in this report is an analysis of the behaviour of sorting algorithms on modern caches. Over the last decade there has been considerable interest in optimizing sorting algorithms to reduce the number of cache misses. We experimentally study the cache performance of both classical sorting algorithms, and a variety of cache-optimized algorithms proposed by LaMarca and Ladner. Our experiments cover a much wider range of algorithms than other work, including the $O(N^2)$ sorts, radixsort and shellsort, all within a single framework. We discover a number of new results, particularly relating to the branch prediction behaviour of cache-optimized sorts.

We also developed a number of other improvements to the algorithms, such as removing the need for a sentinel in classical heapsort. Overall, we found that a cache-optimized radixsort was the fastest sort in our study; the absence of comparison branches means that the algorithm causes almost no branch mispredictions.

Contents

List of Figures

Chapter 1

Introduction

Most major sorting algorithms in computer science date to the 50s and 60s. Radixsort was written in 1954, quicksort in 1962, and mergesort has been traced back to the 1930s. Modern processors meanwhile have features which were not thought of when these sorts were written: branch predictors, for example, were not created until the 1970s, and advanced branch prediction techniques were not used in consumer processors until very recently. It is hardly surprising, then, that sorting algorithms rarely take these architectural features into account.

Papers considering sorting performance typically analyse instruction count. LaMarca and Ladner break this trend in [LaMarca 96b] by discussing the cache implications of several algorithms, including sorts. This report investigates these claims and tests and analyses high performance sorts based upon their work.

LaMarca devotes a chapter of his thesis to sorting. Another chapter is spent discussing implicit heaps, which are later used in heapsort. In [LaMarca 97], radixsort was added for comparison. Radixsort and shellsort are added here for the same purpose. For comparison, elementary sorts - insertion sort, selection sort, bubblesort and shakersort - are also included.

In addition to repeating LaMarca's work, this report also discusses hardware branch prediction techniques and their effects on sorting. When algorithms are altered to improve their performance, it is important to observe if any change in the rate of branch prediction misses occurs.

Over the course of the cache-conscious algorithm improvements, branch prediction results are generated in order to observe the side-effects of the cache-conscious improvements on the branch predictors, and whether they affect different branch predictors in different ways.

Several important discovered results are reported here. The branch prediction rates of insertion sort and selection sort, for example, and the difference in performance between binary searches and sequential searches due to branch misses. Several steps to remove instructions from standard algorithms are devised, such as double tiling and aligning mergesort, removing a sentinel and bounds check from heapsort, and adding a copy-back step to radixsort.

Chapter 2 provides background information on caches and branch predictors. It also discusses the theory of sorting, and conventions used in the algorithms in this report.

Chapter 3 discusses the method and framework used in this research, including the tools used and the reasons for using them.

Chapter 4 discusses the elementary sorts and their implementations. Results are included for later comparison.

Chapters 5 through 8 discuss base algorithms and cache-conscious improvements for heapsort, mergesort, quicksort and radixsort. Results are presented and explained, and branch prediction implications of the changes are discussed.

Chapter 9 presents two shellsort implementations and discusses their performance in the areas of instruction count, cache-consciousness and branch prediction.

Chapter 10 presents conclusions, summarises results from previous chapters, compares our results with LaMarca's, presents a list of cache and branch prediction results and improvements, discusses the fastest sort in our tests and lists contributions made by this research.

Chapter 2

Background Information

2.1 Sorting

Sorting algorithms are used to arrange items consisting of *record*s and *key*s. A key gives order to a collection of data called a record. A real-life example is a phone book: the records, phone numbers, are sorted by their keys, names and addresses.

Sorts are frequently classified as *in-place* and *out-of-place*. An out-of-place sort typically uses an extra array of the same size as the array to be sorted. This may require the results to be copied back at the end. An in-place sort does not require this extra array, though it may use a large stack, especially in recursive algorithms.

A *stable* sort is one in which records which have the same key stay in the same relative order during the sort. All the elementary sorts are stable, as are mergesort and radixsort. Quicksort, heapsort and shellsort are not stable.

Sorting algorithms tend to be written using an *Abstract Data Type*, rather than for a specific data type. This allows efficient reuse of the sorts, usually as library functions. An example from the C standard library is the `qsort` function, which performs a sort on an arbitrary data type based on a comparative function passed to it.

The ADT interface used here is taken from [Sedgewick 02a]. An `Item` is a record, and a function is defined to extract its key. Comparisons are done via the `less` function, and swapping is done using the `exch` function. These are replaced with their actual functions by the C preprocessor. The code used for ADTs is in Figure 2.1 on the following page. The tests presented in this report use *unsigned integer*s as Items, which were compared using a simple *less-than* function[1]. In this case, a record and key are equivalent, and the word *key* is used to refer to items to be sorted.

There are special cases among sorts, where some sorts perform better on some types of data. Most data to be sorted generally involves a small key. This type of data has low-cost comparisons and low-cost exchanges. However, if strings are being compared, then the comparison would have a much higher cost than the exchange, since comparing keys could involved many comparisons - in the worst case one on

[1] This abstraction broke down in several cases, such as where a *greater-than* function was required, i.e. in quicksort.

```
#define Item unsigned int
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define exch(A, B)
do {
    unsigned int t = (A);
    (A) = (B);
    (B) = t;
}
while(0)
```

Figure 2.1: Definitions used by the ADT

each letter.

Sorts which are not comparison-based also exist. These sorts are called counting-based sorts, as they count the number of keys to be put in each segment of an array before putting them in place. The most important of these is radixsort, discussed in Chapter 8.

Sorts sometimes use a sentinel to avoid exceeding bounds of an array. If an algorithm spends its time moving a key right based on a comparison with the key next to it, then it will need to check that it doesn't go over the edge of the array. This check, which can be expensive, can be replaced with a sentinel. This is done by placing a maximal key off the right of the array, or by finding the largest key in the array, and placing it in the final position. This technique is used in quicksort and insertion sort.

2.2 Caches

Processors run many times faster than main memory. As a result, a processor which must constantly wait for memory is not used to its full potential. To overcome this problem, a *cache hierarchy* is used. A cache contains a small subset of main memory in a smaller memory, which is much faster to access than main memory. Modern computers typically contain many levels of cache, each with increasing size and *latency*[2]. For example, it may take twice as long to access the level 1 cache as it does to access a register. It may take ten times as long as that to access the level 2 cache, and fifty times as long to access main memory.

When a cache contains the data being sought, then this is called a cache *hit*. When it does not, it is called a cache *miss*, and the next largest cache in the hierarchy is queried for the data. There are three types of cache miss: *compulsory* or *cold-start* misses, *conflict* misses and *capacity* misses. A compulsory miss occurs when the data is first fetched from memory. Since it has not been accessed before, it cannot already be in the cache. A conflict miss occurs when the addressing scheme used by the cache causes two separate memory locations to be mapped to the same area of the cache. This only occurs in direct-mapped or set-associative cache, described below. Finally, a capacity miss occurs when the data sought has been ejected from the cache due to a lack of space.

Caches range from direct-mapped to fully-associative. In a direct-mapped cache, every memory address has an exact position in the cache where it can be found. Since the cache is smaller than main memory,

[2]Latency is the time between a request being made and its successful execution.

many addresses map to the same position. Fully-associative is the opposite of this: data can be stored anywhere in the cache. A set-associative cache is a mixture of both: more than one piece of data can be stored in the same place. A 4-way set-associative cache allows four items to be stored in the same place; when a fifth item needs to be stored in that position, one of the items is ejected. The item may be ejected randomly, in a specific order, or the *least recently used* item may be removed.

When data is loaded into the cache, data near it is loaded with it. This set of data is stored in a *cache line* or *cache block*. The associativity and length of the cache line dictate the mapping of memory addresses to cache storage. Figure 2.2 shows how a memory address is divided.

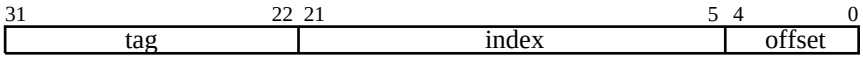


Figure 2.2: Address Division

This cache is a 2MB direct-mapped cache. There is a 32-byte cache line, which is indexed by the 5 bit *offset*. There are 2^{16} cache lines, which are indexed by the middle 16 bits of the address. The most significant 11 bits form the *tag*. A cache line is tagged to check if the data stored at an index is the same as the data requested. For a fully-associative cache, there is no index, and all the bits except the offset make up the tag. A fully-associative cache has no conflict misses for this reason; similarly, a direct-mapped cache has no capacity misses, since even when full, misses are due to conflicts.

The address used by the cache is not necessarily the address used by a program. The addresses used within a program are called *virtual addresses*. Some caches are addressed by virtual address, some caches by *physical address*, and some caches by *bus address*[3]. Several optimizations in later chapters use the address to offset arrays so that conflicts are not caused. However, because a program cannot be sure of what type of addressing is used, the actual manipulated address may not directly relate to the index used to address the cache. This problem is not encountered in our simulations, as SimpleScalar addresses its caches by virtual address.

Caches work on the principle of *locality*. It is expected that after some data has been used once, it is likely to be used again in the near future. This is called *temporal locality*. Similarly, if some data is being used, it is likely that data near it will also be used soon, such as two consecutive elements of an array. This is called *spatial locality*. Many of the algorithms presented here are designed to take advantage of locality. Spatial locality is exploited by ensuring that once a cache line is loaded, each key in the cache line is used. Temporal locality is exploited by ensuring that an algorithm uses data as much as it can as soon as it is loaded, rather than over a large time scale, during which it may have been ejected from the cache.

2.3 Branch Predictors

A branch is a conditional statement used to control the next instruction a machine must execute. Typically, these are `if` and `while` statements, or their derivatives. Branches cause delays in a processor's instruction pipeline. The processor must wait until the branch is resolved before it knows what the next instruction is, and so it cannot fetch that instruction, and must stall. To avoid this problem, modern

[3]A bus address is used to address parts of the computer connected over a bus, including main memory, multimedia controllers such as sound and graphics cards, and storage devices such as hard drivers and floppy disk controllers.

processors use branch predictors, which guess the branch direction and allow the processors to execute subsequent instructions straight away.

Each branch can be resolved in two ways: *taken* or *not-taken*. If the branch is taken, then the program counter moves to the address provided by the branch instruction. Otherwise, the program counter increments, and takes the next sequential instruction.

If a branch is mispredicted, then a long delay will occur, though this delay is the same as if a branch predictor had not been used. During this delay, the pipeline must be emptied, the instructions already executed must be discarded, and the next instruction must be fetched.

Branch mispredictions are sometimes called *branch misses*, due to the similarity between dynamic predictors and caches. In this report, we use the terms interchangeably.

In this report, we split branches into two types. The first is a *flow control branch*. Flow control refers to controlling the flow of execution of a program, and branches of this type are loops and other control statements. The second type is a *comparative branch*: these are comparisons between two pieces of data, such as two keys in an array being sorted. A branch can be both a flow control and a comparative branch, but these will be referred to as comparative branches here. This is due to the properties of the branches: a comparative branch should not be easily predictable; a flow control branch should be easy to predict based on previous predictions of the same branch.

There are several kinds of branch predictors: *static*, *semi-static* and *dynamic*. This report only deals with dynamic predictors, which are of interest due to their high accuracy. The other types are none-the-less discussed below, as an introduction to dynamic predictors. A good discussion on predictor types is [Uht 97], from which the percentages below are taken. These predictors were tested by Uht on the SPECint92 benchmarking suite. This is a standard suite of integer-based programs used to measure the performance of an architecture against another. Programs in this benchmark include `gcc`, a major C compiler, `espresso`, which generates Programmable Logic Arrays and `li`, a LISP interpreter.

2.3.1 Static Predictors

A static predictor makes hardwired predictions, not based on information about the program itself. Typically the processor will predict that branches are all taken, or all not-taken, with 60% and 40% accuracy, respectively. Another type of static branch predictor is Backwards-Taken-Forwards-Not-taken. Backward branches are usually taken, so this increases the prediction accuracy to 65%.

2.3.2 Semi-static Predictors

Semi-static predictors rely on the compiler to make their predictions for them. The predictions don't change over the execution of a program, but the improvement in predicting forward branches, which predominantly execute one way, increases the accuracy to 75%.

2.3.3 Dynamic Predictors

Dynamic predictors store a small amount of data about each branch, which predicts the direction of the next branch. This data changes over the course of a program's execution. Examples of this type of predictor are *1-bit*, *bimodal* and *two-level adaptive*.

1-bit

A 1-bit dynamic predictor uses a single bit to store whether a particular branch was last taken or not-taken. If there is a misprediction, then the bit is changed. 1-bit predictors have the potential to mispredict every time, if the branch is alternately taken and not taken. In the case of a loop with the sequence T-T-T-N-T-T-T-N, a 1-bit predictor will mispredict twice per loop, or about half the time. Typically, though, it achieves an accuracy of 77 to 79%.

Bimodal

A 2-bit dynamic predictor is called a bimodal predictor. It is a simple 2-bit counter, each combination representing a state. In states 00 and 01, not-taken is predicted; in states 10 and 11, taken is predicted. Each taken prediction increments the count, to a maximum of 11, and each not-taken prediction decrements the counter, to a minimum of 00.

In the sequence T-T-T-N-T-T-T-N, a 2-bit predictor will only mispredict once per loop iteration. In general, it will achieve 78 to 89% accuracy.

Two-Level Adaptive

A two-level adaptive predictor uses branch history to predict future branches. A global branch history register keeps track of the last few branches, and uses this to index a table of 2-bit predictors, which predict the result of the branch. Frequently, the program counter is also included, either concatenated or exclusive-ORed with the branch history to form the table index.

This type of predictor can predict the two cases above perfectly. However, the table can take time to fill, leading to cold-start misses, much like those of a cache. Typically, a two-level adaptive predictor will be accurate 93% of the time.

2.4 Big O Notation

Big O notation is used to describe the complexity of an algorithm. It says that the algorithm will complete in a certain number of steps, in relation to number of items of data being worked on, without giving specifics of how many instructions each step will take to execute. It is possible, therefore, that an algorithm with a lower complexity will complete more slowly than an algorithm where each step is executed very quickly, but which has a greater order of complexity.

Big O notation is used to describe how an algorithm scales. Doubling the number of elements to be sorted by an $O(N)$ algorithm will roughly double the time it takes to execute, but will quadruple the running time of an $O(N^2)$ algorithm.

An algorithm which is $O(N)$ is said to complete in linear time. An algorithm which is $O(N^2)$ is said to complete in quadratic time. $O(NlogN)$ algorithms may take several orders of magnitude more time than $O(N)$ algorithms, and typically use the *divide-and-conquer* principle, or use a binary tree to reduce complexity. Quicksort, mergesort and heapsort are all of this type, whereas elementary sorts, such as insertion sort, selection sort, bubblesort and shakersort are $O(N^2)$ algorithms. Radixsort is $O(N)$, but can be more expensive than quicksort, due to the cost of each step. These are discussed in later chapters. The complexity of shellsort, meanwhile, is quite complex, and is discussed in [Sedgewick 96].

An important thing to note is that the time the algorithms take to execute are proportion to their complexity, but different constants of proportionality are used for different algorithms. An algorithm with $10N$ instructions is $O(N)$, and is not differentiated from an algorithm with $1000N$ instructions by this notation.

Chapter 3

Tools and Method

This section discusses the method and framework used to complete this research. The first section discusses the steps involved in creating a testing framework, programming the sorts and getting results. The second section discusses tools that were used or created to do this.

3.1 Method

The following were the objectives of the research:

- Implement heapsort, mergesort, quicksort, radixsort, shellsort and the four elementary sorts: insertion sort, selection sort, bubblesort and shakersort.
- Implement cache-conscious sorts from [LaMarca 96b] and [LaMarca 97].
- Simulate these sorts using the SimpleScalar architecture simulator, using a variety of branch predictors and cache configurations.
- Measure the performance of these sorts using the PapiEx performance counters.
- Simulate bimodal branch predictors in software and use them to measure the sorts.
- Investigate and analyse the increase in speed, reduction in cache misses, and change in branch prediction accuracy as a result of the changes made.
- Determine if any changes can be made to improved the sorts' performance.

Several steps were necessary to prepare for the research. A framework was required by which sorts could be tested as they were being developed. This is discussed in Section 3.2.3. It was also necessary to fully understand the technologies to be used in the report. Several texts were used for this: cache and branch predictors are both in [Patterson 90]; branch predictors in [Uht 97], as well as in [McFarling 93] and [Smith 81]; sorts are discussed in [Sedgewick 02a] and [Knuth 98]; cache-conscious sorts are discussed [LaMarca 96b], [LaMarca 97] and [LaMarca 96a].

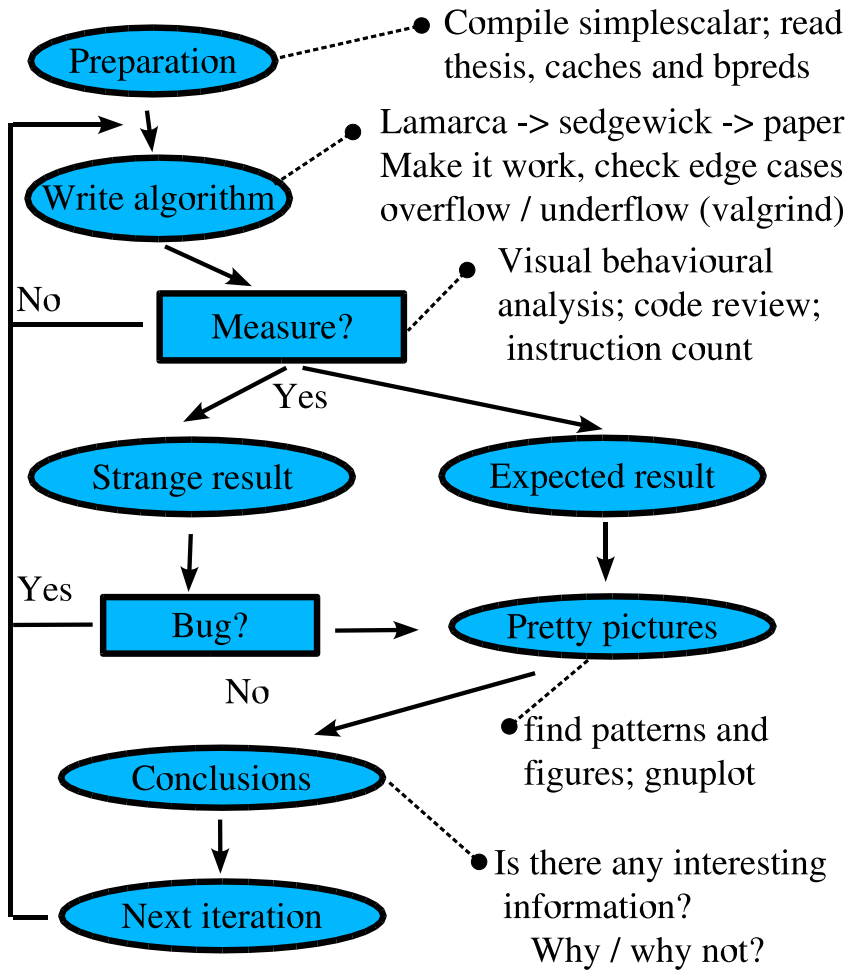


Figure 3.1: Flowchart describing the development of an algorithm

There were many steps involved in writing each iteration of the algorithms. Insertion and selection sort had just one of these iterations; the other sorts had more than one[1]. A flowchart of these steps appears in Figure 3.1.

The first step in each case was to write the algorithm. [Sedgewick 02a] was the primary source of these, though in several cases LaMarca specified a particular version which he used. When a step involved in an algorithm wasn't clear, the paper describing the algorithm was used to clarify the meaning.

Once the algorithm was written, a testing framework was used. This tested for edge-cases in the algorithms by testing sorts sized from `8` to `255` keys, with different random data sets. Testing was also performed on a very large data set. This test was timed and the results were considered. The results of this test were not accurate: the program took up to a minute to run, during which time the computer was being used by multiple users and processes, leading to context switches, sharing of resources and scheduling. Despite these inaccuracies, it was possible to use the results relative to each other, and to

[1] Quicksort, for example, had four: base quicksort, tuned quicksort, multi-quicksort and sequential multi-quicksort.

determine whether, for example, memory-tuned quicksort was running faster than base quicksort.

Early on, it was decided that exceeding the bounds of an array during a sort was to be forbidden. No serious sorting library or application could acceptably work in this manner, and any results obtained while allowing this were only of academic interest. It isn't possible, from a normal run of a program, to determine if an arrays bounds are exceeded slightly, though exceeding the bounds by a large amount will result is a segmentation fault. Because of this, Valgrind (see Section 3.2.4), was used to report these errors.

Once it was certain that a sort actually sorted without error, it was necessary to ensure that this was not a fluke occurrence. Many of the sorts had multiple passes, and if one pass did not properly sort the array segment it was meant to, the array might still be sorted by the next pass, though potentially much more slowly. To test that this did not occur, a set of functions to visually display arrays and variables at certain points was developed, which created a HTML page showing the progression of the algorithm. An example of this is in Figure 3.2 on the next page.

This shows two stages of a bitonic mergesort which is described in Section 6.1.3. The top of the image shows the source array being merged into the target array below. The source array shows three large segments of the array and four smaller segments. The first two large segments have been merged into one segment in the target array. The third of these has been merged into the corresponding target array segment. The first of the smaller segments has begun being merged with the other small segments. The rest of the target array is uninitialised. Below that is a small amount of debugging information. $i == j$ is a key point in the program and is printed as debugging information. In the two arrays below this, the merge continues.

While it was not known in advance the desired instruction count of an algorithm, a certain range was implied by LaMarca's descriptions. The relationship between one iteration of an algorithm, or between base versions of several algorithms was known ahead of time. One configuration of the `simple` script ran quick tests using the `sim-fast` simulator, giving instruction counts which could be compared to other versions.

If both these tests and the visual display of the sort's behaviour seemed correct, then the full collection of simulations was run. The results of the simulation were compared to results of other versions and to the expected results. If the results were surprising, then it was necessary to explain the reasons for this. If the cause was a bug, then the bug was fixed and the tests repeated.

The results were then turned into graphs, from which conclusions about the performance of the algorithm could be reached, and patterns could be spotted. More details of this are in Section 3.2.2. The next iteration of the algorithm was then begun, and the process repeated for each of the algorithm's iterations.

Finally, a software predictor was added in order to pin-point the cause of the branch mispredictions. This meant altering the code of the algorithms to call functions which updated simulated branch predictors. Each important branch was allocated a predictor, which was updated independently of the other predictors. This allowed the performance to be analysed down to the individual branch, and blame apportioned appropriately.

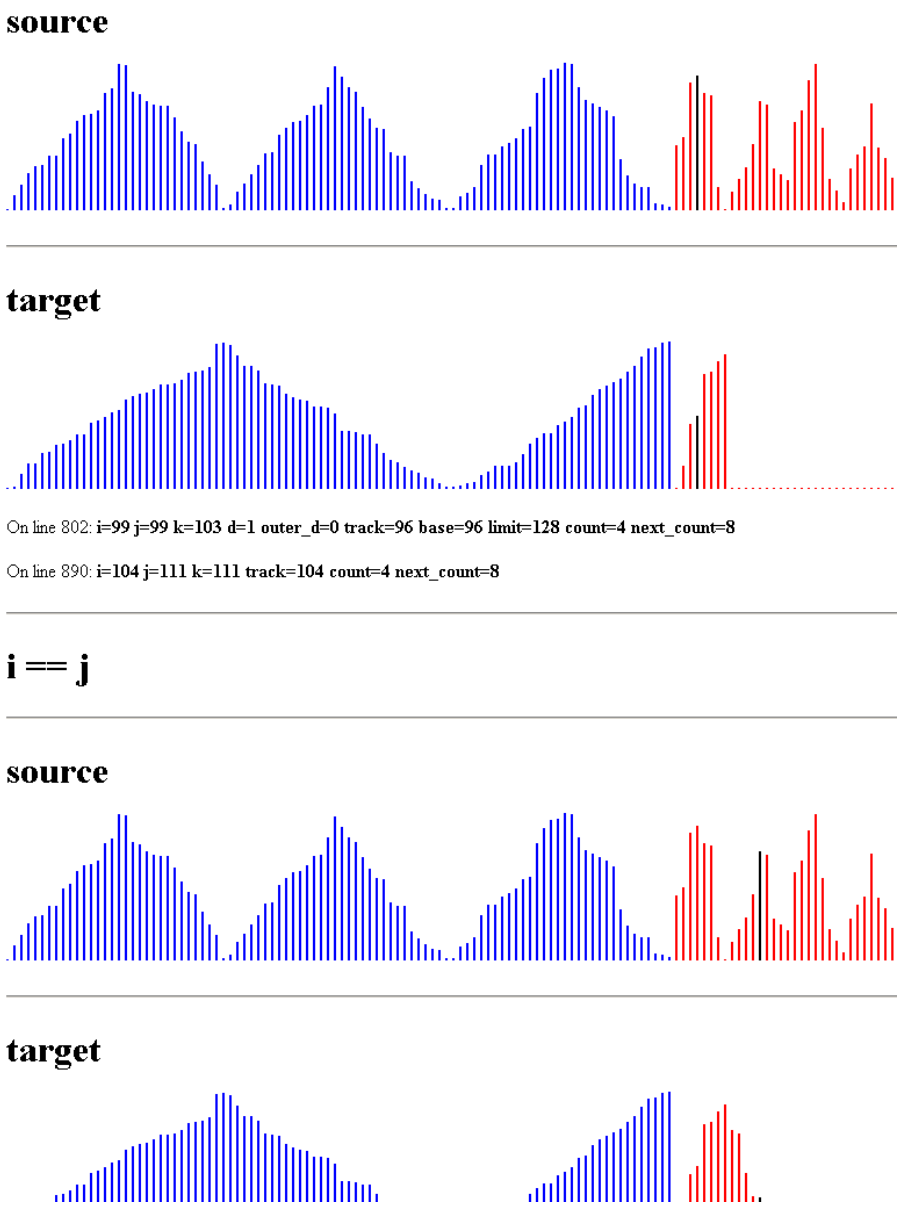


Figure 3.2: A sample visual sort routine, showing a merge from the source array to the auxiliary array

3.2 Tools

3.2.1 SimpleScalar

SimpleScalar is an architecture simulator. It works by compiling code into its native instruction set, *PISA*, then emulating a processor as it runs the program. Documentation on how to use and modify SimpleScalar are [Burger 97b], [Austin 02], [Austin 01], [Austin 97] and [Burger 97c]. SimpleScalar can be found at *http://www.simplescalar.com*.

SimpleScalar works by compiling code into its own instruction set. It provides a compiler - a port of gcc - with related tools such as a linker and assembler. Several simulators are then provided: `sim-fast` only counts instructions executed; `sim-cache` simulates a cache and counts level 1 and level 2 cache misses; `sim-bpred` simulates a branch predictor and counts branch hits and misses. Other simulators are provided, but they were not used in our tests.

The output of SimpleScalar is human readable, but is not conducive to collecting large amounts of data. To that end, a script was written to run each required set of simulations. It recorded data from the simulations into a single file for later retrieval. This file was suitable to reading with a `diff` program.

3.2.2 Gnuplot

Gnuplot is a tool for plotting data files. It provides visualisations of data in two and three dimensions, and is used to create all of the graphs in this document. Its homepage is *http://www.gnuplot.info/*.

There were several steps involved in using gnuplot for the data. Firstly, the data was not in the correct format for gnuplot, and had to be converted. It also needed to be offset by the cost of filling the array with random variables, and scaled to so that the y-axis could be in the form of *plotted data per key*, both to be more readable, and to be consistent with LaMarca's graphs. Once the data was converted, it was necessary to generate a gnuplot script for each graph. The number of graphs required made this an impossible manual task, and so another script had to be written. Some graphs still needed to be tweaked by hand, due to difficulties in the script, but the task was much shortened by this, and the tweaking was mostly accomplished using macros in the development platform.

Finally, a script to compare the data together was required. This compared each sort against each other sort in key metrics, such a branch or cache misses. It also compared each cache size against other cache sizes for a particular sort, and branch predictors against other branch predictors for each sort.

3.2.3 Main

The main program provided the framework for the development of the sorts. Imaginatively called `main.c`, this program provided the means for testing and (roughly) timing each sort. Figure 3.3 on the following page contains a sample of this code.

```
test_array(const char* description) {
   if (memcmp(sorted_array, random_array2,
         RANDOM_SIZE * sizeof(Item)) == 0)
      printf("%s is sorted\n\n", description);
   else {
      printf("Not sorted:\n");
      print_int_array(random_array2, RANDOM_SIZE,
         description, -1, -1, -1);
      printf("\n\nWhat it should be:\n");
      print_int_array(sorted_array, RANDOM_SIZE,
         "Sorted array", -1, -1, -1);
   }
}

time_sort(void sort(Item*, int), char* description) {
   memcpy(random_array2, random_array,
       RANDOM_SIZE * sizeof(int));
   start_timer();
   sort(random_array2, RANDOM_SIZE);
   stop_timer();
   print_timer(description);
   test_array(description);
}

test_sort(void sort(Item*, int), char* description) {
   for(int i = 8; i < 256; i++) {
      int* a = malloc(i * sizeof(Item));
      int* b = malloc(i * sizeof(Item));
      fill_random_array(a, i, i);
      memcpy(b, a, i * sizeof(Item));
      qsort((void*)a, i, sizeof(Item), compare_ints);
      sort(b, i);
      if (memcmp(a, b, i * sizeof(Item)) != 0) {
         printf("%s is not sorted (size == %d):\n",
            description, i);
         print_int_array(b, i, description, -1, -1, -1);
         printf("\n\nWhat it should be:\n");
         print_int_array(a, i, "Sorted array", -1, -1, -1);
         exit(-1);
      }
      free(a); free(b);
   }
   printf("Testing of %s complete\n", description);
}
test_sort(insertsort, "O(N squared) Insertion");
time_sort(insertsort, "O(N squared) Insertion");
```

Figure 3.3: Key sections of main.c

3.2.4 Valgrind

Valgrind is a set of tools which are used for debugging and profiling. By default it works as a memory checker, and detects errors such as the use of uninitialised memory and accessing memory off the end of an array. This use makes it particularly useful for bounds checking. No configuration was required for this, and simply running `valgrind a.out` ran the program through valgrind. The homepage for valgrind is at *http://valgrind.kde.org/*.

3.2.5 PapiEx

PapiEx is a tool used to access performance counters on a Pentium 4. It uses the *PAPI* framework to count certain characteristics across a program's execution. We used it to count the number of cycles a sort took to run. This provided an accurate measure of how fast a sort executes on a Pentium 4. The PapiEx homepage is at *http://icl.cs.utk.edu/~mucci/papiex/*.

PapiEx was used to create the graphs comparing the running time of sorts. The gnuplot scripts for this were short and easily written by hand.

PapiEx was an alternative to using SimpleScalar. By using its hardware registers, actual real world results could have been measured. Using PapiEx instead of SimpleScalar was an option reviewed at the start, but because PapiEx measures results on a real machine, only one type of cache and branch predictor could have been measured. We have instead provided limited measurements to supplement the SimpleScalar results.

3.2.6 Software Predictor

The software predictor used was a series of macros and functions which maintained the state of a bimodal branch predictor at a particular point in the program. Statistics about each branch were collected, and dumped into text files, from which scripts created graphs to display them. Some sample code is shown in Figure 3.4. Each of the graphs produced by the predictors uses the same scale; this way, it is possible to visually compare the results from one sort to another.

3.3 Sort Implementation

To show an example of the creation of a sort, mergesort will now be discussed. Descriptions of some of the terms here is saved for discussion in Chapter 6.

The base mergesort described by LaMarca is Algorithm N. Three improvements were prescribed by LaMarca: changing the sort to be bitonic to eliminate bounds checks, unrolling the inner loop, and pre-sorting the array with a simple in-place sort. He also recommended making the sort pass between two arrays to avoid unnecessary copying.

Algorithm N is described in [Knuth 98]. A flow control diagram and step by step guide is provided, but very little detail of how the sort actually works is included. Therefore, the first step was to understand how this sort worked. This was done by first analysing the flow control of the algorithm. The steps

```
static int
partition(unsigned int a[], int l, int r)
{
   unsigned int v;
   int i, j;

   v = a[r];
   i = l - 1;
   j = r;

   for(;;)
   {
      while (less(a[++i], v))
      {
         branch_taken(&global_predictor[0]);
      }
      branch_not_taken(&global_predictor[0]);

      while (less(v, a[--j]))
      {
         branch_taken(&global_predictor[1]);
      }
      branch_not_taken(&global_predictor[1]);

      if (i >= j)
      {
         branch_taken(&global_predictor[2]);
         break;
      }
      branch_not_taken(&global_predictor[2]);

      exch(a[i], a[j]);
   }

   exch(a[i], a[r]);

   return i;
}
```

Figure 3.4: Sample code using the software predictor

of the sort are described in assembly, and while they can be written in C, the result is a very low-level version. An attempt was made to rewrite the algorithm without `goto` statements and labels, instead using standard flow control statements such as `if`, `while` and `do-while`. Despite the time spent on this, it was impossible to faithfully reproduce the behaviour without duplicating code.

As a result, it was necessary to use the original version, and to try to perform optimizations on it. However, several factors made this difficult or impossible. LaMarca unrolled the inner loop of mergesort, but it is not clear which is the inner loop. The two deepest loops were executed very infrequently, and unrolling them had no effect on the instruction count. Unrolling the other loops also had no effect, as it was rare for the loop to execute more than five or six times, and once or twice was more usual.

Pre-sorting the array was also not exceptionally useful. The instruction count actually increased slightly as a result of this, indicating that algorithm N does a better job of pre-sorting the array than the inlined pre-sort did.

Algorithm N does not have a bounds check. It treats the entire array bitonically; that is, rather than have pairs of array beings merged together, the entire array is slowly merged inwards, with the right hand side being sorted in descending order and the left hand side being sorted in ascending order. Since there is no bounds check, there is no bounds check to be eliminated.

As a result of this problems, algorithm S, from the same source, was investigated. This behaved far more like LaMarca's description: LaMarca described his base mergesort as sorting into lists of size two, then four and so on. Algorithm S behaves in this manner, though Algorithm N does not. Fully investigating algorithm S, which is described in the same manner as algorithm N, was deemed to be too time-consuming, despite its good performance.

As a result, a mergesort was written partially based on the outline of algorithm N. The outer loop was very similar, but the number of merges required were calculated in advance, and the indices of each merge step. It was very simple to turn this into a bitonic array; adding pre-sorting reduced the instruction count, as did unrolling the loop. The reduction in instruction count from the initial version, which performed the same number of instructions as the same as algorithm N, and the optimized version, was about 30%.

A tiled mergesort was then written, based on LaMarca's guidance. Firstly, this was aligned in memory so that the number of misses due to conflicts would be reduced. Then it fully sorted segments of the array, each half the size of the cache, so that temporal locality would be exploited. It also fully sorted segments within the level 1 cache. Finally, all these segments were merged together in the remaining merge steps.

LaMarca's final step was to create a multi-mergesort. This attempted to reduce the number of cache misses in the final merge steps of tiled mergesort by merging all the sorted arrays in one step. This required the use of an k-ary heap, which had been developed as part of heapsort.

Once these sorts were written it was attempted to improve them slightly. Instead of a tiled mergesort, a double-aligned tiled mergesort was written. This aligned the arrays to avoid conflicts in the level 1 cache, at a slight cost to level 2 cache performance. Double-aligned multi-mergesort was a combination of this and of multi-mergesort. We also rewrote base mergesort far more cleanly, with the results being more readable and efficient. These were also found to reduce the instruction count, as well as the level 2 cache miss count. These results are discussed in more detail in Section 6.6.3.

Each of these sort variations had to be fully written from the descriptions in LaMarca's thesis. Only the base algorithm was available from textbooks, and this needed to be extensively analysed and rewritten. Each step which was written had to be debugged and simulated. The results were compared with the

expected results and frequently the sorts needed to be edited to reduce the cost of a feature.

Radixsort, shellsort and each of the four $O(NlogN)$ sorts were treated in this fashion.

3.4 Testing Details

Our tests were designed so that we could compare our results to LaMarca's, and so that our results would hold in the general case. Both our testing method and our data sets were chosen with these goals in mind.

3.4.1 SimpleScalar Simulations

We tested each of our sorts using SimpleScalar, using set sizes from 4096 to 4194304, increasing by a factor of two each time. Each of these set sizes was simulated using `sim-cache` and `sim-bpred` in a variety of configurations.

The SimpleScalar settings were chosen to be similar to LaMarca's simulations. LaMarca used the *Atom* simulator, which was used to test software for the DEC AlphaStation. It simulated a 2MB direct-mapped cache with a 32-byte cache line. Other settings were also chosen to correspond to the DEC Alpha: 8KB level 1 data cache, 8KB level 1 instruction cache and shared instruction and data level 2 cache.

In order to be able to extrapolate more information, we increased the number of cache simulations. We simulated both 2MB and 4MB caches, configured as both direct-mapped and fully-associative.

We chose a similar approach for our branch predictor caches. We aimed to simulate bimodal predictors as well as two-level adaptive, in order to be able to compare them together. We simulated bimodal predictors with a 512, 1024 and 2048 byte table, and two-level adaptive predictors with 512, 1024, 2048 and 4096 entries.

3.4.2 Software Predictor

In order to explore the behaviour of the individual branches, we performed a second set of branch prediction experiments. Rather than using SimpleScalar, we instead added additional statements to the source code to simulate a separate bimodal predictor for each interesting branch. This allowed us to measure each branch individually, whereas SimpleScalar gives only a total figure for all branches.

The software predictor results are averaged over ten runs, using arrays containing 4194304 keys. To compare these results to the SimpleScalar ones, it is necessary to divide the number of branch mispredictions by 4194304.

3.4.3 PaPiex

Our papiex tests ran significantly faster than either of our other simulations, and so we were able to run them over a significantly greater quantity of data. Each sort and each size from 4096 to 4194304 keys was

run 1024 times, using the system's random number generator to generate data, using seeds of between 0 and 1023. The results displayed are averaged over the 1024 runs.

These simulations were run a on Pentium 4 1.6GHz, which had an 8-way, 256K level 2 cache, with a 64-byte cache line. The level 1 caches are 8K 4-way associative caches, with separate instruction and data level 1 caches.

The instruction pipeline of the Pentium 4 is 20 stages long, and so the branch misprediction penalty of the Pentium 4 approaches 20 cycles. The Pentium 4's static branch prediction strategy is backward taken and forward not taken; its dynamic strategy is unspecified: though [Intel 04] mentions a branch history so it is likely to be a two-level adaptive predictor.

3.4.4 Data

Both our SimpleScalar simulations and our software predictor tests used *http://www.random.org* to provide truly random data. The software predictor used ten sections containing 4194304 *unsigned integer*s, and the SimpleScalar simulations used the same ten sections. When SimpleScalar simulations were run on sized smaller than 4194304 keys, only the left hand side of the sections were used.

3.4.5 32-bit Integers

LaMarca used 64-bit integers, as his test machine was a 64-bit system. Since our machine is 32-bit, and SimpleScalar uses 32-bit addressing on our architecture, LaMarca's results will not be identical to ours. Rather, the number of cache misses in his results should be halved to be compared to ours; when this is not the case, it is indicated.

3.4.6 Filling Arrays

It takes time to fill arrays, and this can distort the relationship between the results for small set sizes, and larger set sizes, for which this time is amortised. As a result, we measured this time, and subtracted it from our results.

However, this practice also distorts the results. It removes compulsory cache misses from the results, so long as the data fits in the cache. When the data does not fit in the cache, the keys from the start of the array are faulted, with the effect that capacity misses reoccur, which are not discounted. As a result, on the left hand side of cache graphs, a number of compulsory misses are removed, though they are not on the right hand side. LaMarca does not do this, and so this needs to be taken into account when comparing his results to ours.

3.5 Future Work

An improvement that would have yielded interesting results would be to do simulations which only consisted of flow control, with no comparisons. This would allow a division of flow control misses and

comparative misses. However, this would not always be possible; sorts like quicksort and heapsort use their data to control the flow of the program. In addition, bubblesort, shakersort and selection sort have (analytically) predictable flow control, while radixsort has no comparison misses at all. Mergesort may benefit from this, however, especially versions with difficult flow control diagrams, such as *algorithm N* and *algorithm S* (see Section 6.1.1).

A metric not considered so far is the number of instructions between branches. This should be an important metric, and investigating it could point to important properties within an algorithm.

Valgrind comes with a tool called `cachegrind`, which is a cache profiler, detailing where cache misses occur in a program. Using this on a sort may indicate areas for potential improvement.

Chapter 4

Elementary Sorts

Elementary sorts are also referred to as $O(N^2)$ sorts. This is due to the manner in which they check every key in the array in order to sort a single key; for every key to be sorted, every other key is checked. Four of these sorts are discussed here: insertion sort, selection sort, bubblesort and shakersort, along with improved versions of bubblesort and shakersort.

4.1 Selection Sort

Selection sort works in a straightforward manner. It begins by searching the unsorted array for the smallest key, then swapping it into the first position in the array. This key is now the first element of a sorted array, being built from the left. The remainder of the array is still unsorted, and it is now searched for its smallest key, which is swapped into place. This continues until the entire array is sorted. Sample code is in Figure 4.1.

```
void selection(Item a[], int N)
{
   for (int i = 0; i < N-1; i++)
   {
      int min = i;
      for (j = i+1; j < N; j++)
         if (less(a[j], a[min])) min = j;

      exch(a[i], a[min]);
   }
}
```

Figure 4.1: Selection sort code

4.1.1 Testing

All elementary sorts are only sorted up to 65536 keys in the simulations, and 262144 keys with the performance counters. Due to the quadratic nature of these sorts, days and weeks would be required to run tests on the larger arrays.

Expected Performance

Sedgewick provides a discussion on the performance of elementary sorts. The observations on instruction count are his, and come from [Sedgewick 02a].

Selection sort uses approximately $N^2/2$ comparisons and exactly N exchanges. As a result, selection sort can be very useful for sorts involving very large records, and small keys. In this case, the cost of swapping keys dominates the cost of comparing them. Conversely, for large keys with small records, the cost of selection sort may be higher than for other simple sorts.

The performance of selection sort is not affected by input, so the instruction count varies very little. Its cache behaviour is also not affected by input, and selection sort performs very badly from a cache perspective. There are N traversals of the array, leading to bad temporal reuse. If the array doesn't fit inside the cache, then there will be no temporal reuse.

The level 1 cache should have bad performance as a result. Since the level 1 cache is smaller that the arrays tested, it should perform analogously to a level 2 cache being tested with a much larger array.

The branch prediction performance is not expected to be bad. Flow control predictions are very straightforward, and should result in very few misses. Comparison predictions, however, are very numerous. Each traversal of the array has an average of $N/2$ comparisons, as we search for the smallest key in the unsorted part of the array. During the search, the algorithm compares each key with the smallest key seen so far. In a random array, we would expect that the candidate for the smallest key (the left-to-right minimum) would change frequently towards the start of the array, making the comparison branch rather unpredictable. However, as the traversal continues, the left-to-right minimum will become smaller, and thus more difficult to displace. It will become very unusual to find a smaller key, and thus the branch will almost always resolve in the same direction. Therefore, the comparison branch will quickly become very predictable.

Knuth [Knuth 97] analyses the number of changes to the left-to-right minimum when searching for the minimum element of an array. He shows that for N elements, the number of changes is $H_N - 1$, where H_N is the harmonic series:

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N} = \sum_{k=1}^{N} \frac{1}{k}, \quad N \geq 1$$

Within practical sorting ranges, H_N grows at a rate similar to $log(N)$. Thus, in the average case, we can expect that the comparison branch will mostly go in the same direction, making the branch highly predictable.

Knuth also presents the number of changes to the left-to-right minimum in the best case (0 changes)

and worst cases ($N - 1$ changes). In the worst case, the array is sorted in reverse order, and so the left-to-right minimum changes on every element. Interestingly, the worst cases for the number of changes is not the worst case for branch mispredictions. If the minimum changes for every element, then the comparison branch will always resolve in the same direction, and thus be almost perfectly predictable. From the point of view of branch prediction, frequent changes (preferably with no simple pattern) are the main indicator of poor performance, rather than the absolute number of times that the branch resolves in each direction. The worst case for selection sort would probably be an array which is partially sorted in reverse order - the left-to-right minimum would change frequently, but not predictably so.

4.2 Insertion Sort

Insertion sort, as its name suggests, is used to insert a single key into its correct position in a sorted list. Beginning with a sorted array of length one, the key to the right of the sorted array is swapped down the array a key at a time, until it is in its correct position. The entire array is sorted in this manner.

An important improvement to insertion sort is the removal of a bounds check. In the case that the key being sorted into the array is smaller than the smallest key in the array, then it will be swapped off the end of the array, and continue into the void. To prevent this, a bounds check must be put in place, which will be checked every comparison, doubling the number of branches and severely increasing the number of instructions. However this can be avoided by using a sentinel. This is done by finding the smallest key at the start, and putting it into place. Then no other key can be smaller, and it will not be possible to exceed the bounds of the array. Another technique would be to put 0 as the smallest key, storing the key currently in that position, then putting it in place at the end. This would use a different insertion sort, with a bounds check, going in the opposite direction. The first of these is shown in the sample code in Figure 4.2 on the following page.

4.2.1 Testing

Expected Performance

These instruction count observations are again due to Sedgewick. Insertion sort performs in linear time on a sorted list, or slightly unsorted list. With a random list, there are, on average, there are $N^2/4$ comparisons and $N^2/4$ half-exchanges[1]. As a result, the instruction count should be high, though it is expected to be lower than selection sort.

The cache performance should be similar to selection sort as well. While selection sort slowly decreases the size of the array to be considered, insertion sort gradually increases the size. However, each key is not checked with every iteration. Instead, on average, only half of the array will be checked, and there will be high temporal and spatial reuse due to the order in which the keys are accessed. This should result in lower cache misses than selection sort.

The flow control of insertion sort is determined by an outer loop, which should be predictable, and an inner loop, which, unlike selection sort, is dependent on data. As a result, the predictability of the flow control is highly dependant on the predictability of the comparisons.

[1] A half-exchange is where only one part of the swap is completed, with the other value stored in a temporary to be put in place later.

```
void insertion(Item a[], int N)
{
   // get a sentinel
   int min = 0;
   for(int i = 1; i < N; i++)
      if(less(a[i], a[min])) min = i;
   exch(a[0], a[min]);

   // sort the array
   for(int i = 1; i < N; i++)
   {
      int j = i;
      Item v = a[i];

      while(less(v, a[j-1]))
      {
         a[j] = a[j-1];
         j--;
      }
      a[j] = v;
   }
}
```

Figure 4.2: Insertion sort code

4.3 Bubblesort

Bubblesort, as its name suggests, works by bubbling small keys from right of the array to the left. It begins by taking a key from the end and moving it left while it is less than any keys it passes. Once it comes across a smaller key, it changes to this key, and moves it towards the left of the array. In this way, every iteration the smallest key moves left to its final position, and other small keys are slowly moved left, so that the unsorted part of the array becomes more sorted as bubblesort continues.

An improvement made to bubblesort is that it stops as soon as the array is sorted. Sample bubblesort code is in Figure 4.3 on the next page.

4.3.1 Testing

Expected Performance

These instruction count observations are again due to Sedgewick. Bubblesort has $N^2/2$ comparisons and exchanges. The number of passes over the array, and size of each pass is the same as selection sort, and similar instruction count results are expected. The extra check to see if the array is sorted adds $O(NlogN)$ instructions, but this is small in comparison to the number of instructions saved if the sort ends early.

The cache performance of bubblesort should be the same as that of selection sort. Each key loaded into the cache is used just once, and will be ejected from the cache if the data set is large enough.

```
void bubblesort(Item a[], int N)
{
   for (int i = 0; i < N-1; i++)
   {
      int sorted = 1;
      for (int j = N-1; j > i; j--)
      {
         if (less(a[j],a[j-1]))
         {
            exch(a[j-1],a[j]);
            sorted = 0;
         }
      }
      if (sorted) return;
   }
}
```

Figure 4.3: Bubblesort code

The branch predictive performance is also expected to be similar. While selection sort has $O(NH_N)$ mispredictions, each of bubblesort's comparisons has the potential to be a misprediction, and not just in the pathological case. However, as the sort continues and the array becomes more sorted, the number of mispredictions should decrease. The number of flow control mispredictions should be very low.

4.4 Improved Bubblesort

Improved bubblesort incorporates an improvement to end bubblesort early. The bubblesort implementation described in the previous section ends early if it detects that the array is fully sorted. It keeps a binary flag to indicate if there were any exchanges along the way. Instead, improved bubblesort keeps track of the last exchange made, and starts the next iteration from there. This means that any pockets of sorted keys are skipped over, not just if they occur at the end. Improved bubblesort code is shown in Figure 4.4 on the following page.

4.4.1 Testing

Expected Performance

It is expected that improved bubblesort should perform faster than the original in all metrics. Instruction count, were there no conditions that allowed keys to be skipped, would be the similar, since few extra instructions are added. The flow of the algorithm is the same, though there should be fewer iterations, especially at the start, when iterations are longer, so cache accesses, and branch misses should be reduced along with instruction count.

```
void improved_bubblesort(Item a[], int N)
{
   int i = 0, j, k = 0;
   while(i < N-1)
   {
      k = N-1;
      for (j = N-1; j > i; j--)
      {
         if (less(a[j],a[j-1]))
         {
            k = j;
            exch(a[j-1],a[j]);
         }
      }
      i = k;
   }
}
```

Figure 4.4: Improved bubblesort code

4.5 Shakersort

Shakersort is sometimes called back-bubblesort. It behaves like a bubblesort which moves in both directions rather than just one way. Both ends of the array become sorted in this manner, with the larger keys slowly moving right and the smaller keys moving left. Sample code for shakersort is shown in Figure 4.5 on the next page.

4.5.1 Testing

Expected Performance

The instruction count of shakersort is expected to be exactly the same as bubblesort, while its cache performance is expected to be superior. A small amount of temporal reuse occurs at the ends of the array, as each key is used twice once its loaded into the cache. In addition, when the array is not more than twice the size of the cache, then the keys in the centre of the array are not ejected from the cache at all.

Finally, the branch prediction results are expected to be the same as in bubblesort, with one difference. Keys which must travel the entire way across the array are taken care of earlier in the sort. If the largest key is in the leftmost position, it must be moved the entire way right, which causes misses for every move in bubblesort, since each move is disjoint from the next. In shakersort, however, each move in the correct direction will follow a move in the same direction, each of which should be correctly predicted. This works for a lot of keys, which are more likely to move in continuous movements than they are in bubblesort, where only small keys on the right move continuously. The performance should improve noticeably as a result.

```
void shakersort(Item a[], int N)
{
   int i, j, k = N-1, sorted;
   for (i = 0; i < k;  )
   {
      for (j = k; j > i; j--)
         if (less(a[j],a[j-1]))
            exch(a[j-1],a[j]);
      i++;
      sorted = 1;
      for (j = i ; j < k; j++)
      {
         if (less(a[j+1],a[j]))
         {
            exch(a[j],a[j+1]);
            sorted = 0;
         }
      }
      if (sorted) return;
      k--;
   }
}
```

Figure 4.5: Shakersort code

4.6 Improved Shakersort

Improved shakersort incorporates the same improvements over shakersort as improved bubblesort does over bubblesort, that is, it skips over pockets of sorted keys in both directions, as well as ending early. The code for improved shakersort is in Figure 4.6 on the following page.

4.6.1 Testing

Expected Performance

It is expected that the improvements will make shakersort faster in all cases than its original form.

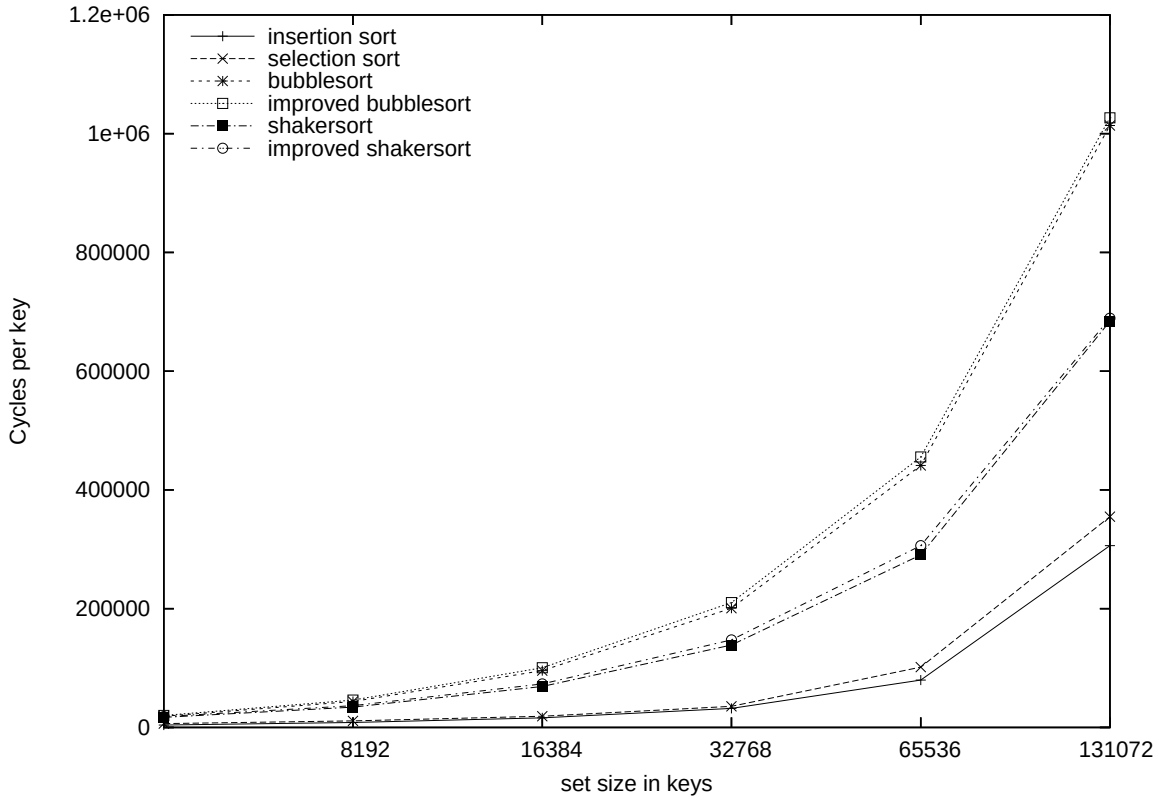
4.7 Simulation Results

Figure 4.7(b) shows the instruction count of each elementary sort. Each sort's instruction count is as predicted. Selection sort has the most instructions, and insertion sort has approximately half that number, as predicted. Bubblesort has almost the same number of instructions as selection sort, but shakersort has a significant number less. The improvement to bubblesort in improved bubblesort have increased the instruction count, contrary to predictions, and similarly in the improved shakersort. It appears the cost of skipping over a segment of the array is not worth the additional book-keeping cost of determining

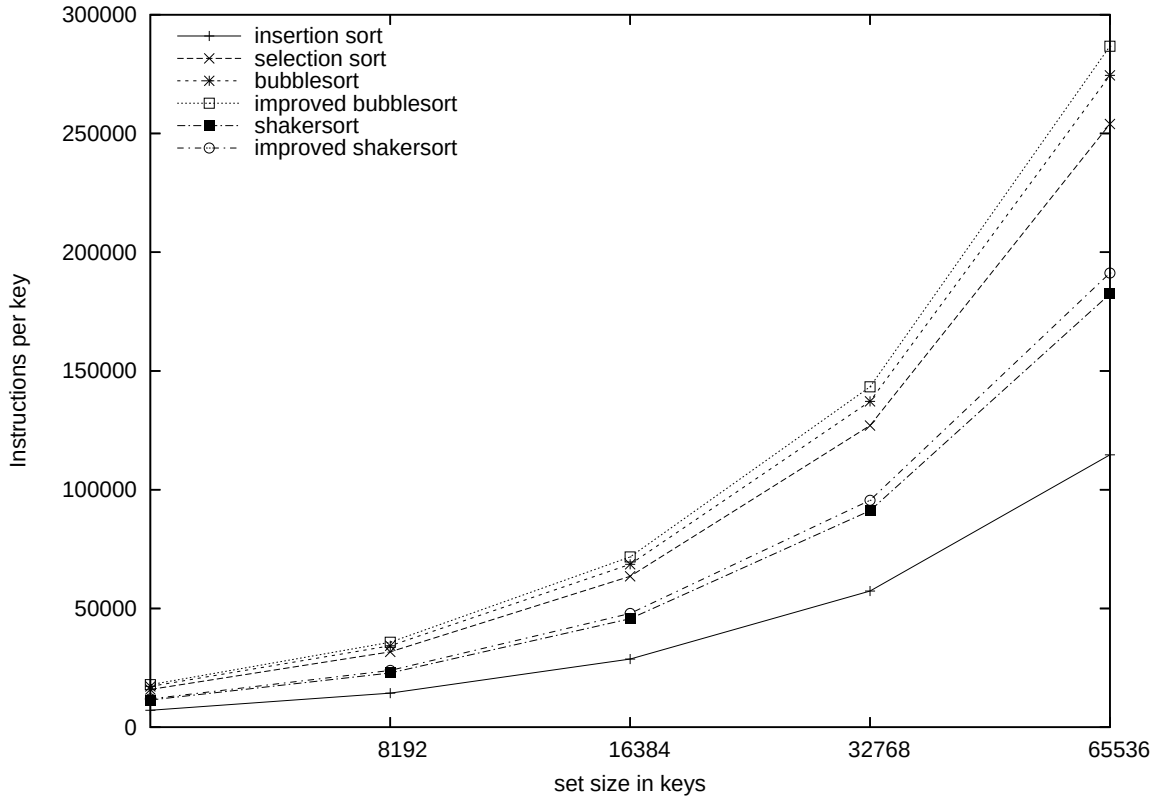
```
void improved_shakersort(Item a[], int N)
{
   int i = 0, j, k = N-1, l = N-1;
   for ( ; i < k; )
   {
      for (j = k; j > i; j--)
      {
         if (less(a[j],a[j-1]))
         {
            exch(a[j-1],a[j]);
            l = j;
         }
      }
      i = l;

      for (j = i ; j < k; j++)
      {
         if (less(a[j+1],a[j]))
         {
            exch(a[j],a[j+1]);
            l = j;
         }
      }
      k = l;
   }
}
```

Figure 4.6: Improved shakersort code

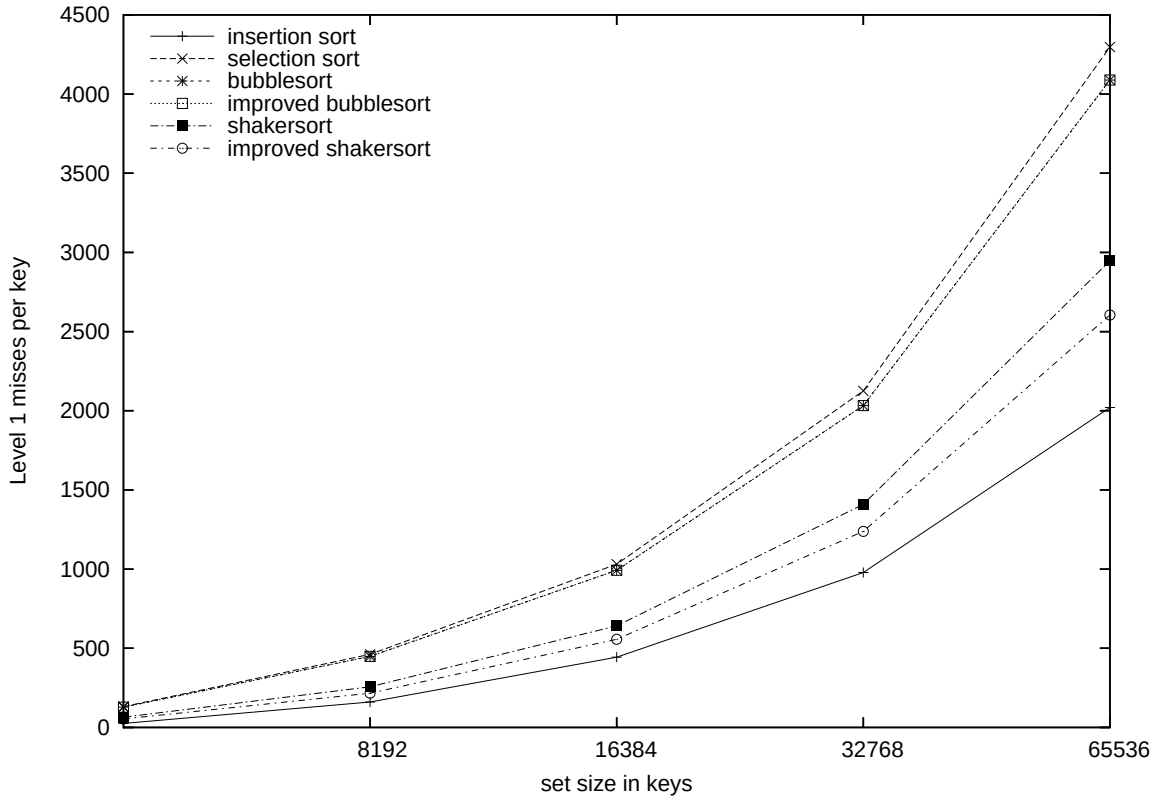


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

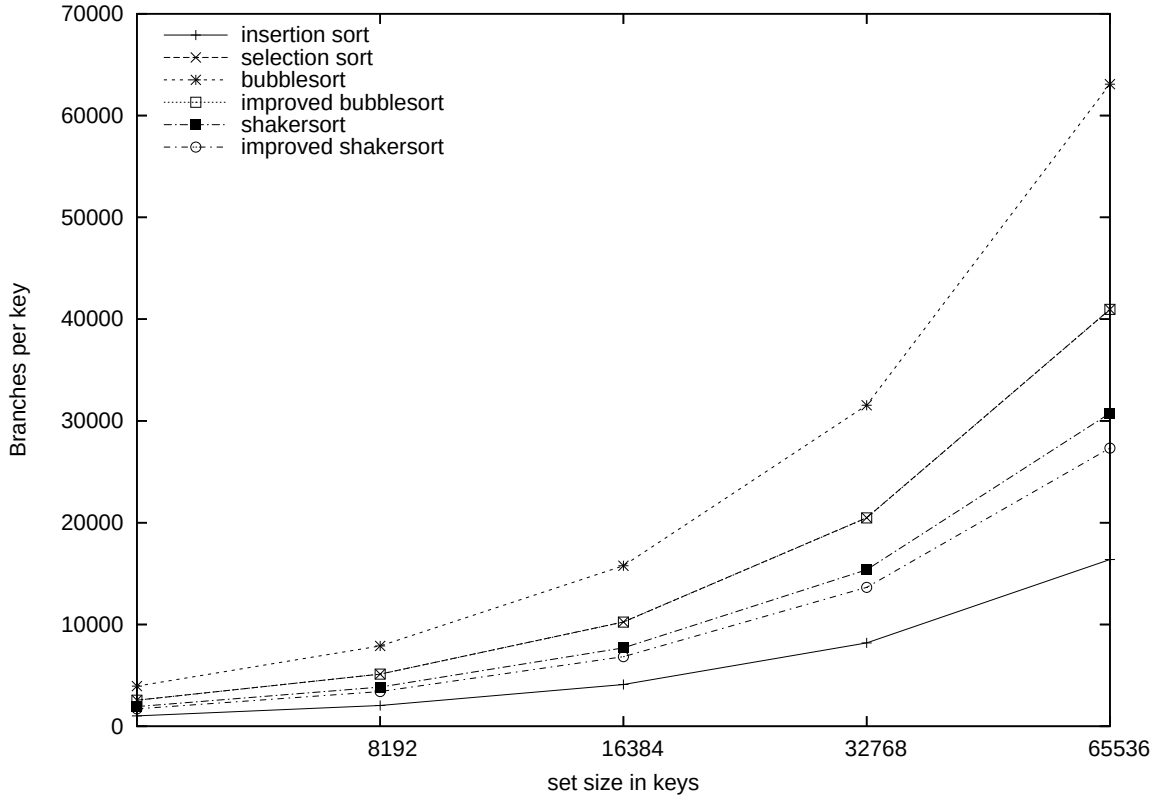


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 4.7: Simulated instruction count and empiric cycle count for elementary sorts

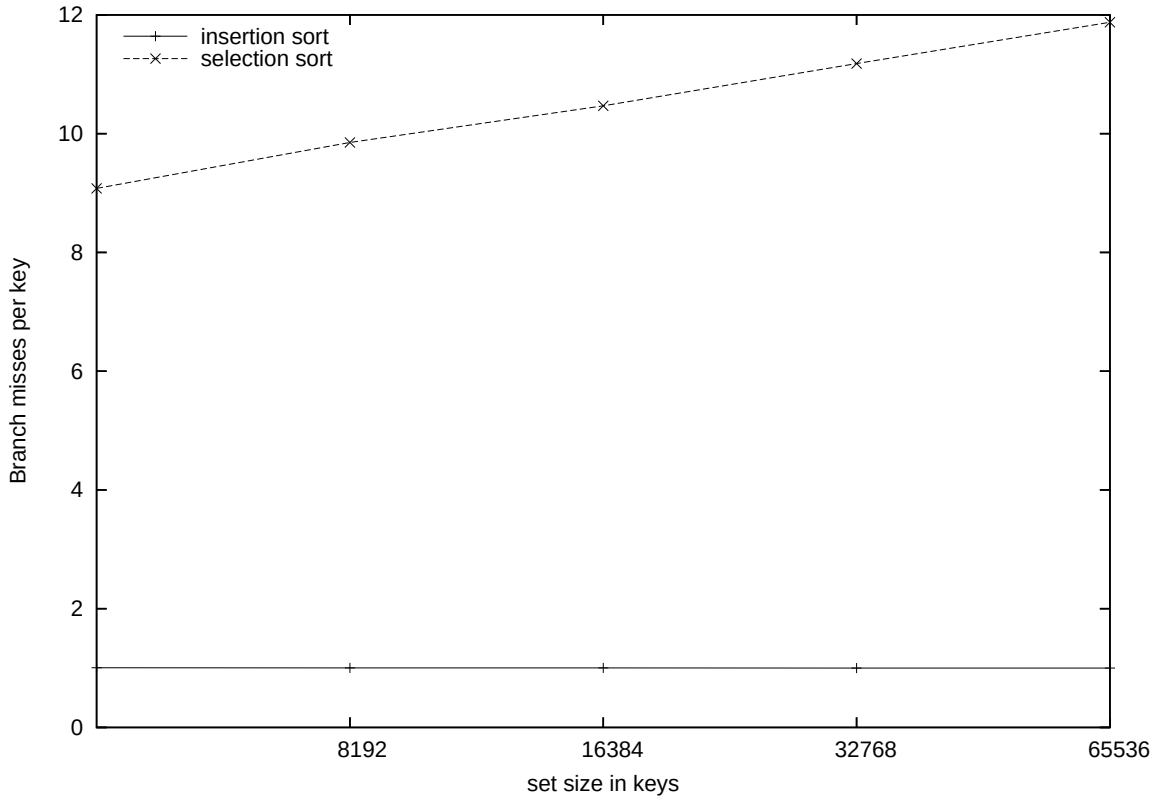


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache. Results shown are for a direct-mapped cache.

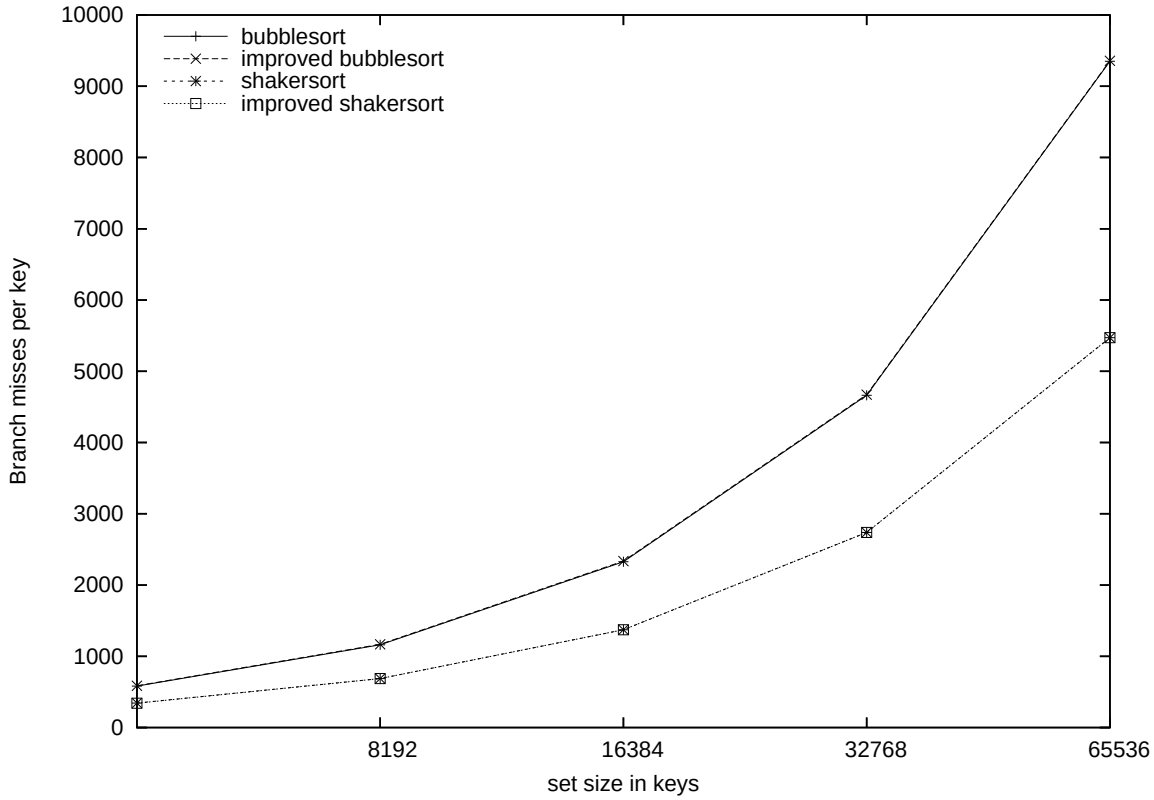


(b) Branches per key - this was simulated using `sim-bpred`.

Figure 4.8: Cache and branch prediction simulation results for elementary sorts



(a) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10 bit branch history register which was XOR-ed with the program counter. Results shown are for a 512 byte bimodal predictor.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10 bit branch history register which was XOR-ed with the program counter. Results shown are for a 512 byte bimodal predictor.

Figure 4.9: Branch simulation results for elementary sorts

when this can be done.

The level 2 cache performance of these sorts is not considered, as the set sizes are never larger than the level 2 cache. Instead, the level 1 caches are used, and their performance can be used to approximate the level 2 cache in these cases. These results are shown in Figure 4.8(a). Each set size is larger than the cache, and each sort iterates across the entire array each time, leading to very poor performance.

As expected, selection sort has the worst cache performance. This is due to the fact that it considers the entire unsorted segment of the array for each key to be put in place. Bubblesort, with similar behaviour, is only slightly better. Shakersort has a small amount of temporal reuse, reducing its number of misses. Improved shakersort is placed roughly between insertion sort and shakersort, showing that ending early is helpful in its case. This does not appear to be the case with improved bubblesort though, which performs identically to bubblesort.

Insertion sort performs best of these sorts in relation to cache misses. This is due to the fact that the keys it considered most recently are used again for the next key. These keys will be swapped out fairly regularly, however, but not nearly as often as selection sort, where they are swapped out at nearly every iteration.

Figure 4.8(b) shows the number of branches per key of these sorts, while Figure 4.9 shows the branch misses per key. Due to huge variations in predictability, we split the numbers of branch mispredictions into two charts. Selection sort causes around H_N mispredictions per key, which is remarkably small. The comparison branch in the inner loop is highly predictable on random data, because a good candidate minimum is usually found quickly.

Selection sort, meanwhile, has the same number of branches as bubblesort, but only $O(H_N)$ misses per key. The most interesting result, though, is insertion sort. This has exactly one miss per key. Although this result was not predicted in advance, on reflection this behaviour is obvious: every iteration involves moving the key left. The only time it will not move left is when its being put into place, and this only happens once for every key. This shows that it is possible to completely eliminate even comparative branch misses. It also establishes insertion sort as the best of the simple sorts, and it is used later in quicksort and mergesort as a result of this.

The number of misses for bubblesort and shakersort are so huge that we had to plot them on a separate chart. Nearly one in four comparisons are misses. As the data sets gets large, there are nearly 10000 branch misses involved in sorting a single key. The improved bubblesort had exactly the same number of misses and branches. The improved shakersort has a lot less branches, but the same number of misses. This is because skipping over sorted keys is predictable from the first branch miss.

The branch behaviour of bubblesort is particularly interesting when compared to selection sort. Although we usually think of the two algorithms as being different, they are actually very similar. Like selection sort, bubblesort finds the smallest (or largest) key in the unsorted part of the array and moves it to start (end) of the array. Thus, we would expect that bubblesort's branch prediction behaviour would be similar to that of selection sort.

However, in the process of moving the smallest element into position, bubblesort also rearranges other array elements. Selection sort sweeps across the array and always keeps track of the left-to-right minimum up to the current point. However, bubblesort actually moves this minimum, exchanging pairwise with each element it encounters which is larger. Thus, even the unsorted part of the array becomes partially sorted. We might expect that bubblesort's comparison branch for a partially sorted array might be more

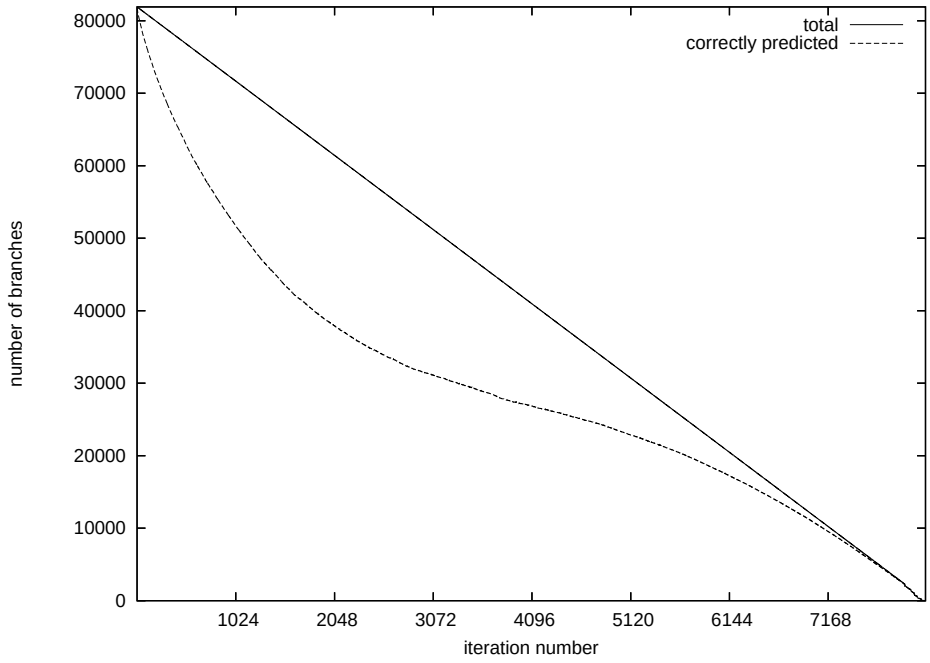


Figure 4.10: Behaviour of branches in bubblesort compared to the "sweep" number

predictable than for a random one, but it is not. With a random array, the smaller elements are equally likely to appear anywhere in the array. With a partially sorted array, the smaller elements tend to appear towards the start of the array which is, in bubblesort, the last place that we look. Thus, we change the candidate minimum much more often than if we were searching a randomly ordered array.

To demonstrate this effect, we measured how the predictability of bubblesort's comparison branch varies with successive sweeps across the array. We added a simple software simulation of a bimodal predictor to the source code of base bubblesort, and ran it on a large number of randomly generated arrays, each with 8192 elements.

In Figure 4.10 we see that on the early iterations the number of correct predictions is very close to the number of executed comparison branches. The comparison branch is highly predictable because bubblesort's search for the smallest element is similar to selection sort. However, with successive sweeps even the unsorted part of the array becomes increasingly sorted and the branch becomes more unpredictable. We see that the number of executed comparison branches falls linearly, but that the number of correctly predicted comparison branches falls much more quickly. Eventually, the array becomes so close to fully sorted that the branch becomes increasingly predictable in the other direction, and the number of correct predictions again approaches the number of executed branches. However, during the intermediate stage, a huge number of branch mispredictions have already occurred. Paradoxically, by partially sorting the array, bubblesort greatly increases the number of branch mispredictions and thus greatly increases its running time. Bubblesort would be substantially faster if, instead of attempting partially sort these intermediate elements, it simply left them alone.

A particularly interesting result comes from a comparison between shakersort and selection sort. The number of cycles to sort an array (shown in Figure 4.7(a)) is nearly twice as high on shakersort as it is for

selection sort. Shakersort has a significantly lower instruction count and cache miss count than selection sort, but has several orders of magnitude worse branch misses. This shows how important it is to consider branch misses in an algorithm, especially comparative ones. It also establishes that algorithms such as bubblesort and shakersort, will perform considerably worse than algorithms which have a correlation between successive branches, as in insertion and selection sorts.

4.8 Future Work

Simulations from elementary sorts take a lot of time. Running the simulation for bubblesort with 65536 keys takes nearly a day, and the simulation for 4194304 keys would take nearly 11 years to run. Due to this, it is necessary to extrapolate results for larger data sets from the results from these smaller sets. The only place where it's necessary to have larger data sets is in the case of cache results. The cache behaviour of these sorts should change dramatically once the keys no longer fit in the cache, as they do in subsequent chapters with other sorts. The results can be simulated, by using a smaller cache, but in this case all the data sets are too large. It would be easy to use data sets smaller than the cache, but in this case many factors would not have time to be amortised, such as filling the cache and branch prediction tables.

Each of these sorts could have the number of iterations across the array reduced by a half or a third by considering two or three keys at a time. The resultant reduction in cache misses is predictable, and instruction count should reduce. The new branch prediction results, especially in the case of insertion sort, may be significantly worse, or may stay the same. The results of this would be interesting.

It should be possible to make more cache-conscious versions of these sorts. Making a bubblesort which performs a bubble across an entire cache line would be an example. Another would be a cache-conscious selection sort. It might switch the direction of the search to reuse lines at the end of the array. A cache-conscious bubblesort could also do something similar, which would make it into shakersort. However, these would only make a difference on large data sets, when better sorts such as quicksort should be used. In addition, the complexity of improving elementary sorts means it may be easier to code a simple $O(NlogN)$ sort, which would perform better for the same effort. Due to this, a cache-conscious elementary sort is but an academic exercise.

Chapter 5

Heapsort

Heapsort is an in-place sort which works in $O(NlogN)$ time in the average and worst case. It begins by creating a structure known as a *heap*, a binary tree in which a parent is always larger its child, and the largest key is at the top of the heap. It then repeatedly removes the largest key from the top and places it in its final position in the array, fixing the heap as it goes.

5.0.1 Implicit Heaps

A *binary tree* is a linked data structure where a node contains data and links to two other nodes, which have the same structure. The first node is called the *parent*, while the linked nodes are called *children*. There is one node with no parents, called the *root*, from which the tree branches out, with each level having twice as many nodes as the previous level.

A heap is a type of binary tree with several important properties. Firstly, the data stored by the parent must always be greater than the data stored by its children. Secondly, in a binary tree there is no requirement that the tree be full. In a heap, it is important that every node's left child exists before its right child. With this property, a heap can be stored in an array without any wasted space, and the links are implied by a nodes position, rather than by a pointer or reference. A heap done in the fashion is called an *implicit heap*.

To take advantage of the heap structure, it must be possible to easily traverse the heap, going between parent and child, without the presence of links. This can be done due to the nature of the indices of parents and their children. In the case of an array indexed between 0 and $N-1$, a parent whose index is i has two children with indices of $2i+1$ and $2i+2$. Conversely, the index of a parent whose child has an index of j is $\lfloor (j+1)/2 - 1 \rfloor$. It is also possible to index the array between 1 and N, in which case the children have indices of $2i$ and $2i+1$, while the parent has an index of $\lfloor j/2 \rfloor$. Using this scheme can reduces instruction cost, but results in slightly more difficult pointer arithmetic, which cannot be performed in some languages[1].

[1] In C this can be done by `Item a[] = &array[-1];`

5.0.2 Sorting with Heaps

To sort an array with a heap, it is necessary to first create the heap. This is known as *heapifying*, and can be performed in two ways.

The first method involves *sifting*[2] keys up the heap. Assuming that a heap already exists, adding a key and re-heapifying will maintain the heap property. This can be done by adding the key at the end of the heap, then moving the key up the tree, swapping it with smaller parents until it is in place. An unsorted array can be sorted by continually heapifying larger lists, thereby implicitly adding one key every time to a sorted heap of initial size one. This technique was invented by Williams, and LaMarca refers to it as the *Repeated-Adds* algorithm.

The second technique is to heapify from the bottom up, by sifting keys down. Beginning at half way through the array (this is the largest node with children), a parent is compared to its two children and swapped down if it is smaller. This continues upwards until the root of the heap is reached. This is the technique specified by Floyd and LaMarca refers to this as Floyd's method.

The second step, described next, is to destroy the heap, leaving a sorted array.

In-place Sorting

For an in-place sort, a heap is created as specified above. Then the root of the heap (the largest key) is swapped with the last key in the heap, and the new root is sifted down the heap which has been shrunk so as not to include the old root, which is now in its final position. This is repeated for the entire heap, leaving a fully sorted array.

Floyd suggested an improvement to this, where the new root is sifted straight down without regard for whether it is larger than its child. Typically, it will end near the bottom, so sifting up from here should save comparisons. Sedgewick notes speed improvements only when comparisons are move expensive than swaps, such as in the case of sorting strings. In our case, the comparisons are very predictable moving down the heap, and so this step should not be done.

Out-of-place Sorting

Up until this point, the heap property used has been that a parent will be larger than its child. If this property is change so that the parent is always smaller than its child, then the result is an out-of-place sort. The smallest key is removed from the heap, and placed in its final place in the array. The heap is then re-heapified. LaMarca refers to the steps involved as *Remove-min*.

5.1 Base Heapsort

LaMarca recommends to use of the Repeated-Adds algorithm for reducing cache misses. As such, his base heapsort uses Floyd's method, ostensibly at the behest of major textbooks, each of whom claim Floyd's

[2]Sifting up and down is also referred to as fixing up and down, and algorithms that do this are called *fix-up* or *sift-up*, and *fix-down* or *sift-down*.

```
void fixDown(Item a[], int parent, int N)
{
   Item v = a[parent];
   while(2*parent <= N) /* check for children */
   {
      int child = 2*parent;

      /* find larger child */
      if (less(a[child], a[child+1])) child++;

      /* stop when not larger than parent */
      if (!less(v, a[child])) break;

      a[parent] = a[child];
      parent = child;
   }
   a[parent] = v;
}

void base_heapsort(Item a[], int N)
{
   /* construct the heap */
   for(int k=N/2; k >= 1; k--) fixDown(&a[-1], k, N);

   /* destroy the heap for the sortdown */
   while(N>1)
   {
      Item temp = a[0];
      a[0] = a[--N];
      fixDown(&a[-1], 1, N);
      a[N] = temp;
   }
}
```

Figure 5.1: Simple base heapsort

method gives better results. This is certainly backed up by Sedgewick, who recommends this method, but in initial tests, the out-of-place Repeated-Adds technique had a lower instruction count. Regardless, the base algorithm used here is Floyd's in-place sort. Code for a base heapsort is in Figure 5.1.

A difficulty of heapsort is the need to do a bounds check every time. In a heap, every parent must have either no children, or two children. The only exception to this is the last parent. It is possible that this has only one child, and it is necessary to check for this as a result. This occurs every two heapifies, as one key is removed from the heap every step. Therefore it is not possible to just pretend the heap is slightly smaller, and sift up the final key later, as it might be with a statically sized heap.

LaMarca overcomes the problem by inserting a maximal key at the end of a heap if there's an even number of nodes in the array. This gives a second child to the last element of the heap. This is an effective technique for building the heap, as the number of keys is static. When destroying the heap, however, the maximal key must be added to every second operation. We found this can be avoided with an innovative technique, with no extra instructions. If the last key in the heap is left in place as well as

being sifted down the heap, it will provide a sentinel for the sift-down operation. The key from the top of the heap is then put into place at the end. This technique reduces instruction count by approximately 10%.

The only other optimization used is to unroll the loop while destroying the heap. This has minimal affect, however.

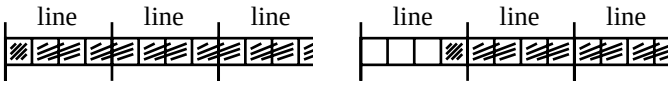
5.2 Memory-tuned Heapsort

LaMarca notes that the Repeated-Adds algorithm has a much lower number of cache misses than Floyd's. The reason for this is simple. Consider the number of keys touched when a new key is added to the heap. In the case of Floyd's method, the key being heapified is not related to the last key that was heapified. Reuse only occurs upon reaching a much higher level of the heap. For large heaps, these keys may be swapped out of the cache by this time. As a result, Floyd's method has poor temporal locality.

The Repeated-Adds algorithm, however, displays excellent temporal locality. A key has a 50% chance of having the same parent, a 75% of the same grandparent and so on. The number of instructions, however, increases significantly, and the sort becomes out-of-place. For this reason, the Repeated-Adds algorithm is used for memory-tuned heapsort.

The cost of building the heap is completely dominated by the cost of destroying it. Less than 10% of heapsort's time is spent building the heap. Therefore optimizations must be applied when destroying the heap.

LaMarca's first step is to increase spatial locality. When sifting down the heap, it is necessary to compare the parent with both children. Therefore it is important that both children reside in the same cache block. If a heap starts at the start of a cache block, then the sibling of the last child in the cache block will be in the next cache block. For a cache block fitting four keys, this means that half the children will cross cache block boundaries. This can be avoided by padding the start of the heap. Since the sort is out of place, an extra cache block can be allocated, and the root of the heap is placed at the end of the first block. Now, all siblings will reside in the same block.



(a) a normal heap (b) a padded heap

A second point about heapsort's spatial locality is that when a block is loaded, only one set of children will be used though two or four sets may be loaded. The solution is to use a heap higher *fanout*, that is, a larger number of children. The heap used so far could be described as a 2-heap, since each parent has two children. LaMarca proposes using a *d-heap*, where d is the number of keys which fit in a cache line. In this manner, every time a block is loaded, it is fully used. The number of cache blocks loaded is also decreased, since the height of the tree is halved by this.

This step changes the indexing formula for the heap. In this method, a parent i has d children, indexed from $(i * d) + 1$ to $(i * d) + d$. The parent of a child j is indexed by $\lfloor (i-1)/d \rfloor$.

The height of a 2-heap is $\lceil log_2(N) \rceil$, where N is the number of keys in the heap. By changing to a

4-heap, the height is reduced to $\lceil log_4(N) \rceil = \lceil \frac{1}{2} log_2(N) \rceil$. The height of an 8-heap is reduced again to $\lceil log_8(N) \rceil = \lceil \frac{1}{3} log_2(N) \rceil$. However, the number of comparisons needed in each level is doubled, since all the children will be unsorted. LaMarca claims that as long as the size of the fanout is less than 12, a reduced instruction count will be observed.

In the base heapsort, an improvement was added that removed the need for a sentinel. Since the 4- and 8-heaps were unsorted, this optimization had to be removed. Leaving the key being copied in place was still possible, but it needs to be replaced by a maximal key before trying to sift the next key down the heap, lest it were chosen to be sifted up. Placing the sentinel is still cheaper than the bounds check, however, as the bounds check will need to be performed $NlogN$ times, compared to N times for the sentinel. The cost of the sentinel write is more expensive but the extra $logN$ instructions saved will mean that this increase will be an order of magnitude lower than the bounds check.

5.3 Results

All simulations presented here were performed on 32-byte cache lines. As a result, 8-heaps were used for heapsort, since 8 4-byte `int`s fit the cache line. LaMarca used 64 bit ints, and while this doesn't affect results, it means that fewer keys fit into his cache lines. Since modern cache lines are typically longer than 4 ints, we stick with 32-byte cache lines. `UINT_MAX` was used as the sentinel.

5.3.1 Expected Results

The instruction count of heapsort is $O(Nlog_d(N))$, although a higher d means that each step will take longer. As the fanout of the heap doubles, then the height of the heap halves, but each heap step is more expensive. It is expected though, that increasing d will reduce the instruction count until the point at which the cost of a heap step doubles, as compared to a heap with a smaller fanout.

An additional consideration is that the base heapsort has more instructions than its out-of-place equivalent. This means that the 4-heap will have a better instruction count, as well as better cache performance than the base heapsort. The 8-heap should have the best cache performance of them all, due to fully using each cache line, taking advantage of the spatial locality of the cache line.

LaMarca reports base heapsort has very poor temporal locality, and that the memory-tuned heapsort will have half the cache misses of base heapsort.

It is expected that the number of comparison branches stays the same with an increased fanout. When the height of the heap halves, the number of comparisons in each step doubles. However, the proportion of those which are predictable changes. The key being sifted down the heap is likely to go the entire way down the tree. Therefore the comparison between the smallest child and the parent is predictable. However, the comparisons to decide which child is smallest are not predictable. In a 2-heap, there is one comparison between children, meaning that half the branches will probably be correctly predicted. An 8-heap has seven comparisons between children, and will probably have about 1.59 mispredictions, as it behaves similarly to selection sort (See 4.1.1). It is possible, therefore, that the 8-heap will have slightly fewer mispredictions due to the combination of the shallower heap and the larger fanout.

However, the creation of the heap will have fewer misses with a larger fanout, due to the reduced height. In practice though, the implications of this are minimal. Most of the time is spent in the destruction of

the heap, and this will have far more branches than the creation phase.

5.3.2 Simulation Results

As expected, the 4-heap performs better than the base heapsort. However, the increased number of comparisons of the 8-heap increases the instruction count to even higher than base heapsort, due to the disparity between the increase in comparisons in set of children, and the lesser reduction in the height of the heap. This is shown in Figure 5.2(b).

The level 1 cache results, shown in Figure 5.3(a), show the difference between the two types of heap. The slope of the base heapsort result is much greater than all the others, showing base heapsort's poor temporal locality, with the 8-heap having the fewest misses. This continues into the level 2 cache, where the flattest slope belongs to the 8-heap. The difference between fully-associative and direct-mapped is only minor, with the fully-associative performing slightly better than the direct-mapped cache.

In the graph, the base heapsort's results flat-line for a step longer than the other heapsorts' results. This is due to being in-place, since twice as many keys fit in the cache without a corresponding increase in misses. For the same reason, the base heapsorts have no compulsory misses, while the out-of-place sorts have one compulsory miss in eight, when the auxiliary heap is being populated.

The branch prediction behaves differently to the expected results, as the prediction ratio gets worse as the fanout gets larger. The reason we predicted differently was due to selection sort's $O(H_N - 1)$ mispredictions; however, this is over the entire length of the array, and it is not accurate to extrapolate these results over a sequence only seven keys long.

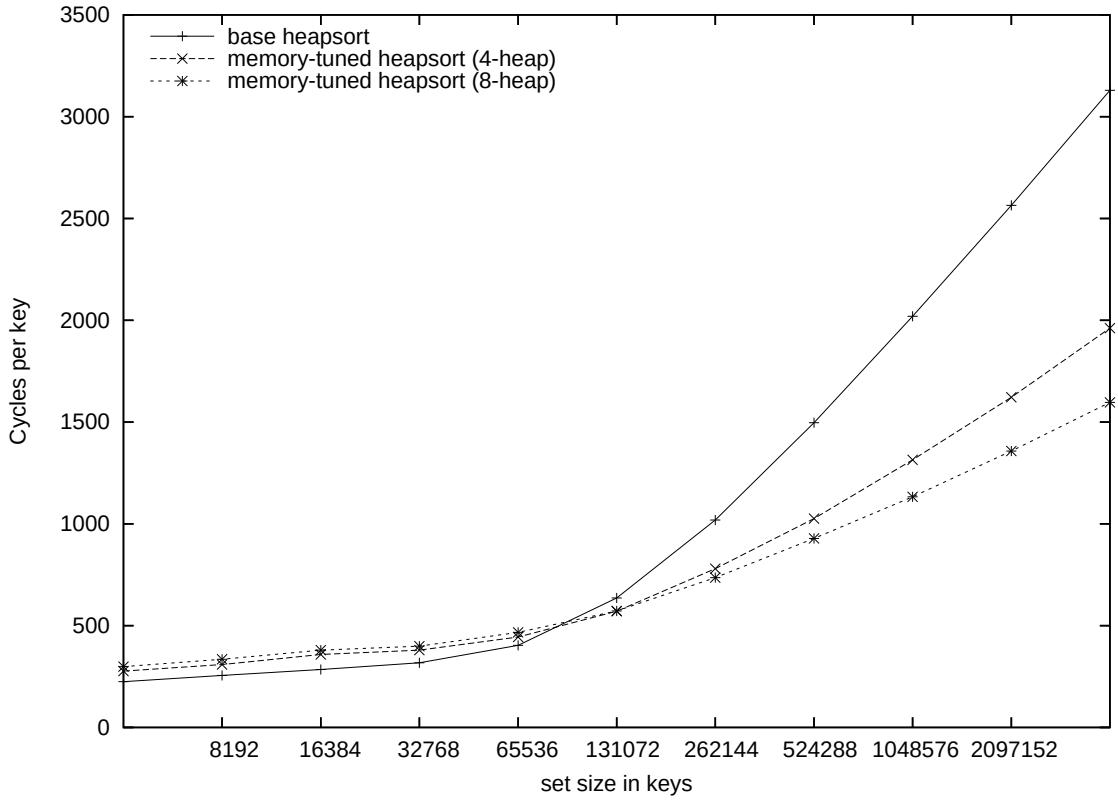
The difference between the 4-heap and 8-heap is minor, and the difference for larger fanout would be less still. The difference between the 2-heap and 4-heap result is interesting though, the 2-heap has more branches, but less misses per key.

The difference between different types of predictors is also visible in the results, shown in Figure 5.4(b). The largest table size is shown for the two-level predictor. The smallest table size is shown for the bimodal predictor, as all the bimodal table sizes show identical results. In the case of the 2- and 8-heaps, the bimodal predictor works better than the two-level adaptive predictor. The warm-up time of the table-driven predictors is visible in the base heapsort results, though it's amortised for larger data sets.

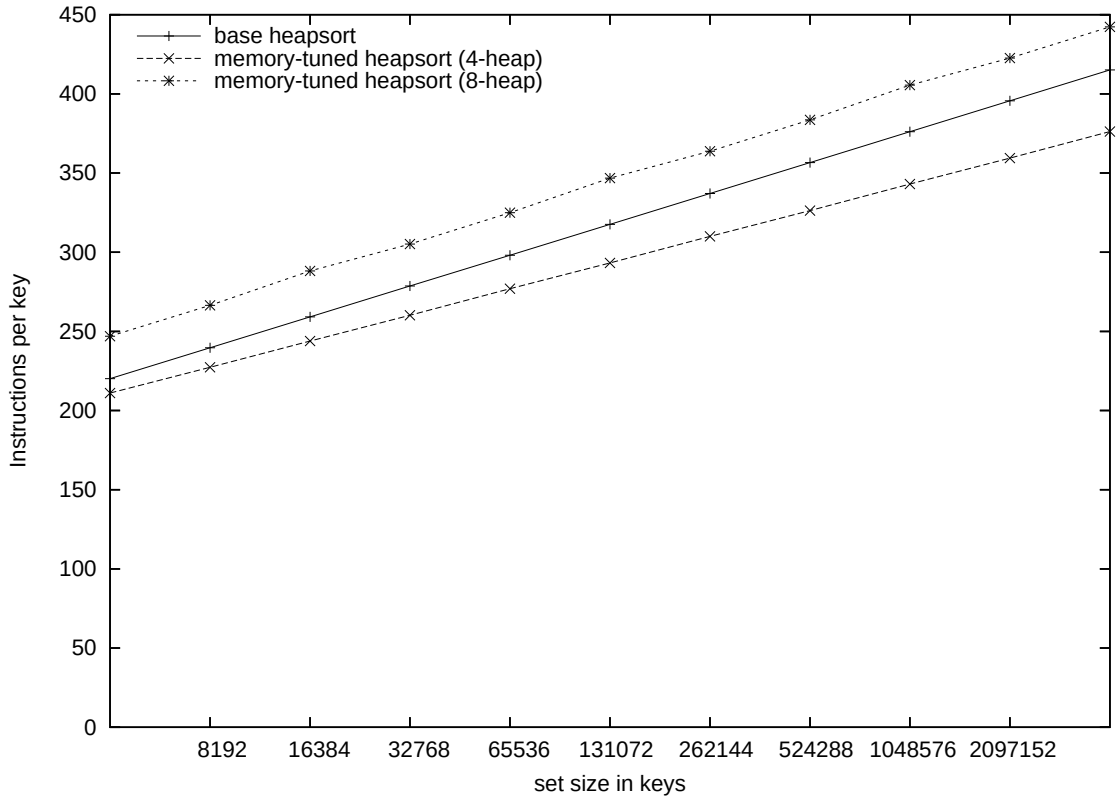
Figure 5.2(a) shows the actual time taken for heap sort on a Pentium 4 machine. The figure shows that the base heapsort performs best when the data set fits in the cache. After that point, the 8-heap performs best, and its cycle count increases with a smaller slope. This shows the importance of LaMarca's cache-conscious design even nine years after his thesis. It also shows the importance of reduced branch mispredictions: the base heapsort has more instructions and level 1 cache misses, but still outperforms the 4-heap, until level 2 cache misses take over. However, this is hardly conclusive: the lower level 2 cache misses may account for this.

The similarities to LaMarca's results are startling. The shape of the instruction and cache miss curves are identical, as are the ratios between the base and memory-tuned versions in the case of level 1 and 2 cache misses, instruction count and execution time.

Slight differences exist though. The level 2 cache miss results here indicate that while still sorting in the cache, the number of misses in the in-place heapsort is less than that of the out-of-place sorts. LaMarca,

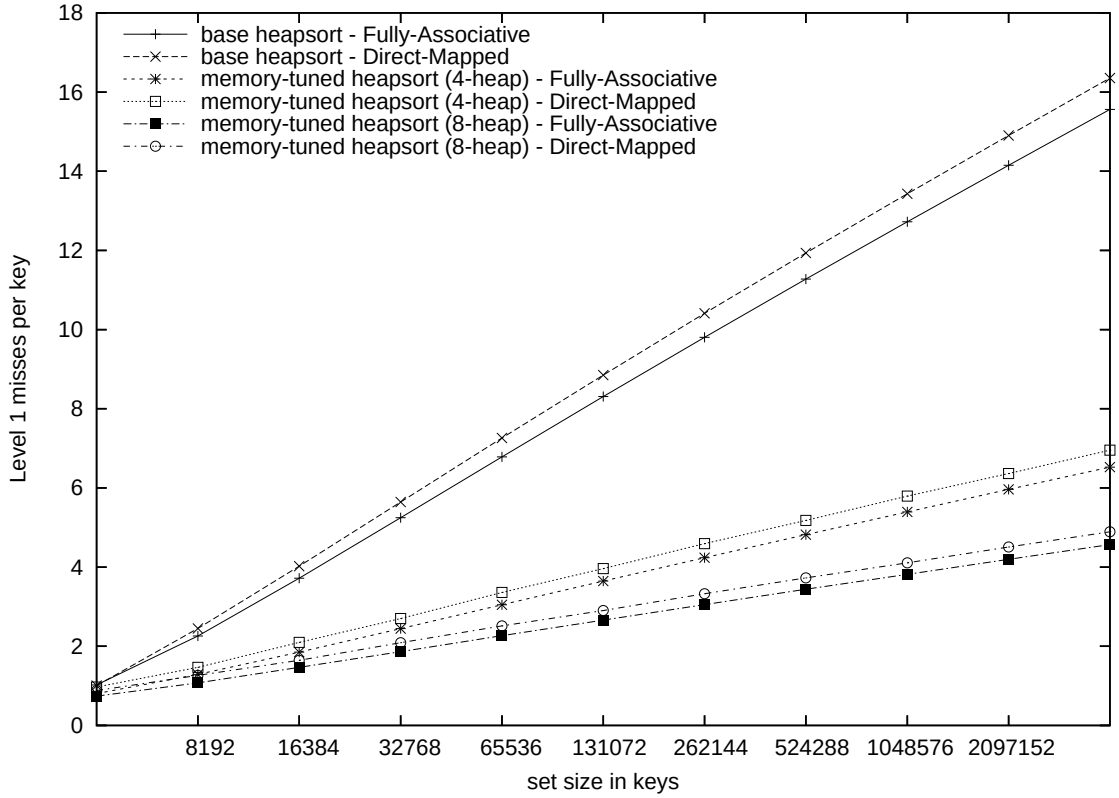


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

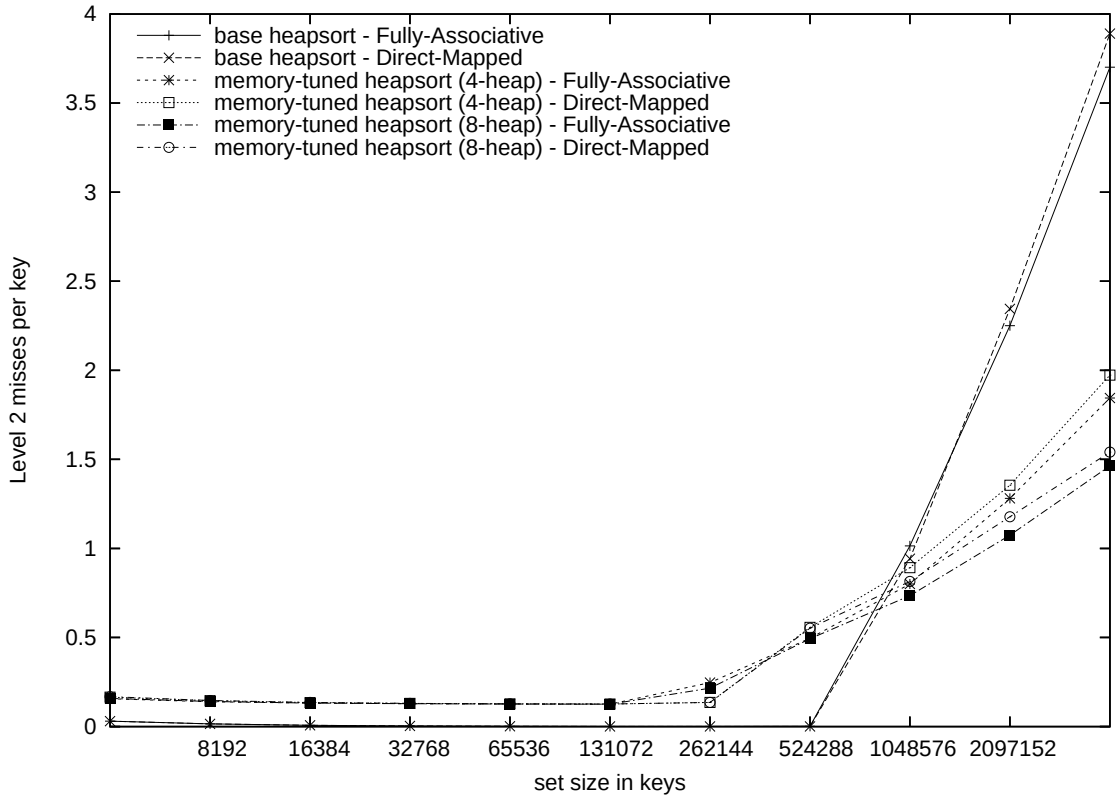


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 5.2: Simulated instruction count and empiric cycle count for heapsort

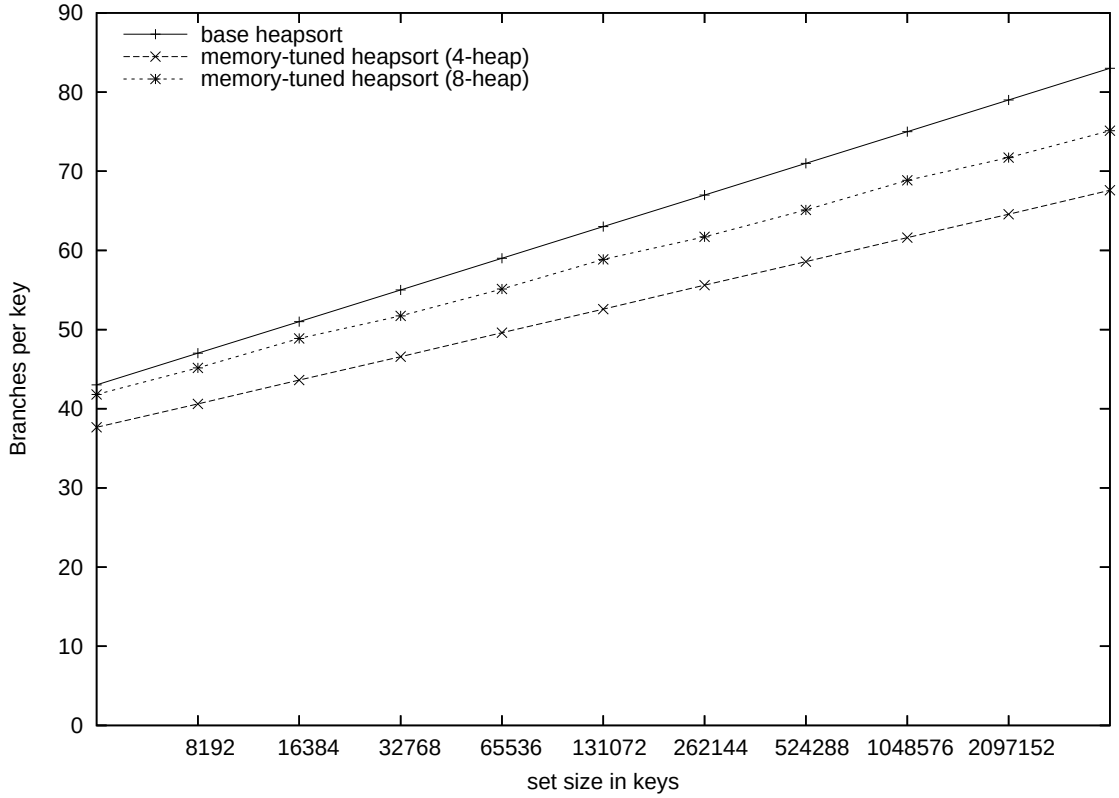


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache.

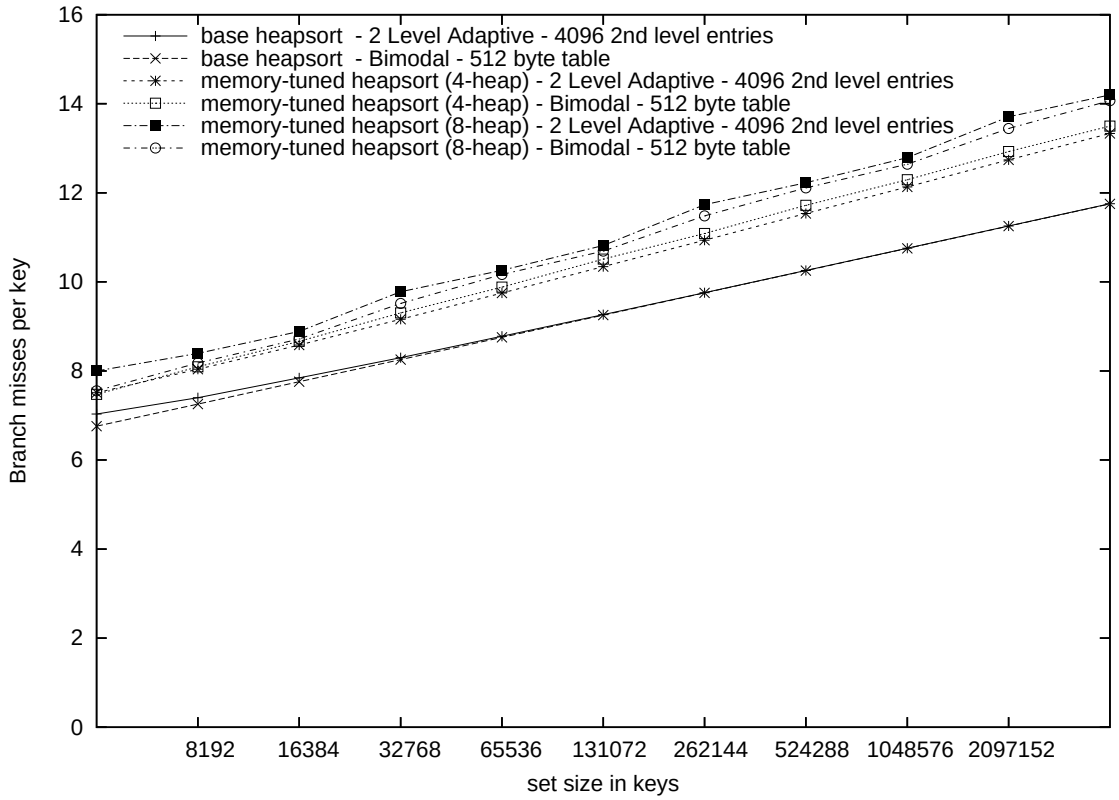


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32-byte cache line.

Figure 5.3: Cache simulation results for heapsort



(a) Branches per key - this was simulated using `sim-bpred`.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10-bit branch history register which was XOR-ed with the program counter.

Figure 5.4: Branch simulation results for heapsort

however, has the same amount of misses in both cases. Similarly, the results here indicate that base heapsort experiences a sharp increase in cache misses. Memory-tuned heapsort also has a similar increase, though LaMarca's results show that these occur at the same position in the graph, in other words, when the set size is the same. Our results, however, show that this occurs earlier in memory-tuned heapsort, due to the fact that it is an out of place sort.

Finally, though this is likely to be related the previous differences, base heapsort performs slightly better for small sets than the memory-tuned heapsort's do in the results presented here. LaMarca's results indicate that the memory-tuned heapsort has better results the entire way through the cache. This may relate to the associativity: LaMarca's tests were performed on a DEC AlphaStation 250 with a direct-mapped cache. The Pentium 4's cache is 8-way associative, and so the unoptimized algorithm would not suffer as much.

5.4 A More Detailed Look at Branch Prediction

Figure 5.5(a) shows the behaviour of each branch in the base heapsort algorithm. The '*Creation, Has Children*' branch refers to the check as to whether a key has children, during the creation of the heap. The '*Creation, Child Comparison*' branch is the comparison between the two children in the 2-heap, and the '*Creation, Child Promotion*' refers to checking if the child is larger than its parent. The same checks are later made during the destruction of the heap, and are referred to as '*Destruction, Has Children*', '*Destruction, Child Comparison*' and '*Destruction, Child Promotion*'.

In the memory-tuned heapsort, the creation phase is simpler, involving merely one branch, called '*Fix-up*' here. The other columns refer to the destruction phase, where most of the work takes place in memory-tuned heapsort. When a 4-heap is used (see Figure 5.5(b)), there are three comparisons instead of just one. Similarly, in an 8-heap (Figures 5.5(c) and 5.5(d)), there are seven comparisons.

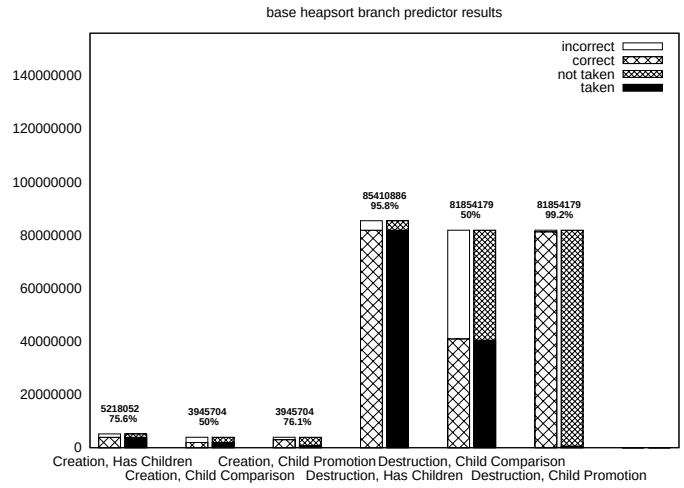
The results of the software predictor show the relation of the branches in the creation and destruction of the heap. The base heapsort results show that the time taken to create the heap is not significant relative to the time taken to sort from the heap.

The most interesting thing is the relation between the predictability of the comparisons during the destruction of the 2-, 4- and 8-heaps. Compare the columns marked '*Destruction, Child Comparison*' and '*Destruction, Child Promotion*' in Figure 5.5(a) (the 2-heap), with the columns marked '*Comparison 0*' to '*Child Promotion*' in Figure 5.5(b) (the 4-heap) and Figures 5.5(c) and 5.5(d) (the 8-heap).

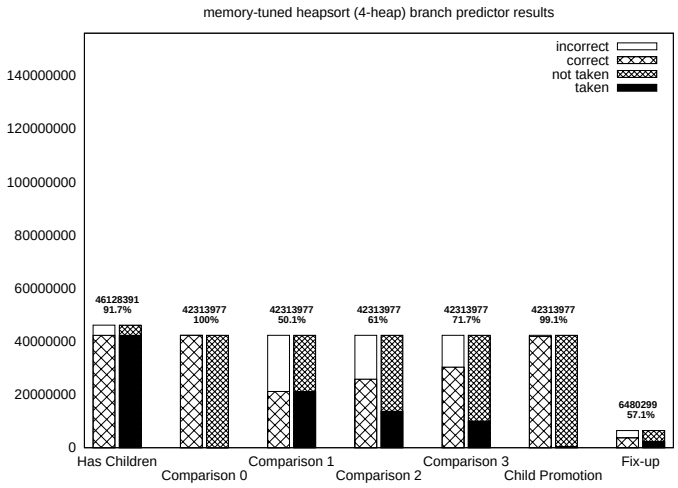
Though the number of times each comparison is used shrinks as the size of the fanout increased, overall the number of branches increases. In total, there are 50% more comparisons in the 4-heap than the 2-heap, and 150% more in the 8-heap.

More importantly, the accuracy of the each successive comparison increases in the same way as selection sort (see Section 4.1.1), becoming more predictable as the fanout increases. For example, in the 8-heap, the first comparison is predicted correctly 50% of the time, while the next is predicted correctly 61% of the time, then 71%, 78%, and so on in a curve, which is shown in Figure 5.6.

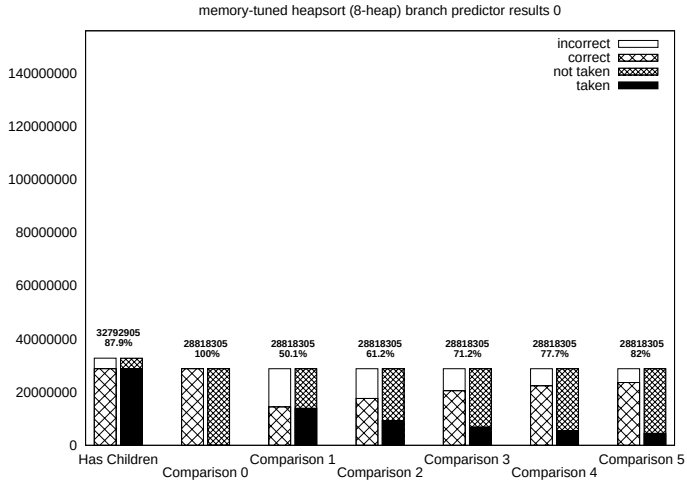
The relationship between these numbers and $H_N - 1$ is relatively straightforward. $H_N - 1$ represents the cumulative number of changes in the minimum. There first element is guaranteed to be the minimum. The second element has a 50% chance of being smaller than the first. If we examine a third element, then



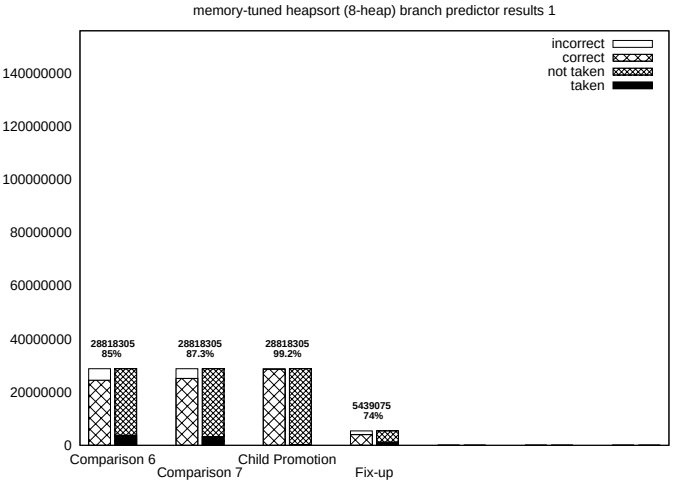
(a)



(b)



(c)



(d)

Figure 5.5: Branch prediction performance for base and memory-tuned heapsorts. All graphs use the same scale. Above each column is the total number of branches and the percentage of correctly predicted branches for a simulated bimodal predictor.

the chances of that element being smaller than two other elements is one in three. Therefore, this branch will be taken one third of the time. As the number of elements we examine increases, the probability of the nth element being smaller than all elements seen so far is $\frac{1}{n}$.

Meanwhile, Figure 5.6 shows the rate of correct predictions. The loop has been fully unrolled, so there is a separate branch (with its own predictor) for each of the seven comparisons. As a result, we expect $1-y = \frac{1}{x+1}$, where x and y are the Cartesian coordinates of each point on the graph. $1-y$ is the number of mispredictions from the graph, while $\frac{1}{x+1}$ is the increase in H_N for step N.

An interesting question is whether fully unrolling this loop, and thus separating the comparison branch into seven separate branches, has an impact on branch prediction. Essentially, there are two effects at work if we replace the sequence of branches with a single branch. First, the single branch predictor will spend most of its time in the state where it predicts not-taken, rather than taken. This is because all branches apart from the first are biased in the not-taken direction. This should improve accuracy. On the other hand, on those occasions when the bi-modal branch predictor is pushed towards predicting taken, the next branch is highly likely to be mispredicted. This is especially true because the first two or three branches are most likely to push the predictor in the taken direction, whereas the later branches are those that are mostly likely to result in a misprediction in this state (because they are almost always not-taken).

We investigated the trade-offs in these two effects and we discovered that using a single branch does indeed cause the branches for the earlier comparisons to become more predictable. However, this improvement is outweighed by a loss in predictability of the later comparisons. The overall prediction accuracy fell by around 2% when we used a single branch, rather than unrolling the loop. This result is interesting because it shows that relatively small changes in the way that an algorithm is implemented can have an effect on branch prediction accuracy. In this case, unrolling does not just remove loop overhead. It also improves the predictability of the branches within the loop.

When the array is short, as in this example, the branch predictor has too much feedback to be close to $H_N - 1$. For $N = 2$, we would approximate a 33% misprediction ratio using $H_N - 1$, but we measure 39%. The approximated and actual ratios become closer as N increases, as we would expect, and for $N = 7$, we approximate 12.5% and measure 12.7%. We conclude that initial suggestion is correct: while the branch prediction ratio of finding the minimum is very close to $H_N - 1$, the relationship is less accurate for very short arrays.

We observe also that the more branches there are, the more are correctly predicted, and that there may a threshold below which the number of branches will not drop. This is discussed further in Section 7.5.1.

5.5 Future Work

LaMarca later published heapsort code in [LaMarca 96a]. The ACM Journal of Experimental Algorithms requires that source code be published, and comparing simulated results from there to the sorts created here might have interesting results and would explain the discrepancies highlighted in this chapter.

The formula we use to explain selection sort's branch prediction performance breaks down for short arrays, as in our 8-heap. It should be possible to more accurately model and therefore predict the branch prediction ratio when the array is short. Our results show us what we expect the answer to be, but we have not attempted to create a formula to predict it.

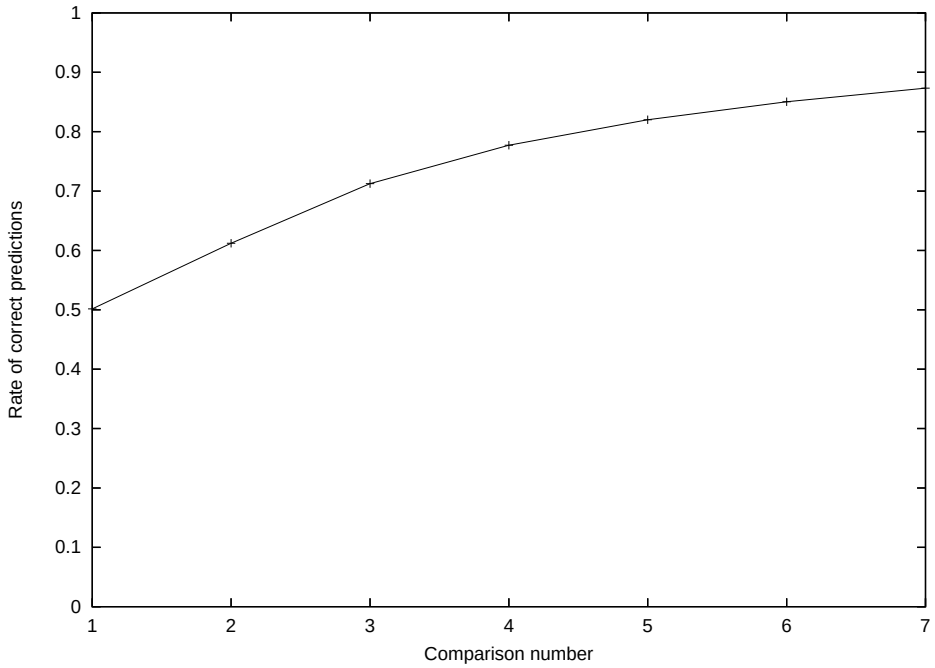


Figure 5.6: Predictability of branches in memory-tuned heapsort, using an 8-heap.

Chapter 6

Mergesort

Merging is a technique which takes two sorted lists, and combines them into a single sorted list. Starting at the smallest key on both lists, it writes the smaller key to a third list. It then compares the next smallest key on that list with the next smallest on the other, and writes the smaller to the auxiliary list. The merging continues until both lists are combined into a single sorted list.

Mergesort works by treating an unsorted array of length N as N sorted arrays of length one. It then merges the first two arrays into one array of length two, and likewise for all other pairs. These arrays are then merged with their neighbours into arrays of length four, then eight, and so on until the arrays are completely sorted.

Mergesort is an out-of-place sort, and uses an auxiliary array of the same size for the merging. Since a lot of steps are required, it is more efficient to merge back and forth - first using the initial array as the source and the auxiliary array as the target, then using the auxiliary array as the source and the initial array as the target - rather than merging to the auxiliary array and copying back before the next merge step.

The number of merges required is $log_2(N)$, and if this number is even then the final merge puts the array in the correct place. Otherwise, the sorted array will need to be copied back into the initial array.

6.1 Base Mergesort

6.1.1 Algorithm N

For his base algorithm, LaMarca takes Knuth's algorithm N [Knuth 98]. LaMarca describes this algorithm as functioning as described above; lists of length one are merged into lists of length two, then four, eight, and so on. However, algorithm N does not work in this fashion. Instead, it works as a *natural* merge. Rather than each list having a fixed length at a certain point, lists are merged according to the initial ordering. If it happens that there exists an ascending list on one side, this is exploited by the algorithm. While this results in good performance, the merge is not perfectly regular, and it is difficult or impossible to perform several of the optimizations LaMarca describes on it. We attempted to contact him in order

to discuss this, but received no response. It is likely that the improvements were actually applied to Knuth's algorithm S.

6.1.2 Algorithm S

Algorithm S is a *straight* merge, and works as described above. A disadvantage of this method is that if an array has slightly more than 2^x keys, an entire extra merge step would be required, which would considerably slow the algorithm. This problem is shared by all other versions of mergesort considered here, except algorithm N. However, since our tests use exact powers of two as set sizes, this problem does not affect them. LaMarca, however, used set sizes such as 4096000, which at times would suffer from this type of problem.

Algorithm S has 15% lower instruction count than Algorithm N[1]. Algorithm S, like algorithm N, is a very difficult algorithm to modify. Knuth describes it in assembly, and making the optimizations that LaMarca describes on it is more difficult than rewriting the algorithm more simply. The rewritten algorithm, referred to from here on as 'base mergesort', is a simple higher level version of algorithm S, and should perform only slightly more slowly.

6.1.3 Base Mergesort

Three optimizations are described by LaMarca for the base mergesort. Firstly, a bitonic sort, as described in [Sedgewick 02a], removes instructions from the inner loop. Every second array is created in reverse, so that the larger keys are next to each other in the middle. The smallest keys are at the outside, meaning that the lists are iterated through from the outside in. As a result, it is not possible to go outside the array, and it is not necessary to consider the lengths of the arrays, apart from when setting the initial array indices. When the first array is exhausted, it will point to the last key on the other array, which will be larger than all other keys in that array. When both are exhausted, the last key of both arrays will be equal. A check is put in, in this case, and merging ends if both indices are the same. Figure 3.2 on page 17 shows a bitonic merge.

The overhead of preparing arrays to be merged is small. However, for small lists, the overhead is large enough that it can be more efficient to sort using a much simpler sort, such as insertion, selection or bubblesort. In Section 4.7, it is shown that insertion sort is the fastest elementary sort. Sedgewick's quicksort uses this, and LaMarca recommends the use of a simple inline pre-sort, which can be made from insertion sort. To make lists of length four, the smallest of the four keys is found, and exchanged with the first key, in order to act as a sentinel. The next three keys are sorted with insertion sort. The next set of four keys is sorted in reverse using the same method. It is possible to hard code this behaviour, in an attempt to reduce the instruction count, but in fact the instruction count stays the same, and the branch prediction suffers as a result.

Finally, the inner loop is *unrolled*[2]. There are actually two inner loops, and they are both unrolled 8 times. The compiler is also set to unroll loops. However, the observed difference between unrolling manually and compiler unrolling is up to 10% of instruction count, so manual unrolling is used.

[1] This is observed on random data, which algorithm N is not designed for. On more regular data, or semi-sorted data, this will probably not occur.

[2] Unrolling a loop means executing several steps of the loop between iterations and reducing the number of iterations, which lowers the loop overhead and allows for better scheduling of the instructions.

Overall, these three optimizations reduce the instruction count by 30%.

6.2 Tiled Mergesort

LaMarca's first problem with mergesort is the placement of the arrays with respect to each other. If the source and target arrays map to the same cache block, the writing from one to the other will cause a conflict miss. The next read will cause another conflict miss, and will continue in this manner. To solve this problem, the two arrays are positioned so that they map to different parts of the cache. To do this, an additional amount of memory the same size as the cache is allocated. The address of the auxiliary array is then offset such that the block index in the source array is $CacheSize/2$ from the block index in the target array. The is achieved by inverting the most significant bit of the block index in the target array.

While this step makes sense, whether this is valuable is very much a platform dependant issue. Since virtual addresses are used, it may be possible that the addresses used by the cache are not aligned as expected. Even worse, an operating system with an excellent memory management system may do a similar step itself. The program's attempt to reduce cache misses could then increase them. In the event that the *Memory Management Unit* in the processor uses virtual addresses to index its cache, or has a direct mapping between virtual and physical or bus addresses, then this problem is very unlikely. The simulations presented in this report address the cache by virtual address, and so this optimization is effective in this case.

The next problem LaMarca considers is that of temporal locality. Once a key is loaded into the array, it's only used once per merge step. If the data does not fit in half the cache, then conflict misses will occur, and no temporal reuse will occur at all. The solution is to apply a technique used in compilers called *tiling*. Using this technique, an entire cache sized portion of the array will be fully sorted before moving on to the next set. When all the sets have been sorted in the manner, they are all merged together as usual.

It is also possible to avoiding the copy back to memory. In the event of an odd number of merge steps, the auxiliary array will contain the sorted array, which needs to be copied back. This can be avoided by pre-sorting to make lists sized two or eight. A large increase of instruction count was realized by sorting to lists of length two, so a length of eight was used. When an even number was required, four was used.

6.3 Double-aligned Tiled Mergesort

Offsetting the target array to avoid conflicts reduces significantly the number of level 2 cache misses. However, it does this at the expense of increasing the number of level 1 cache misses. To avoid this, we further offset the target array by $Level1CacheSize/2$. It is expected that the number of level 1 misses should be greatly decreased, at the expense of very few level 2 misses. The expected improvement is not as great as between base mergesort and tiled mergesort, but some improvement should be seen.

6.4 Multi-mergesort

Multi-mergesort has two phases. The first is the same as tiled mergesort, and sorts the array into lists half the size of the cache. The second phase attempts to improve temporal locality by merging these lists together in a single merge step. There are k lists and this is called a *k-way merge*. It is possible to do a search across each of these keys, maintaining a list of indices to the smallest unused key from each list. When there is a small number of lists, this isn't a problem, however, when the number of lists increases significantly, the overhead involved in this is substantial. It is necessary to check each of the lists, and each key searched would cause a cache miss, since each array segment is aligned, and map to the same cache block.

To avoid this, LaMarca uses a priority queue. The smallest key from each list is added to the queue, and then put into the final target array. This reduces the problem of the search overhead, but does not reduce the problem of *thrashing*[3]. To avoid this, an entire cache line is added at a time to the priority queue. There will still be conflict misses, but there will be no thrashing since the keys overwritten in the cache will not be needed again.

LaMarca used a 4-heap for his priority queue, and an 8-heap is used here, for the same reasons as before (see Section 5.2.). The heap is aligned, and a cache line is put into the array each time. The smallest key is at the top of the heap, and this is put into its final position in the array. The heap is then fixed.

LaMarca uses a special tag to mark the last key from each list inserted into the array. No more details are provided on this, and a sorted list of length k is used here, in its place. If the key at the top of the heap is the same as the smallest value in the list, another eight keys are added from the array segment to which the number belonged. This continues until the array is full sorted.

Using a sorted list instead of LaMarca's tagging leads to a variety of problems. The reason the sorted list is used is that no details of the tagging is provided. The sorted list is k keys long, and only needs to be properly sorted once. The next time it needs to be sorted is when a new line is added, but in this case only one key is out of place, and can be put in place with a single insertion sort run. This is linear at worst, and will have only 1 branch misprediction.

When a list is fully consumed, the next keys inserted will be from the next list. This could be fixed with a bounds check, however, since the lists are bitonic, the next keys inserted are going to be the largest keys from the next array. These will be added to the heap, and take their place at the bottom, without harmful effect. A harmful effect is possible, though, when lots of keys with the same value are in the array. All these keys would be inserted at a similar time, and when the first one is taken off the heap, a list would add eight keys, even though it had just added eight. In this manner, the size of the heap could be overrun. With random keys over a large range, as in our tests, this is unlikely, and so this is not addressed.

The heap used for the k-way merge should result in a large increase in instructions, and a significant reduction in level 2 cache misses. The addition of the heap should not result in an increase in misses, since the heap is small and will only conflict with the merge arrays for a very short time. The heap is aligned as described in the previous chapter, and has excellent cache properties. LaMarca estimates that there should be one miss per cache line in the first phase, and the same in the second phase, although each miss will occur in both the source and target arrays. This sums to four misses per cache block, for a total of one miss for every two keys in the array.

[3]Thrashing occurs when two arrays are mapped to the same cache block, and the constant access of one followed by the other causes every access to be a miss.

6.5 Double-aligned Multi-mergesort

Double-aligned Multi-mergesort combines the double alignment of Double-aligned tiled mergesort with multi-mergesort.

6.6 Results

6.6.1 Test Parameters

The level 1 cache is 8K, so 1024 keys from each array can be sorted in the level 1 cache. 262144 keys can be sorted in the level 2 cache. When arrays were pre-sorted, they were pre-sorted to a size four, or eight if the number of passes was going to be odd. These were reversed for the double-aligned multi-mergesort; since it does a k-way merge into the source array, the sorted cache-sized segments need to be in the auxiliary array. An 8-heap was used for the multi-mergesorts.

6.6.2 Expected Performance

Algorithm N is not optimized for random data, and should not perform as well as algorithm S. Algorithm S is expected to have a lower instruction count than base mergesort. Neither algorithm is expected to have good cache performance.

The instruction count of mergesort is $O(NlogN)$, and does not vary according to input. Multi-mergesort should have a significantly higher instruction count once its threshold is hit. The double merges have exactly the same flow control as the single merges, so will only differ in cache misses.

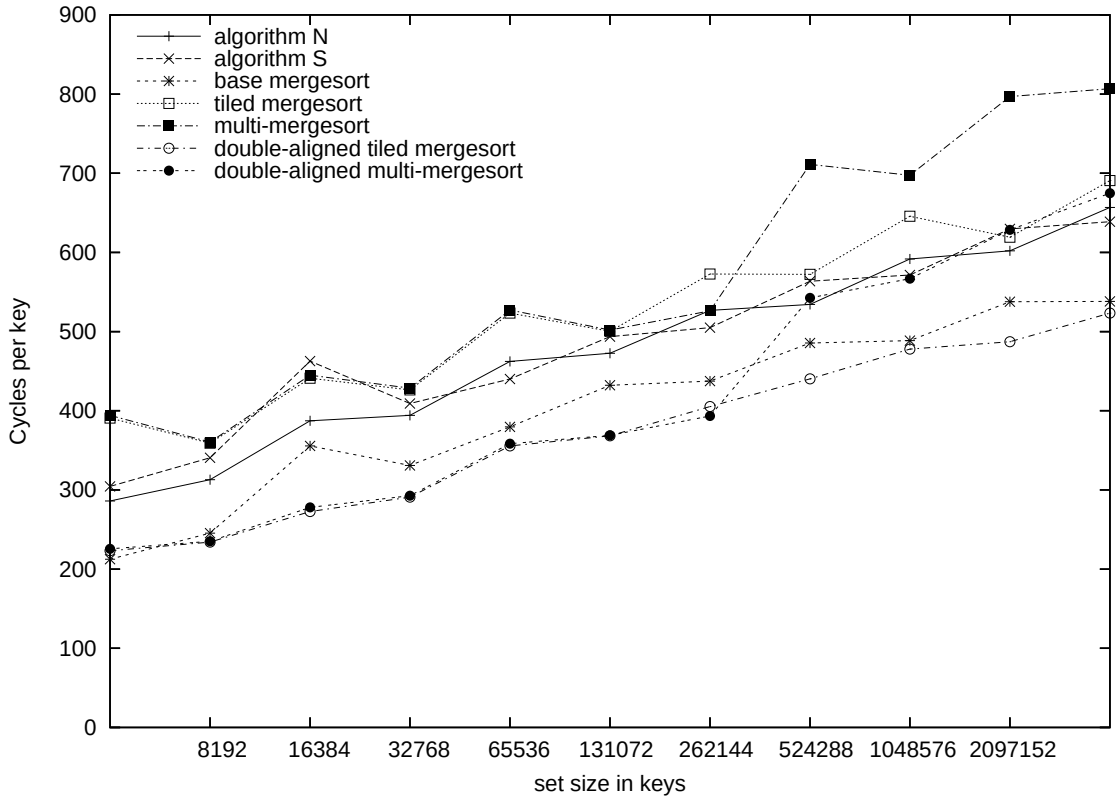
The number of memory references is also $O(NlogN)$, and mergesort's cache performance should be very regular, so long as the data set fits in the cache. Once the data set exceeds the cache, the number of cache misses should leap. For multi-mergesort, they should flatten out after this, perhaps with a slight slope to take into account the conflict misses of the aligned array segments. The fully-associative caches should not have these misses, and the results should be correspondingly lower. The double mergesorts should have better performance than their counterparts in level 1 misses, though there should be a slight increase in level 2 misses. It is expected that they should perform slightly faster as a result.

The flow control of mergesort is very regular, but there is no way to predict the comparisons. As a result the misprediction rate should be $O(NlogN)$ for all but the multi-mergesorts, which should increase once the threshold is passed.

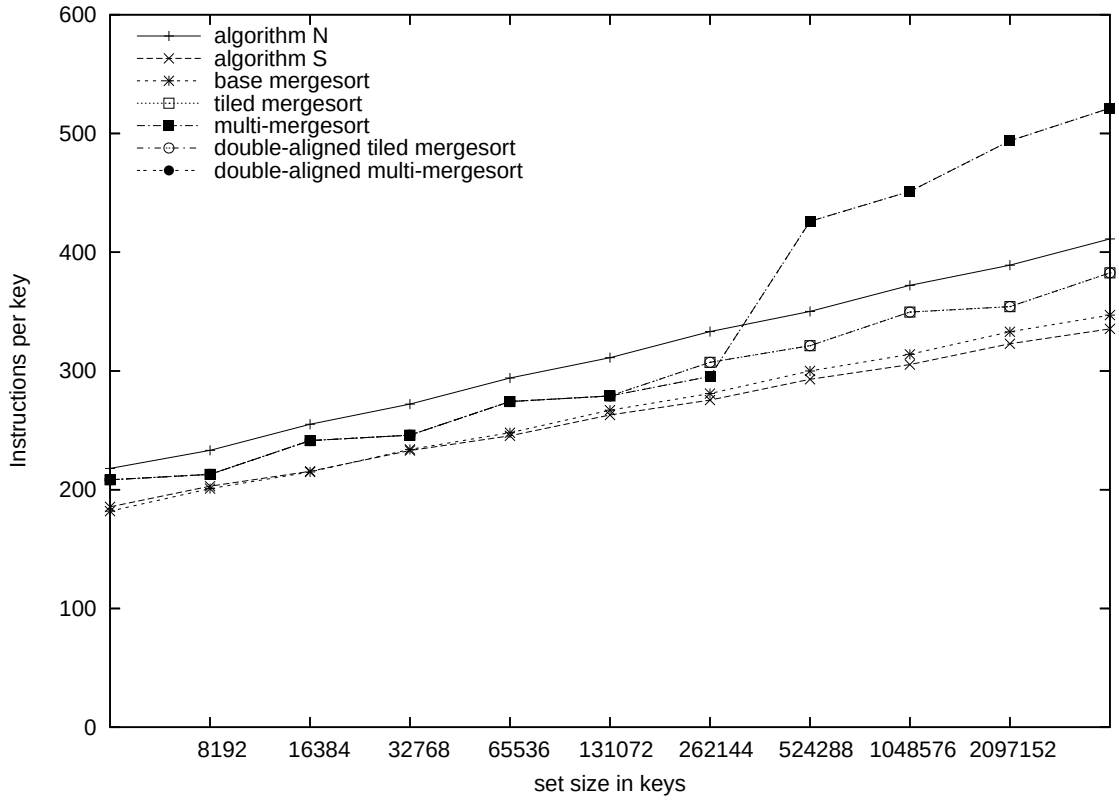
6.6.3 Simulation Results

As expected, algorithm S has lower instruction count than algorithm N, and also slightly outperforms base mergesort. The number of instructions, shown in Figure 6.1(b) is exactly as predicted, with a 50% spike at multi-mergesort.

Level 2 misses, shown in Figure 6.2(b) are also as expected, although the multi-mergesort does not show

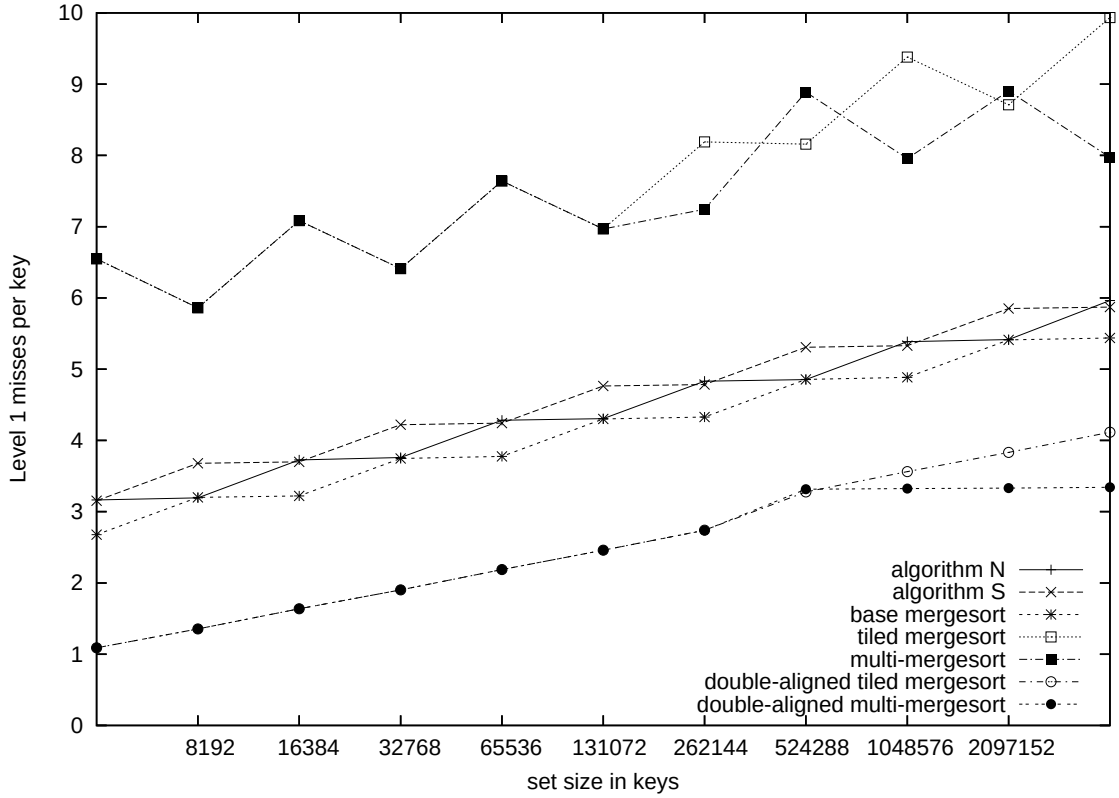


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

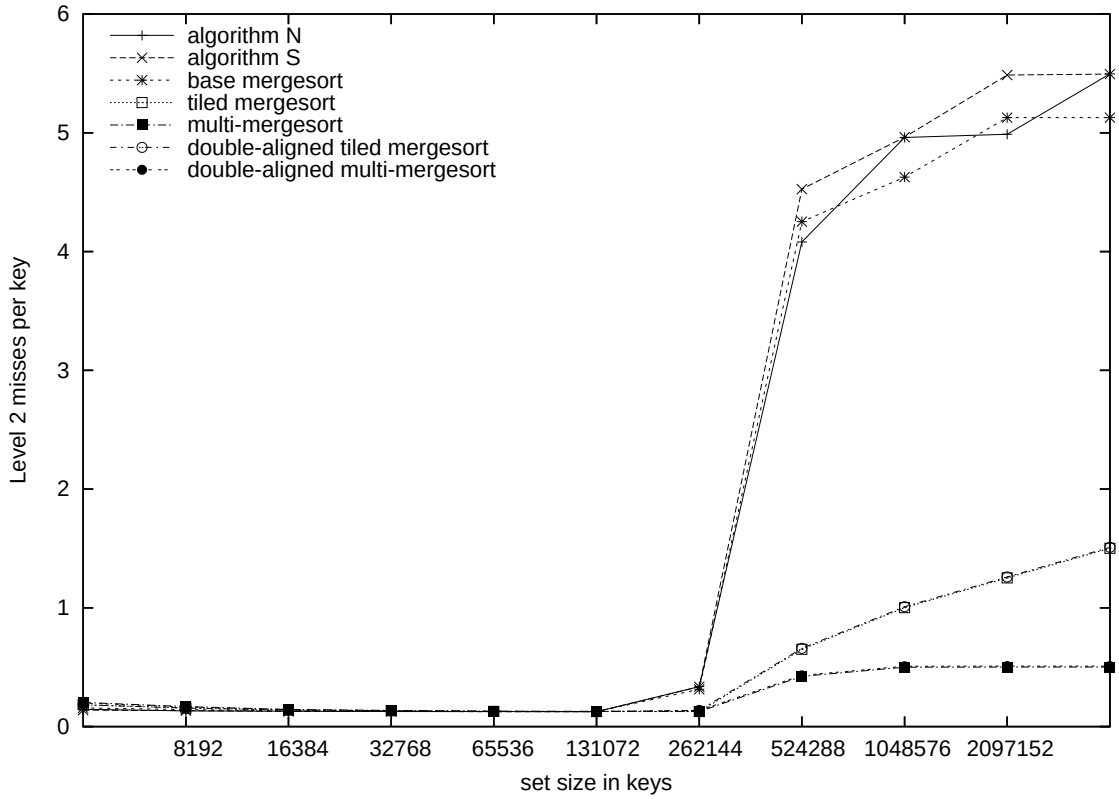


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 6.1: Simulated instruction count and empiric cycle count for mergesort

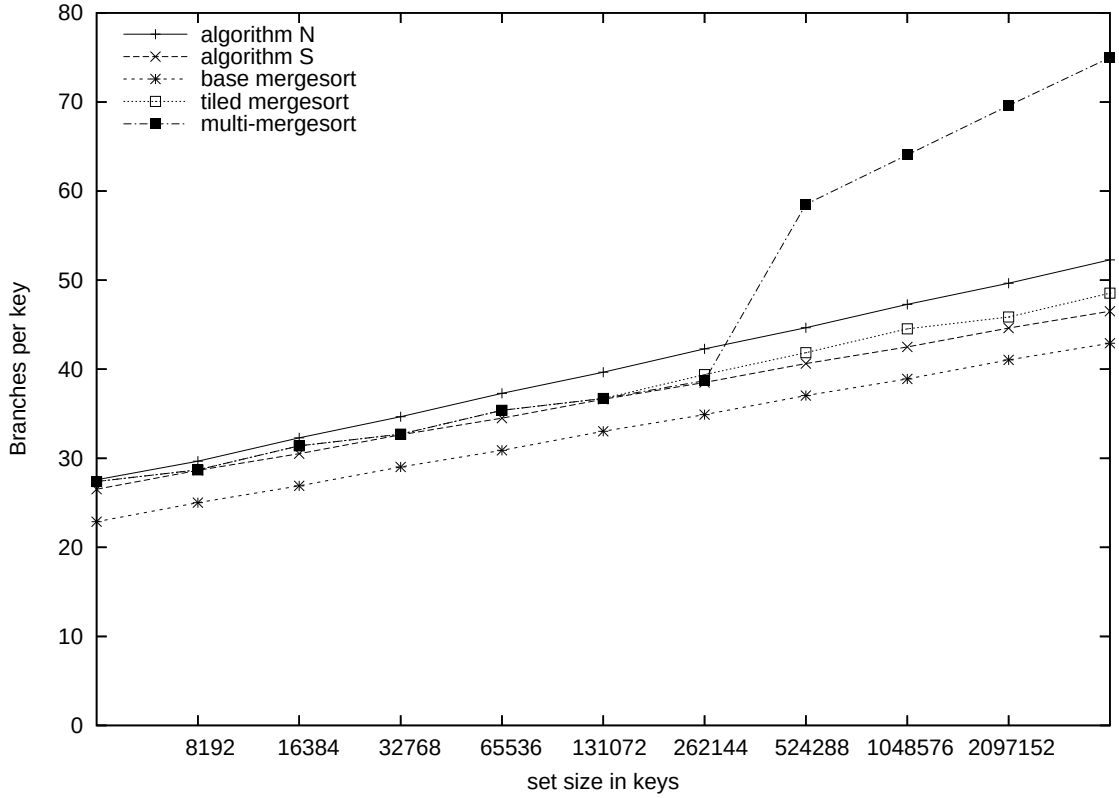


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache. Results shown are for a direct-mapped cache.

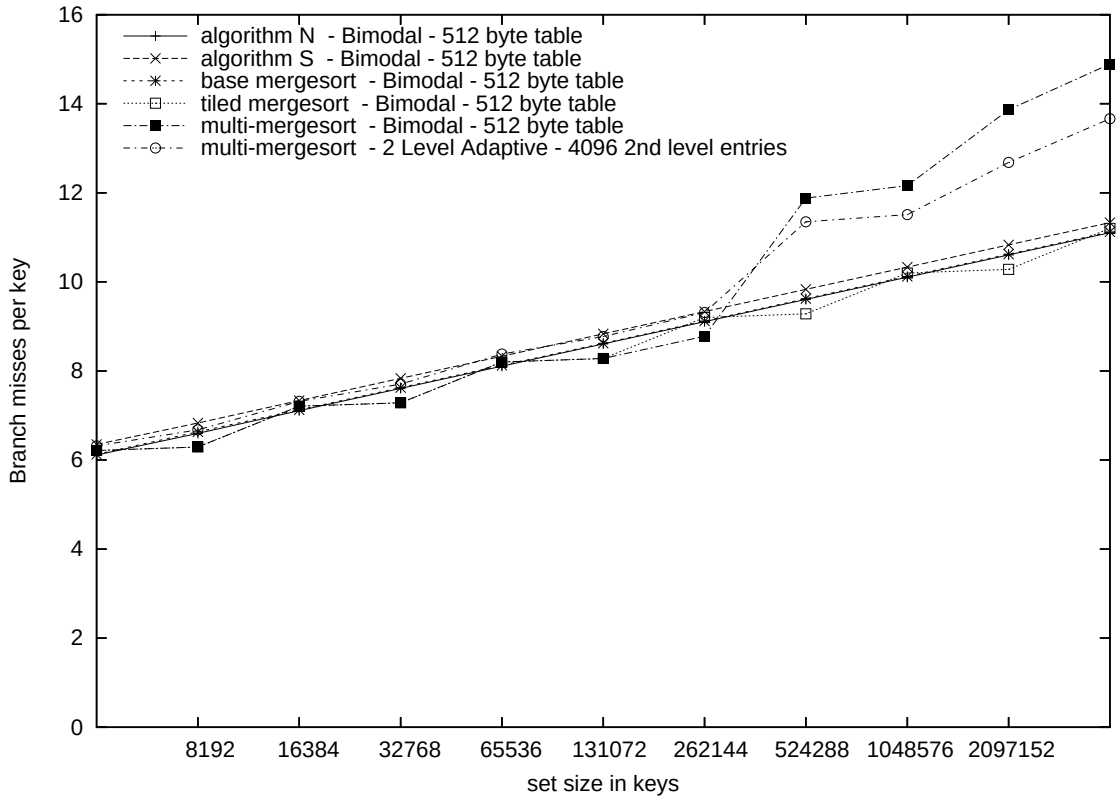


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32-byte cache line. Results shown are for a direct-mapped cache.

Figure 6.2: Cache simulation results for mergesort



(a) Branches per key - this was simulated using `sim-bpred`.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10-bit branch history register which was XOR-ed with the program counter.

Figure 6.3: Branch simulation results for mergesort

the same improvement over tiled mergesort as in LaMarca's results, as our tiled mergesort performs better. The cache performance of algorithms N and S are poor, as expected. They perform roughly the same as base mergesort.

It is notable that our prediction of half a miss per key for multi-mergesort, which was taken from LaMarca, is exactly right. However, our assertion that the fully-associative cache will perform slightly better is false. In actual fact, the direct-mapped cache performs significantly better (see Figure A.14(d) on page 118). This is due to the replacement policy: our fully-associative cache removes the least recently used keys from the cache to make room, while the direct-mapped caches are designed to make best use of the keys while they are in the cache. They have, effectively, an optimized replacement policy.

The prediction results in Figure 6.3 are as expected. Very little variation exists between the types of mergesort. It is noticeable, though, that the spike in branch misses due to the multi-mergesorts is considerably lessened by the two level adaptive predictor. This shows there must be regular access patterns in the k-way merge at the end of multi-mergesort, which are predictable using a branch history.

As predicted, the rate of level 1 misses is greatly reduced by the double tiling, as shown in Figure 6.2(a). Double-aligned multi-mergesort has consistently four times fewer level 1 misses than multi-mergesort, and both double tiled sorts perform significantly better than the unoptimized sorts. The expected increase in level 2 misses did not materialise, and is in fact negligible.

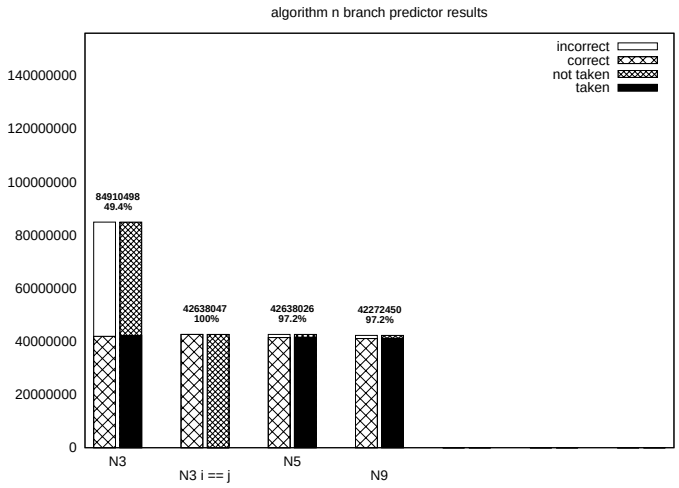
The level 2 cache results are startlingly similar to LaMarca's. The shape of the Figure 6.2(b) is exactly that provided by LaMarca. The ratio's between the each of the sorts is also the same, as each of our results is exactly half that of LaMarca, due to different `int` sizes in our tests.

Despite the improvement in cache misses due to multi-mergesort, it still cannot compensate for the increase in instruction count as a result, and it has a higher cycle count on the Pentium 4 (see Figure 6.1(a)) as a result. The other sorts generally perform in $O(NlogN)$ time, with double-aligned tiled mergesort being the fastest. Note that the alignment of the mergesort may not work as desired due to the size of the cache. This was optimized for a 2MB cache, while the Pentium 4 cache is only 256KB. This means that the two arrays are aligned to the same cache block. However, since the cache is 8-way associative, this is not serious, and should not result in a lot of thrashing.

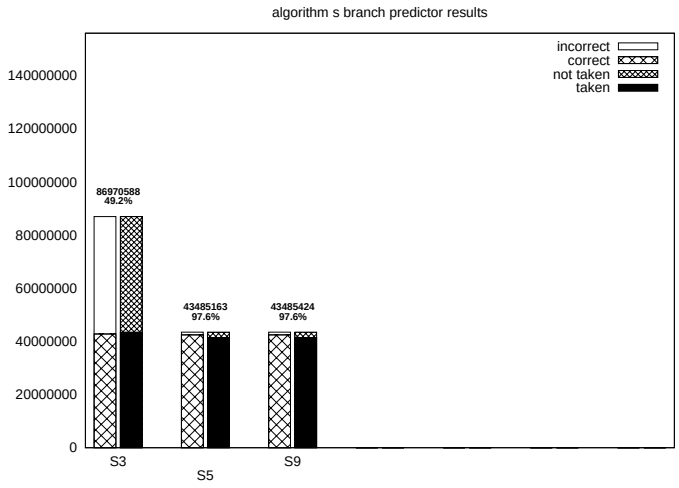
The cache misses end up being the deciding factor between the performance of double-aligned tiled mergesort and base mergesort, the next best contender. Despite the reduction in level 2 cache misses, the sharp increase in level 1 misses result is poor performance for tiled and multi-mergesort. Multi-mergesort is further encumbered with a much higher instruction count. Double-aligned tiled mergesort, with its low instruction count, and low cache miss rate in both levels 1 and 2, emerges the winner.

6.7 A More Detailed Look at Branch Prediction

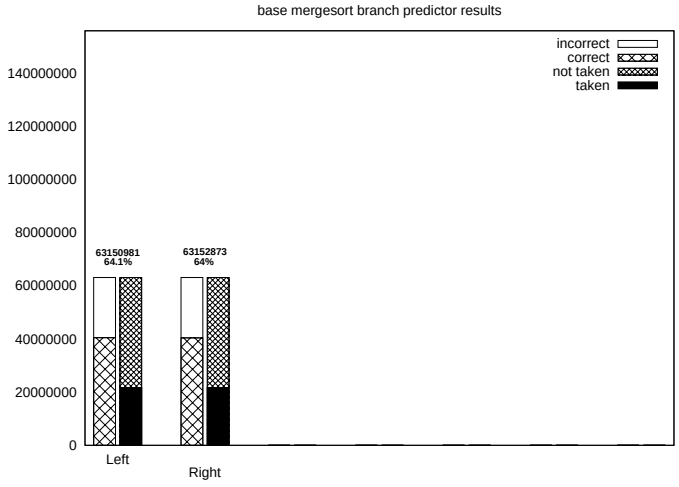
Figure 6.4(a) shows the behaviour of the individual branches within algorithm N. The labels in the graphs of algorithms N and S correspond to the names of the steps in Knuth's algorithm (we also used the same names for the labels of branch targets in our C source code. Branch '*N3*' is the main comparison branch; it is used to choose the smaller of the elements from the start of the two sub-arrays being merged. The direction of this branch is completely random - it depends entirely on the (in this case random) data. Figure 6.4(a) also shows that this branch is almost entirely unpredictable using a bimodal predictor. The result for a two-level predictor is much the same, because there is no pattern in the outcome of



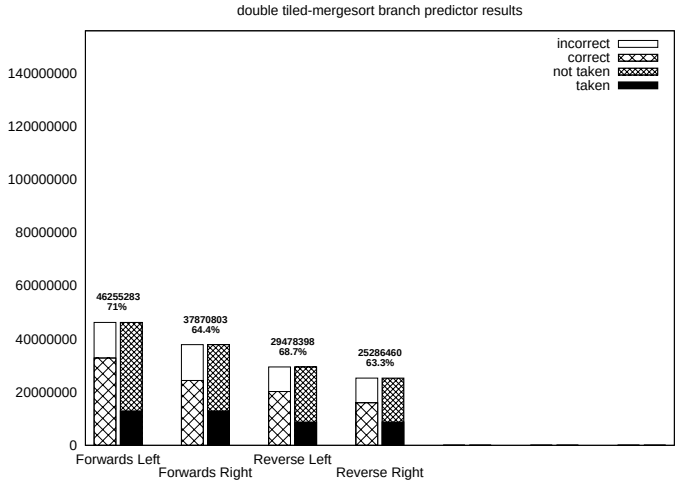
(a)



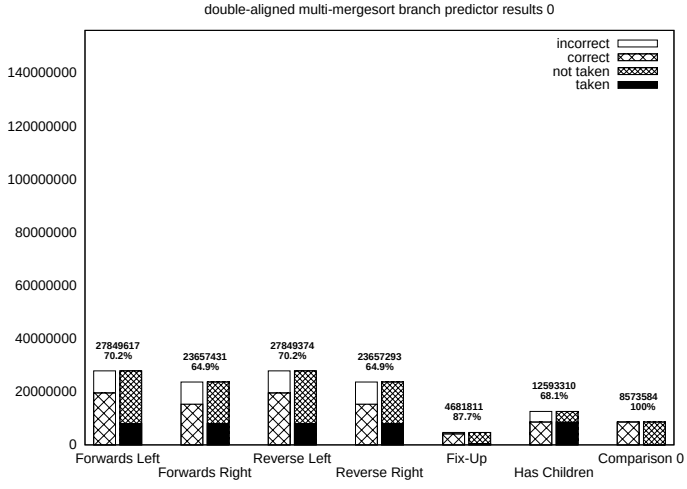
(b)



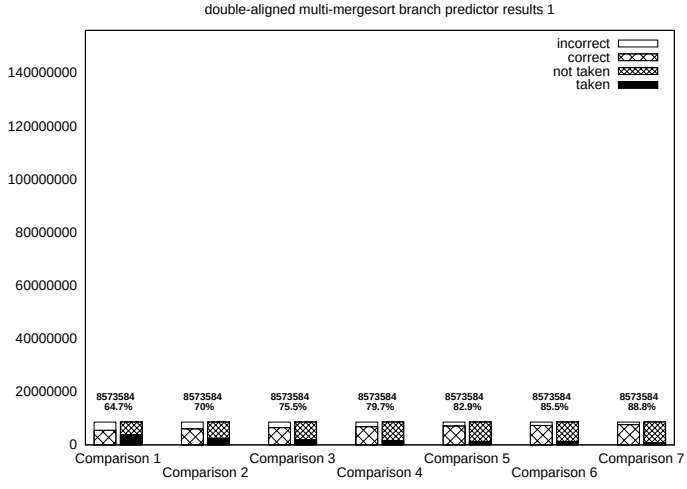
(c)



(d)



(e)



(f)

Figure 6.4: Branch prediction performance for mergesorts. All graphs use the same scale. Above each column is the total number of branches and the percentage of correctly predicted branches for a simulated bimodal predictor.

these branches. Branches '*N3 i==j*' (check for the current merge being complete), '*N5*' and '*N9* (both checks for the end of a run of increasing elements) are control-flow branches, and are almost entirely predictable. Figure 6.4(b) shows that the comparison branch in Algorithm S is just as unpredictable as that in algorithm N. However, given that the merge in algorithm S is entirely regular (see section 6.1.2) it is possible to eliminate one of the control-flow branches.

Base mergesort is an optimized algorithm that reverses the sorting direction of every second sub-array, pre-sorts small sub-arrays, and applies a number of other optimizations. As Figure 6.4(c) shows, our base mergesort has eliminated some of the control-flow branches. The main comparison branch is split into two separate branches - one iterating from left-to-right over the increasing sub-array ('*Left*'), and the other from right-to-left over the decreasing sub-array ('*Right*'). However, base mergesort appears to need a total of 50% more comparison branches that either Algorithm S or N.

In fact, base mergesort eliminated control-flow branches by using the same branch that is used for comparison to also deal with the case where one (or both) of the merging sub-arrays becomes empty. Thus, the sum of the number of executions of the branches '*Left*' and '*Right*' is almost exactly the same as the sum of the number of executions of the branches N3 and N3 i==j in Algorithm N. The total number of mispredictions is also almost the same.

The comparison branch in double-aligned tiled mergesort is divided into four separate branches. As in base mergesort, there are '*Left*' and '*Right*' branches. However, for each of these there are separate branches for when creating a new sub-array in increasing order ('*Forwards*') or decreasing order ('*Reverse*'). There are slightly more executions of '*Forwards*' than '*Reverse*' because the final array is always sorted into increasing order. The left merge step is also favoured, but this is merely due to the fact that it occurs first in the code. While there are more left branches, the actual number of branches taken are the same.

Double aligned multi-mergesort is an even more complicated algorithm. It sorts the arrays fully in the cache, leaving them bitonically arranged (sorted into sub-arrays, sorted into alternately increasing and decreasing order). The final merge is performed separately, and hence no bias for the forward branches, as there was in tiled mergesort. The final sort is performed with a k-way merge, which is implemented using a heap data structure. This is shown in Figures 6.4(e) and 6.4(f). The '*Comparison*' branches are comparisons from the merge. The labels in these graphs are the same as in Figure 6.4(d) and 5.5(b).

During regular two-way merging, the number of branches is 20% lower than base mergesort, while the number of mispredictions in only 10% lower. This implies that most of the mispredictions occur early in the merging process, and not in the final merge, which multi-mergesort leaves out.

Overall, when the k-ary heap is used, there are approximately the same total number of branch mispredictions as in the other mergesort algorithms. Although the final k-way merge executes a lot more comparison branches, these heap branches are quite predictable, as we saw in Section 5.4. This means that using the technique to reduce cache misses does not adversely affect branch prediction rates (it does, however, still affect the instruction count). The rate of branch misses is much the same as that in heapsort, and progresses in the same manner (see Figure 5.6). Note that '*comparison 0*' is included for illustration purposes only. It is a comparison between two values, one of which has just been assigned to the other, and which are therefore equal. It is guaranteed to be taken every time, and would be optimized out by the compiler.

6.8 Future Work

Several avenues could be pursued in order to improve the algorithms presented here. In multi-mergesort, LaMarca spoke of tagging keys as they went into the cache. Here, a sorted list was used in its place. Implementing and investigating tagging and comparing of the results of the two techniques would be interesting.

Algorithm N is a mergesort for more natural layouts of data, such as partially sorted lists. It can sort these faster than a straight merge, but won't on random data until it is nearly sorted. Some of the improvements described here may be able to improve its performance, as they did for the straight merge.

To address one of the problems with multi-mergesort, that of the conflict misses on the k-way merge, [Xiao 00] reveals a padded multi-mergesort with reduced conflict misses. It would be interesting to compare these results.

It may also be possible to simply reduce the number of conflict misses of the k-way merge, simply by reducing the size of the initial sort by the size of several cache lines. In this way, the smallest keys would map to different cache blocks, without the need for padding.

Chapter 7

Quicksort

Quicksort is an in-place, unstable sort. It was first described in [Hoare 61a] and [Hoare 61b], and later in more detail in [Hoare 62]. Sedgewick did his PhD on quicksort, and several important improvements to quicksort are due to this. They are discussed in [Sedgewick 78] and [Sedgewick 02a]. [Bentley 93] discusses considerations for a fast C implementation. [Sedgewick 02b] discusses equal keys and proves that quicksort has the lowest complexity of any comparison-based sort.

Quicksort is an algorithm of the type known as *divide and conquer*. It recursively partitions its array around a pivot, swapping large keys on the left hand side with smaller keys on the right hand side. The partition causes the pivot to be put in its final position, with all greater keys to the right, and lesser keys on the left. The left and right partitions are then sorted recursively.

The recursive nature of quicksort means that even though quicksort is in-place, there is a requirement of a stack. In the worst case, this stack can grow to $O(N)$, if the largest or smallest key is always chosen as the pivot. This worst case will occur frequently, especially when trying to sort an already sorted, or almost sorted list.

Figure 7.1 contains a simple implementation of the quicksort algorithm.

```
void quicksort(Item a[], int l, int r)
{
   if (r <= l) return;
   int i = partition(a, l, r);
   quicksort(a, l, i-1);
   quicksort(a, i+1, r);
}
```

Figure 7.1: Simple quicksort implementation

7.1 Base Quicksort

LaMarca uses the optimized quicksort described by Sedgewick in [Sedgewick 02a] for his base quicksort. This is an iterative quicksort which uses an explicit stack. In the worst case, this stack must be the same size as the array, and must be initialised at the start of the algorithm. To reduce the size of the stack, Sedgewick proposes sorting the smallest partition first. This optimization reduces the maximum size of the stack to $O(logN)$. In practice, two `int`s, the left and the right index, are required for each partition.

Sedgewick's second improvement is the use of an elementary sort for small sub-files. Once a partition is smaller than an arbitrary threshold, insertion sort is used to sort the partition. Sedgewick states that roughly the same results are obtained for thresholds between five and 25. An additional reduction in instruction count can be had by doing the insertion sort over the entire array at the end, rather than across each small array as soon as the threshold is reached. At the end of the partition sorting, LaMarca places a sentinel at the end of the array and uses an insertion sort, eliminating the bounds check from the end. In this implementation, the sentinel is placed at the front, and the smallest of the first `THRESHOLD` keys is used as a sentinel, in keeping with our policy of not exceeding the bounds of the array.

Choosing a pivot which is close to the median of the keys to be sorted ensures quicksort achieves optimal performance. Hoare suggested a random pivot, but the added cost of a random number generator would severely slow the algorithm. Sedgewick recommends that the median of three keys is chosen, the three keys being the first, middle and last in the list. This reduces the chance of a worst case occurring significantly, notably in the case of sorted and almost sorted lists.

An additional benefit of Sedgewick's median-of-3 implementation is that it provides a sentinel for the partition. It is now possible to remove the bounds check from the innermost loops of `partition`, saving a lot of instructions.

Between these three improvements, Sedgewick estimates a reduction in instruction count of roughly 30%. The size of the auxiliary stack is also reduced significantly, and the possibility of stack overflow, in the worst case of the recursive version, is removed.

Quicksort has been well studied, and several important improvements have been noted. In [Bentley 93], using a median-of-9 partition is discussed. This is discussed below. Modern discussions on quicksort refer to the problem of equal keys, which is discussed there, as well as by Sedgewick in [Sedgewick 02b]. These improvements are not included here for several reasons. Firstly, a quicksort which takes into account equal keys puts the equal keys at the front of the array, then copies them into place later. These copies would represent a lot of cache misses, and the results may overwhelm the improvements LaMarca makes in the next phase of the algorithm. Secondly, when sorting random keys over a large range, the number of equal keys will be very low. Using a median of greater than three is discussed in Section 7.5.1.

7.2 Memory-tuned Quicksort

LaMarca's first step was to increase the temporal locality of the algorithm by performing insertion sort as soon as the partitioning stops. Once a partition is reduced below the size of the threshold, insertion sort is preformed immediately. By doing this, the keys being sorted are in the cache already. By leaving the insertion sort until the end, as Sedgewick does, the reduction in instruction count is made up by the increase in cache misses, once the lists being sorted are larger than the cache.

As discussed above, it is necessary to insert either a bounds check or a sentinel. If the partition to be sorted is the left most partition, then the smallest key in the partition will be chosen as the sentinel. No other partition needs a sentinel. A possible way to avoid the sentinel in that partition is to place zero at 0^{th} position in the list to be sorted, and sort whichever number was in this position back at the end. This sort could be done with one iteration of insertion sort, and would have just one branch miss. However, the number of level 2 cache misses would be great in this case, and the cost of this would be substantial. This could be reduced by choosing the key to remove from a small list, such as the eight keys at the start, middle and end of the array. This would introduce a constant number of cache misses, but would significantly reduce the chances of having to sort across the entire array at the end. This, however, is not implemented; instead a check is done to fetch a sentinel from the appropriate sub-file if required. This check isn't very expensive, and is significantly cheaper than either a bounds check or putting a sentinel into every sub-file.

7.3 Multi-Quicksort

In the case that the array to be sorted fits in a particular level of the cache, then quicksort has excellent cache properties at that level. Once the array is larger than the cache, however, the number of misses increases dramatically. To fix this problem, LaMarca uses a technique similar to that used in multi-mergesort. Instead of multi-merging a series of sorted lists at the end, a *multi-way partition* is done at the start, moving all keys into cache sized containers which can be sorted individually.

At the start of the algorithm, a set of pivots is chosen and sorted, and keys are put into containers based on which two pivots their value lies between. A minimal and maximal key are put on the edges of this list to avoid bounds checks as the list is iterated over. Because of the way the pivots were chosen, two pivots which happen to be close together would result in a very small number of keys being put into their container. While this is not a bad thing, this will result in a larger number of keys in another container. Conversely, a large space between two consecutive pivots will mean that the keys in this container will not all fit into the cache at once.

There are two possible solutions to this. The first is to choose the pivots well, so that they are very evenly distributed. The second is to choose the number of pivots so that the average size of the containers is not larger than the cache. LaMarca presents calculations showing that by choosing the average subset size to be $C/3$, where C is the number of keys which fit in the cache, on average only 5% of subsets will be larger than the cache.

Since the maximum number of keys which must fit in a container is not known, it is necessary to implement containers using linked lists. Each list contains a block of a few thousand keys. LaMarca notes very little performance difference between 100 and 5000 keys in each block. The size is therefore chosen so that each list fits exactly into a page of memory. When a list is filled, another list is linked to it, and these lists hold all the keys in the container.

The initial implementation of this allocated a new list every time an old one was filled. This lowers performance significantly, as it results in a lot of system calls, which increase instruction count and cache misses. Instead, it is possible to work out the maximum possible number of lists required to create the containers, and allocate them at the start. When a list runs out of space, it is linked to an unused list from the large allocation. This wastes space, since it is almost guaranteed that less lists will be required than available, though the number of wasted lists allocated will be less than the number of pivots, and therefore not exceptionally high.

The entire array is iterated though a key at a time. For each key, the list of pivots are searched until the appropriate container is found, and the key is added to that container. Sequentially searching the lists increases the instruction count by 50% over the memory-tuned version. For this reason, the search was replaced with an efficient binary search, which reduced by half the extra instructions.

Once the entire array has been put into containers, each container is emptied in turn and the keys put back into the array. When a container is empty, the pivot greater than that container is put into position, which is guaranteed to be its final position. The indices of these positions is pushed onto the stack, and the sub-array is then sorted.

While emptying the containers back into the array, an opportunity is taken to find the smallest key in the leftmost partition. This is placed as a sentinel, ensuring that every partition has a sentinel, and that no bounds checks are required.

7.4 Results

7.4.1 Test Parameters

When the size of a partition to be sorted is less than a certain threshold value, insertion sort is used instead of quicksort. The chosen threshold was 10, simply because this is the number used in Sedgewick's book.

The number of keys in the linked list is 1022 since a word is needed for the count, and another as a pointer to the next list. 1024 is the number of 32-bit words which fit in a 4KB page of memory. In earlier versions where each page was `malloc`ed individually, the number of keys in the linked list was 1018, as 4 words were used by the particular version of `malloc` used by SimpleScalar[1].

A 2MB cache was assumed, so the number of 32-bit integer keys which fit in the cache for this in-place sort is 524288. Multi-quicksort began once the number of keys was larger than 174762.

7.4.2 Expected Performance

The instruction count for each quicksort is expected to be $O(NlogN)$, though not as high as the other algorithms due to the efficient inner loop. The number of instructions for multi-quicksort should increase by about 50% in the case of the sequential sort, and 25% in the case of the binary sort. Memory-tuned quicksort should also have a slightly higher instruction count than base quicksort, due to the change in use of insertion sort.

LaMarca notes that quicksort is already very cache efficient[2], as it has excellent spatial and temporal locality. Once it no longer fits in the cache, however, the number of misses should increase significantly. This is especially true in the case of the base quicksort, which does an insertion sort across then entire array at the end.

[1] This particular `malloc` forms part of *glibc*, the GNU C library, version 1.09.

[2] We speculate wildly at this point and suggest that since, historically, an average of 25% of computing time has been spent sorting (see [Knuth 97, Page 3]), the development of caches has been subconsciously affected by the cache performance of quicksort.

Memory-tuned quicksort should have fewer misses than base quicksort as soon as the array is larger than the cache. It is expected that these misses will make up for the increase in instruction count.

LaMarca estimates a large reduction in level 2 misses in multi-quicksort, and puts a figure on it of four misses per cache block, meaning, in our case, half a miss per key.

The number of branches should increase in multi-quicksort along with the instruction count. When a binary search is used, the increase in mispredictions is expected to be significant. With a sequential search, however, the number of misses should not increase at all, and should flat-line from that point on.

7.4.3 Simulation Results

The instruction count, shown in Figure 7.2(b) is mostly as predicted. The increase in instructions in both multi-quicksorts is as predicted, with the binary search version increasing only 25% against the sequential search version's 50% increase. However, the increase we predicted in the instruction count of memory-tuned quicksort based on the results of both Sedgewick and LaMarca - that is, once the efficient insertion sort was replaced with many small insertion sorts - we observe that the instruction count remains exactly the same.

The cache results in Figure 7.3(b) were similar to predictions, except that the multi-quicksorts did not behave as expected, and performed as badly as the memory-tuned version. However, the rate at which the cache misses are increasing is very low, being almost a flat line on our graph. LaMarca's calculation of four misses per cache block is very accurate, as we show 0.55 misses per key, or 4.4 misses per block.

The memory-tuned quicksort performed better than the base version, as predicted. Both the memory-tuned version and the base quicksort both have a significant increase once the arrays no longer fit in the cache, also as predicted.

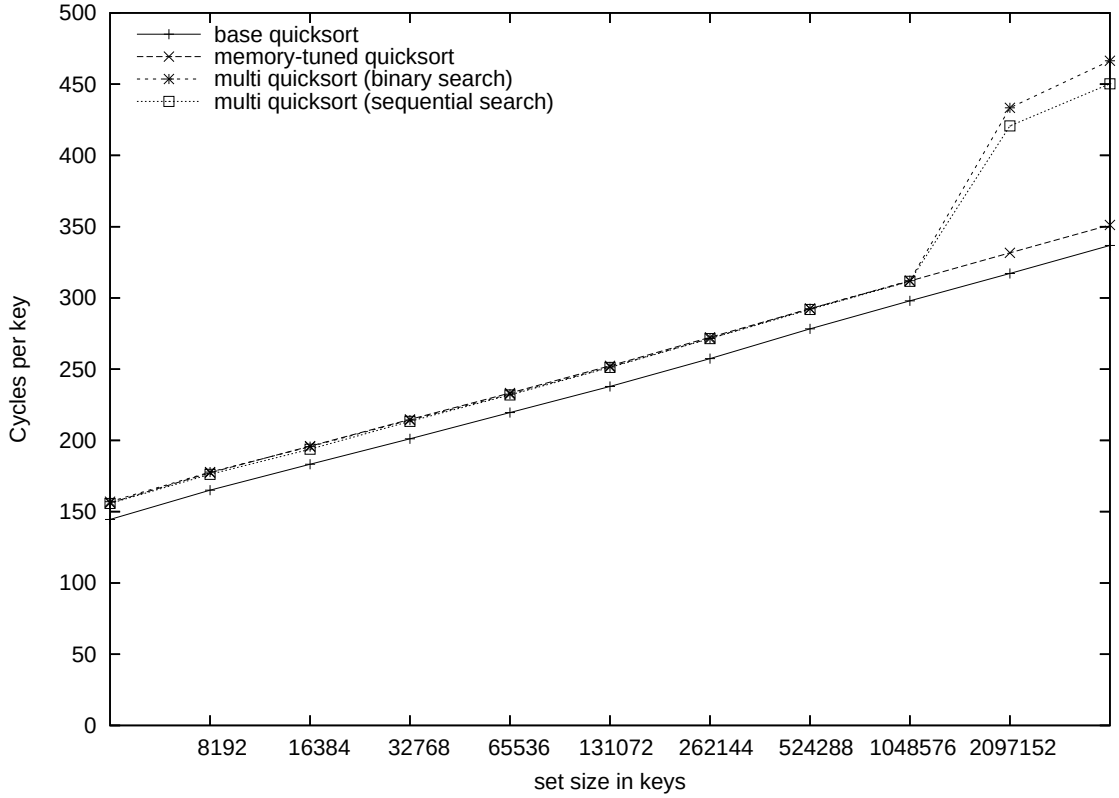
The results show again (see Section 6.6.3) that direct-mapped caches can perform better than fully-associative ones. This is true in the case of all the quicksorts, as can be seen in Figures A.17(d) to A.24(d) graph that this is true in the case of all the quicksorts. As in Section 6.6.3, this is due to the replacement policy of the fully-associative cache.

The branch prediction results are shown in Figure 7.4(b). As expected, there was a a dramatic increase in the binary search multi-quicksort branch misses, but only when using a bimodal predictor. When using a two-level adaptive predictor, the number of misses is, in general, slightly higher, but it does not spike due to the binary search. While it performs worse than the bimodal predictor, it appears that two-level adaptive predictors can adapt to the erratic branches of a binary search.

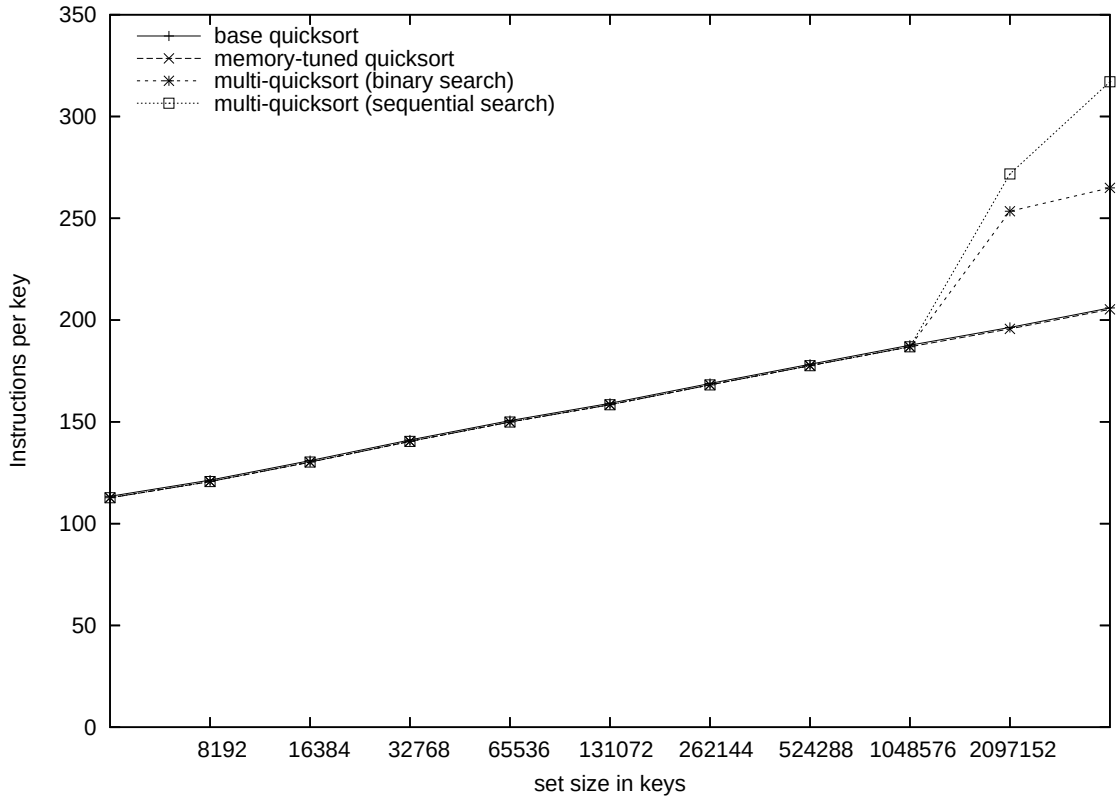
The sequential search stops the increase of branch misses, since every key will only take one miss to find its proper place. The results of this on cycle count, shown in Figure 7.2(a) are enough to make up for the increase in instruction count, which was significantly reduced by using the binary search.

Memory-tuned quicksort had fewer branch misses than base quicksort, due to the fact that fewer comparative branches are executed. In base quicksort, we sort the entire array using insertion sort. In tuned quicksort, we don't sort the pivots, but only the keys between them.

Despite the sequential search multi-quicksort performing better than its binary searched cousin, and its good cache performance, it still failed to approach the performance of either base or tuned quicksort. In

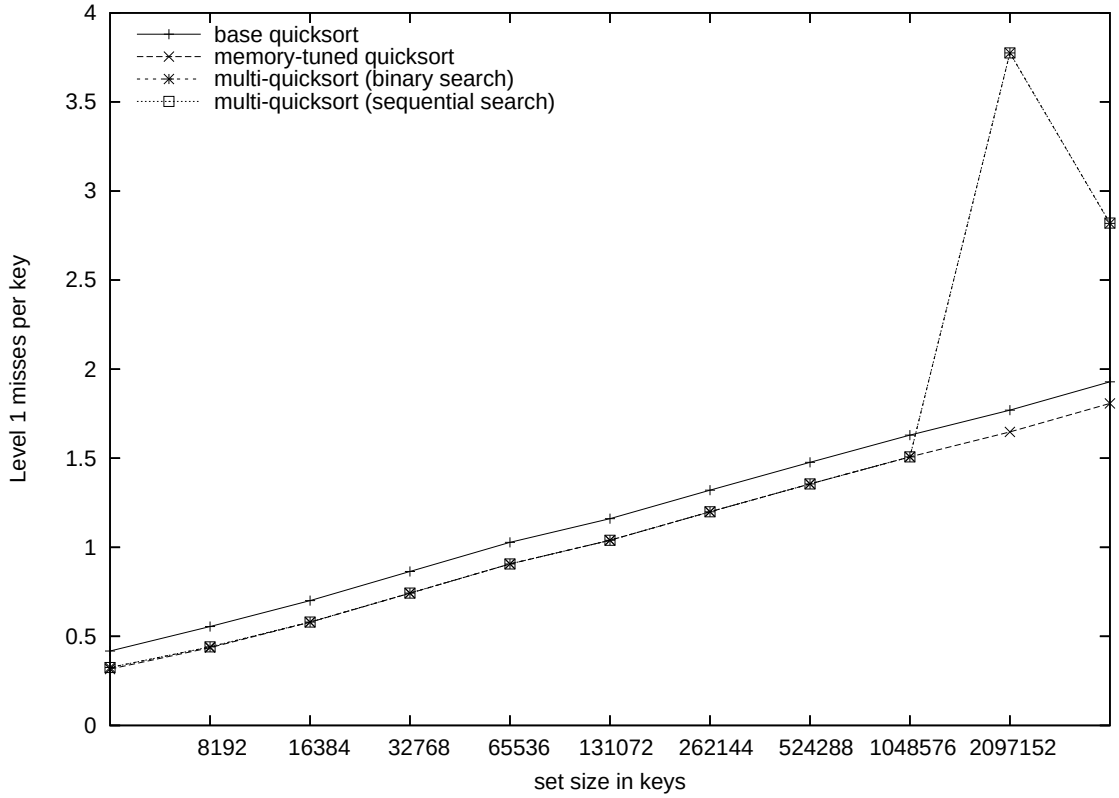


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

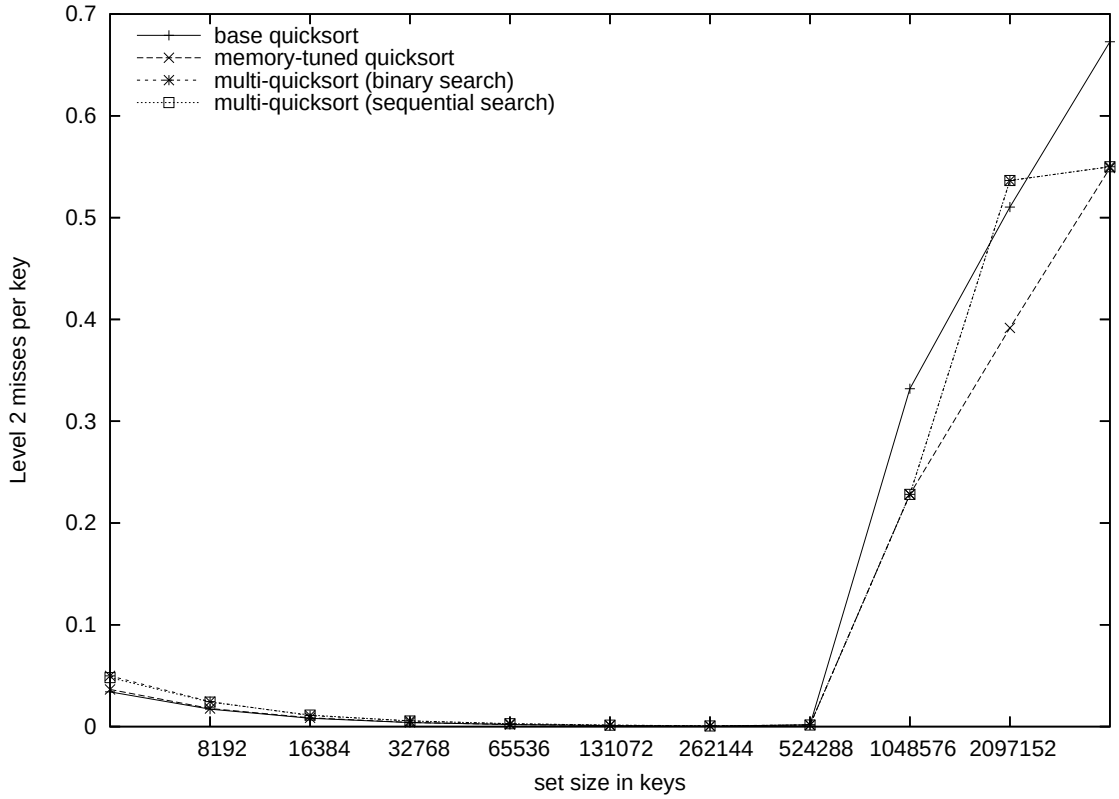


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 7.2: Simulated instruction count and empiric cycle count for quicksort

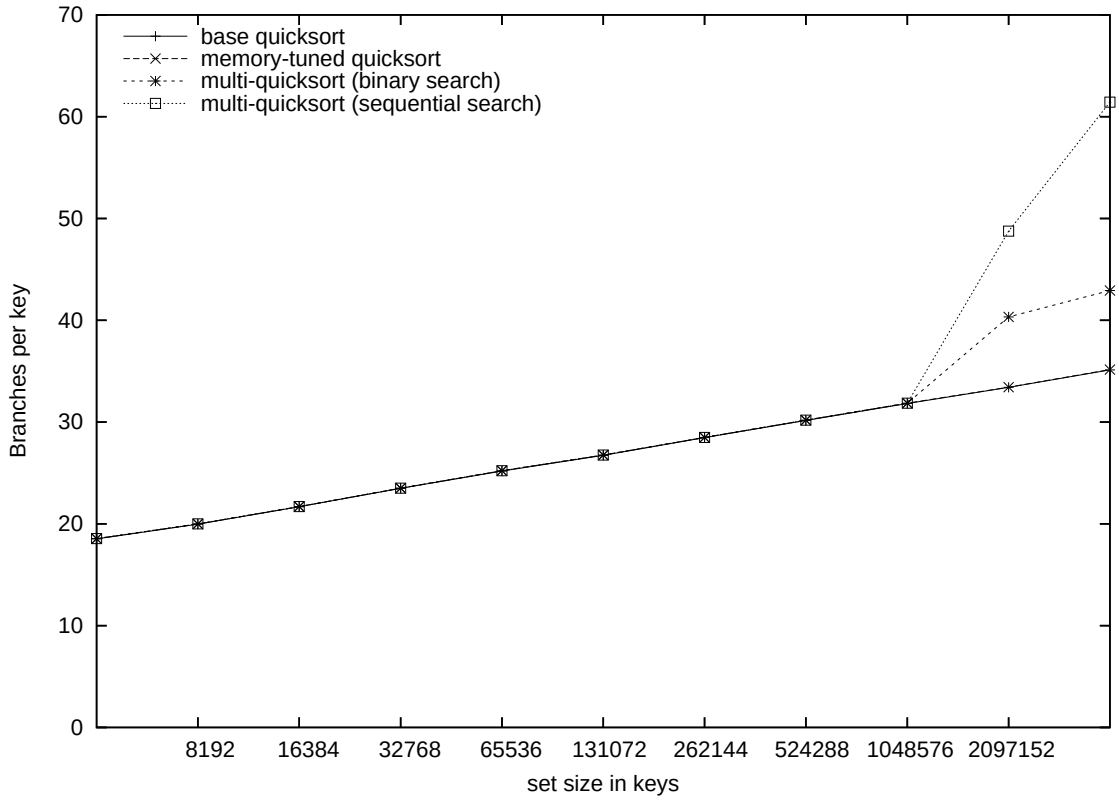


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache. Results shown are for a direct-mapped cache.

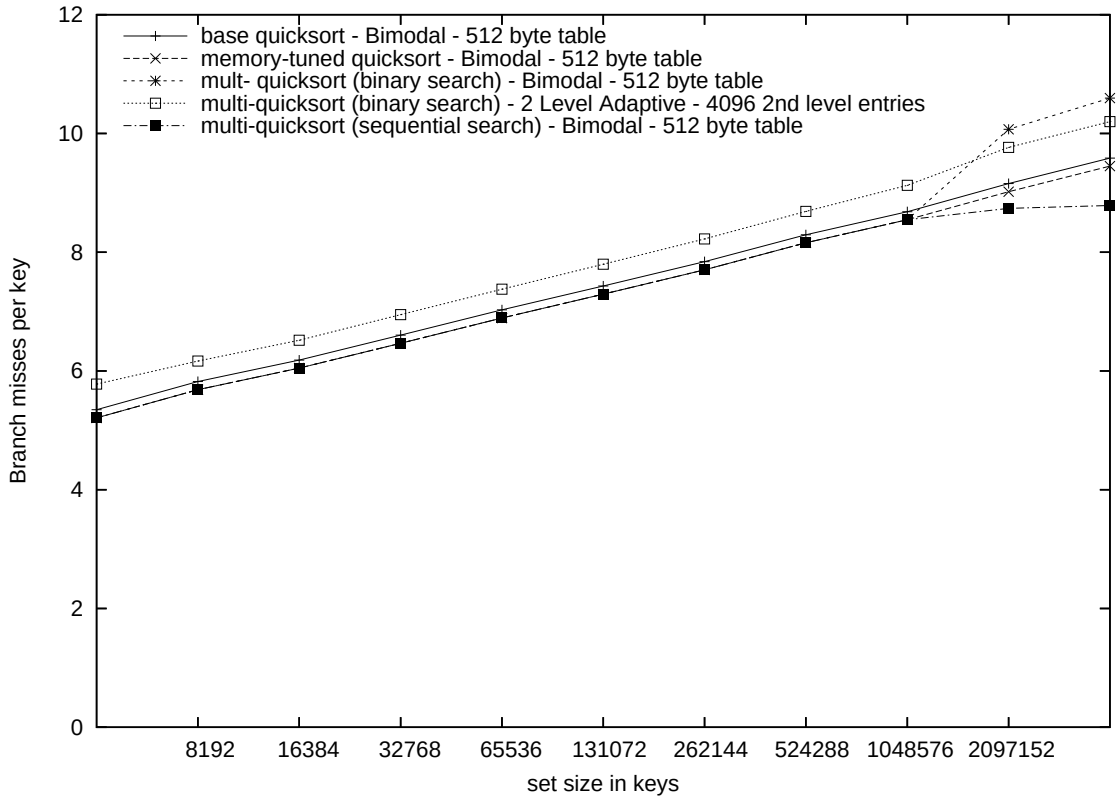


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32-byte cache line. Results shown are for a direct-mapped cache.

Figure 7.3: Cache simulation results for quicksort



(a) Branches per key - this was simulated using `sim-bpred`.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10-bit branch history register which was XOR-ed with the program counter.

Figure 7.4: Branch simulation results for quicksort

fact, base quicksort was the fastest of all the quicksorts in our performance counter test, even though our simulator results show that it has more level 1 misses, level 2 misses, and branch misses than tuned quicksort, and the same number of instructions.

LaMarca's results are again similar to the results presented here, but not identical. The shape of the instruction count graph is the same in both cases, but that of the cache misses is not. The ratio of the cache misses between base and memory-tuned quicksort are the same in both sets of results, but these are lower than the same misses in LaMarca's results. However, the multi-quicksort results appear to be the same, though our results are not conclusive either way.

7.5 A More Detailed Look at Branch Prediction

Figure 7.5(a) shows the behaviour of the individual branches in base quicksort. The i branch is the loop control branch for the index sweeping from left-to-right up the array during partition. The j branch is the corresponding loop sweeping down the array. The partition step terminates when $i \geq j$, and the *Partition End* branch checks that condition. The *Insertion* branch is the comparison branch in the insertion sort algorithm. Recall that our quicksort algorithm does not sort partitions which are smaller than 10 elements, but instead does a final pass of insertion sort over the whole array to sort these small partitions. Finally, the *Median* branch is the sum of the results for the branches involved in computing the median-of-three for each partition.

The only difference between memory-tuned quicksort and base quicksort is that memory-tuned quicksort applies insertion sort to small partitions immediately, whereas base quicksort does a single insertion sort pass at the end. The result is that both algorithms perform almost identically from the viewpoint of branch prediction. Memory-tuned quicksort uses a very slightly smaller number of insertion sort comparison branches, because it sorts only within partitions and does not attempt to move array elements that have acted as pivots. (The reason that quicksort works is that after partitioning, the pivot is in its correct location, and must never move again). In practice, this reduction in branches is probably offset by the increase in mispredictions in the outer loop exit branch of the many insertion sorts.

Overall, base quicksort has a 35% branch misprediction rate, whereas the rate for memory-tuned quicksort is only 34%, while the number of branches also decreases. The main reason for this is the reduction in branches by not attempting to move the pivot elements. Attempting to move these elements usually results in a branch misprediction. The main reason is that most other elements move at least one position, and thus cause the insertion sort loop to iterate at least once. But the pivot elements never move - they are already in the correct place.

When we look at multi-quicksort, there is a 19% drop in the number of iterations of the i and j loops, with a corresponding reduction in the number of mispredictions. Interestingly, the percentage misprediction rate of the i and j branches does not change - it is 37% in both cases.

Multi-quicksort does, however, introduce a lot of additional branches. We already saw in Figure 7.2(b), that the binary search and sequential search variants increase the number of executed instructions by around 25% and 50% respectively. Figure 7.5(c) shows additional measures of the *Binary Left* and *Binary Right* branches, which are the branches within the binary search code. The number of binary search branches is much greater than the reduction in i and j branches. Furthermore, these binary search branches are highly (43%) unpredictable.

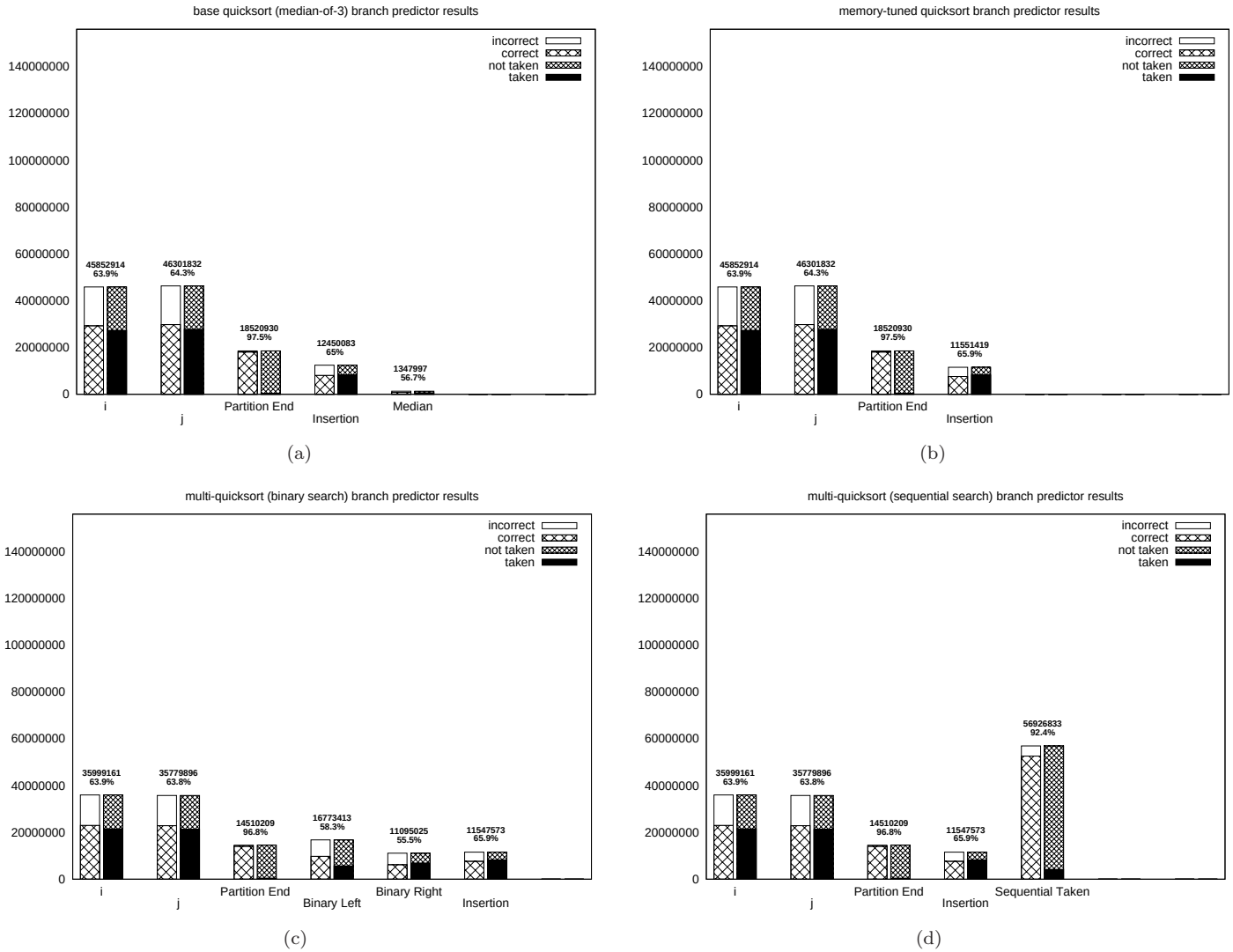


Figure 7.5: Branch prediction performance for base, memory-tuned and multi- quicksorts. All graphs use the same scale. Above each column is the total number of branches and the percentage of correctly predicted branches for a simulated bimodal predictor.

The variation of multi-quicksort that uses sequential search (see Figure 7.5(d)) dramatically increases the number of executed branches, but these branches are highly predictable. This version actually causes fewer branch mispredictions than memory-tuned quicksort.

7.5.1 Different Medians

When we originally looked at quicksort, we expected that the i and j branches in quicksort would resolve in each direction an equal number of times. In fact, this is not the case. The main reason is that the chosen pivot is almost never the true median of the section of the array to be partitioned. Thus, the i and j branches are almost always biased, which makes them predictable. According to Knuth, if the pivot element is chosen randomly, then the i and j branches will be taken (i.e. the i or j loop will execute) twice as often as they are not taken (i.e. the loop exits).

In order to investigate this, we implemented a simple version of quicksort that uses the middle element as the pivot rather than a median-of-three[3]. The arrays we sort are random, so the value of this element will also be random. Figure 7.6(a) shows the behaviour of each of the important branches in this code. We see that the i and j branches are indeed taken about twice as often as not-taken, and the i and j branches are correctly predicted approximately 71% of the time.

The result that the branches are more predictable (71%) than they are biased (66%) is a particularly interesting finding. The reason is that the bias in the results is an average figure over a large number of partitions that quicksort performs to sort the array. On a particular partition, the bias will be more pronounced because the i and j branches will be biased in opposite directions.

For example, consider the case where the pivot element is larger than the true median. In this case, the final resting place of the pivot will be towards the right-hand end of the region to be partitioned. Thus, the i loop will execute a large number of times, and thus the i branch will be biased towards being taken. The j branch, on the other hand, will be biased toward not-taken, because the average element is likely to be smaller than the pivot. Therefore, the j loop will execute only a small number of times. Thus, we see that in a particular partition, we do not normally get an equal number of executions of the i and j loops. Instead we get a large number of executions in one loop (which makes its branch relatively predictable), whereas the other loop often does not enter at all (which makes its branch also predictable, although executed a much smaller number of times). Thus, although overall the i and j branches are taken twice as often as not-taken, on a particular partition one will usually be biased in the not-taken direction, and as a result will be executed a relatively small number of times.

In the preceding paragraphs we have seen that the i and j branches are predictable when we have a pivot that is larger or smaller than the true median of the array section to be partitioned. An interesting question is the effect of choosing a better approximation of the true median on branch prediction. Figure 7.6(b) shows the behavior of these branches when median-of-three is used to select the pivot. Perhaps the biggest effect is that the number of executions of the i and j branches falls by approximately 15%. This fall is partly offset by the comparison branches to compute the median, but these account for only around 1.25% of the original branches from the no-median quicksort. Thus, there is a reduction of almost 14% in the number of executed comparison branches. Even more interesting, however, is the bias in the remaining i and j branches. The no-median i and j branches are 66% biased, but the median-of-

[3]It is important to note that selecting the pivot based on only a single element can be particularly inefficient because it removes the natural sentinels at the top and bottom of the regions to be partitioned that we get from median-of-three. In Figure 7.5.1 we see that this results in an additional branch to handle the edge-case without a sentinel. Although this branch is almost 100% predictable, it is executed a very large number of times.

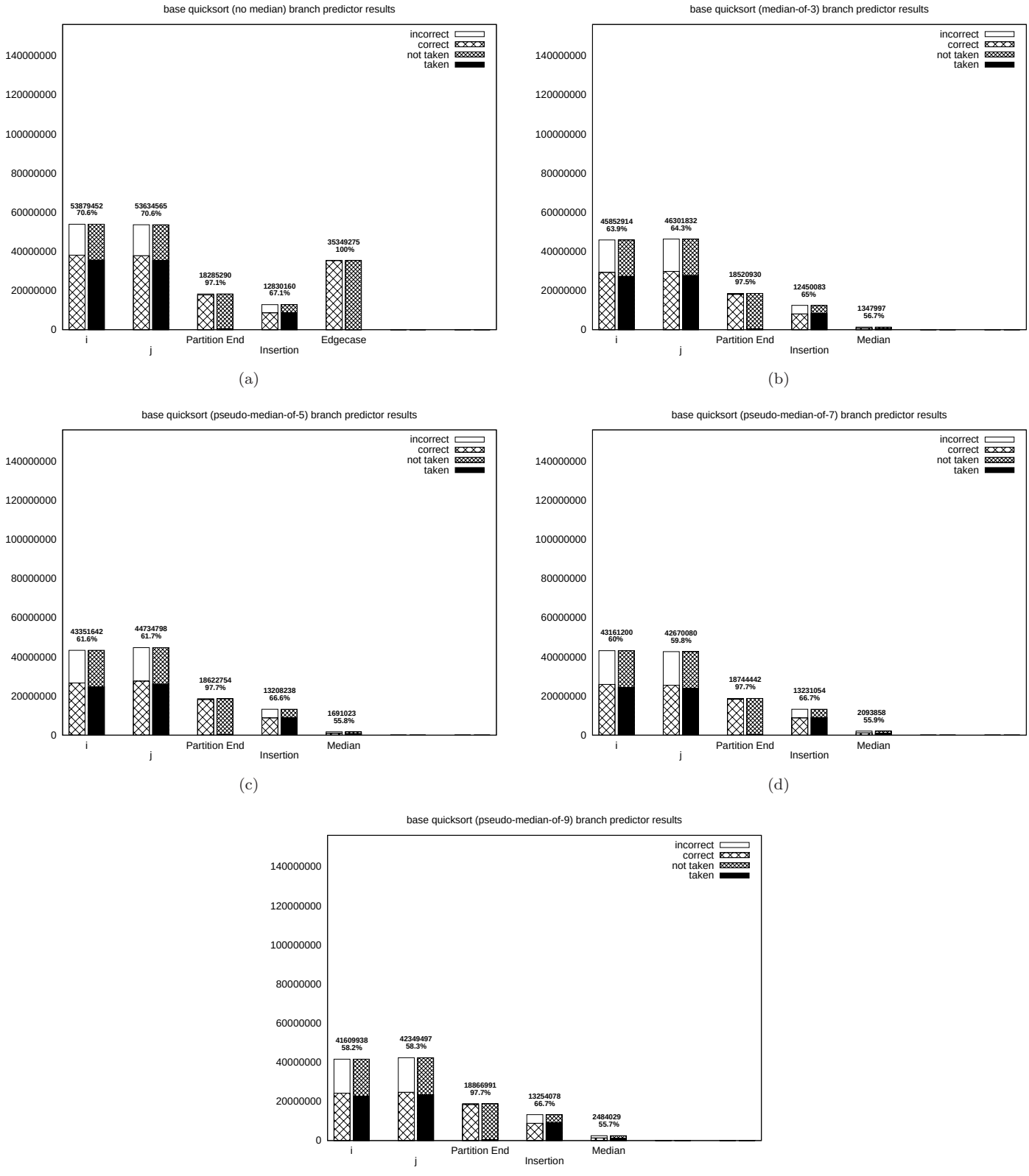


Figure 7.6: Branch prediction performance for base quicksort without a median, with median of 3, and with pseudo-medians of 5, 7 and 9. All graphs use the same scale. Above each column is the total number of branches and the percentage of correctly predicted branches for a simulated bimodal predictor.

three branches are only 60% biased in the taken direction. The accuracy with which these branches are correctly predicted falls from 71% to 64%.

The reason for the reduction in bias and prediction accuracy is that median-of-three gives a much better estimate of the true median of the section to be partitioned. The final resting place of the pivot will usually be closer to the middle of the section, to there is less bias in the branches. Thus, although median-of-three gives around a 14% reduction in executed comparison branches, each of those branches is much less likely to be correctly predicted. Overall there is actually a 6% *increase* in the total number of branch mispredictions.

To further investigate this phenomenon, we measured the effects of using pseudo-medians-of-five, -seven and -nine to choose the pivot. A pseudo-median uses repeated medians-of-three for parts of the computation to reduce the number of comparisons in computing the pivot. The reductions in executed comparison branches are 16.5%, 17.8% and 19.6% respectively. However, the biases of the i and j branches in these algorithms are only 57.8%, 56.3% and 55.1%. The corresponding predictabilities are 61.6%, 59.9% and 58.3%. Overall, the increases in comparison branch mispredictions (including computing the pseudo-median) are 9.2%, 11.8% and 14.3%.

Picking a better estimate of the true median substantially reduces the number of executed branches, but each of those branches is much more unpredictable. Intuitively, this makes sense. Sorting the array involves putting an order on it, and thus reducing its entropy. If we want to sort it in a smaller number of steps, then each step must remove more randomness. Thus, we would expect that the outcome of the branch that removes the randomness would itself be more random.

Figure 7.7 shows the cycle count of each of the quicksorts, showing that a median-of-three offers the greatest trade-off between the reduced number of branches and instructions, and the increase in branch mispredictions. This shows the cost of the extra branch mispredictions. It appears that it can be more costly to remove unnecessary entropy than to endure it.

An additional advantage of using a higher order median is that it dramatically reduces the probability of quicksort exhibiting worst case ($O(N^2)$) time complexity. Although the average case reduction in running time is small, reducing the chance of worst-case performance, which might cause the program to pause for minutes or hours, is very valuable.

7.6 Future Work

The threshold in quicksort is the point at which a simpler sort is used to sort the partition. A larger threshold will results in fewer checks for a sentinel. A smaller threshold means that the instruction count due to the insertion sort will be lower, but more partition steps may be taken. A smaller threshold also makes it more likely that the keys to be sorted all fit in a single cache block. However, the partition is likely to occur across a cache block boundary, though this is not necessarily a bad thing; the next partition to be sorted will be adjacent to the current block, and will already be in memory. Observing how results, especially instruction count and branch misses, vary with changes to the threshold may be interesting. In particular, we expect that a larger threshold would be beneficial, as insertion sort causes far fewer branch mispredictions than quicksort.

Two areas of current research into quicksort are those of equal keys, and median-of-greater-than-three. The former affects the algorithmic properties, and some steps may need to be varied to avoid extra cache

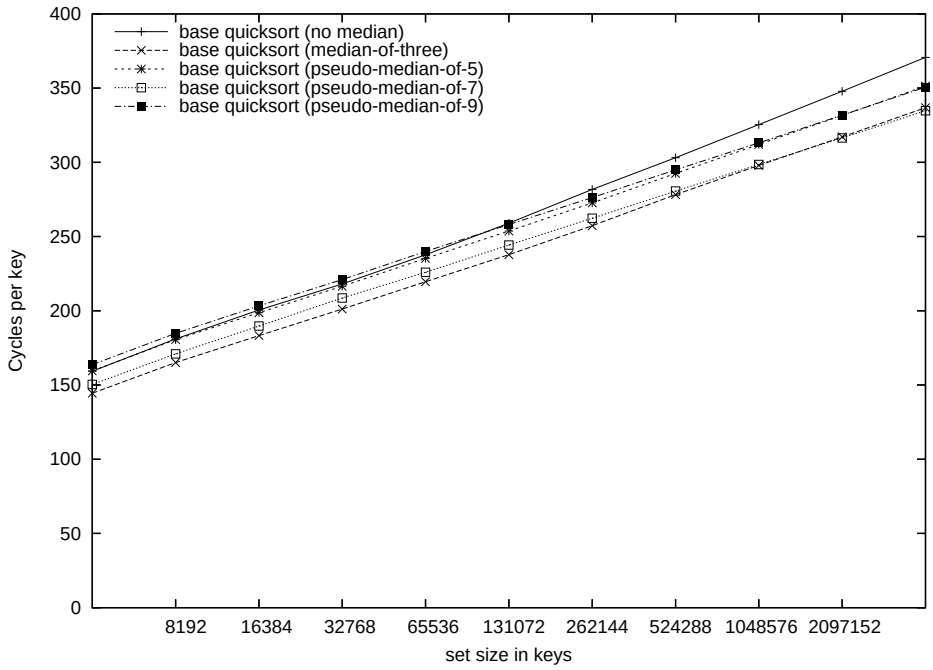


Figure 7.7: Empiric cycles count for quicksort with varying medians

misses as a result of extra copying. The latter simply affects the instruction count, though the extra comparisons are likely to be mispredicted. These results could be measured, giving greater insight to this debate.

Memory-tuned quicksort is the only algorithm here that requires a sentinel check. The branch predictor will only mispredict this once, so the instruction count increase is only that of removing a single check which occurs relatively few ($N/THRESHOLD$) times[4]. Still, the problem is interesting and has applications in other algorithm designs.

Finally, in multi-quicksort, it may be possible to choose better pivots. Choosing better (i.e., more evenly distributed pivots) would reduce the number of containers. The advantage of this is that it reduces linearly the number of pivots to be searched by the sequential search. The most this can be reduced by is a factor of three, indicating perfectly distributed pivots. This would reduce the instruction overhead of the multi-quicksort considerably.

[4]This is a minimum. The figure may be higher since partitions are likely to be smaller than the threshold.

Chapter 8

Radixsort

All other sorts considered in this report are *comparison-based* sorts. They compare two keys at a time, and determine the larger and smaller key, sorting on that basis. Radixsort is a *counting-based* sort. It works by separating the keys into small parts, and processing these parts without regard for other keys. For radix of two, these small parts are bits, and for a radix of 10, these parts would be digits.

The steps to sort by radix are simple. First a radix is chosen, based on the size of the part to be sorted. For 32-bit integers, a byte is used. This means the radix is 256, as there are 256 possible byte values. An array of 256 counters is created to count the number of keys with a particular least significant byte. The counters are then summed in turn to store the cumulative count of keys with a smaller least significant byte. Now each key can be put in place by accessing the counter indexed by its least significant byte, after which that counter is incremented.

The keys are written into an auxiliary array, which is written back to the initial array before the next iteration. The process is then repeated using the second least significant byte. Since every key is processed in order, when the next most significant byte is sorted, the order of the less significant bytes are maintained, so that once the most significant byte is sorted, the entire array is.

8.1 Base Radixsort

This type of radixsort is called *Least Significant Digit* radixsort. Other implementations are possible, including a *most significant bit* version, which behaves as a divide and conquer algorithm. These are discussed in Section 8.4.

Note that this method is only suitable for `unsigned int`s. Because a `signed int`'s sign is contained in the most significant bit, this sort would place negative numbers after the positive numbers.

8.2 Memory-tuned Radixsort

Many improvements exist to radixsort, and some are suggested by Sedgewick and LaMarca. Some of these improvements help to lower cache misses and instruction count, both of which slow down radixsort significantly. LaMarca suggests an improvement by Friend[1] that while the current byte is being put in place, the counting of the next byte can be done. This increases the memory requirements, but this is not significant.

The first improvement is to reduce the number of runs across the array per step. In the *base radixsort*, presented above, there is one step to count, one to copy, and one to write back (which accesses both arrays). We take LaMarca's suggestion and begin the counting of the next least significant byte in the previous iteration.

In fact, all the counting could be done in a single pass at the start, instead of four passes at various stages, decreasing instruction count and cache misses. The downside is the fourfold increase in memory required for the arrays of counts, with no benefits.

The step to copy the array back can also be removed. In mergesort, copying back to the source array was changed to copying back and forth between the source and auxiliary arrays. The same could be done with radixsort, without any increase in complexity. The number of steps is likely to be even, and depends on the radix. It is known at compile time, so the number of times the copy step is done could be reduced to zero since we choose an even radix. If the number of steps is odd, which is unlikely when sorting integers, then one copy at the end would be required. This is not needed in the implementation presented here.

For unsigned 32-bit integers, memory-tuned radixsort should traverse the two array only nine times in total: Once at the start to count the least significant bytes; six times (once per array per pass) as it copies using the three least significant bytes (counting the next least significant byte at the same time), and a final two as it copies using the most significant byte.

8.2.1 Aligned Memory-tuned Radixsort

In Section 6.2, mergesort is improved by aligning its array with reference to its auxiliary array. Specifically, extra memory is allocated for the auxiliary array, which is then offset so that the start of the two arrays are exactly half the length of the cache apart. We further added an alignment for the level 1 cache (see Section 6.3) to significantly reduce level 1 misses at the expense of a small number of level 2 misses. We adapt this strategy here, and call it Aligned Memory-tuned Radixsort.

8.3 Results

8.3.1 Test Parameters

Keys are to be divided into segments 8 bits long, resulting in four sorting steps to completely sort the array. The radix used is therefore $2^8 = 256$.

[1] See [Friend 56a].

8.3.2 Expected Performance

Radixsort is expected to have very good performance, compared to comparison-based sorts. This is due to that fact that radixsort is $O(N)$; for an array as small as 4096 records, it involves 12 times fewer steps than the $O(NlogN)$ sorts. For an array as large as 4194304, an ($O(NlogN)$ sort will take 22 times as many steps to complete. However, it is not expected that radixsort will perform many times better than quicksort, the reigning champion of $O(NlogN)$ sorts. A 'step' for quicksort is as simple as a comparison each step and a possible exchange every few steps. For radixsort, a step comprises being counted four times and copied four times, with the counting involving several layers of indirection, and causing a complete iteration across the array at the start. LaMarca's results show the instruction count of quicksort and radixsort being in the same order of magnitude, and we expect the same.

The cache performance of base radixsort is not expected to be good, however. Quicksort and mergesort have very good temporal locality. Keys are used repeatedly before being swapped out of the cache. In radixsort, however, every key is used just once each time it is in the cache. When the array is larger than the size of the cache, every key access will be a miss as a result, and every new byte being sorted involves four passes over the array. In addition, if the auxiliary array is pathologically aligned, then the copy back into the array can result in thrashing.

The thrashing can also occur in aligned radixsort. Unlike mergesort, we cannot guarantee that the keys being considered are at the same positions in both arrays. In fact, we have no idea where is being accessed in the source array, so it is unlikely that the alignment will have a great effect.

The number of iterations across the array is constant, however, proportional to $(bitsPerInt/log_2(Radix))$. For a 32-bit integer and a radix of 256, this is four steps and each step has five passes over the array, for a total of 20 passes over the array. With eight keys per cache line, there should be about 2.5 misses per key, when the array doesn't fit into the array, though this depends on the placing of the auxiliary array.

The number of branch prediction misses should be exceptionally low. In comparison-based sorts, there are generally frequent mispredictions[2]. Radixsort has no comparative branches at all. As a result, only flow control branches occur in radixsort, and they are all predictable.

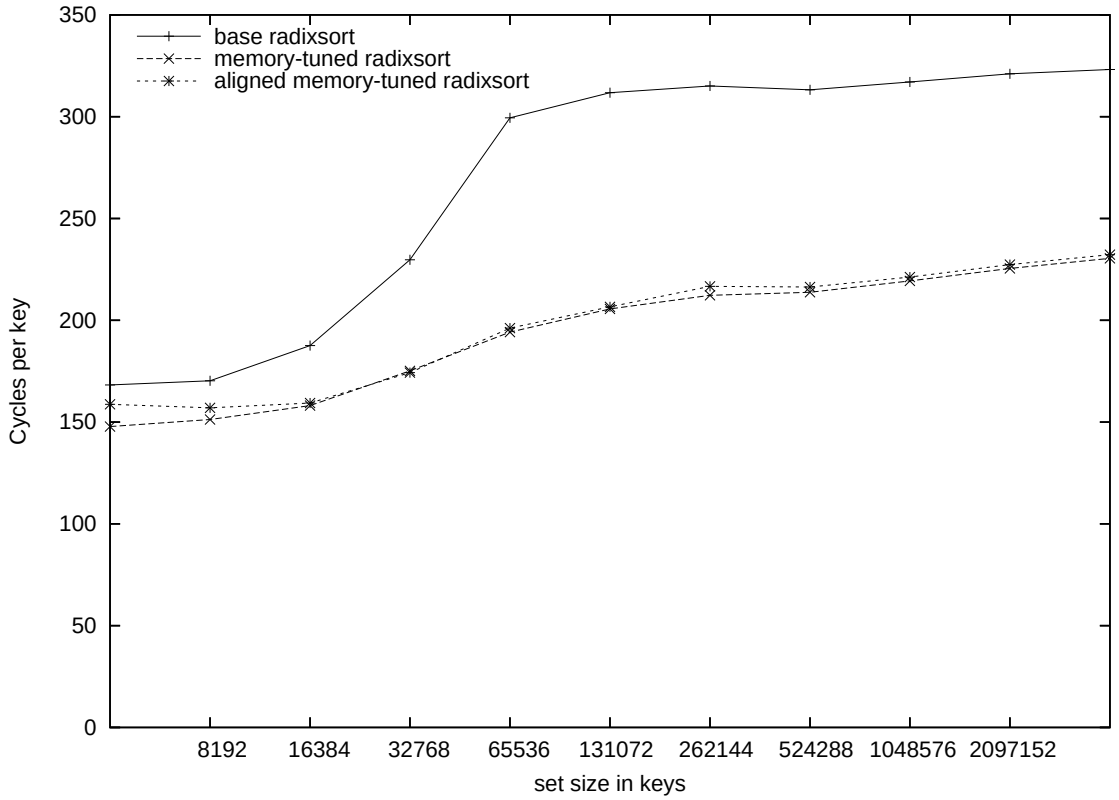
Due to the reduced copying, memory-tuned radixsort should have a lower instruction count and fewer cache misses that base radixsort. There should be nine passes over the array, leading to just over one miss per key, as opposed to two misses per key with base radixsort. The complexity and branch prediction results should remain the same. The instruction count should, however, reduce by nearly a half to $\frac{9}{20}^{th}s$ of base radixsort.

Based on LaMarca's results, showing quicksort to be faster than radixsort by a small amount, we expect memory-tuned radixsort to run faster than quicksort.

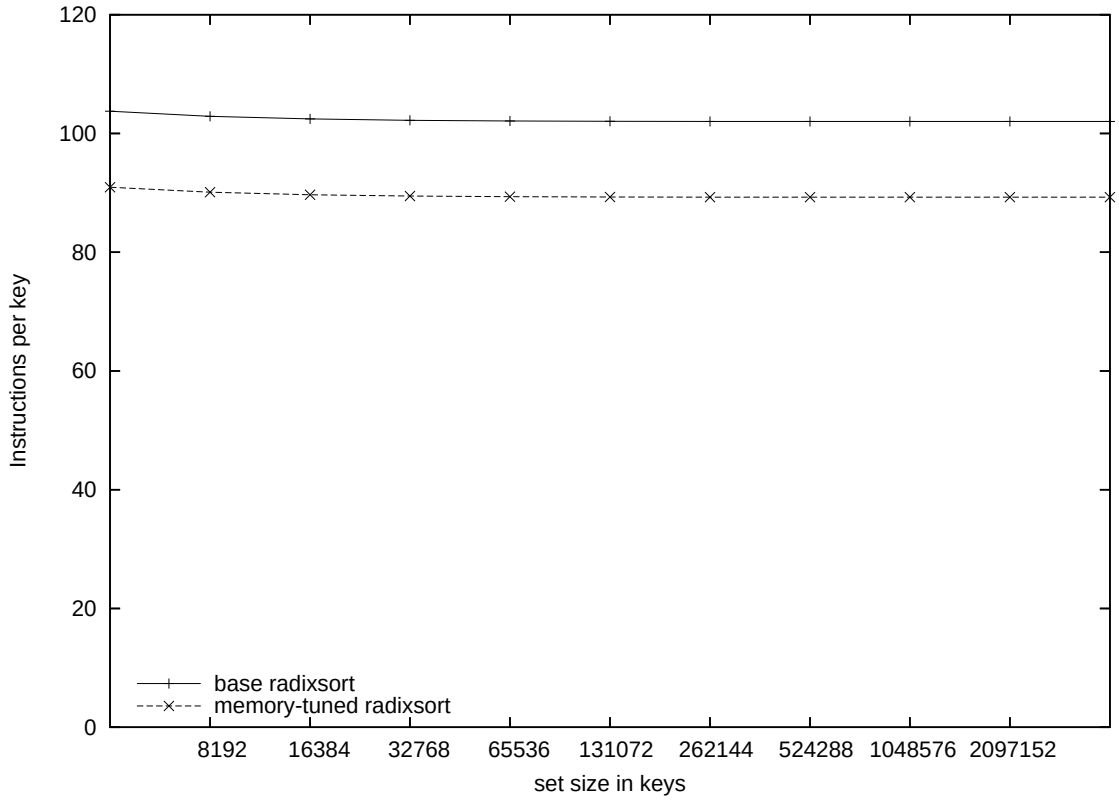
8.3.3 Simulation Results

The instruction count of base radixsort, shown in Figure 8.1(b) is very low. It is lower than even quicksort for the smallest set, and it flat-lines immediately (after slight amortisation of the loop overhead). The number of cache misses in Figure 8.2(b) is only slightly more than quicksort - due to being out-of-place - until the array no longer fits into the cache. The number of misses then leaps up to 2.5 per key, which is

[2]See Section 4.7 for an exception.

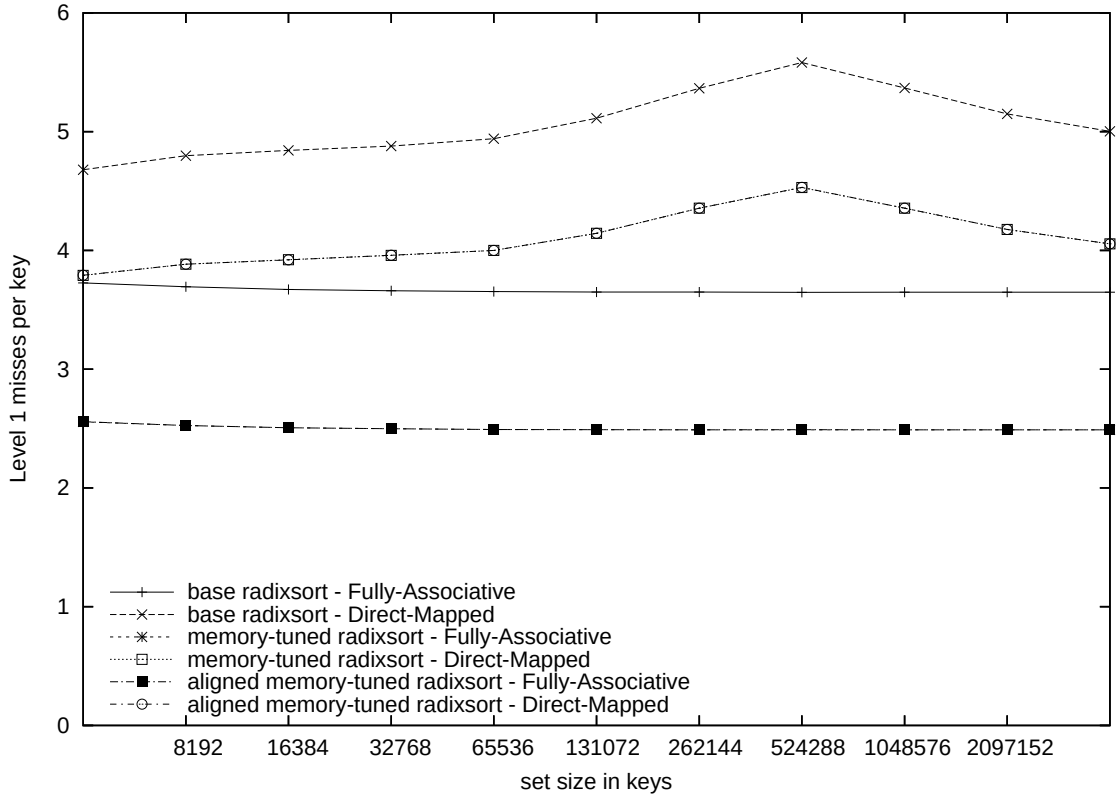


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

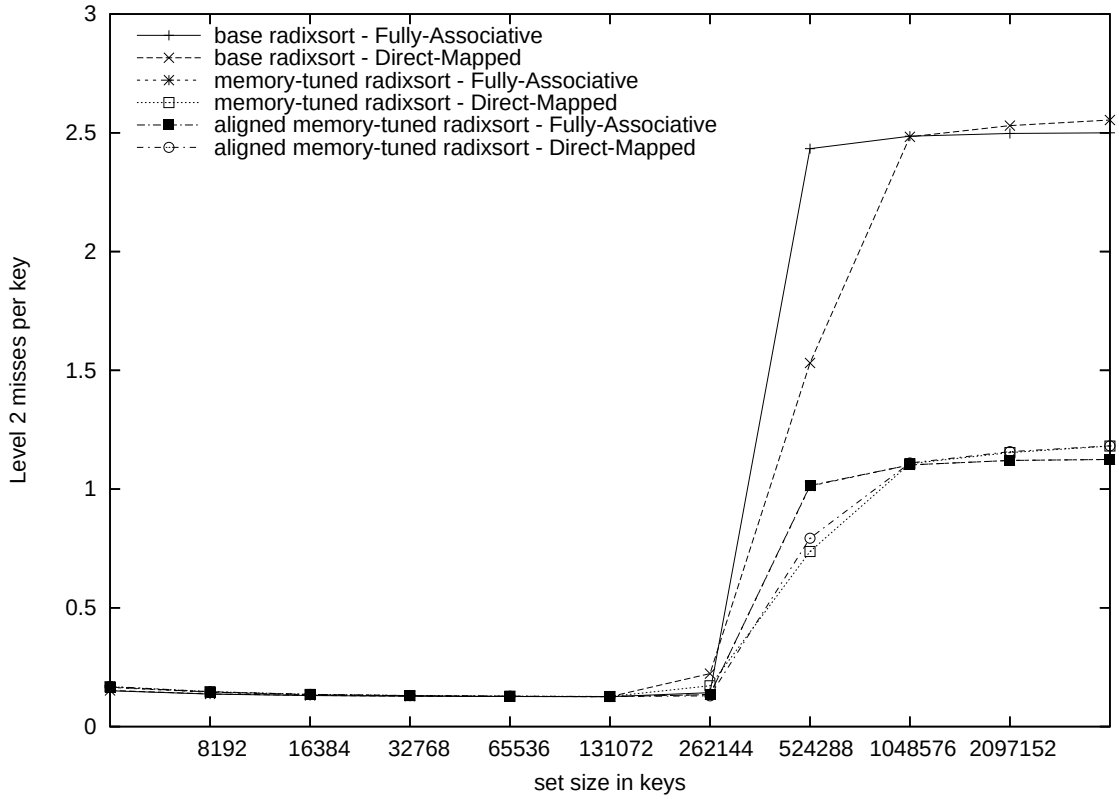


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 8.1: Simulated instruction count and empiric cycle count for radixsort

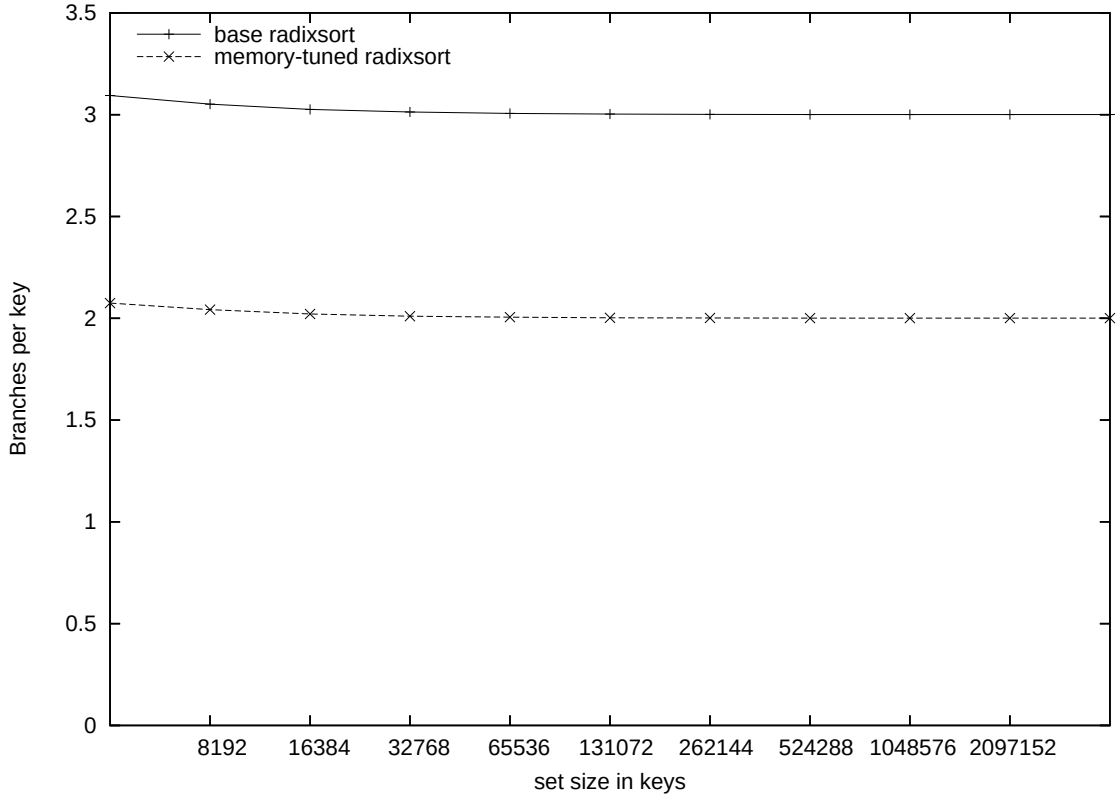


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache.

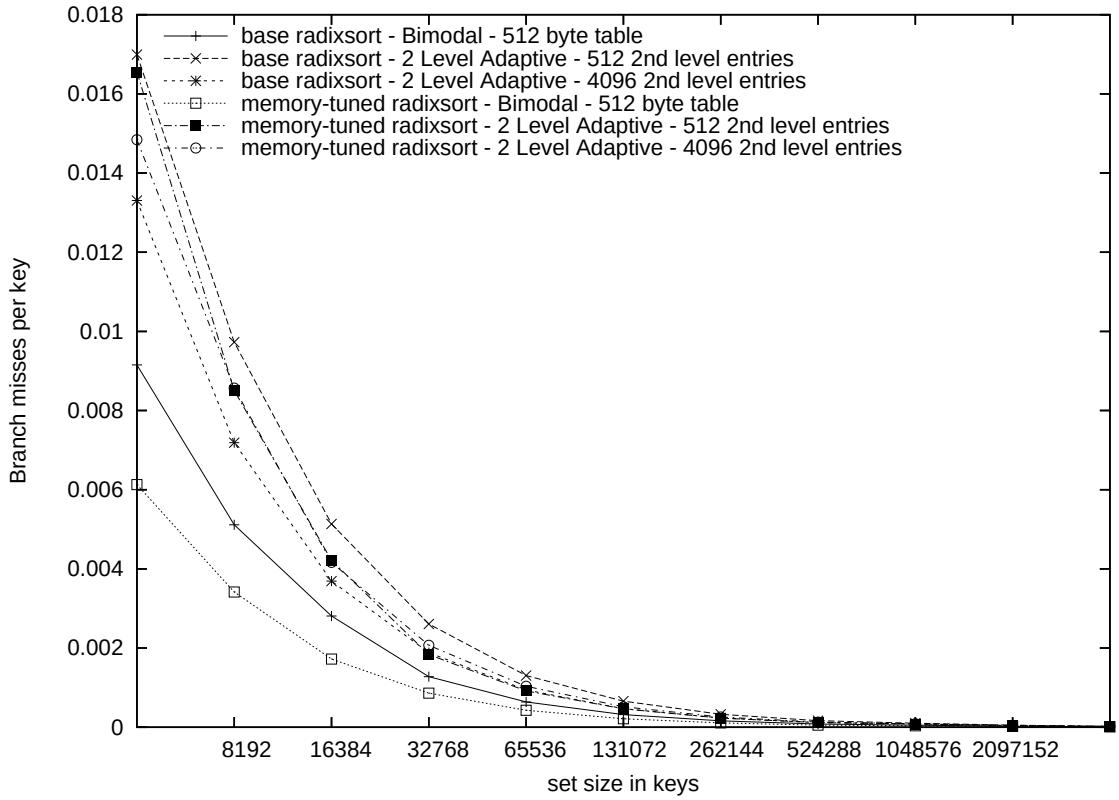


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32-byte cache line.

Figure 8.2: Cache simulation results for radixsort



(a) Branches per key - this was simulated using `sim-bpred`.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10-bit branch history register which was XOR-ed with the program counter.

Figure 8.3: Branch simulation results for radixsort

slightly more than was predicted. The extra is likely due to the unpredictable way that the source array is accessed as the keys are copied to the auxiliary array.

Memory-tuned radixsort performs as predicted as well. By sorting keys between two arrays instead of copying a partially sorted array back, the instruction count has been reduced by more than 10%, and the number of level 2 cache misses have been halved, to one and a quarter miss per key. This is exactly as expected, due to there being nine passes over the array, and a one miss per pass per cache line (coming to slightly under one and a quarter. The extra is again probably due to the unpredictable order of the copying).

The number of misses while the array fits in the cache is obvious: there are two compulsory misses per key - one for each array - which divided by the keys per cache line results in exactly one miss in four (although our graph obscures this, due to the initial subtraction of compulsory misses).

Aligned memory-tuned radixsort performs exactly the same as memory-tuned radixsort in both level 1 and level 2 misses, indicating that the alignment, and the associated memory overhead, are not necessary.

The number of branch misses, shown in Figure 8.3(b) was also extremely low. Despite three branches per key, there was less than .02 of a branch miss per key. The number of branches itself is quite low, as several of the sorts considered had more than that number of mispredictions. However, it is not unexpected. The number of branches reduces when copying back is removed, and while this reduces the number of mispredictions, there are so few to begin with that this result is not significant.

Again we see that the bimodal branch predictor performs better than the two-level adaptive. In this case it is contrary to expectations, however, as there are no comparative branches. However, there are so few misses that these results do not show anything; in the worst case there are slightly over 200 misses in total.

With the low number of cache misses, low instruction count and negligible branch mispredictions, it is no surprise that memory-tuned radixsort performs as well as it does. As shown in Figure 8.1(a), memory-tuned radixsort's performance is almost linear, with only a slight increase when the arrays no longer fit in the cache. Even then, the increase in cache misses are more than made up for by the lack of branch mispredictions, and quicksort ends up taking almost 50% percent longer to sort more than 4 million keys.

LaMarca's radixsort results are quite different to those presented here. His instruction count graph has a logarithmic shape: it begins very high and sharply falls before flattening out. By contrast, the graph here is almost completely flat. He shows a similar result in both cache miss graphs: a sharp dip, followed by a nearly flat-line. This is due to the instructions, time and compulsory misses being amortised over the ever larger array size. The trend is not expected in our graphs due to subtraction of the time taken to fill the array.

Also, in both cases in LaMarca's graphs, once the array no longer fits in the cache, the number of misses jumps to around 3.25 (which would be equivalent to 1.625 misses per key in our results), where they stay. Our results disagree, showing approximately 2.5 misses per key for base radixsort, in line with our predictions.

However, LaMarca's conclusion - that radixsort is not as fast as quicksort - can be easily explained from the base radixsort results. Once an radixsort optimized for the cache is presented, these no longer hold, and the lack of branch misprediction results in a new fastest sort.

8.4 Future Work

Combining other sorts with radixsort could be possible. Radixsort should be be faster than multi-quicksort's initial step. Replacing this with a single radix step would split the array into 256 containers, each of which could be quicksorted. Alternatively, a smaller radix, based on the size of the array, could be generated at run time. Using LaMarca's analysis, $C/3$ containers need to be generated. With N as the number of keys in the array, and C is the number of keys which fit in the level 2 cache, the radix can be calculated with the formula:

$$2^{\lceil log_2(\frac{3N}{CacheSize})\rceil}$$

This will only work for random, evenly distributed keys. If all the keys only vary in their least significant bits, then this will have no effect.

The radixsort presented above is a Least Significant Digit radixsort. A Most Significant Digit radixsort behaves differently. Once the initial sort is done, each bin is recursed into, and continually sorted in this manner. This results in a higher instruction count, and increased stack usage due to the recursion. However, there should be a large increase in temporal reuse due to this, which may make up for it.

Chapter 9

Shellsort

Shellsort is a derivative of insertion sort designed to offer a compromise between the performance of mergesort and radixsort[1], and the space efficiency of selection sort, bubblesort and insertion sort. It was designed by Donald Shell in 1959, and described in [Shell 59][2].

Shell begins by considering the array to be made up of $\frac{N}{2}$ sub-arrays of length two, where the elements of the sub-array are spaced $\frac{N}{2}$ keys apart. These sub-arrays are then sorted using insertion sort. The first iteration will be a simple exchange, but the keys which are exchanged move across the array by $\frac{N}{2}$ spaces. The sort then considers $\frac{N}{4}$ arrays, whose elements are spaced $\frac{N}{4}$ keys apart, which are individually sorted by insertion sort, then $\frac{N}{8}$ and so on. The final iteration will be a standard insertion sort, after which the array will be sorted. A problem with this implementation is that since the increments are powers of two, keys with odd indices are not compared with keys with even indices until the final insertion sort. This can lengthen the final insertion sort's running time to $O(N^2)$.

Since first published, there has been much discussion about the best *increments* to use in shellsort. Shell's original sort, described above, has the increments $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$, $\frac{N}{16}$ ..... 1. [Sedgewick 96] discusses many increments used, and provides code for comparing these. Based on running the `driver` program[3], Gonnet's shellsort was seen to be the fastest on the authors' local machine. As a result, this was used to test the performance of shellsort. This uses increments which are $\frac{5}{11}^{th}s$ of the previous increment.

Two versions of Gonnet's sort were used. The first was taken directly from the code presented by Sedgewick, and only slightly modified to fit into the framework used. The second was modified to reduce the complexity of the loop bounds, and to remove a bounds check in the inner loop. Code for shellsort and improved shellsort is shown in Figures 9.1 and 9.2.

Apart from these changes, the two algorithms algorithms are subtly different. Due to a slight difference in behaviour when $h = 5$, the improved shellsort runs faster than shellsort when this condition occurs. This is due to shellsort running an extra iteration, with $h = 2$, which is skipped by improved shellsort. This occurs when the array is sized 16384, 32768 and 4194304.

[1] Quicksort and heapsort were developed only later, so at the time there was no efficient in-place sort.

[2] Shell describes the former sorts as *Merging Pairs* and *Floating Decimal Sort*, respectively, and the latter sorts as *Finding the Smallest*, *Interchanging Pairs* and *Sifting*, respectively.

[3] See http://www.cs.princeton.edu/~rs/shell/

```
void shellsort(Item a[], int N)
{
   int l = 0, r = N-1, i ,h;
   for (h = r-l+1; h > 0; h = ((h > 1) && (h < 5)) ? 1 : 5*h/11)
   {
      for (i = l+h; i <= r; i++)
      {
         int j = i;
         Item v = a[i];
         while (j >= l+h && less(v, a[j-h]))
         {
            a[j] = a[j-h];
            j -= h;
         }
         a[j] = v;
      }
   }
}
```

Figure 9.1: Shellsort code

9.1 Results

9.1.1 Test Parameters

The increments in Gonnet's shellsort are $\frac{5}{11}N$, $(\frac{5}{11})^2N$, $(\frac{5}{11})^3N$ ... 1;

9.1.2 Expected Performance

Shellsort is designed to perform similarly to an $O(NlogN)$ sort. The instruction count and cycle counts are therefore expected to be similar to $O(NlogN)$.

The cache performance of shellsort should be regular. Each sub-array is sorted in turn without reference to the next. Since each element of a sub-array is quite far from its next element, an entire cache line must be loaded for each key, which will be used once per iteration of the sort across the sub-array. If there are a large number of sub-arrays, as during the iterations with large increments, there will be no spatial reuse, and little temporal reuse, so long as the sub-array is larger than the cache. Once the increment is small enough that each sub-array fits within the cache (i.e. quickly for L2 cache, and much more slowly for the L1 cache), then temporal reuse will occur. Once an entire sub-array fits within an eighth of the size of the cache (eight being the number of keys that fit in a cache line), then spatial reuse will occur as well. This reuse will only occur within a single iteration; once the increment with the next increment begins, most of the contents of the cache will be invalidated, unless the entire array fits within the cache.

The branch performance of shellsort should be predictable. There are $log_{(\frac{11}{5})}N$ iterations, each with N branch misses[4]. It is expected, therefore, that there will be $log_{(\frac{11}{5})}N$ misses per key.

[4]See Section 4.7 for a discussion of the properties of insertion sort.

```
void improved_shellsort(Item a[], int N)
{
   int i, h, min, bound;

   /* shellsort it */
   for (h = 5*N/11; h >= 5; h = 5*h/11)
   {
      for (i = h; i <= N-1; i++)
      {
         int j = i;
         Item v = a[i];
         while (j >= h && less(v, a[j-h]))
         {
            a[j] = a[j-h];
            j -= h;
         }
         a[j] = v;
      }
   }

   /* put a sentinel in place */
   min = 0;
   bound = 11;
   if (N < bound) bound = N;

   for(i = 1; i < bound; i++)
      if(less(a[i], a[min]))
         min = i;
   exch(a[0], a[min]);

   /* do the final insertion sort */
   for(i = 1; i < N; i++)
   {
      int j = i;
      Item v = a[i];

      while(less(v, a[j-1]))
      {
         a[j] = a[j-1];
         j--;
      }
      a[j] = v;
   }
}
```

Figure 9.2: Improved shellsort code

Improved shellsort should have a slightly lower instruction count, and hence a lower cycle count, than shellsort. It's cache performance should be the same, and it should have the same number of comparative branches, and hence a similar number of mispredictions, though its branch count should be higher than shellsort.

9.1.3 Simulation Results

Shellsort's simulation results have the characteristic shape and range of an $O(NlogN)$ sort, albeit, not a very good one. The slopes of its graphs are slightly steeper, as can be seen in Figures 9.3(a), 9.3(b), 9.4(a) and 9.5(b). It performs better than a heapsort, but considerably worse than quicksort. However, it easier to program than either of these two sorts.

Shellsort performs roughly double the number of branches and instructions of quicksort, and has more than twice as many branch mispredictions and nearly four times as many cache misses. It's cycle count is over three times that of quicksort, though it is more than 50 percent faster than the best of the heapsorts.

Shellsort's cache misses are certainly in the range of an $O(NlogN)$ sort, and are not considerably worse than memory-tuned heapsort, and are in fact better than base heapsort. This occurs without any cache improvements being performed upon them. Shellsort's branch mispredictions, are, as predicted $O(Nlog_{\frac{5}{11}}N)$.

Improved shellsort does improve on shellsort, having approximately 10% fewer instructions. It also has slightly fewer cache misses and branch mispredictions. Overall, this leads to a reduction in cycle count of between 35% and 70%, as seen in Figure 9.3(a). While a small amount of this can be attributed to the reduction in instruction count due to the removal of the bounds check, a large amount cannot. We believe that the rest is due to slight differences in the performance of the code generated by the compiler: register allocation may be more efficient, for example, or better performing instruction scheduling may take place.

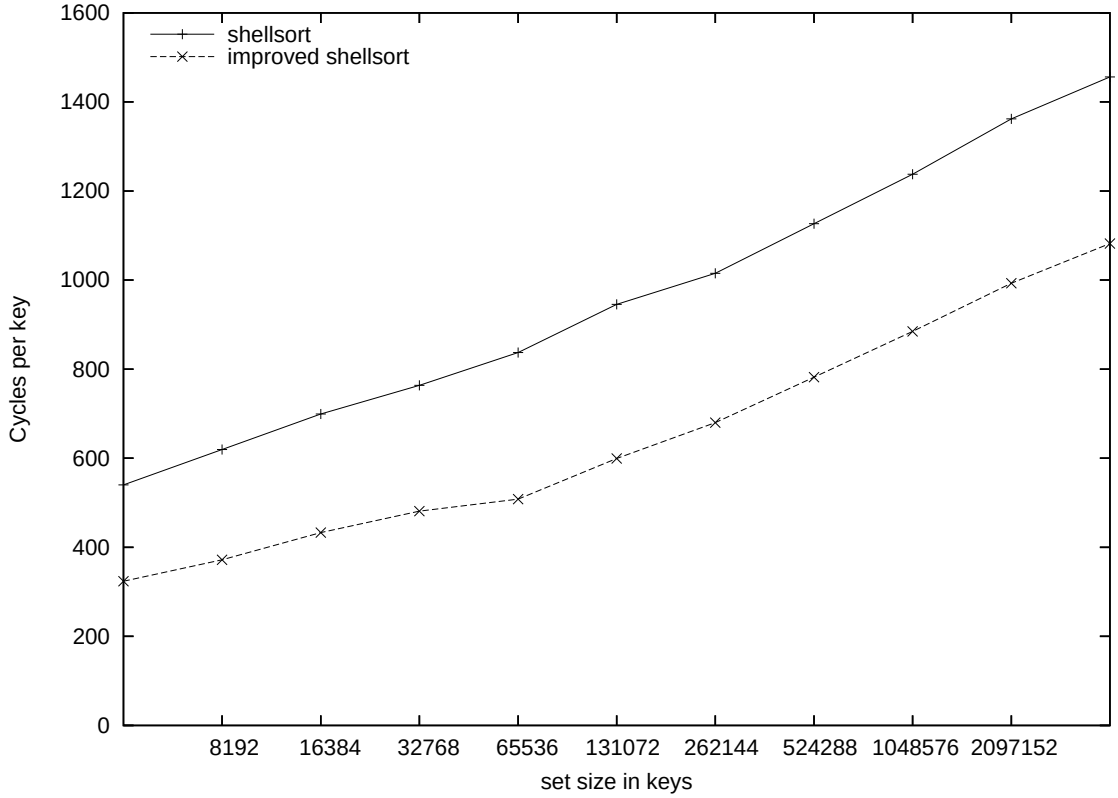
9.2 A More Detailed Look at Branch Prediction

In shellsort, almost exactly half of the important branches are mispredicted. Simply predicting that all branches are taken would result in a slightly higher prediction ratio of 50.8%. Similarly, for improved shellsort, slightly more than half of the branches are correctly predicted, at 50.3%, rising to 53% using a strategy of predicting all branches as taken. These results can be seen in Figures 9.6.

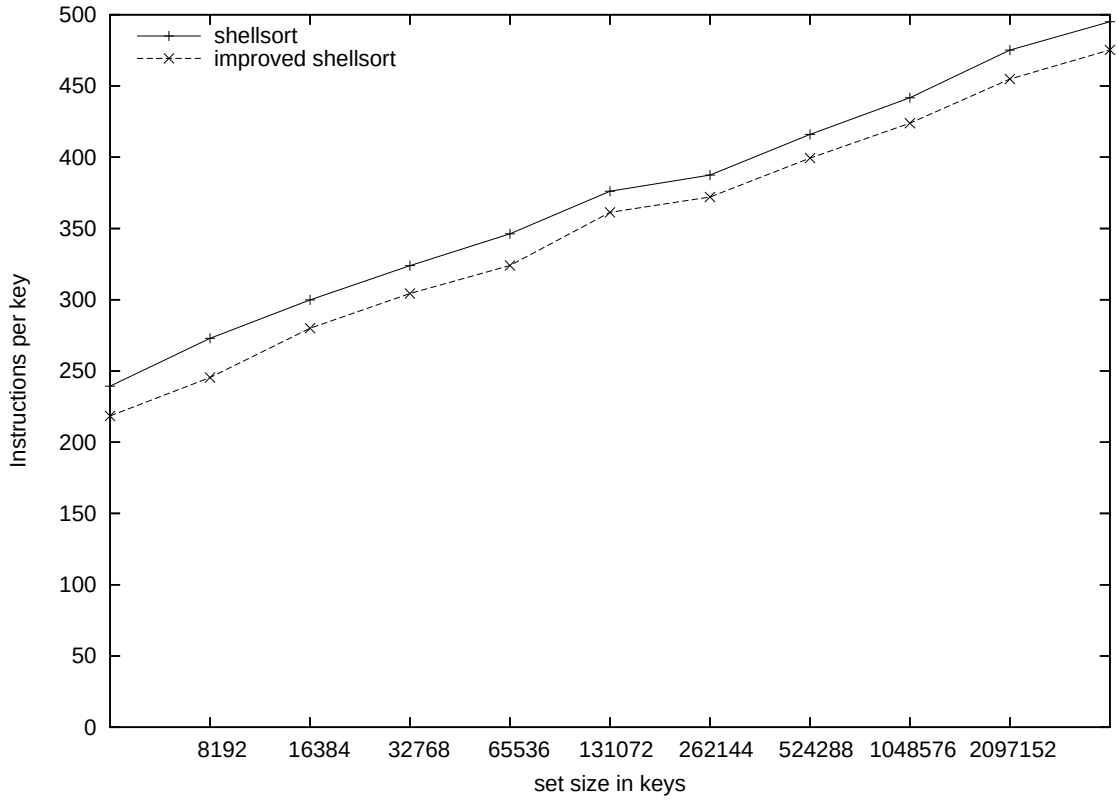
In general, the number of taken branches in both sorts would be almost exactly the same, with only a minor difference due to the positioning of the sentinel. However, during the sorts, the $h = 5$ condition was hit, meaning that there were less branches and less taken branches in the improved shellsort than in the shellsort.

The results show that in improved shellsort, each key moves an average 0.915 places each iteration, while in the final insertion sort, each key moves an average of 2.1 places. We can extrapolate a weighted average from this, to determine that in shellsort, run under the same conditions, each key would move an average of 0.9744 places.

The expected number of branch mispredictions is $log_{(\frac{11}{5})}N$. This is almost exactly correct, being slightly

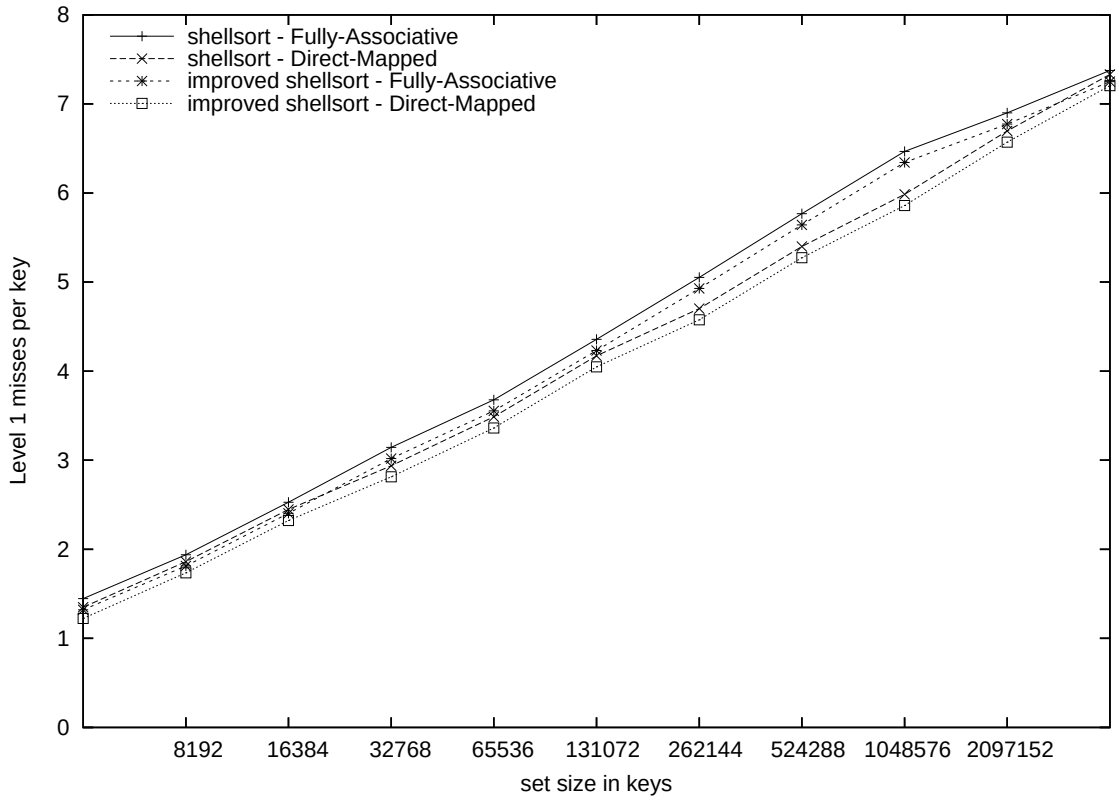


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

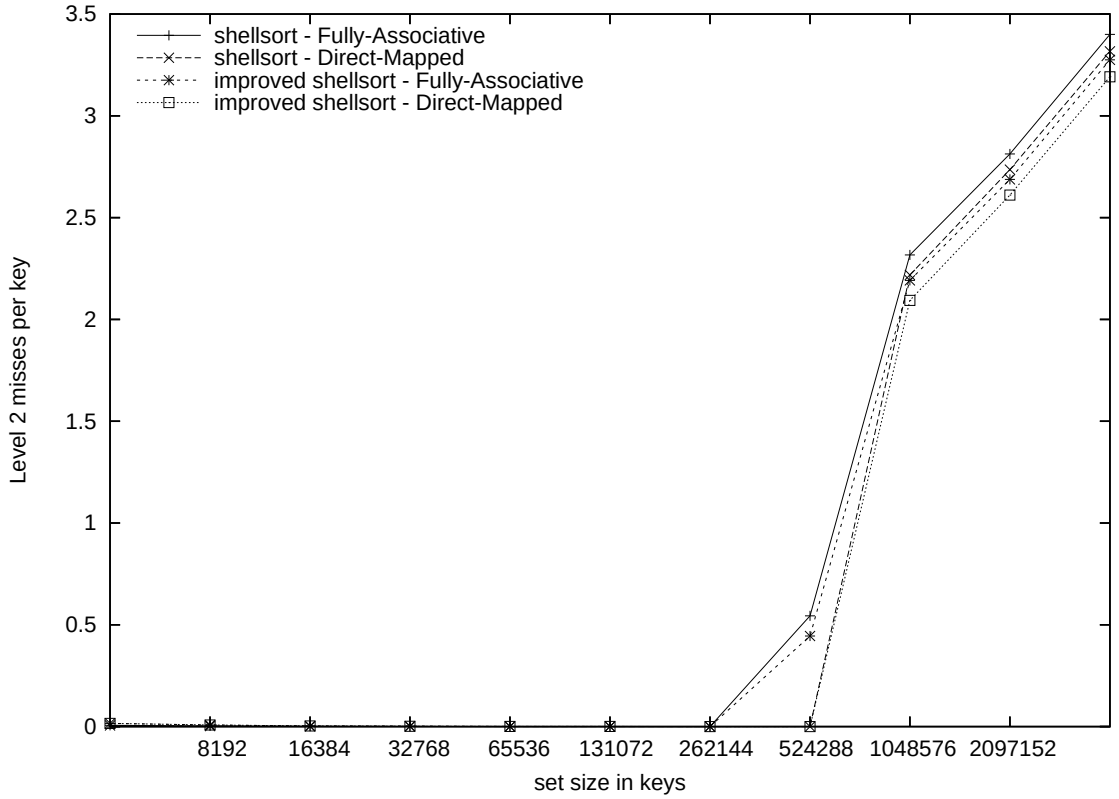


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 9.3: Simulated instruction count and empiric cycle count for shellsort

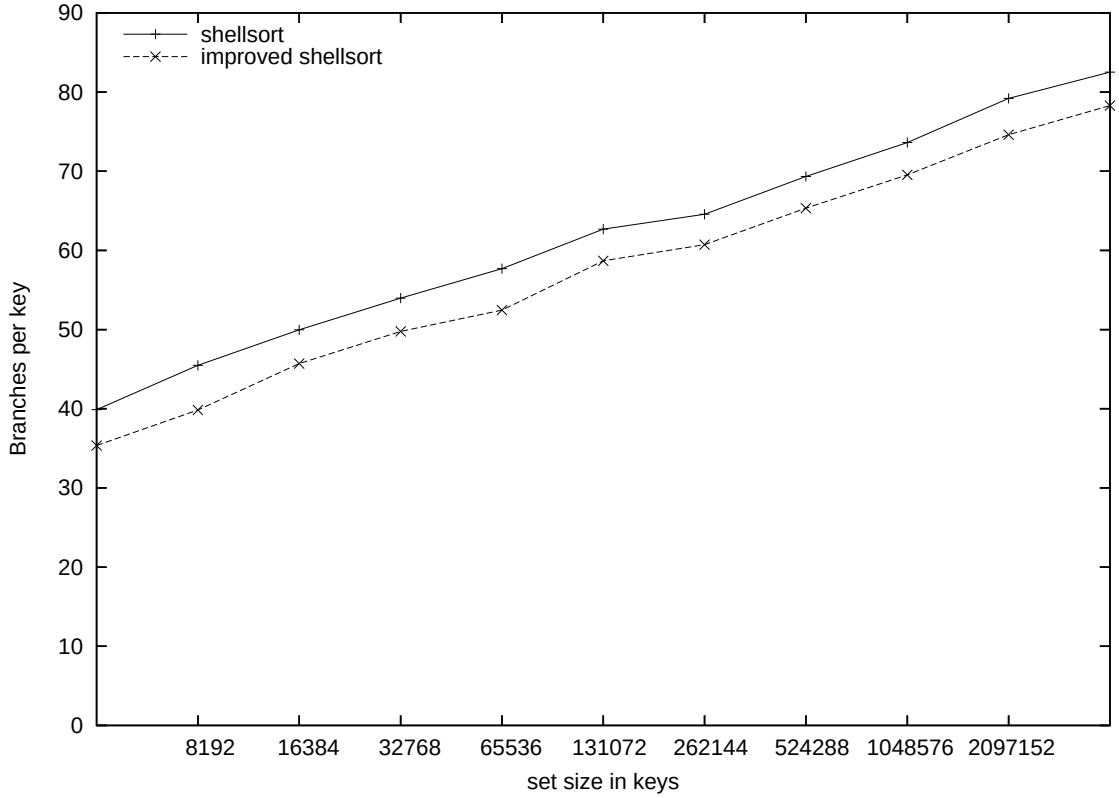


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32-byte cache line and separate instruction cache.

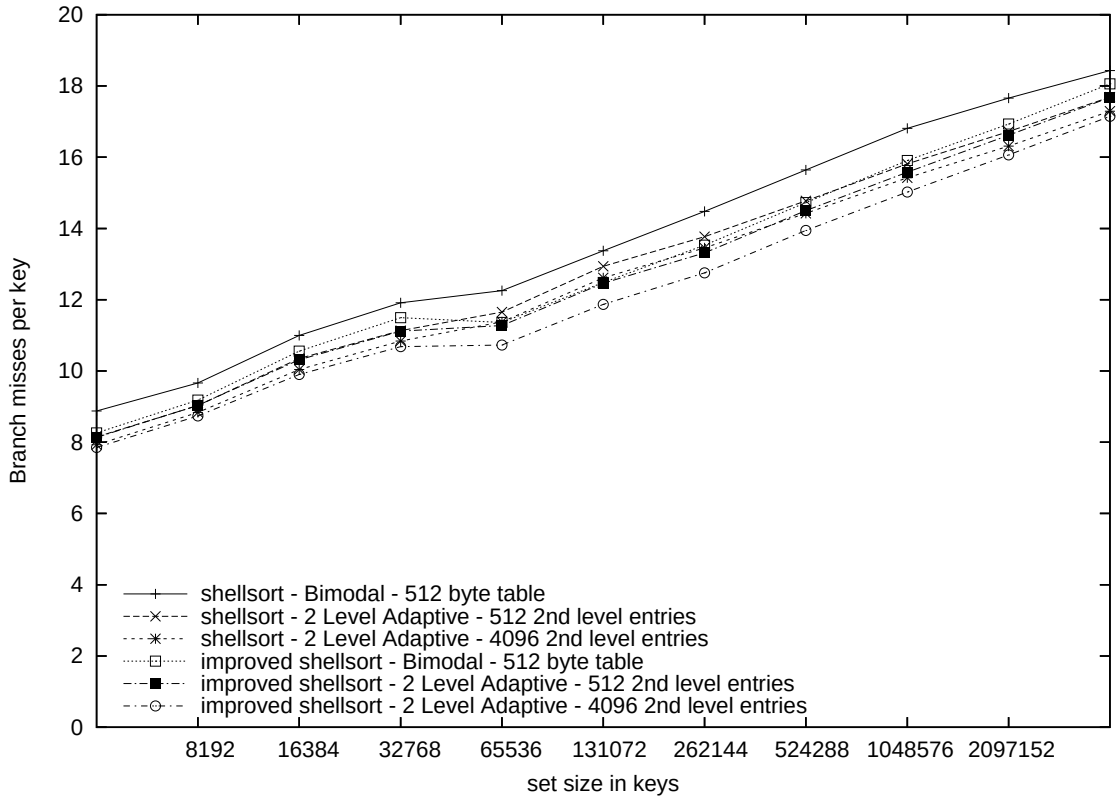


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32-byte cache line.

Figure 9.4: Cache simulation results for shellsort

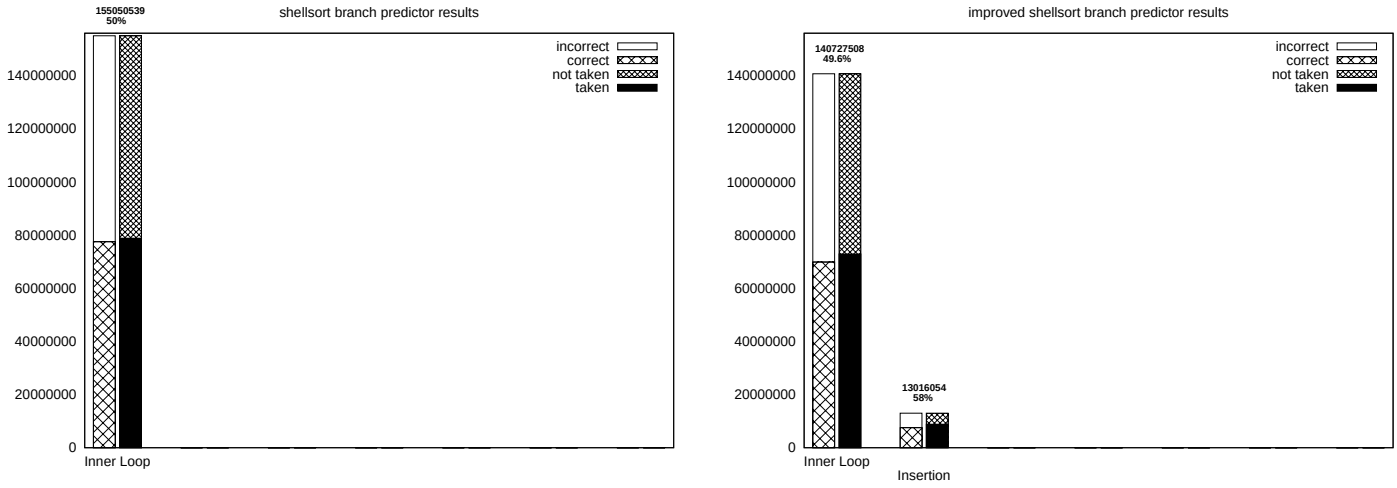


(a) Branches per key - this was simulated using `sim-bpred`.



(b) Branches misses per key - this was simulated using `sim-bpred`, with bimodal and two-level adaptive predictors. The simulated two-level adaptive predictor used a 10-bit branch history register which was XOR-ed with the program counter.

Figure 9.5: Branch simulation results for shellsort



(a) (b)

Figure 9.6: Branch prediction performance for Shellsorts. All graphs use the same scale. Above each column is the total number of branches and the percentage of correctly predicted branches for a simulated bimodal predictor.

lower (18.48 mispredictions per key instead of 19.34, for 4194304 keys).

9.3 Future Work

Only Gonnet's way of producing increments is measured here. Using different increments and different schemes for generating increments would produce different results.

We note that improved shellsort performed in cases where $h = 5$ occurred. Shellsort should also be investigated with this change made.

A cache-conscious shellsort could be created, which would switch the direction of the sweep after each h-sort, to reuse cache lines from the end of the array. This would be especially useful when the array is only slightly larger than the cache, but would have a smaller effect after that.

Chapter 10

Conclusions

This section presents a summary of the results from previous chapters, lists important branch prediction results and discoveries, compares our results with LaMarca's and discusses important cache results, discusses the best sort that we found in a number of categories, and suggests contributions made to computer science by this research.

10.1 Results Summary

10.1.1 Elementary Sorts

We found that insertion sort has the best performance of any of the elementary sorts, with fewer instructions and cache misses than any other elementary sort, and exactly one branch misprediction per key. Selection sort also has very few branch mispredictions, and our results match figures created using Knuth's harmonics formula. We found that bubblesort and shakersort have poor branch prediction performance, and that this cripples their performance, especially compared to selection sort which is otherwise similar in behaviour. We determined that bubblesort has better branch performance at the start and end of its sorting, than in the middle.

We also found that reducing the number of runs performed by shakersort and bubblesort is counterproductive due to additional book-keeping cost. We found that of the elementary sorts, only insertion sort has good cache performance, as the others do not exploit any temporal locality.

10.1.2 Heapsort

The cache optimized heaps perform better than the non-optimized versions. At the cost of a slightly increased instruction count, the number of cache misses can be greatly reduced, and the overall performance of our 8-heap is significantly better than the unoptimized version. This is despite our finding of that increasing the fanout increases branch mispredictions.

Additionally, we created an improvement to heapsort using a 2-heap, which removed the need for a sentinel, without introducing any more instructions. We found that a k-ary heap has similar, though not the same, branch prediction performance as selection sort, in that the fanout becomes more predictable the larger it is.

10.1.3 Mergesort

We found that algorithm S has a lower instruction count than algorithm N, but that this does not translate to better performance. Our results show also that the level 1 cache needs to be taken into account, both in tiling and in alignment; that the extra instructions in multi-mergesort make it slower than tiled mergesort, and that multi-mergesort has regular access patterns during the k-way merge, which can be taken advantage of by a two-level adaptive predictor.

10.1.4 Quicksort

Our results show base quicksort being the fastest of the quicksorts. We note our lowest instruction count occurs with median-of-three partitioning. We also see that using a binary search slows multi-quicksort down compared to a sequential search due to branch mispredictions, but that both have too many extra instructions to compete against memory-tuned quicksort. Despite being slower in our tests, our simulations show memory-tuned quicksort to have fewer cache misses, instructions and branch mispredictions than base quicksort.

10.1.5 Radixsort

We were able to reduce the number of cache misses considerably using a technique adapted from LaMarca's tiled mergesort, and managed to make memory-tuned radixsort the fastest sort in our tests. This is due to its low instruction count, few cache misses and lack of any branch mispredictions. Although we attempted to align it using another technique from mergesort, we found this decreased its performance.

10.1.6 Shellsort

Based on our tests, shellsort performs very similarly to an $O(NlogN)$ sort, outperforming heapsort. We note that our improved version is considerably faster, mostly due to its reduced instruction count. We also measured its branch mispredictions, and determined a formula to accurately calculate them.

The improvement made to shellsort, separating the final loop from the rest, also results in a significantly faster loop, with an extra speed boost from a side-effect skipping an iteration with $h = 2$.

10.2 Branch Prediction Results

To our knowledge, this is the first large-scale, systematic study of branch prediction in sorting algorithms. We discovered some interesting new results.

Among them is the fact that bimodal branch predictors have performed better than two-level adaptive ones in almost every case in this research. This holds across all comparative branches in all the sorts we surveyed, and we believe that where explicit patterns - which can be exploited by two-level adaptive predictors - do not exist, bimodal predictors perform better.

10.2.1 Elementary Sorts

Insertion sort has exactly one miss per key. Using this result we were able to make several other sorts faster by replacing their simple inline sorts with insertion sorts.

Selection sort also has a very low number of branch mispredictions, the results of which can be calculated using Knuth's formula for $H_N - 1$. We found as well that bubblesort and shakersort have incredibly bad branch prediction performance. Both are $O(N^2)$ sorts, and the number of branches and branch misses have the same complexity.

Bubblesort performs similarly to selection sort. It differs in that it partially sorts the array as it moves the smallest key down the array, which selection sort does not do. This has a very large negative effect for branch prediction performance, which is not suffered by selection sort. This occurs in the middle of the sort; at the start, it performs like selection sort, being randomly arranged; at the end, it is largely sorted and is predictable again.

10.2.2 Heapsort

In heapsort, we found a pattern of branch mispredictions when using a fanout in our 8-heap, and that this pattern matched closely that of selection sort, discussed above. We found that the number of mispredictions was close to, but measurably higher than, $H_N - 1$. The main reason was that the number of items from which the selection was being made was small. Although the comparisons are mostly biased in one direction, the early ones are sufficiently unpredictable that the branch predictor often predicts the less likely direction. The later comparisons are highly predictable, and yield prediction accuracies much closer to that indicated by $H_N - 1$.

The result of this is that the number of mispredictions decreases depending on how much the heap fans out. Despite this property, the number of mispredictions using an 8-heap was greater than when using a 4-heap, due to the increased total number of branches.

10.2.3 Mergesort

Using a k-way merge, multi-mergesort had a great number of extra branch predictions. While many of these were mispredicted using the bimodal predictor, the two-level adaptive predictor was able to predict a higher proportion correctly.

10.2.4 Quicksort

Sedgewick proposed replacing individual insertion sorts in quicksort with one elegant sweep at the end. LaMarca proposed replacing this with the previous large number of insertion sorts in order to reduce cache misses. We note that attempting to move the pivots, as Sedgewick does, actually increases both the number of branches and the rate of mispredictions. We conclude that LaMarca's improvement, which slightly improves quicksorts cache performance, also slightly improves it's branch prediction performance.

However, LaMarca's move to multi-quicksort, while not affecting the rate of branch mispredictions in the partitioning phase, results in a large increase in branch mispredictions due to the addition of a binary search. We determine that while this reduces the instruction count and number of memory accesses, the increase in actual running time due to branch mispredictions is significant.

We also analysed the effects of different medians on quicksort. We determine that not using a median to choose a pivot makes the branches more predictable than they are biased, though at a cost of a large increase in instructions (and also significantly greater possibility of a worst-case $O(N^2)$ sort). We also saw that the more keys considered when determining the median, the lower the number of branches. However, this results in an increase not only in the rate of branch mispredictions, but also in the number of actual branch misprediction. We determine that reducing entropy is not the best way of making quicksort perform faster, and that using a median-of-three actually achieves the higher performance.

10.2.5 Radixsort

As predicted, radixsort is highly predictable, having no comparison mispredictions at all. This result, combined with its linear complexity, and its improved cache performance, mean that it is the fastest of any of the sorts presented here.

10.2.6 Shellsort

We are able to accurately predict the number of branch mispredictions in shellsort, using our results from insertion sort.

10.3 Cache Results and Comparison with LaMarca's Results

In several cases, we found that direct-mapped caches performed better than fully-associative ones, resulting in fewer misses. This was due to their replacement policy, and that the sorts in question were heavily optimized for the direct-mapped caches. None of the changes slowed the fully-associative caches however.

Over the course of this research, the vast majority of our results have correlated exactly with LaMarca's. Many of our graphs are exactly the same shape to LaMarca's, can be calculated exactly or almost exactly using his methods, or can be seen to be equivalent when scaled by an appropriate ratio.

10.3.1 Heapsort

LaMarca's heapsort graphs were equivalent to ours in the areas of level 1 misses, instruction count and cycle count. Like LaMarca, we found that a 4-heap lowered our instruction count, and we found better results by using an 8-heap instead, as an 8-heap filled our cache lines exactly. Our level 2 misses also concurred with LaMarca, with only one difference: memory-tuned heapsort, being an out-of-place sort, fills the cache fully with a smaller number of keys than base heapsort, which is in-place. LaMarca's results do not show this. Nor do they show a difference in cache misses between these two sorts when the array does fit in the cache. LaMarca's base heapsort has the characteristics of an out-of-place heapsort. However, our results show that his implicit heap improvements have led to a significantly faster heapsort.

10.3.2 Mergesort

LaMarca's initial improvements to base mergesort were expected to reduce the instruction count. We noted such a reduction with each of the steps he proscribed. We note too that each of his improvement reduce level 2 cache misses, and the shape of our level 2 cache miss graphs exactly match his. Our graphs also have similar characteristics: LaMarca mentions a wobble in the instruction count graphs, due to having to copy the final merge back; we note this same wobble.

However, we disagree with his results in several key places. Firstly, we note that an inlined sorting method does not perform as fast as insertion sort. We find it difficult to perform his improvements on either algorithm N or algorithm S, and write our own mergesort instead. We expected that mergesort would have half the instruction count of heapsort; we found instead they were very similar.

LaMarca did not include details of his tagging system in multi-mergesort. Instead we used a search, which greatly increased the instruction count. As a result, multi-mergesort did not perform as well as the mergesorts which had only been tiled.

We note also that aligning the level 2 cache alone leads to worse performance than not making any improvements at all, due to the thrashing of the level 1 cache. When we aligned the arrays for the level 1 cache, the results were much closer to LaMarca's results, and showed what an improvement his techniques brought to those algorithms.

In addition to the double alignment, we also added two new improvements. The mergesorts were tiled for the level 1 caches as well as the level 2, slightly reducing the number of level 1 cache misses, and we removed the need for a step to copy the array back from the auxiliary array; instead, we presort a step further in cases where this occurs, and find the merge naturally ends in the correct destination.

10.3.3 Quicksort

Our quicksort results, again, we very similar to LaMarca's. We showed an increase is instruction count of 25% due to multi-quicksort - very close to LaMarca's 20%. The shape of our instruction count graphs were the same, and switching to a binary search in multi-quicksort reduced the instruction count. Like LaMarca, we found that the base quicksort had very few cache misses, and that not using Sedgewick's insertion sort improvement reduces cache misses. We found that multi-quicksort's k-way sort changed the slope of the graph, but our result were not conclusive as to whether this would eventually make multi-quicksort faster than the other versions. Our results do show it being slower for all our test cases,

due to the increase in instruction count.

LaMarca predicts two cache misses per key, which translated to one miss per key with our 32-bit integers. We found this calculation to be true, as is his calculation of a decrease of one cache miss in four: we note a reduction of one cache miss in eight.

Where LaMarca found that using Sedgewick's insertion sort optimization decreased instruction count, we found the opposite. We also found that despite improvements due to cache performance and instruction count, our base quicksort performed faster than our memory-tuned version. Finally, we found that although changing multi-mergesort's sequential search to a binary search decreased the instruction count, it greatly increased the number of branch mispredictions incurred, and slowed the sort down accordingly.

10.3.4 Radixsort

LaMarca determines that radixsort could not be improved in the same was as the comparison-based sorts, and that due to poor cache performance, it ran more slowly than quicksort. We showed, however, that one of the improvements applied to mergesort could be applied to it, and that its resulting cache performance is sufficiently good that - partially due to its low instruction count and lack of branch mispredictions - it is the fastest sort in our survey.

10.4 Best Sort

The results also helped answer the question of which is the best sort. However, it is important to remember that sorts have different purposes and properties, so this shall be considered. A comparative chart of the fastest sorts is shown in Figure 10.1 on the following page. This shows the cycles per key of each of the major sorts and their derivatives, measured using Pentium 4 performance counters. None of the elementary sorts or heapsorts are on the chart, as their results are significantly worse than the other results, which makes it difficult to accurately see them.

The best stable sort is memory-tuned radixsort. The best comparison-based stable sort is double-aligned tiled mergesort. The best in-place sort is base quicksort. The best in-place stable sort we examine in this report is insertion sort. The best elementary sort is also insertion sort; as such it should be only used for very small data sets. The best sort for very large data sets is memory-tuned radixsort. The fastest overall sort was memory-tuned radixsort, though the base and memory-tuned quicksorts are all very close.

10.5 Contributions

This section lists the contributions of this report. Each of these is novel and was discovered or developed during the course of this research.

The first main contribution of this report is that it repeats the work of LaMarca on cache-conscious sorting, and validates many of his claims, while questioning others. The other main contribution is having performed a branch prediction analysis of major sorts - which to our knowledge has not been done before - and developing a framework by which these results can be repeated and expanded upon.

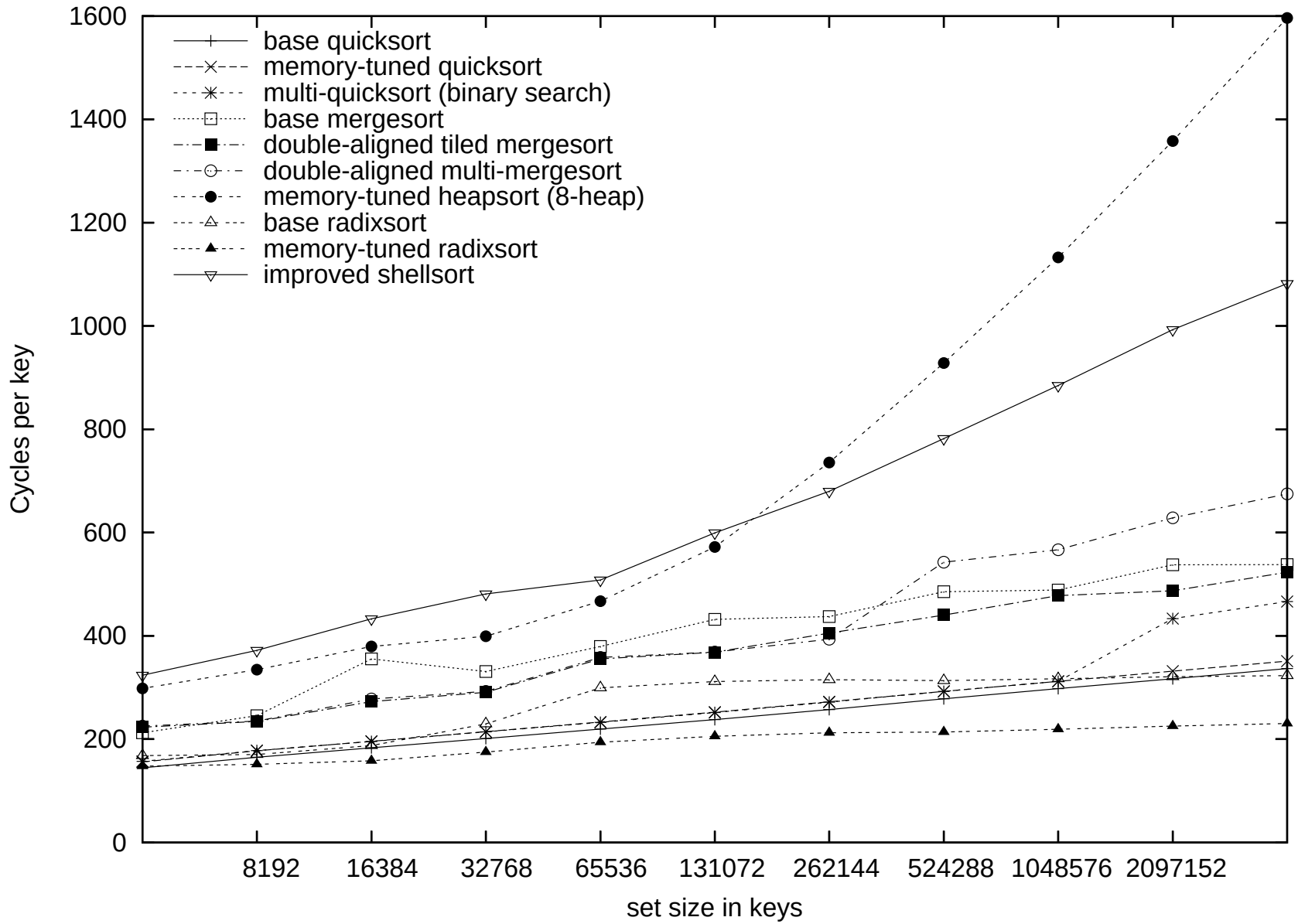


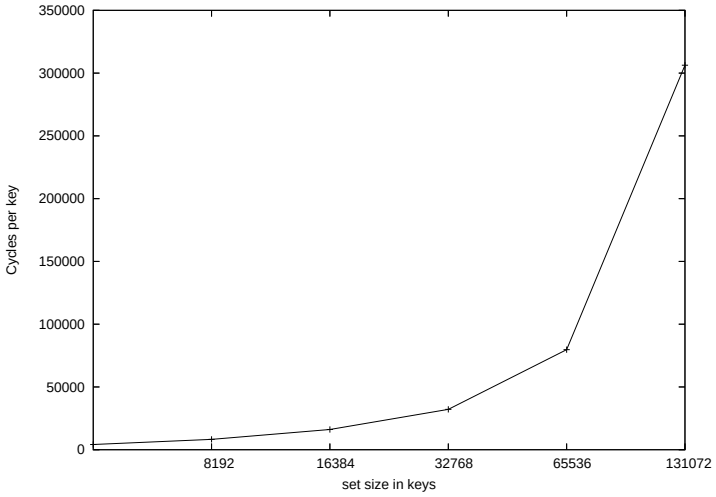
Figure 10.1: Cycles per key of several major sorts and their variations - this was measured on a Pentium 4 using hardware performance counters.

Several discoveries were made and analysed: insertion sort has one misprediction per key; selection sort has $O(H_N - 1)$ branch mispredictions per key, which applies to long arrays, though short arrays become more predictable the longer they get, as in heapsort; tiling mergesort can be done in two steps, reducing the cache miss count; aligning mergesort for a level 2 cache alone is counter-productive - aligning it for level 1 and level 2 is significantly faster; it is possible to avoid using a sentinel and still remove the bounds check from a standard 2-heap, simply by reordering the steps involved; sequential searches are far cheaper than binary searches across small lists, due to a low branch misprediction rate. Bubblesort has terrible branch performance in the middle, due to being partially sorted - it performs much worse than selection sort as a result; multi-mergesort contains regular access patterns which can be exploited by a two-level adaptive branch predictor; in almost all other cases, two-level branch predictors are measurably less accurate than simple bimodal predictors at predicting branches in sorting algorithms. A median-of-three quicksort outperforms other medians; radixsort has no branch mispredictions; shellsort has easily calculable branch mispredictions, and performs very similarly to an $O(NlogN)$ sort.

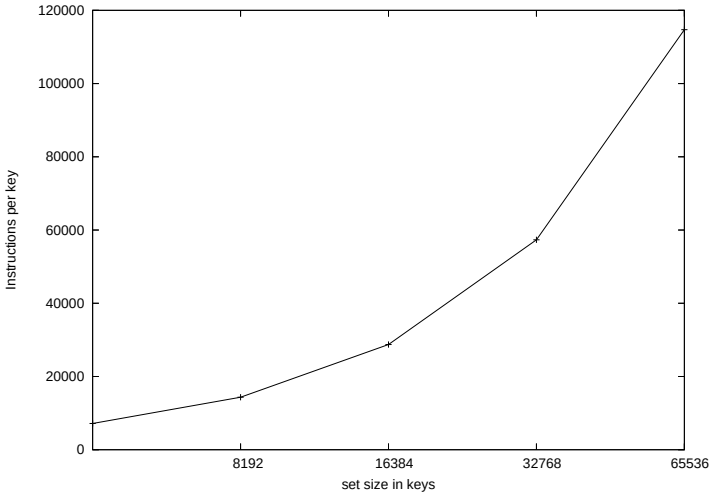
In addition, we provide an analysis of most major sorts currently used, a comparison of $O(NlogN)$ sorts, of different medians for quicksort, of $O(N^2)$ sorts and of cache-conscious sorts. We present branch prediction performance figures for most major sorts, and adapt cache-conscious strategies from mergesort to not only significantly increase the performance of mergesort, but also to make memory-tuned radixsort the fastest performing sort in our tests.

Appendix A

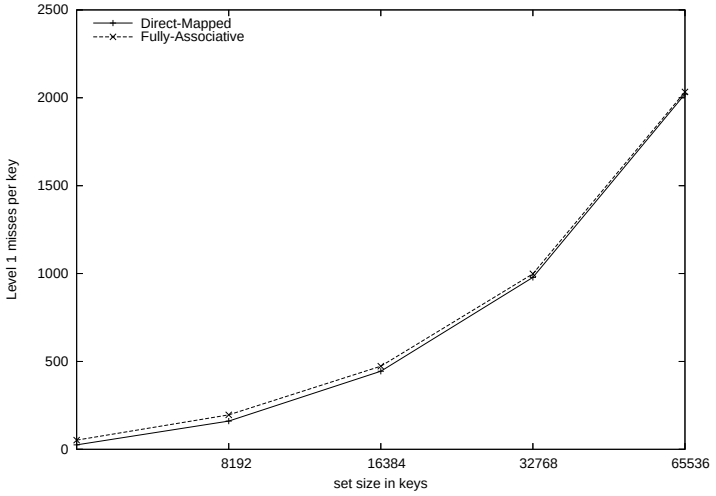
Simulation Result Listing



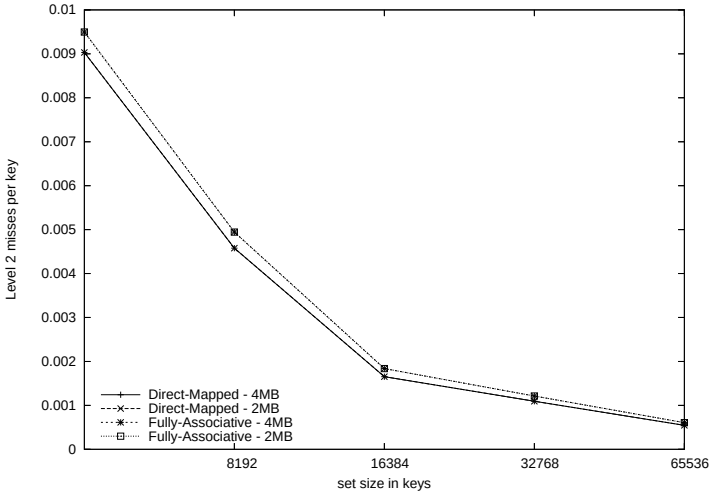
(a) Cycles per key



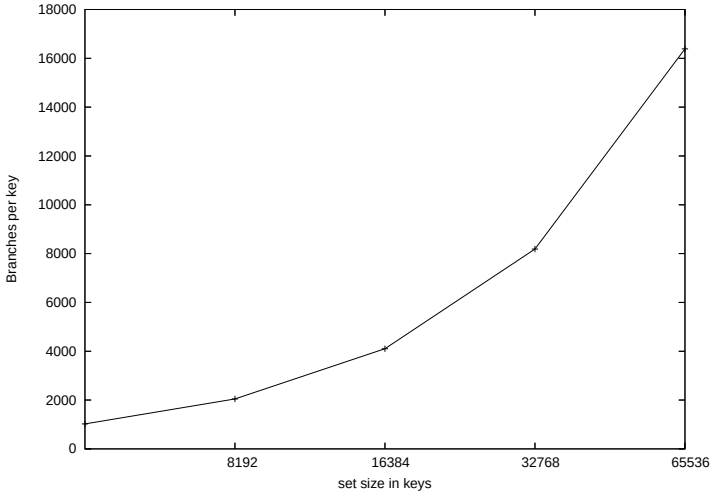
(b) Instructions per key



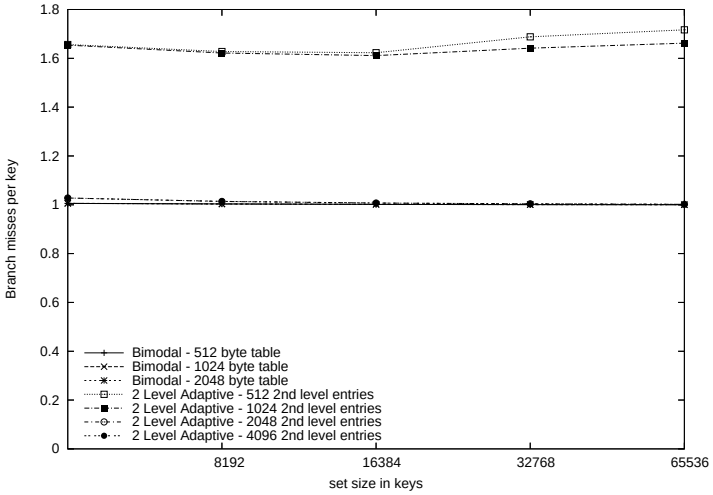
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

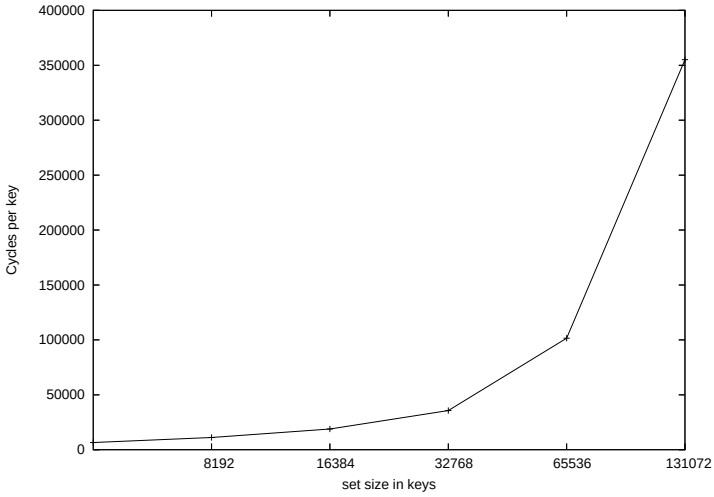


(e) Branches per key

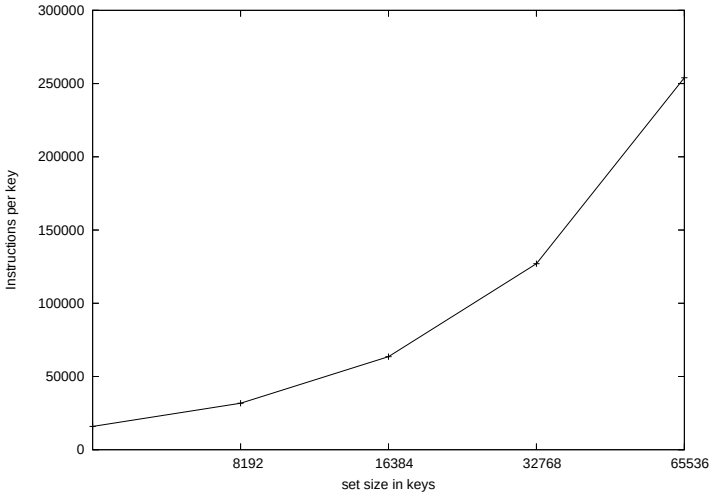


(f) Branch misses per key

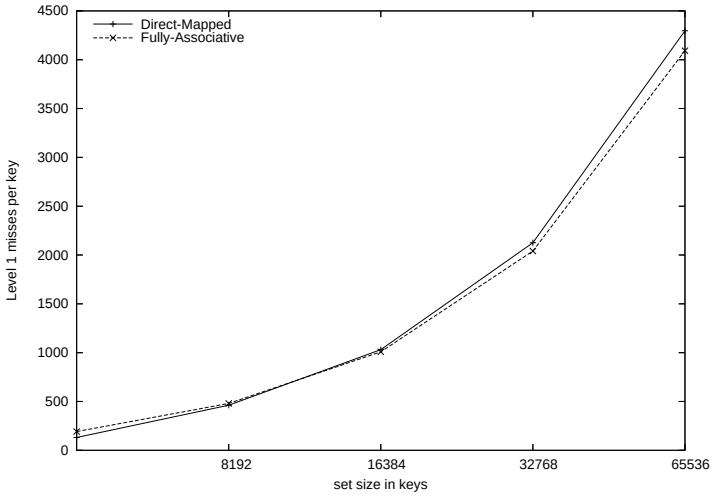
Figure A.1: Simulation results for insertion sort



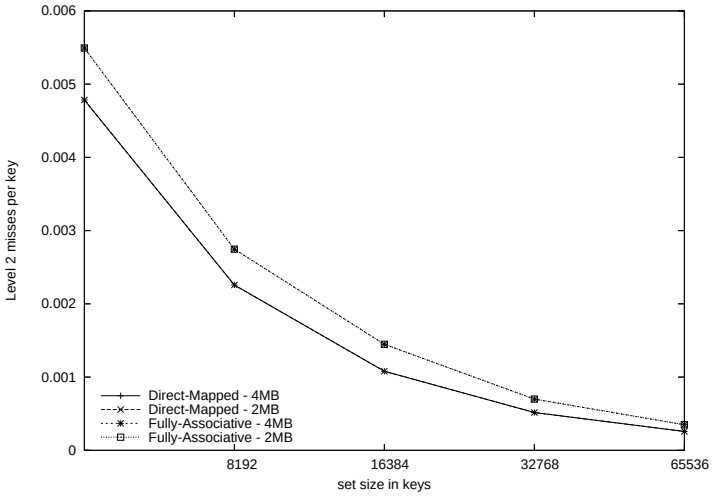
(a) Cycles per key



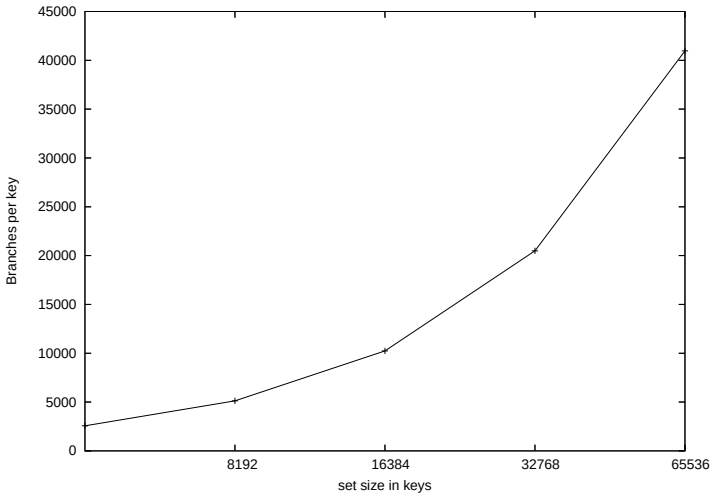
(b) Instructions per key



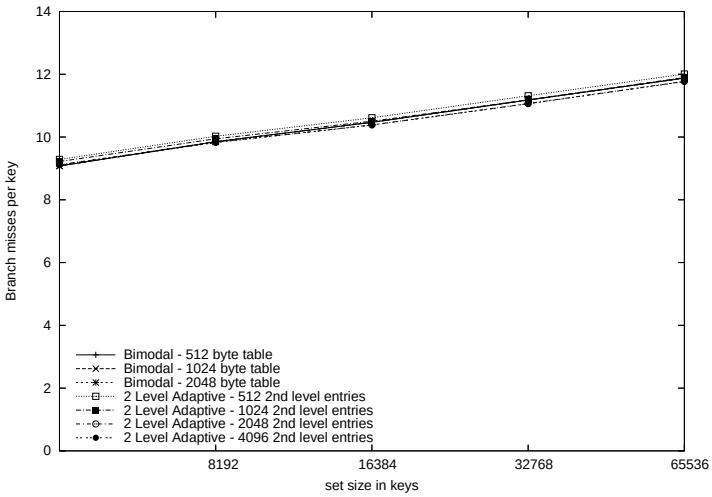
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

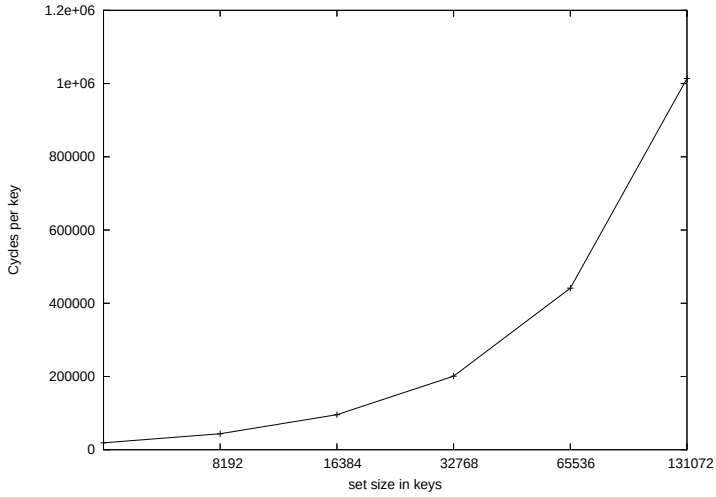


(e) Branches per key

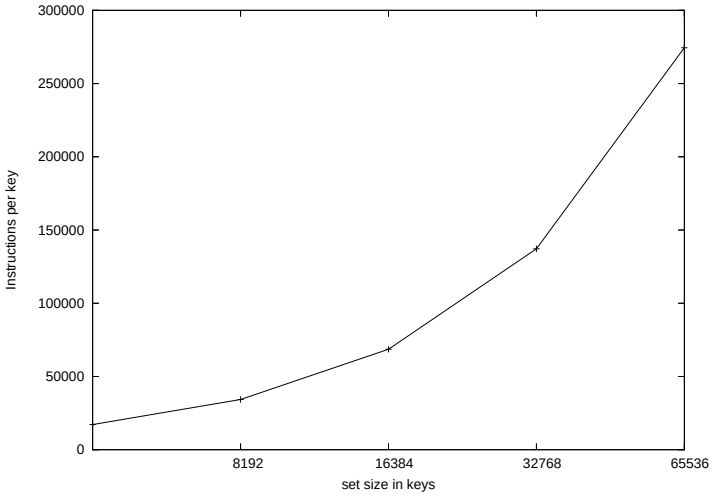


(f) Branch misses per key

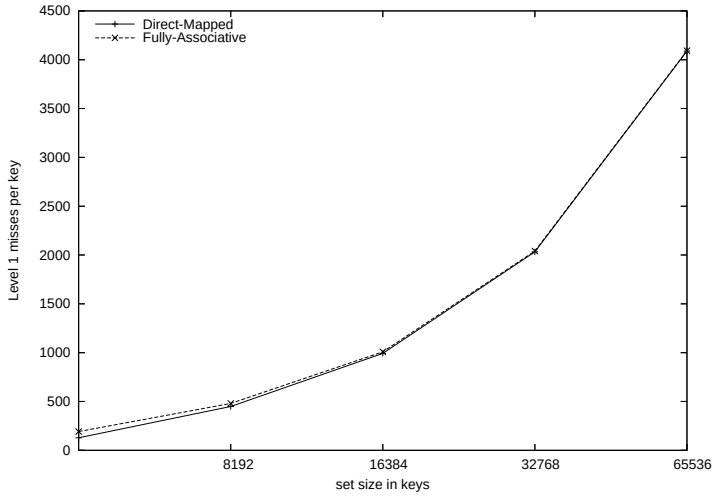
Figure A.2: Simulation results for selection sort



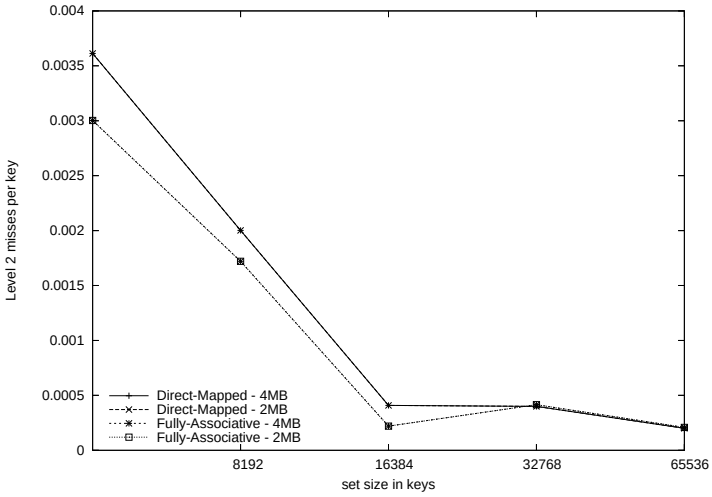
(a) Cycles per key



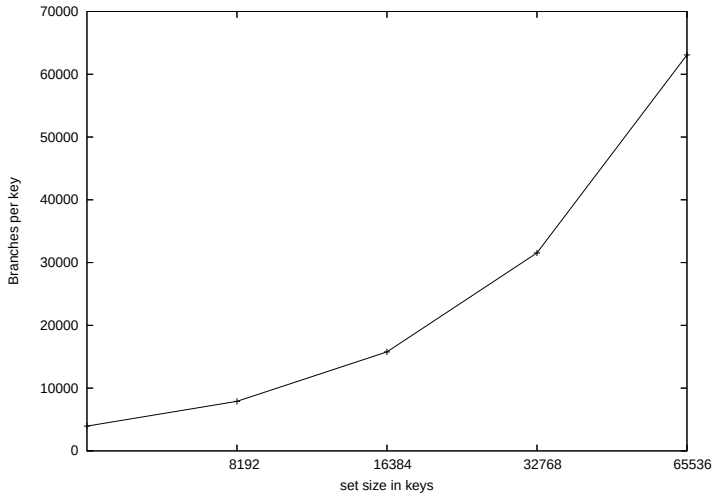
(b) Instructions per key



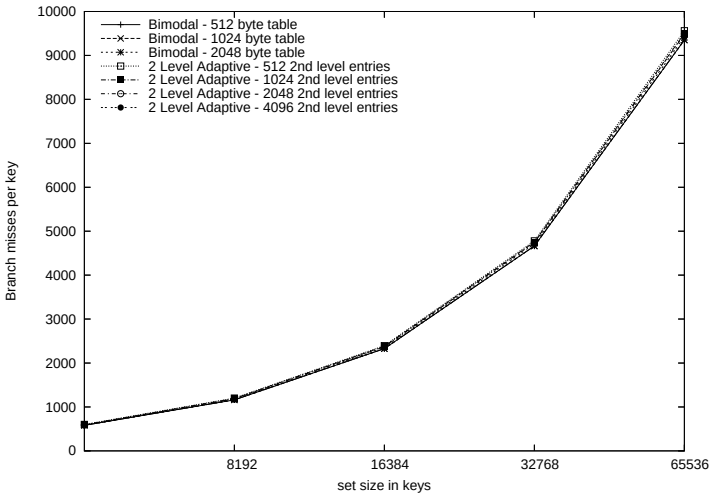
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

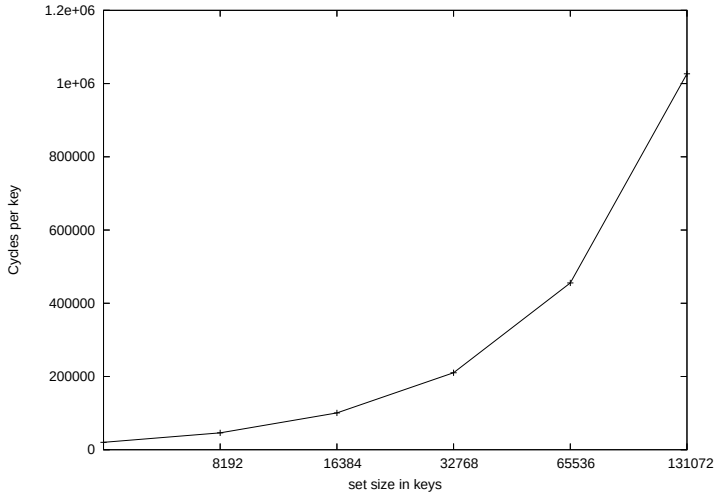


(e) Branches per key

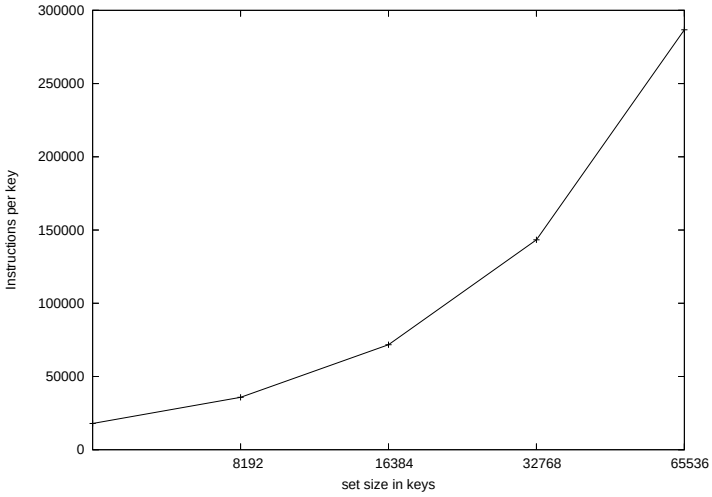


(f) Branch misses per key

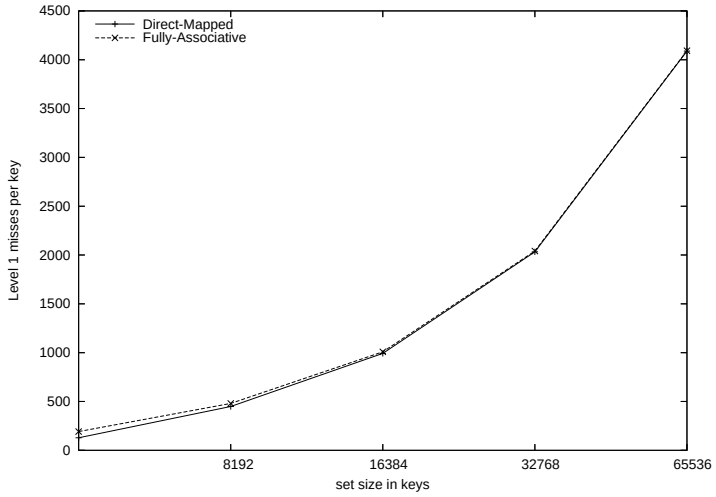
Figure A.3: Simulation results for bubblesort



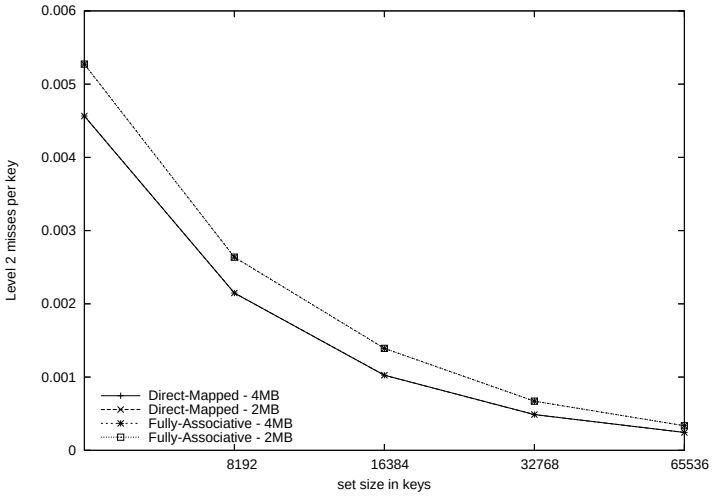
(a) Cycles per key



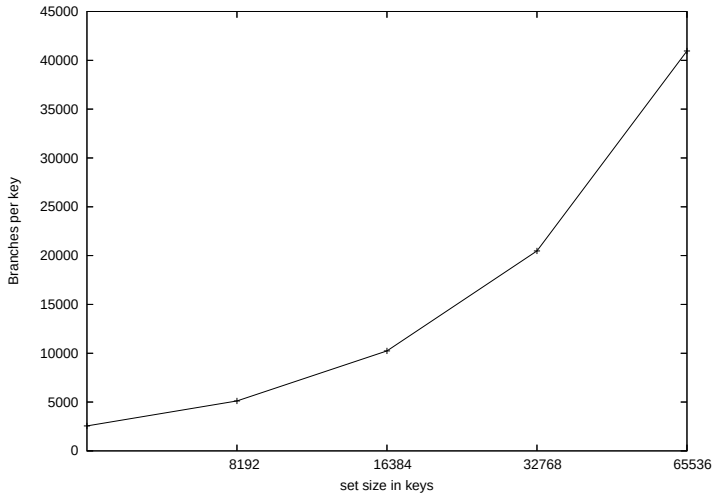
(b) Instructions per key



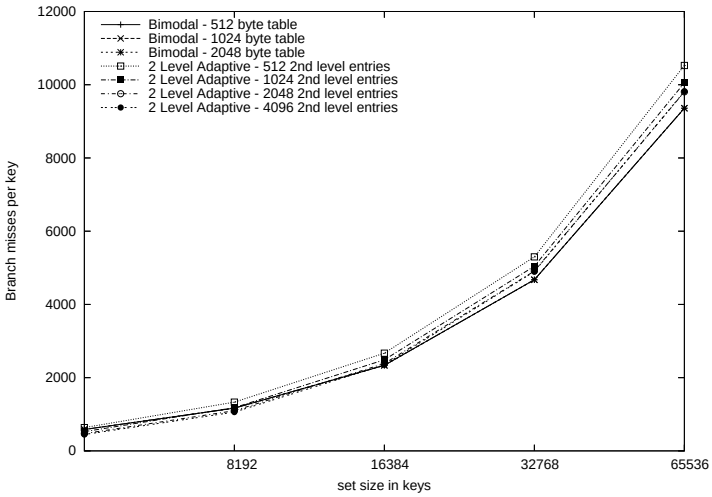
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

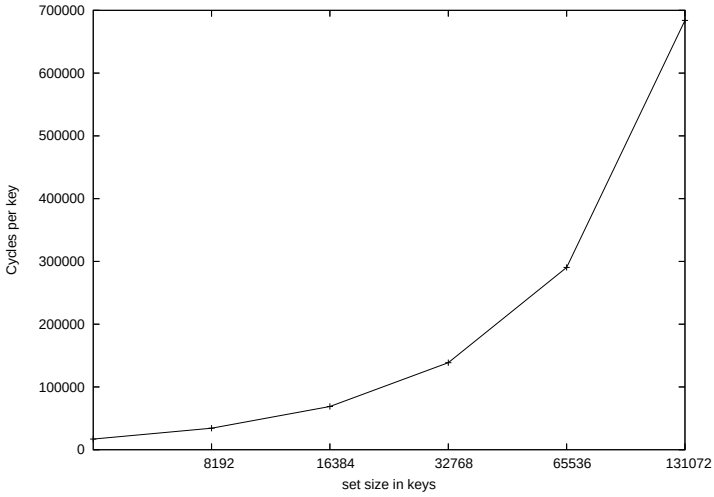


(e) Branches per key

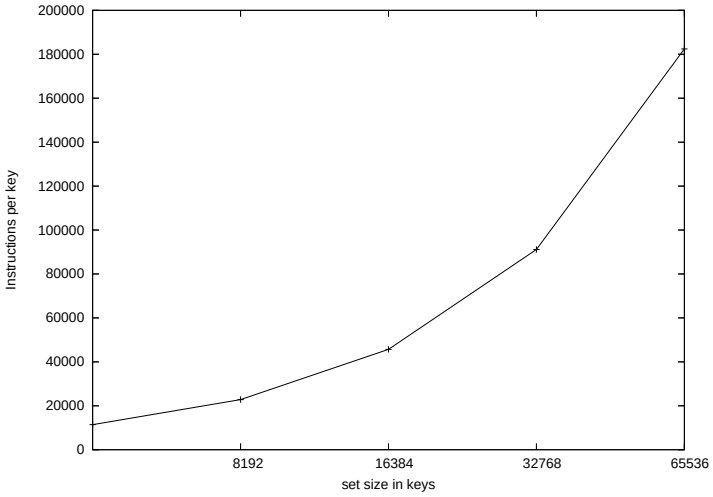


(f) Branch misses per key

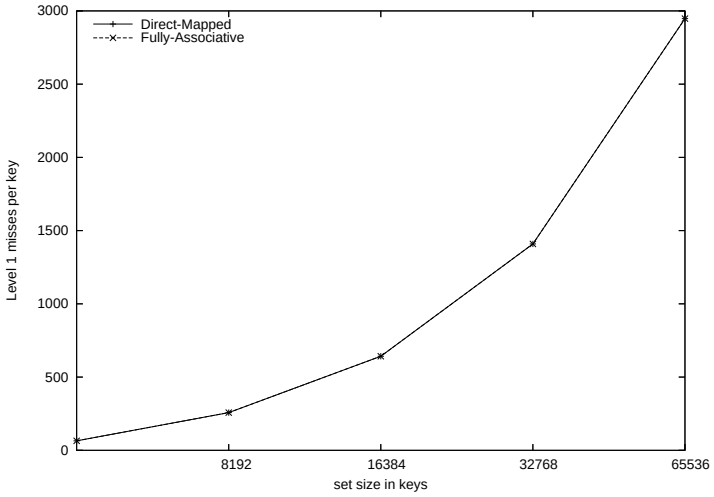
Figure A.4: Simulation results for improved bubblesort



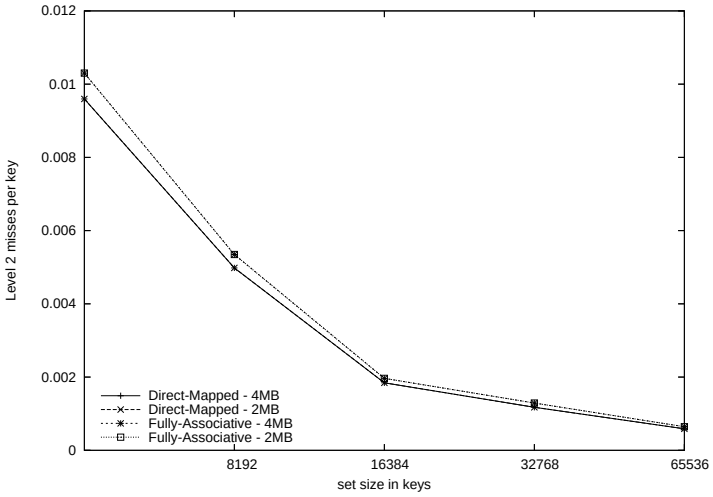
(a) Cycles per key



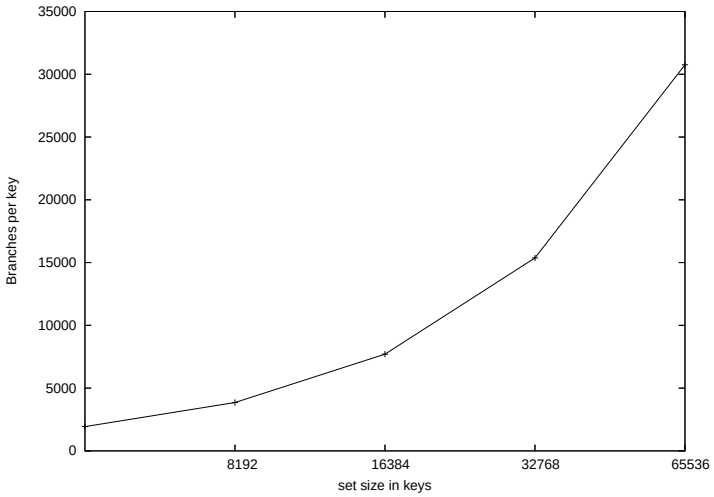
(b) Instructions per key



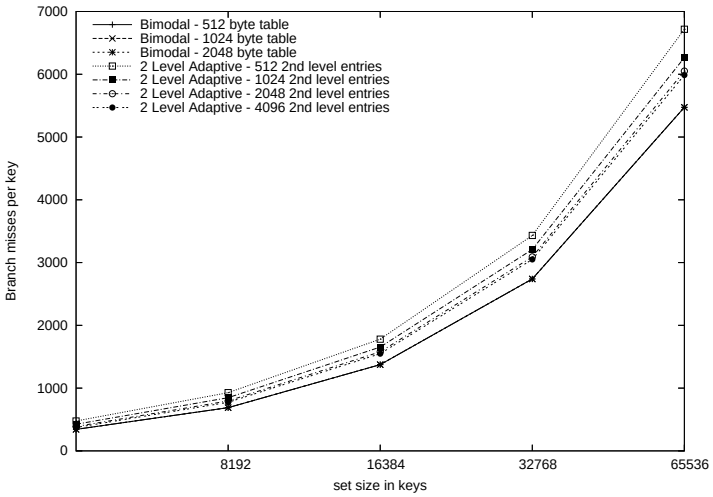
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

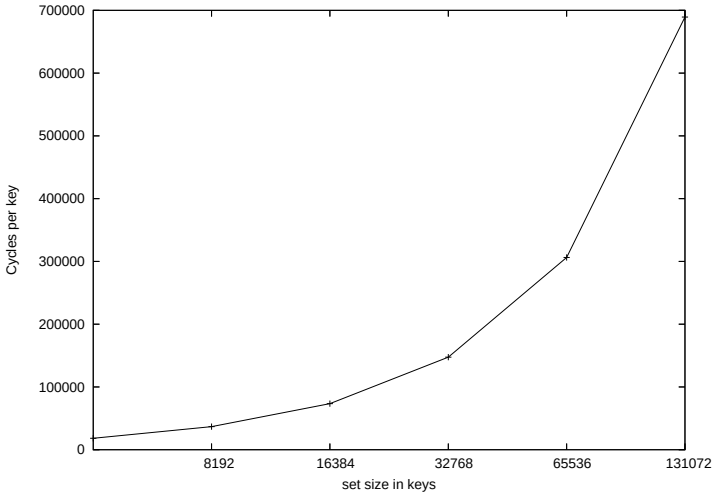


(e) Branches per key

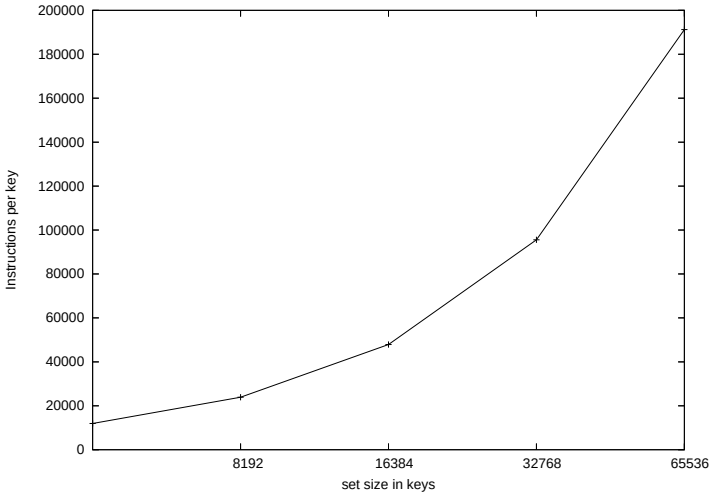


(f) Branch misses per key

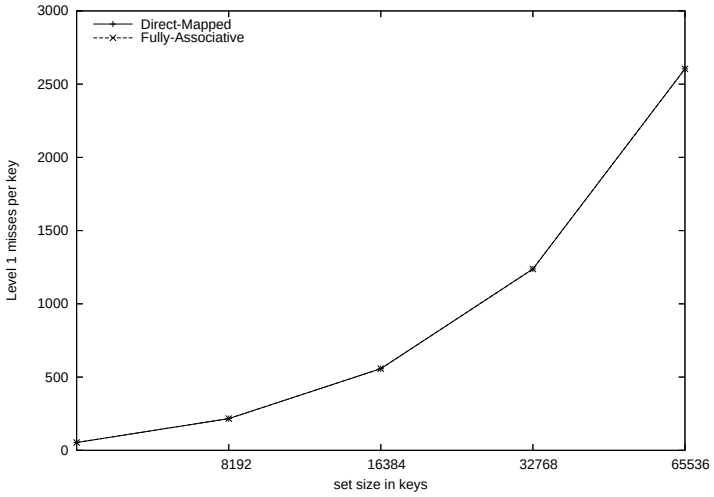
Figure A.5: Simulation results for shakersort



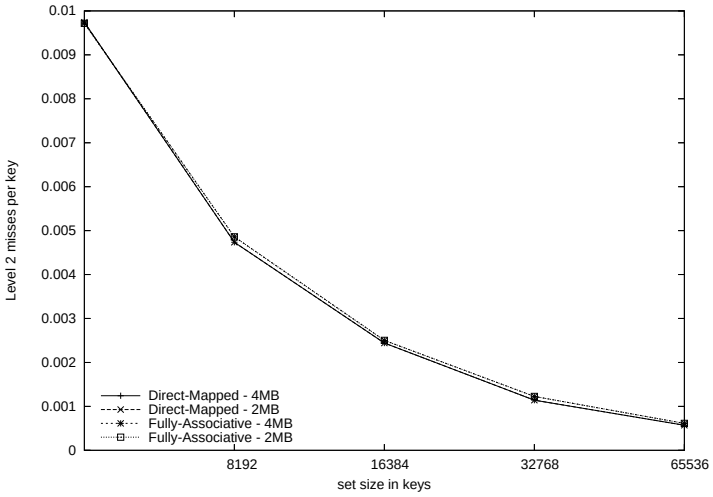
(a) Cycles per key



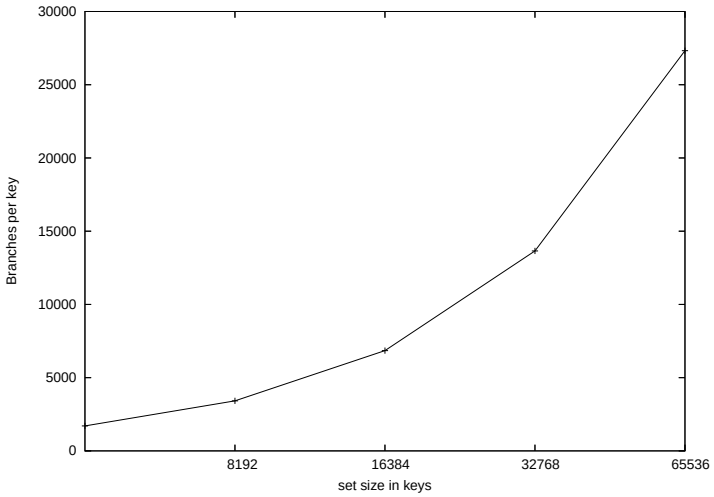
(b) Instructions per key



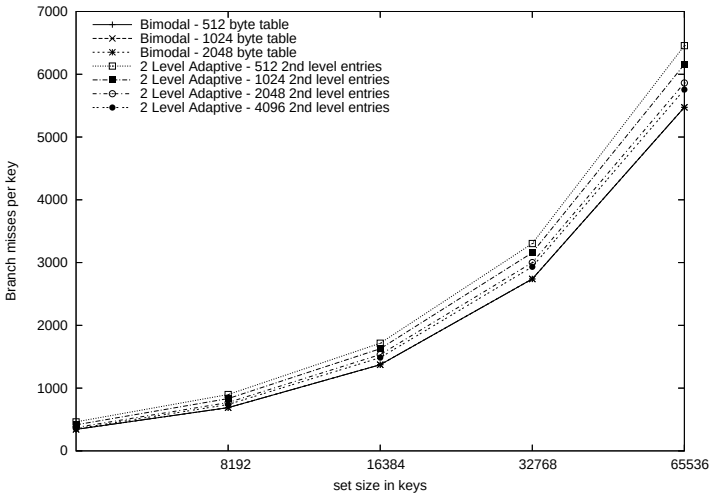
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

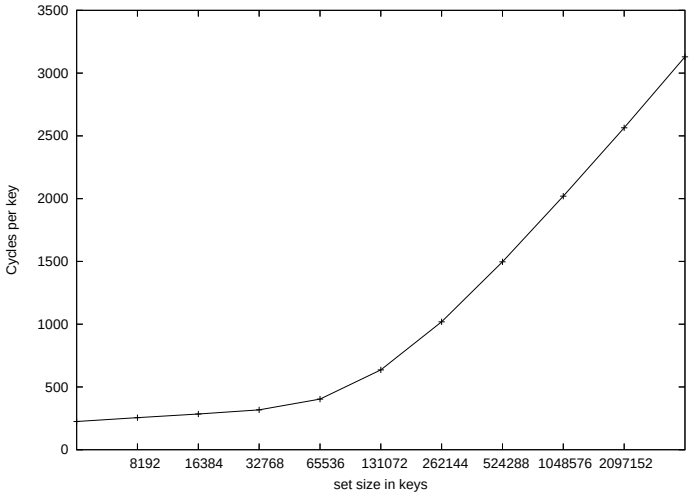


(e) Branches per key

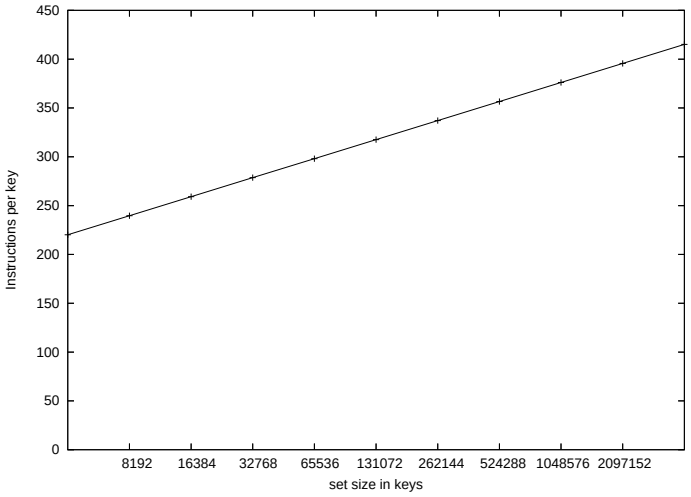


(f) Branch misses per key

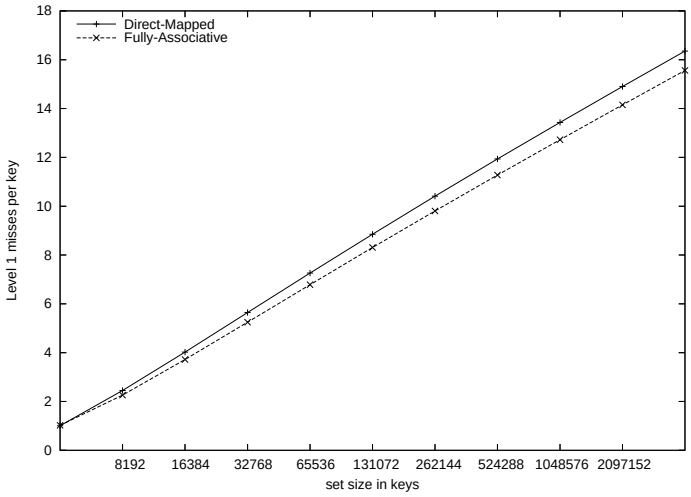
Figure A.6: Simulation results for improved shakersort



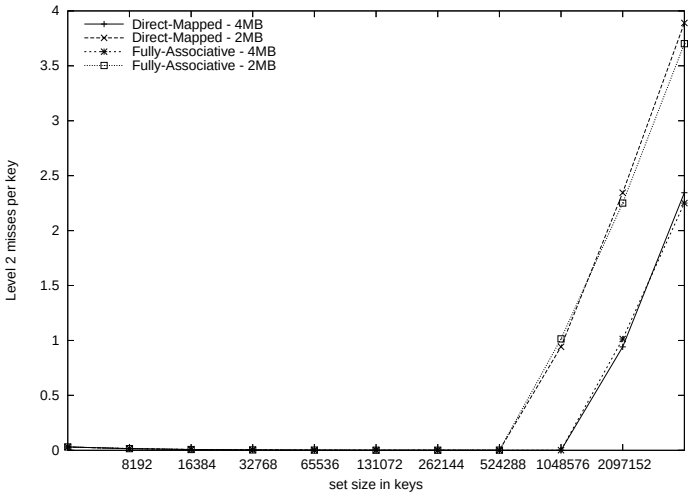
(a) Cycles per key



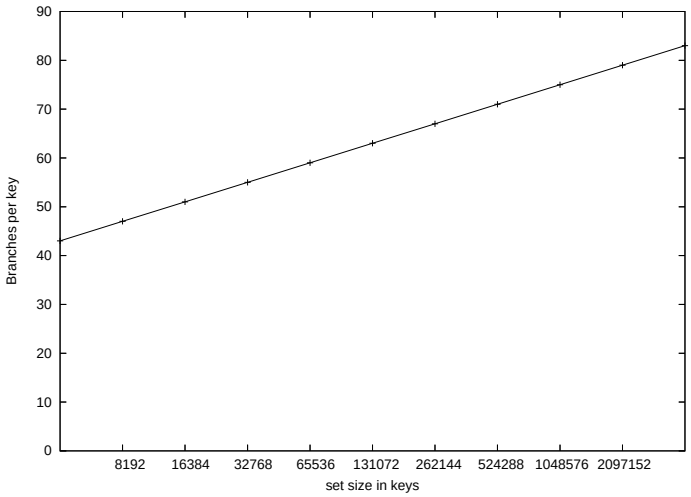
(b) Instructions per key



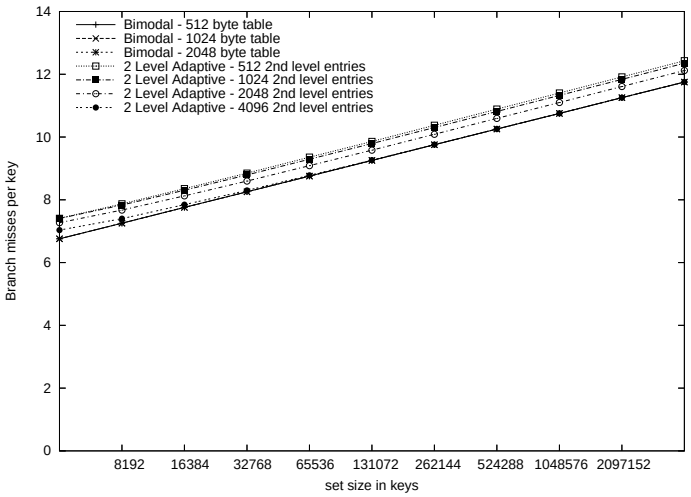
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

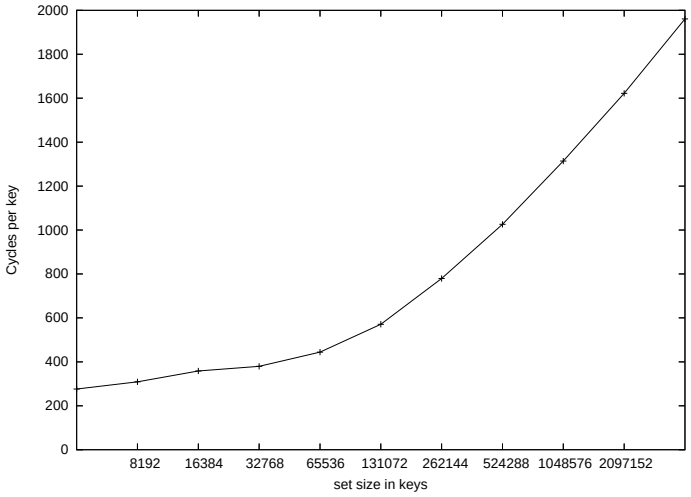


(e) Branches per key

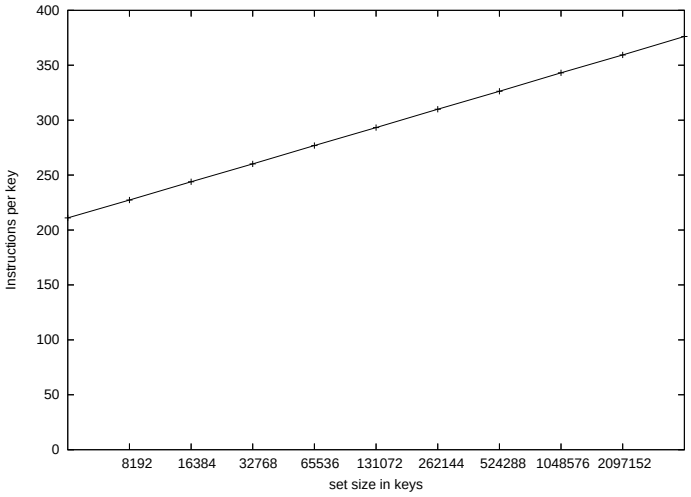


(f) Branch misses per key

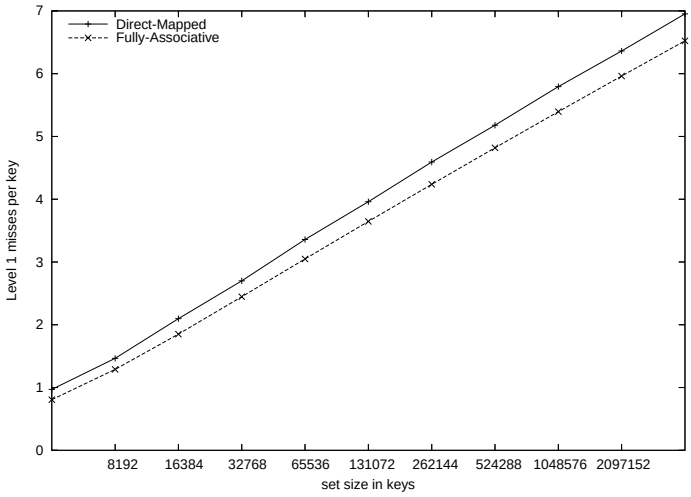
Figure A.7: Simulation results for base heapsort



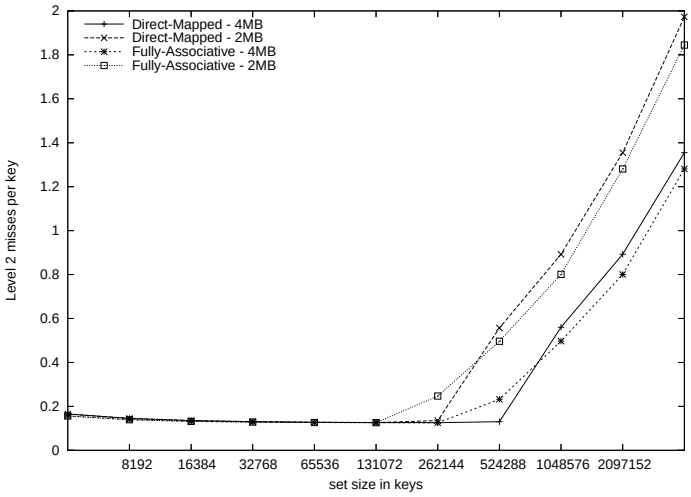
(a) Cycles per key



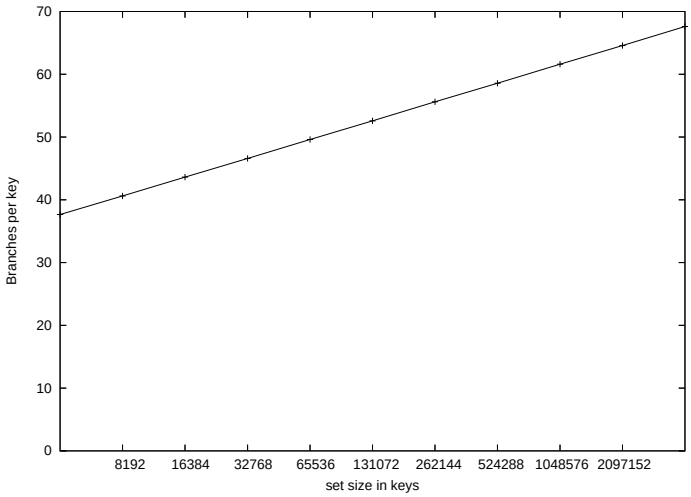
(b) Instructions per key



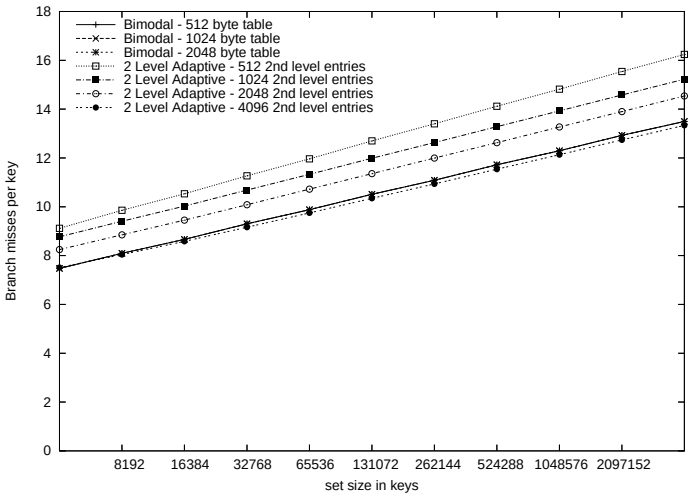
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

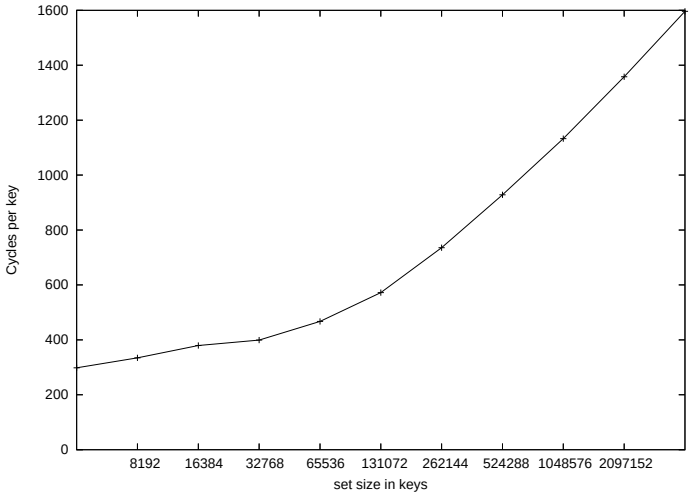


(e) Branches per key

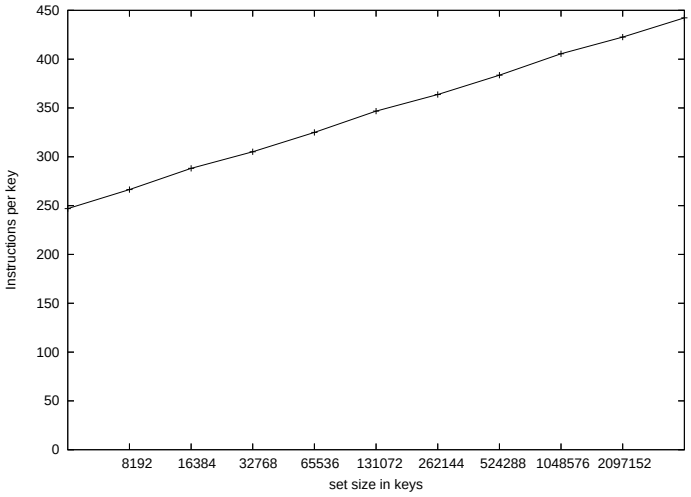


(f) Branch misses per key

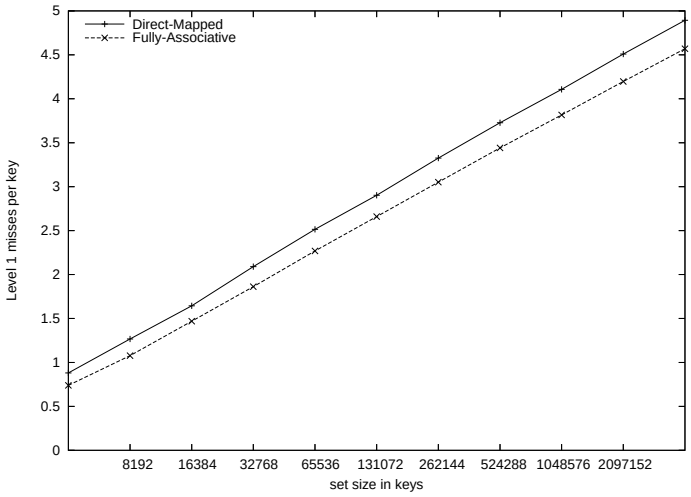
Figure A.8: Simulation results for memory-tuned heapsort (4-heap)



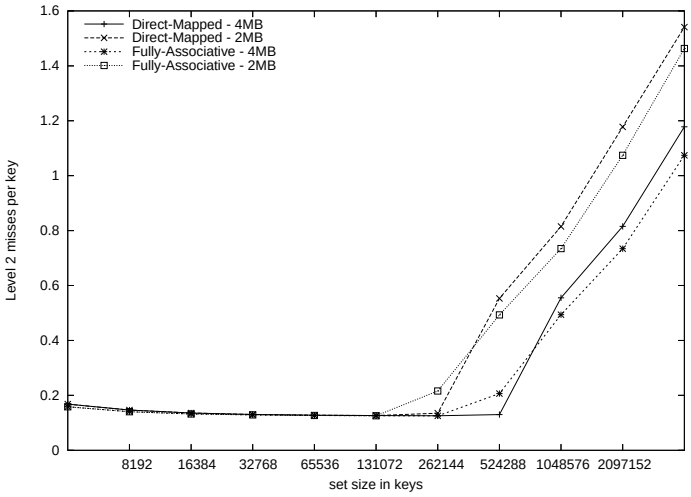
(a) Cycles per key



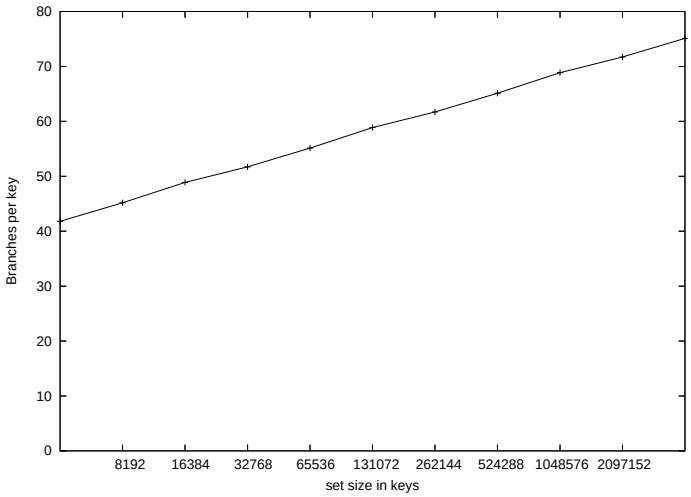
(b) Instructions per key



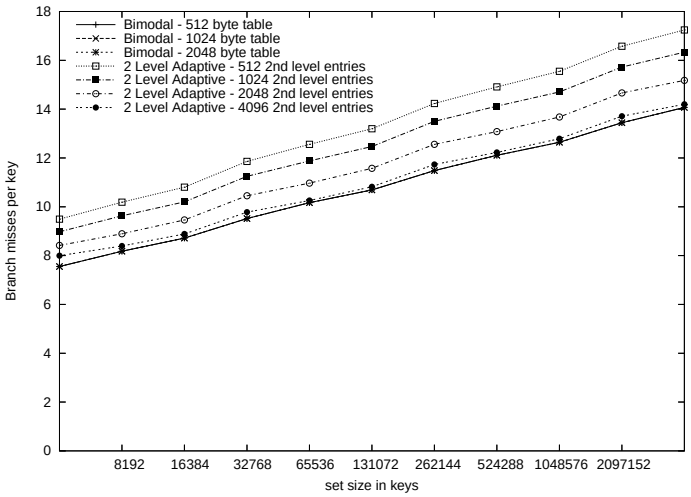
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

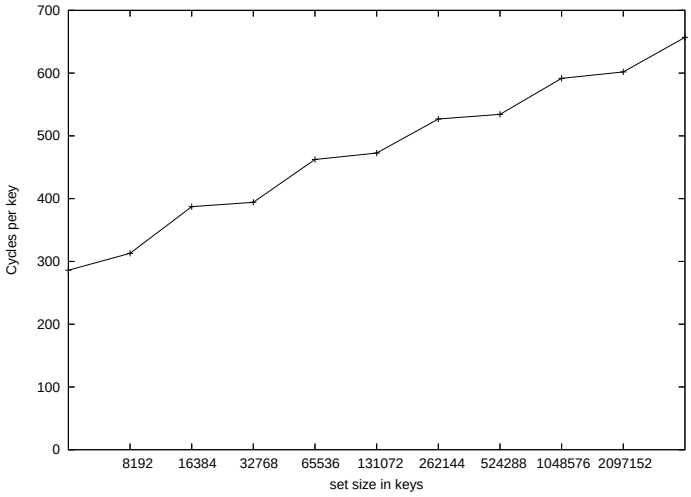


(e) Branches per key

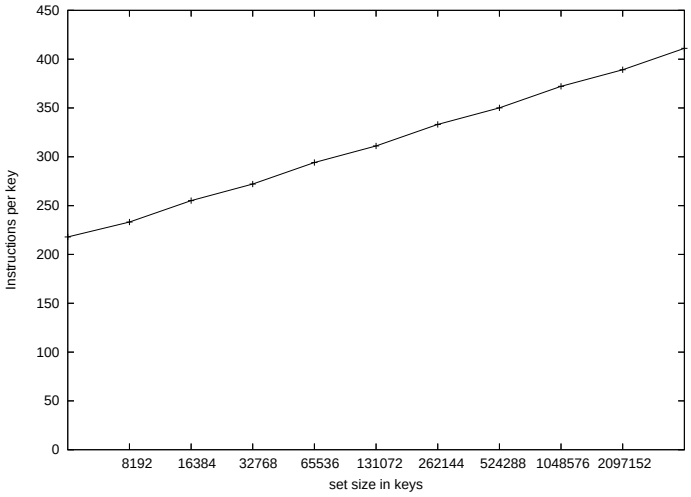


(f) Branch misses per key

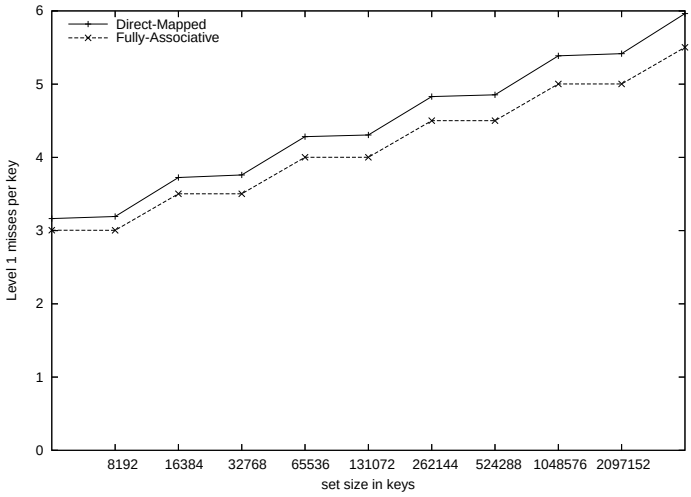
Figure A.9: Simulation results for memory-tuned heapsort (8-heap)



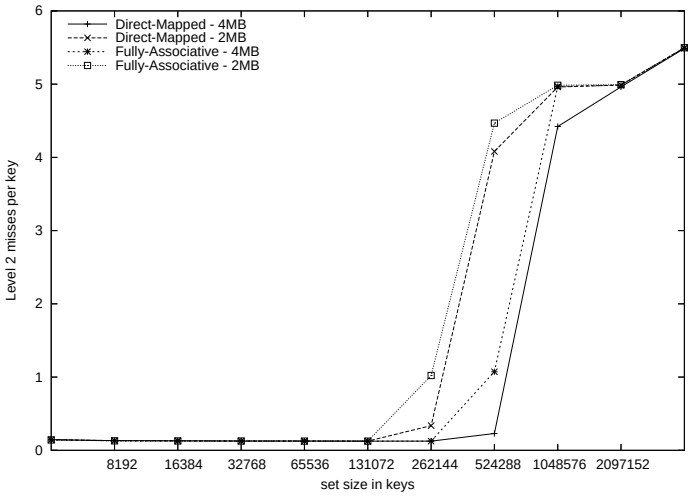
(a) Cycles per key



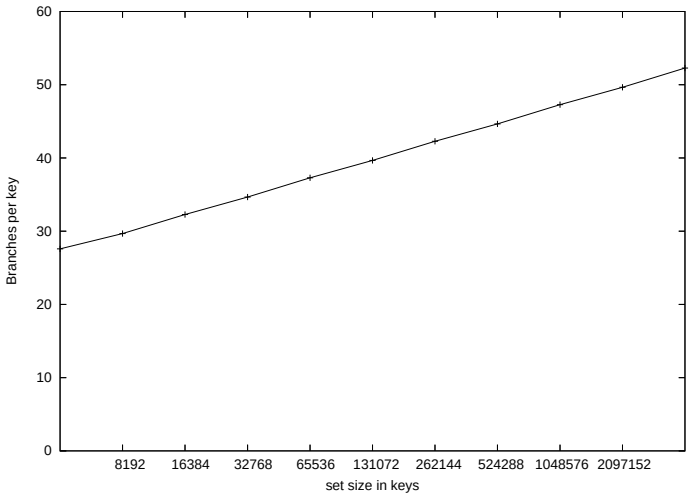
(b) Instructions per key



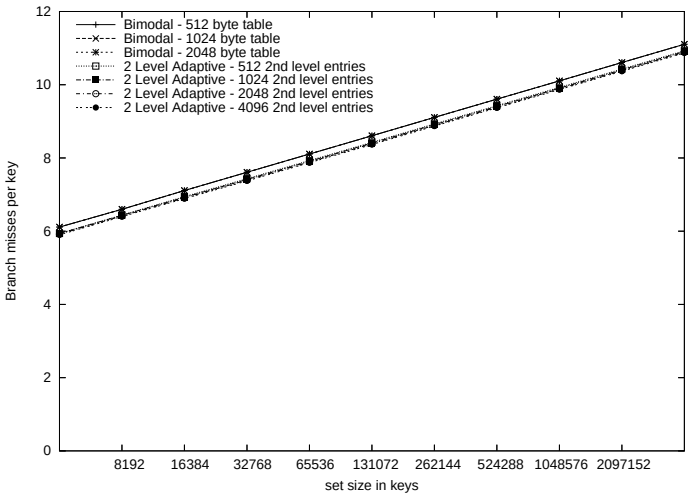
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

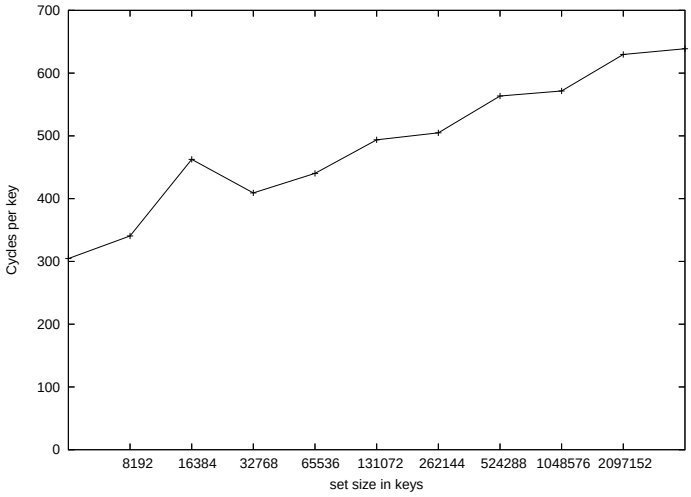


(e) Branches per key

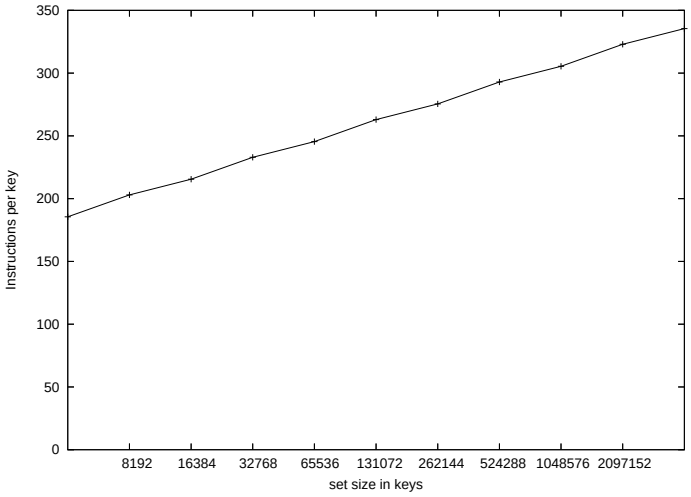


(f) Branch misses per key

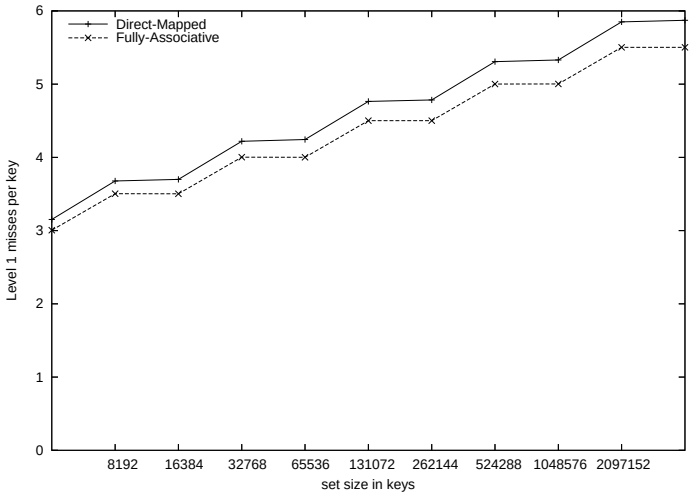
Figure A.10: Simulation results for algorithm N



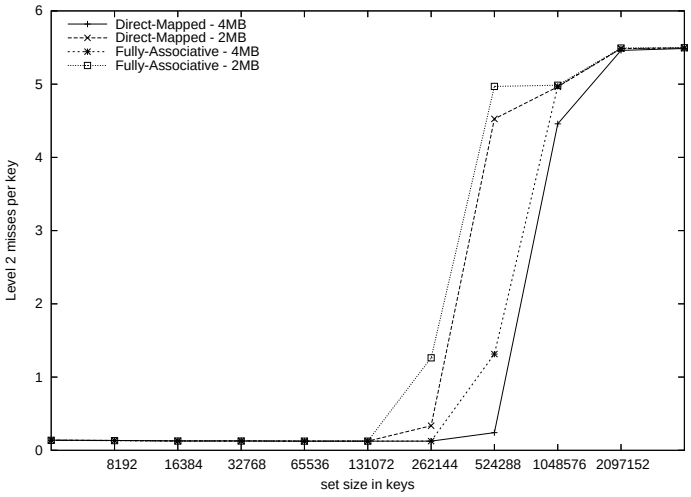
(a) Cycles per key



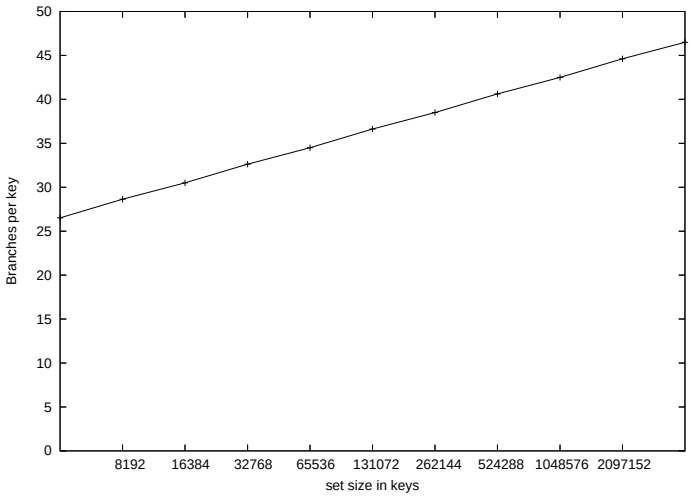
(b) Instructions per key



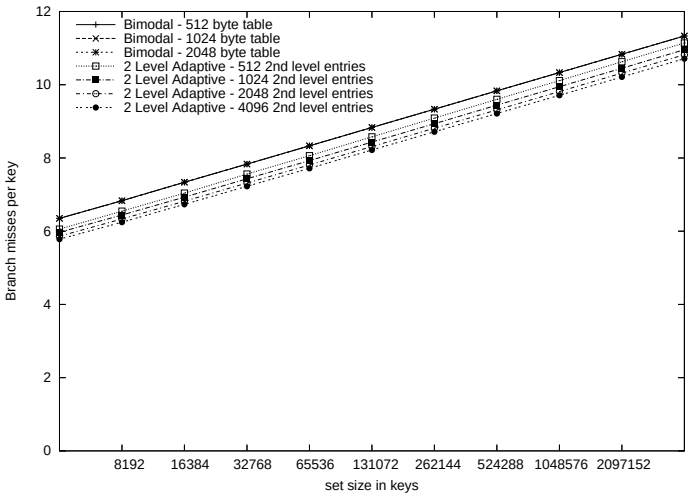
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

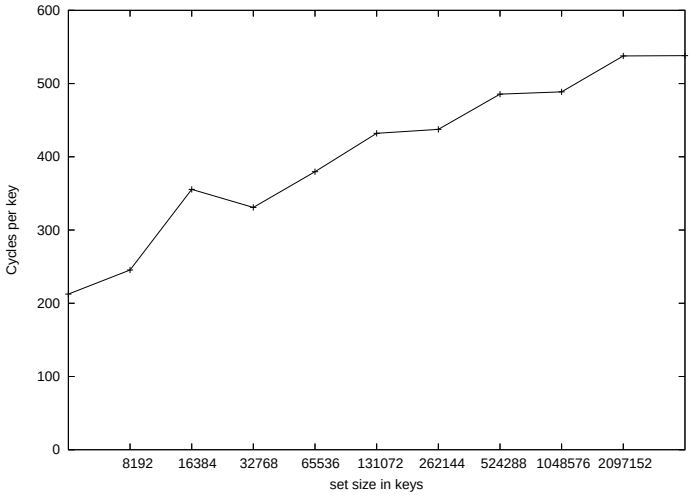


(e) Branches per key

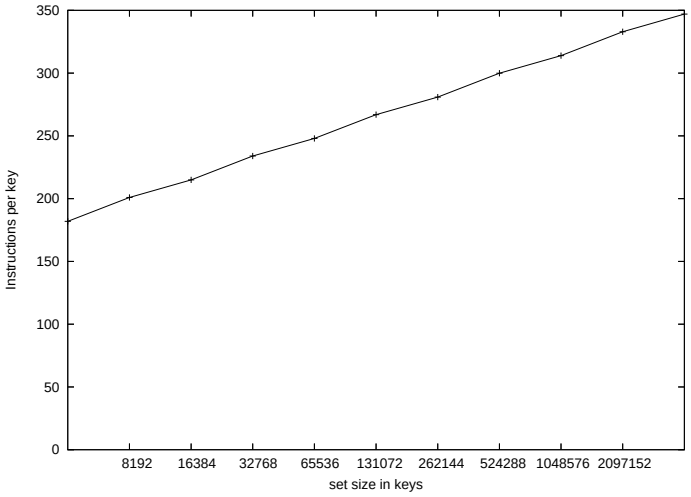


(f) Branch misses per key

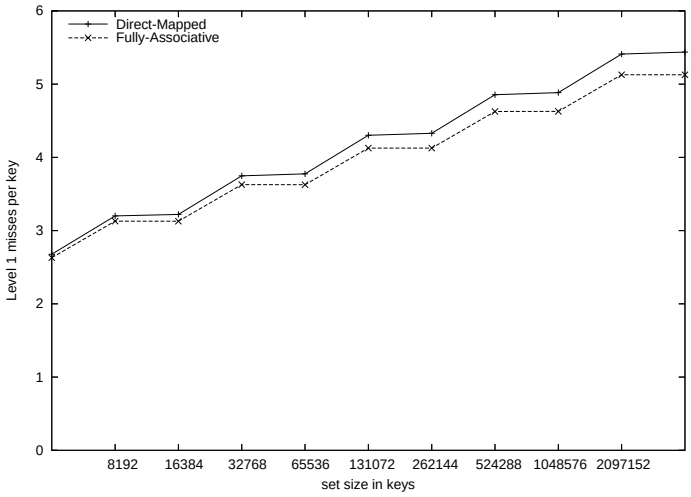
Figure A.11: Simulation results for algorithm S



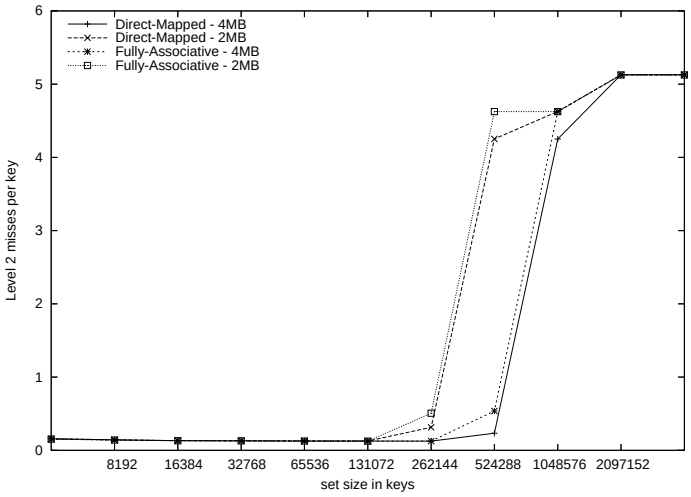
(a) Cycles per key



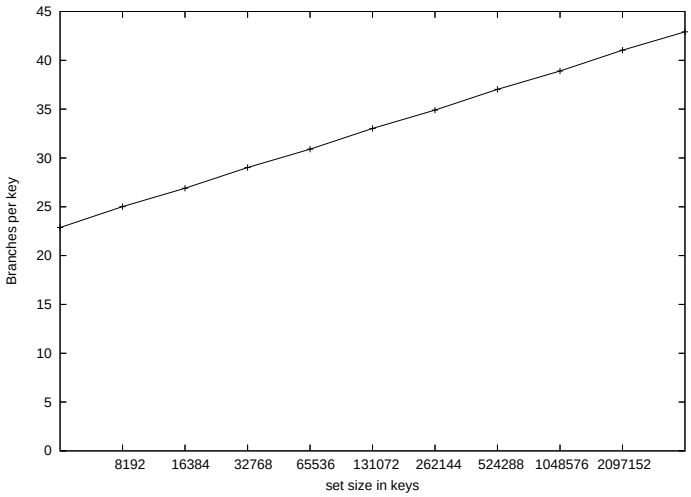
(b) Instructions per key



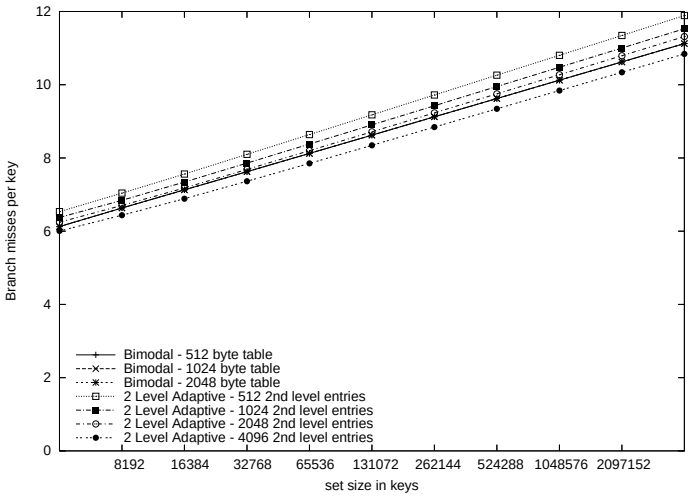
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

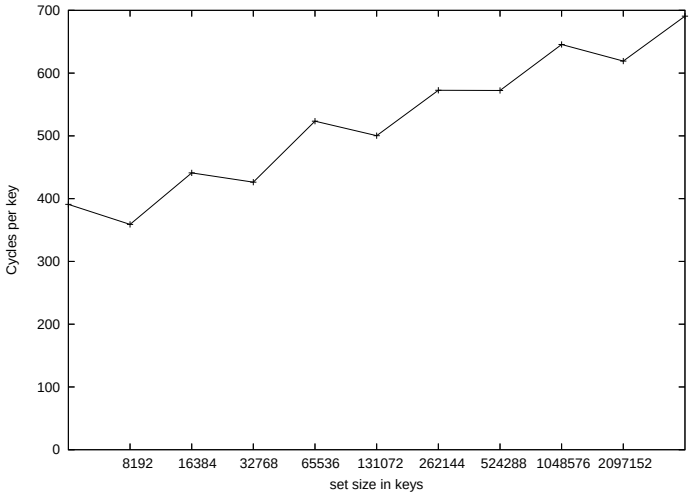


(e) Branches per key

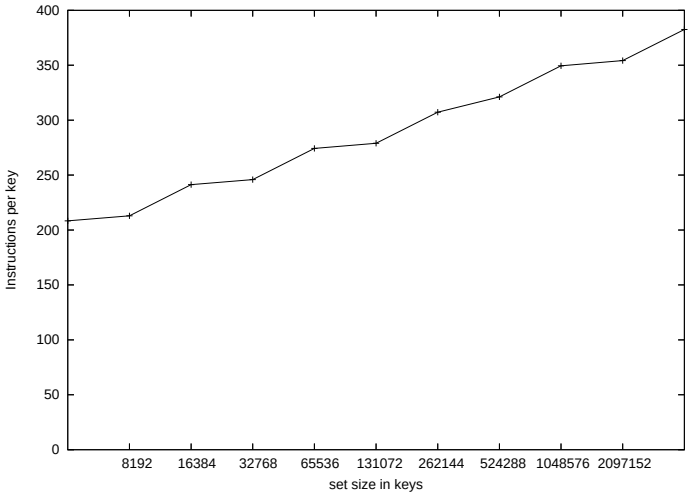


(f) Branch misses per key

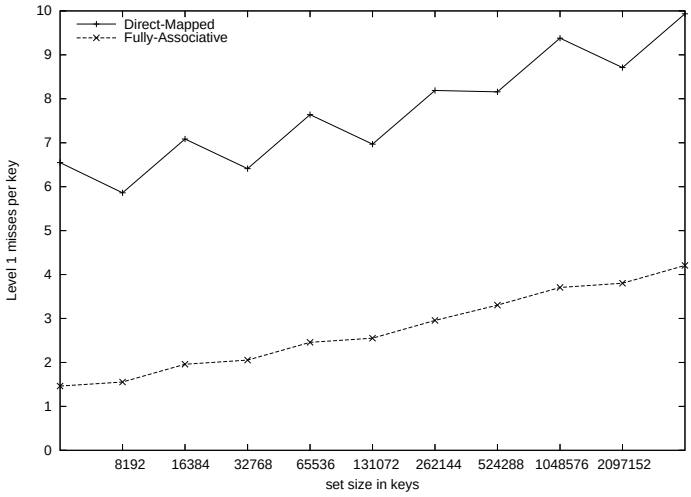
Figure A.12: Simulation results for base mergesort



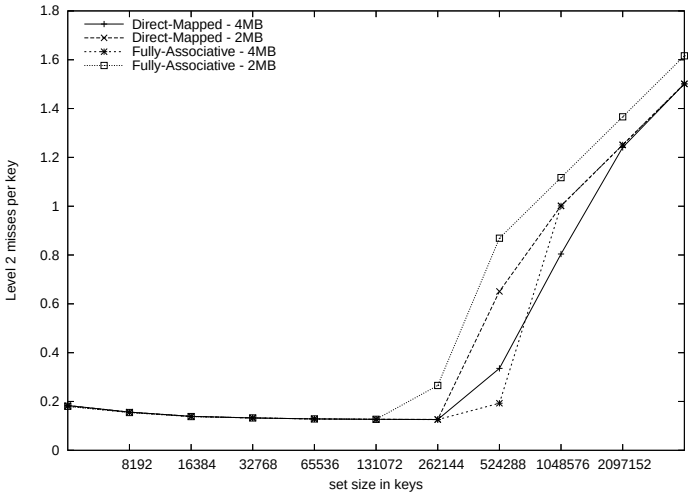
(a) Cycles per key



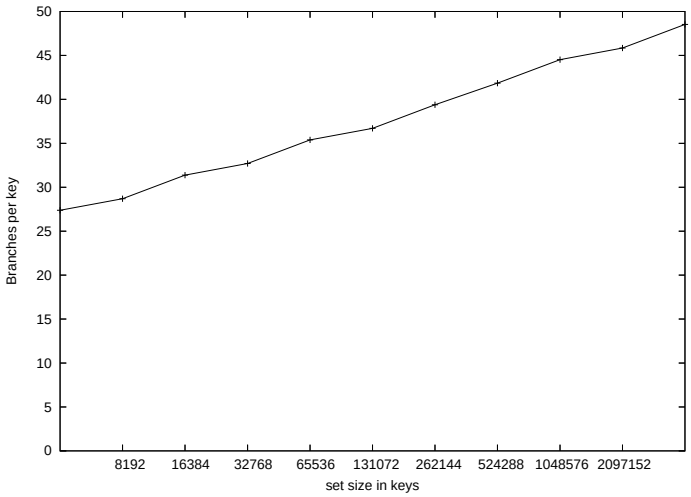
(b) Instructions per key



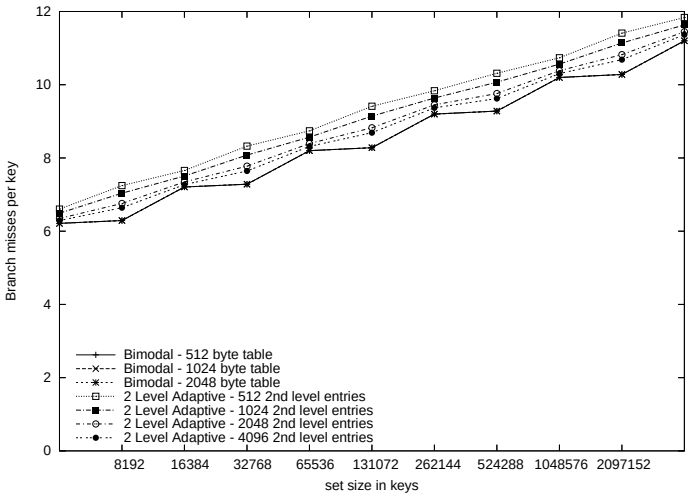
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

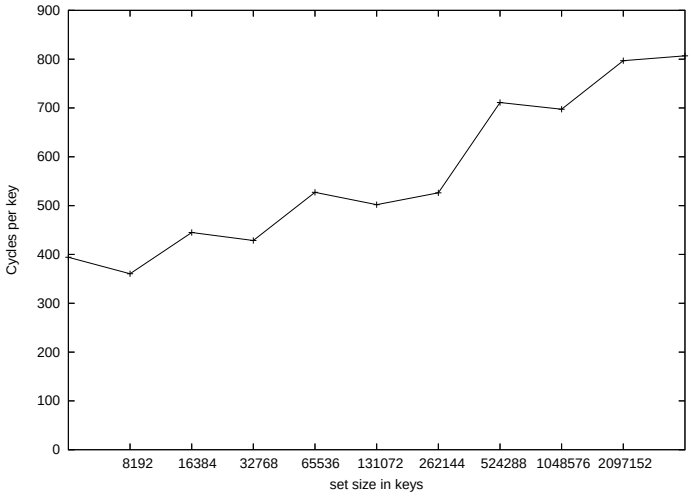


(e) Branches per key

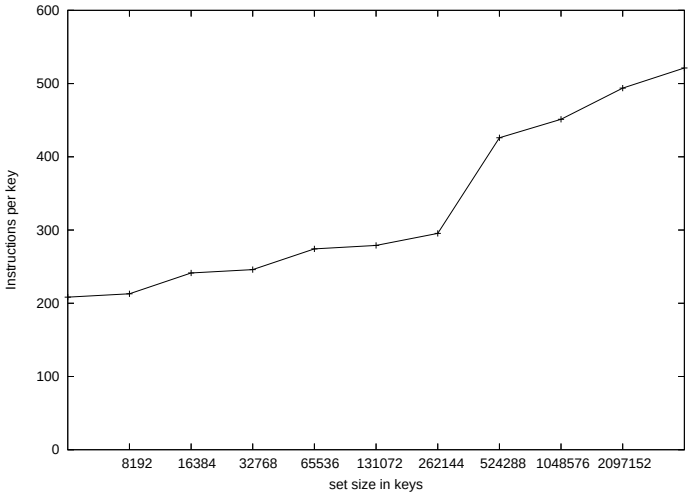


(f) Branch misses per key

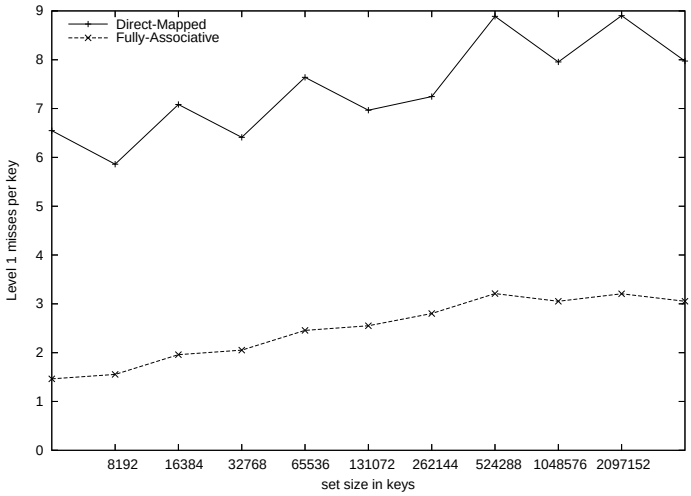
Figure A.13: Simulation results for tiled mergesort



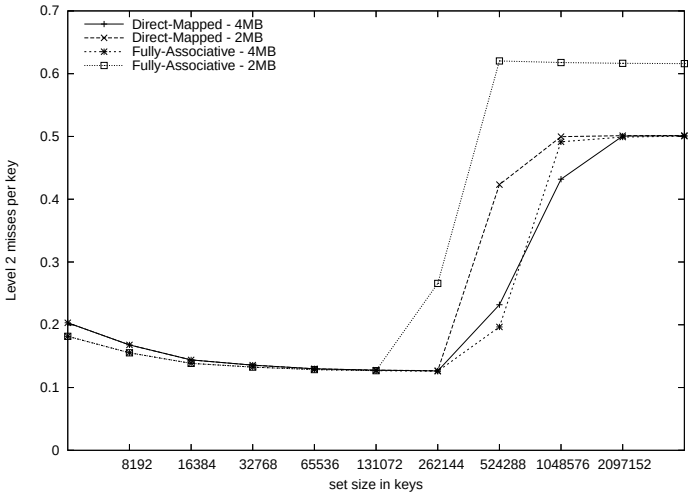
(a) Cycles per key



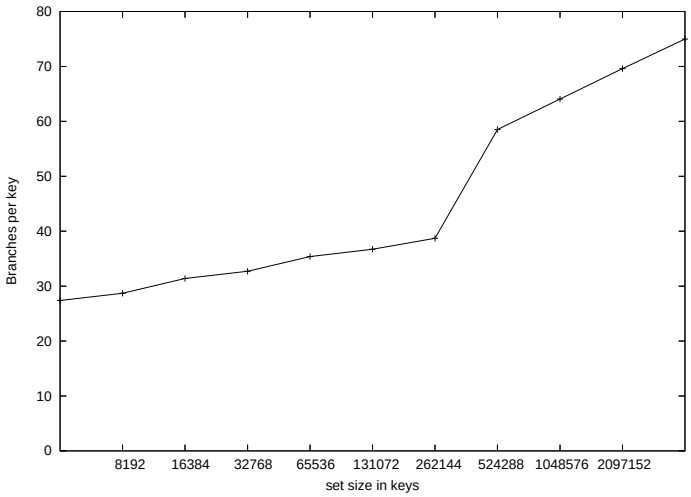
(b) Instructions per key



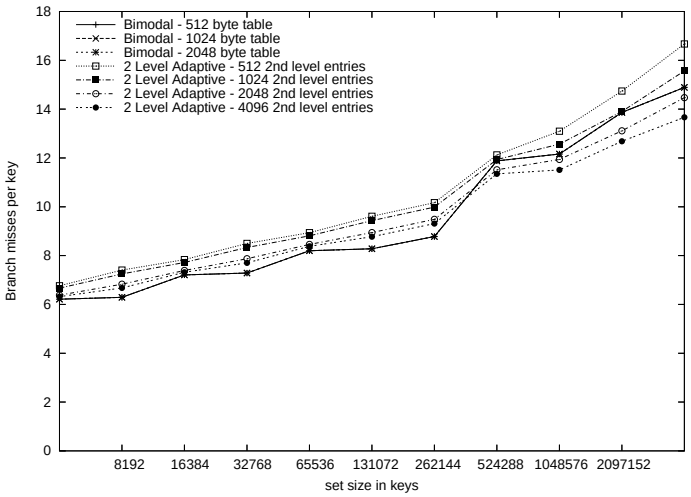
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

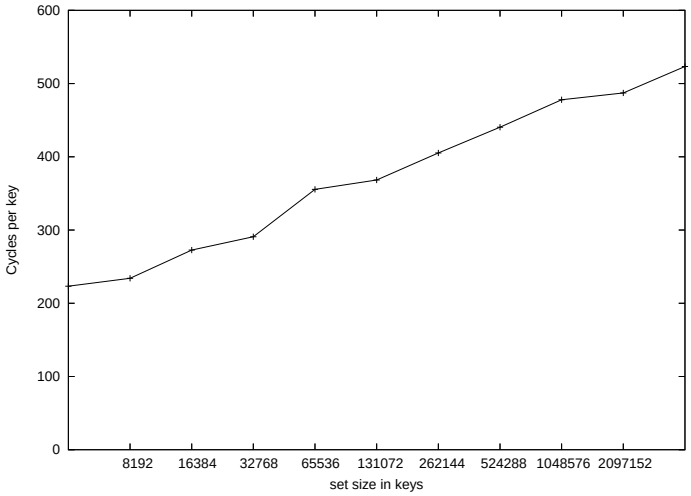


(e) Branches per key

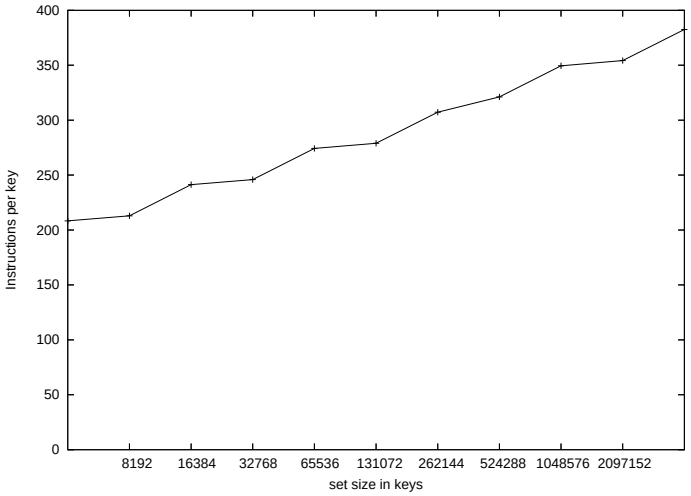


(f) Branch misses per key

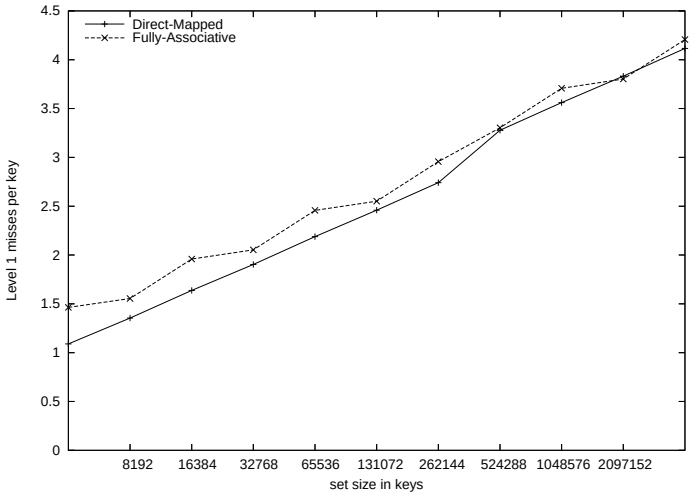
Figure A.14: Simulation results for multi-mergesort



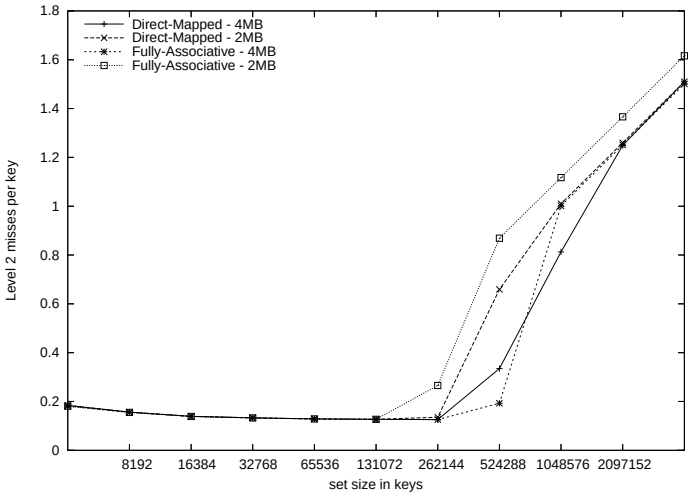
(a) Cycles per key



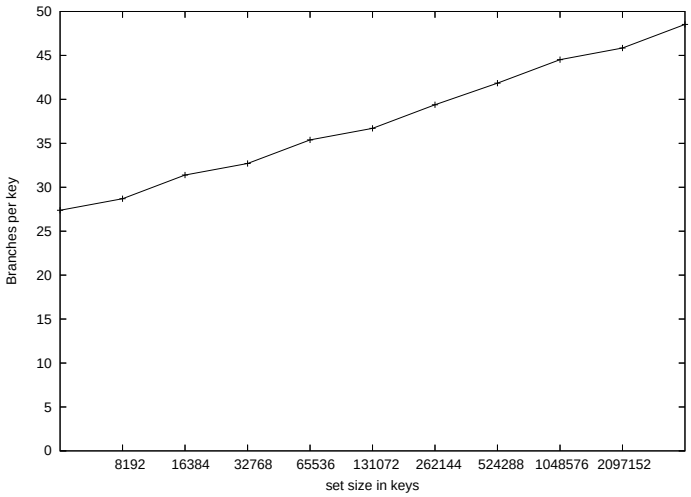
(b) Instructions per key



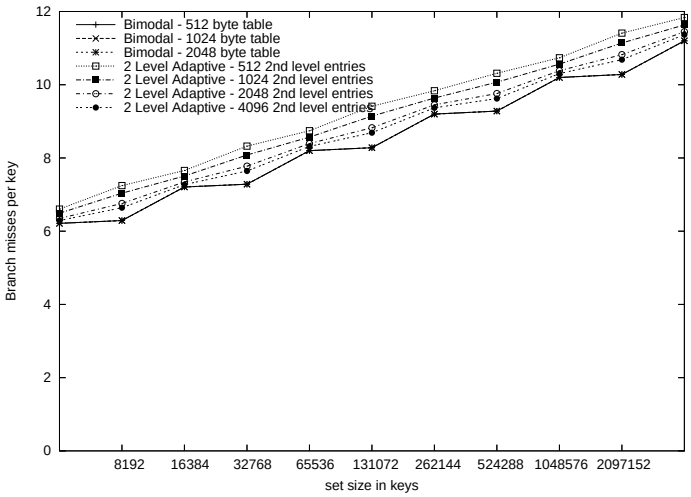
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

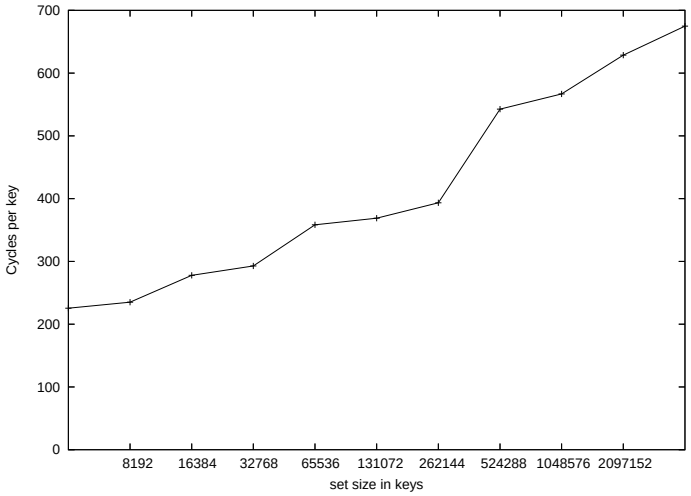


(e) Branches per key

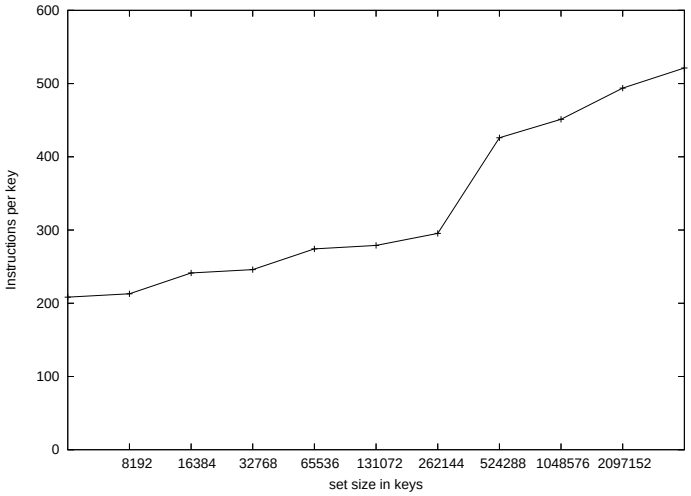


(f) Branch misses per key

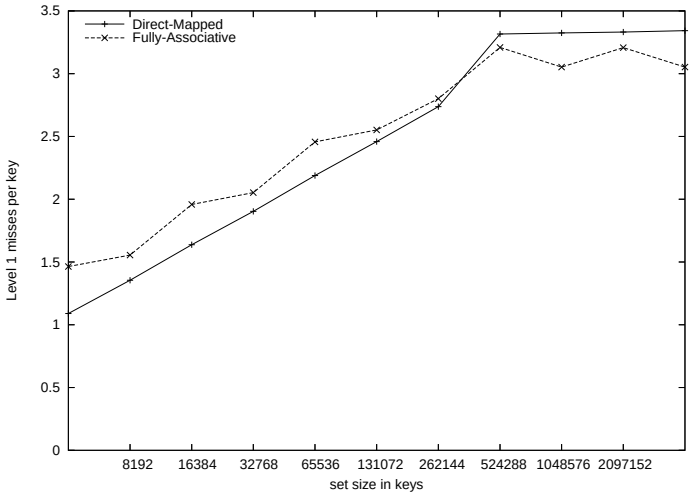
Figure A.15: Simulation results for double-aligned tiled mergesort



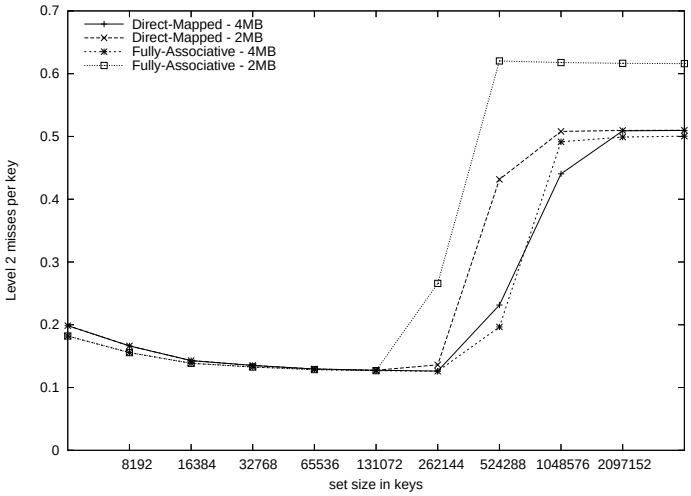
(a) Cycles per key



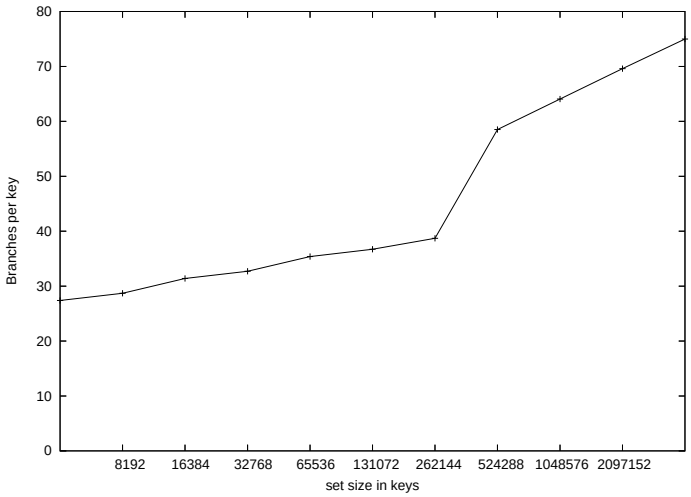
(b) Instructions per key



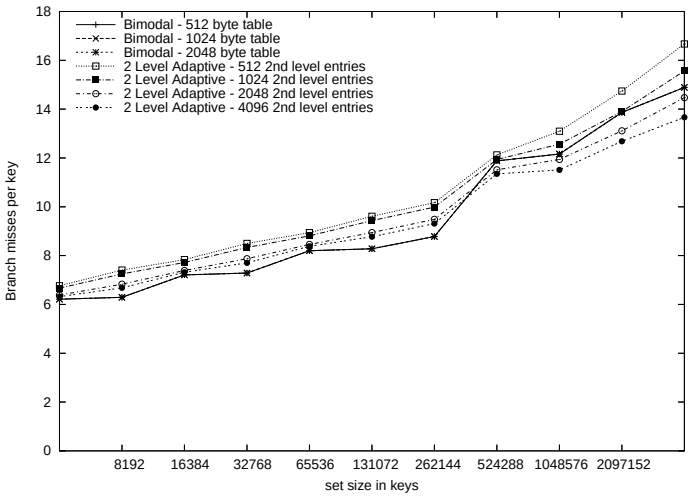
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

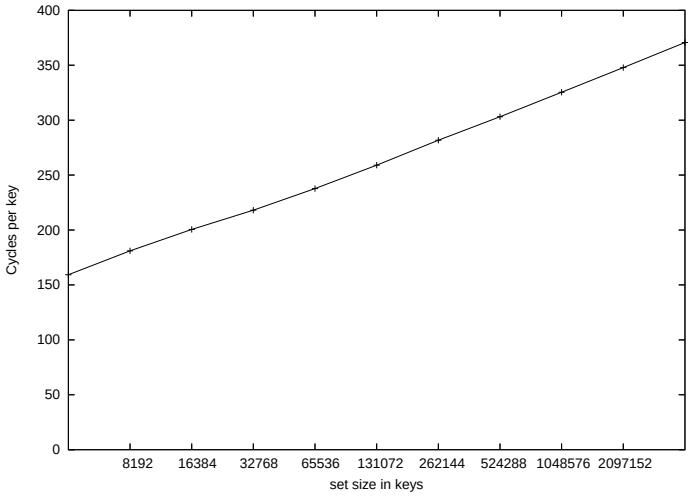


(e) Branches per key

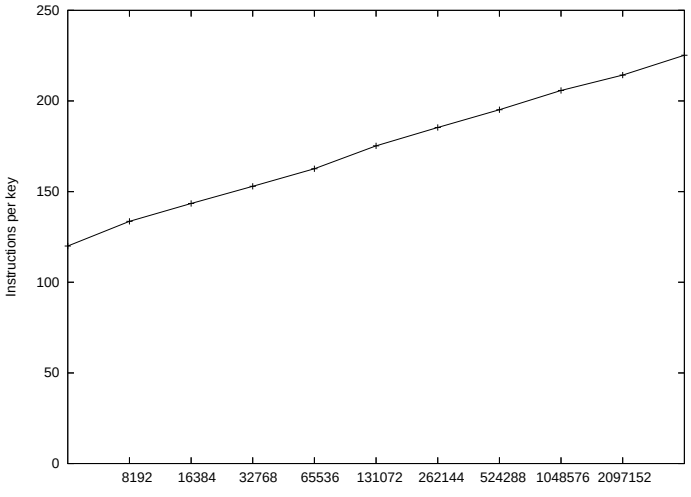


(f) Branch misses per key

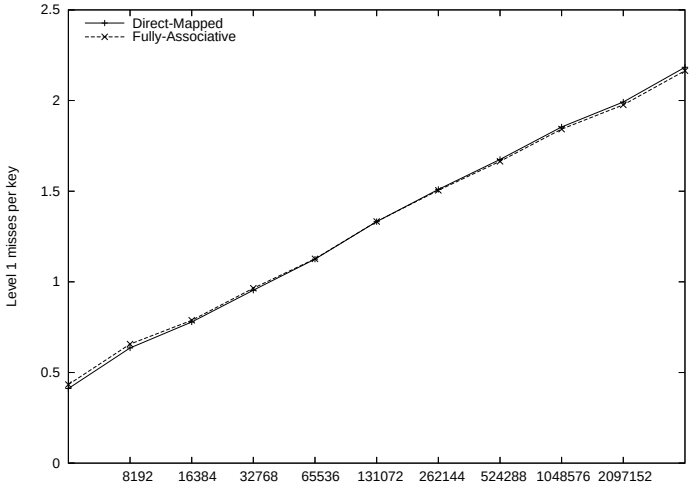
Figure A.16: Simulation results for double-aligned multi-mergesort



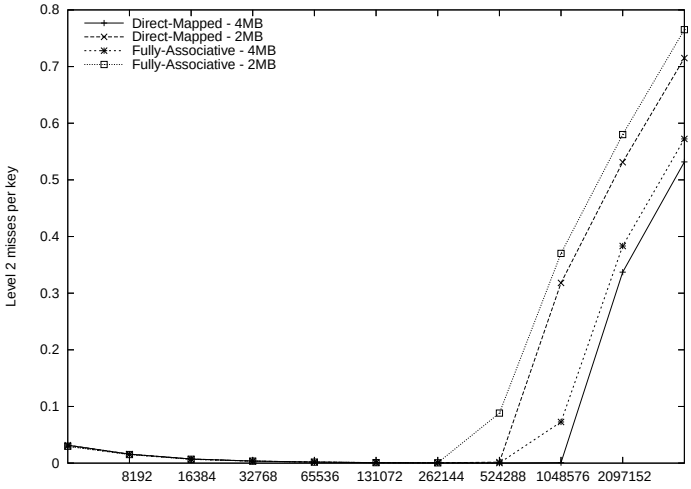
(a) Cycles per key



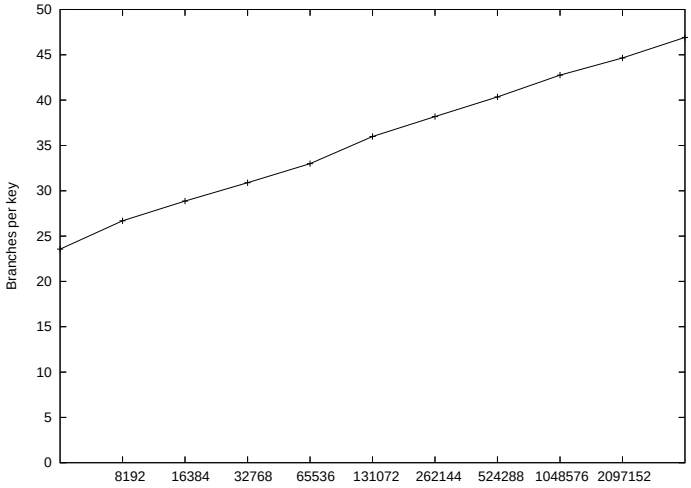
(b) Instructions per key



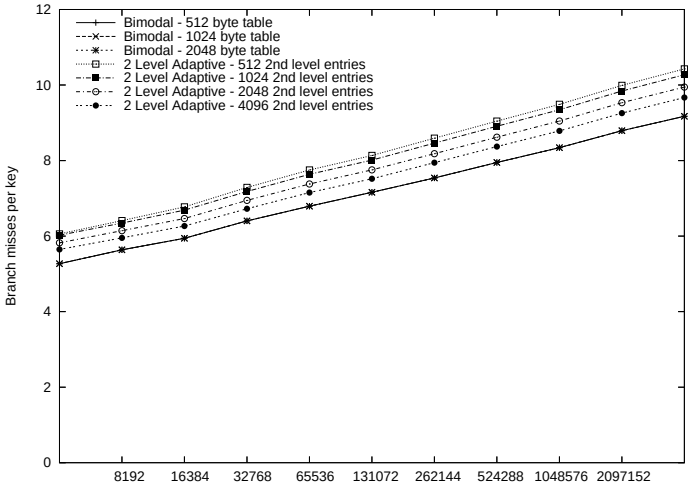
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

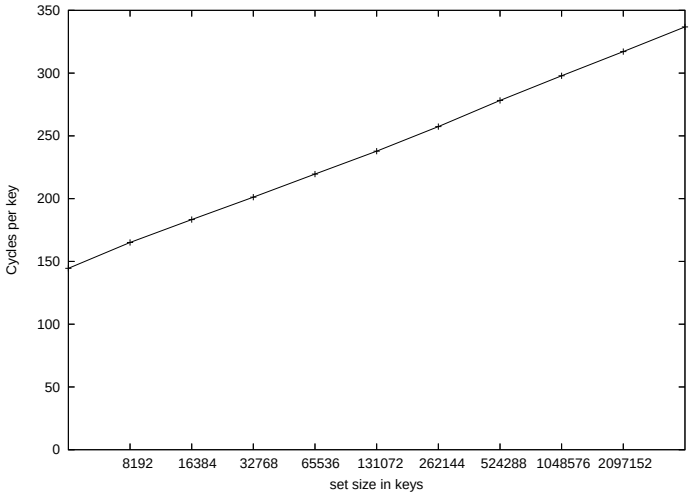


(e) Branches per key

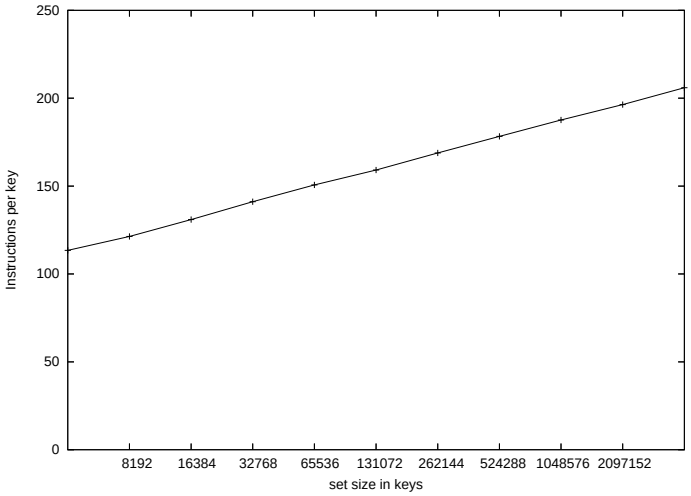


(f) Branch misses per key

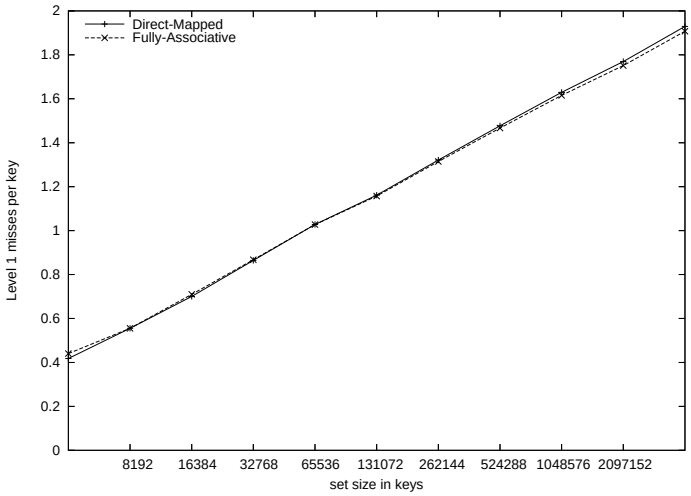
Figure A.17: Simulation results for base quicksort (no median)



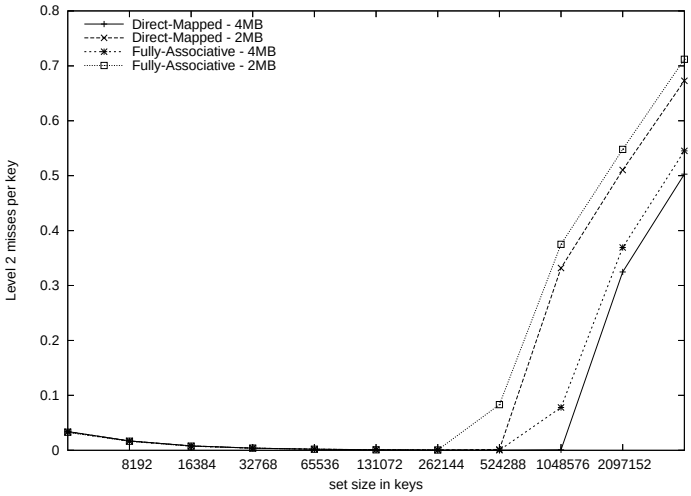
(a) Cycles per key



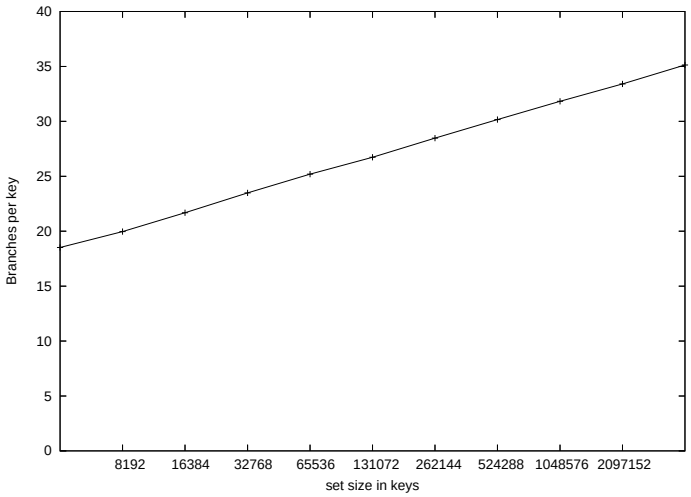
(b) Instructions per key



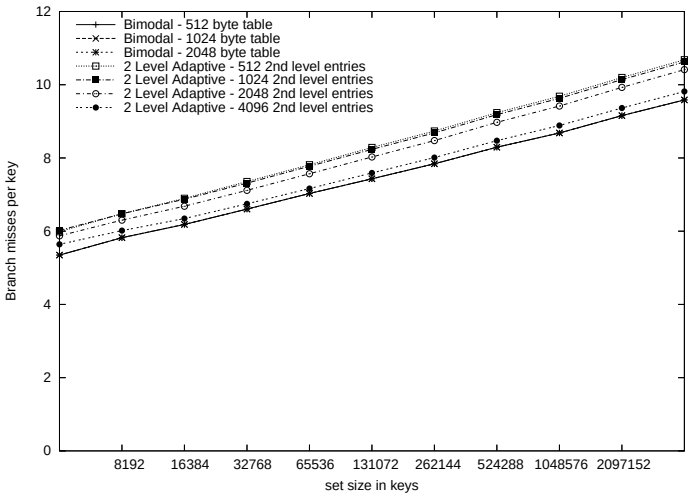
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

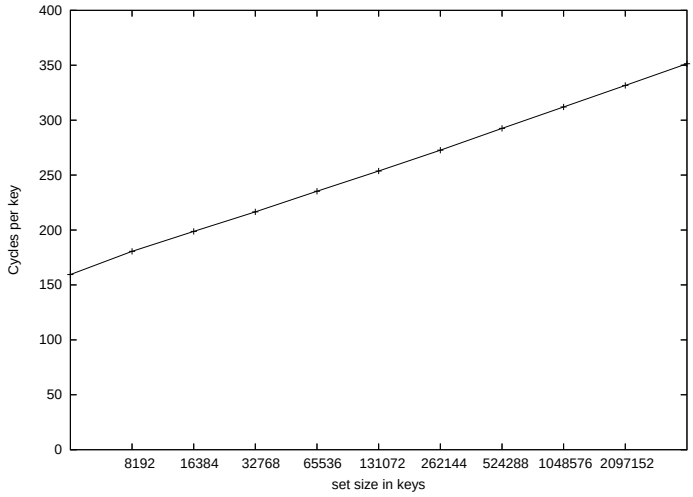


(e) Branches per key

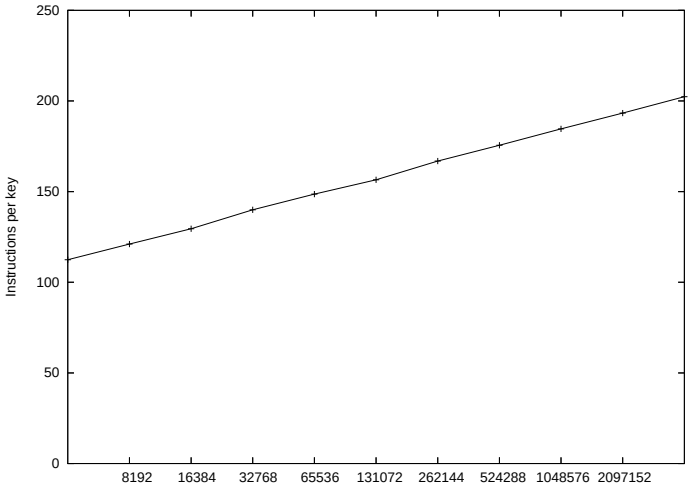


(f) Branch misses per key

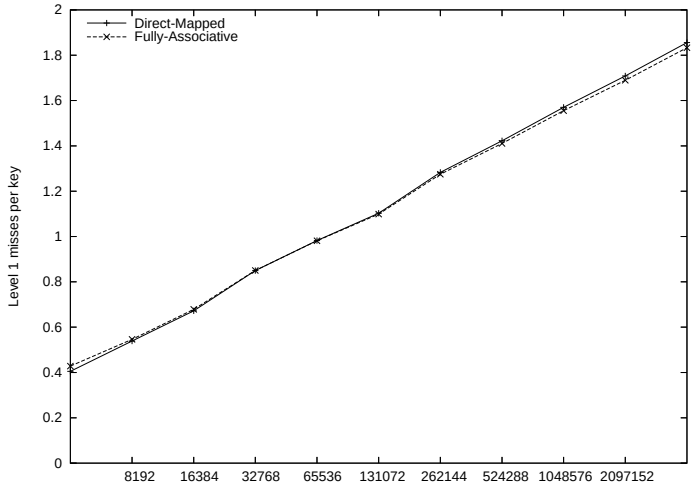
Figure A.18: Simulation results for base quicksort (median-of-3)



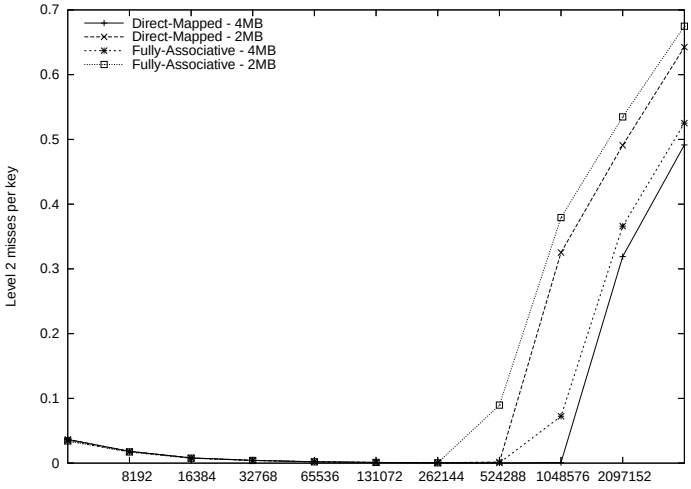
(a) Cycles per key



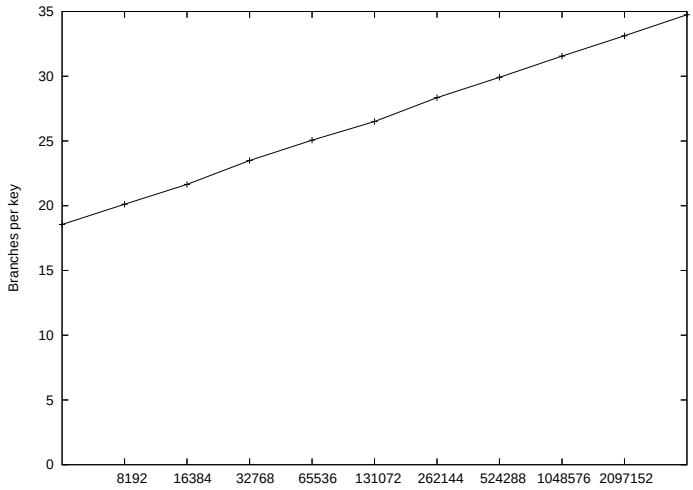
(b) Instructions per key



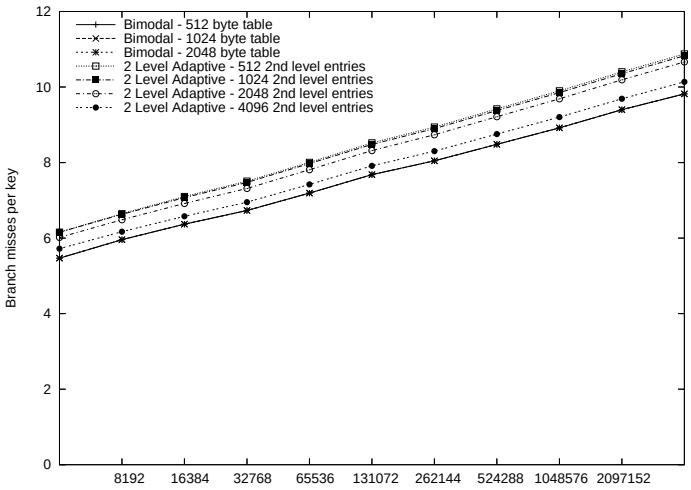
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

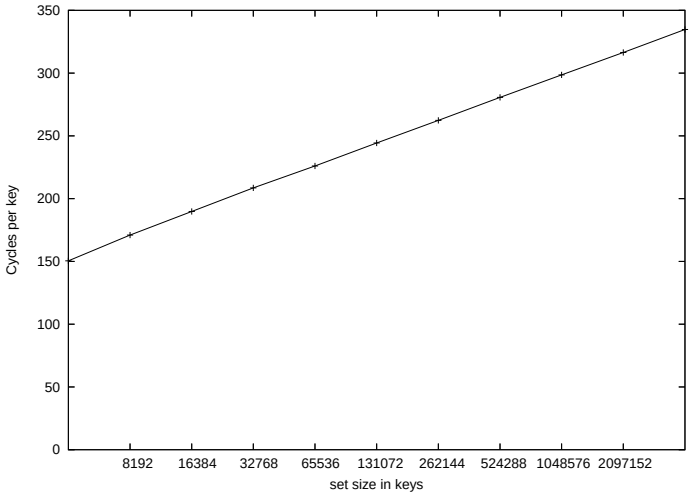


(e) Branches per key

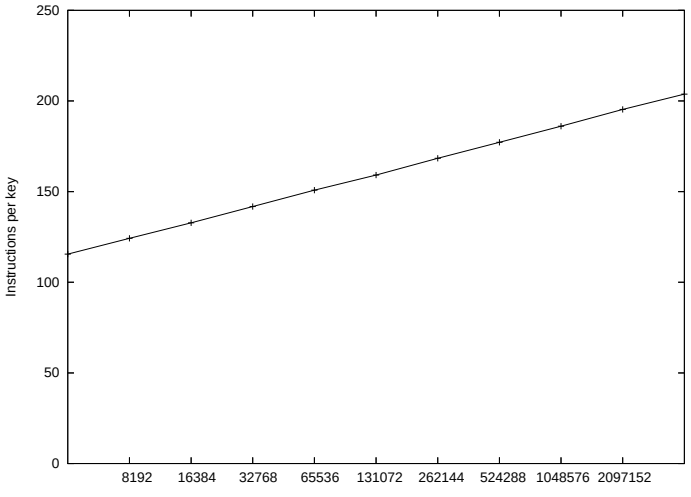


(f) Branch misses per key

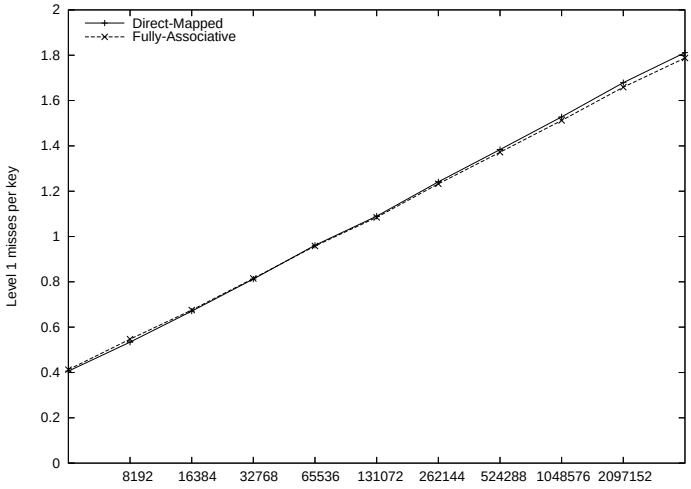
Figure A.19: Simulation results for base quicksort (pseudo-median-of-5)



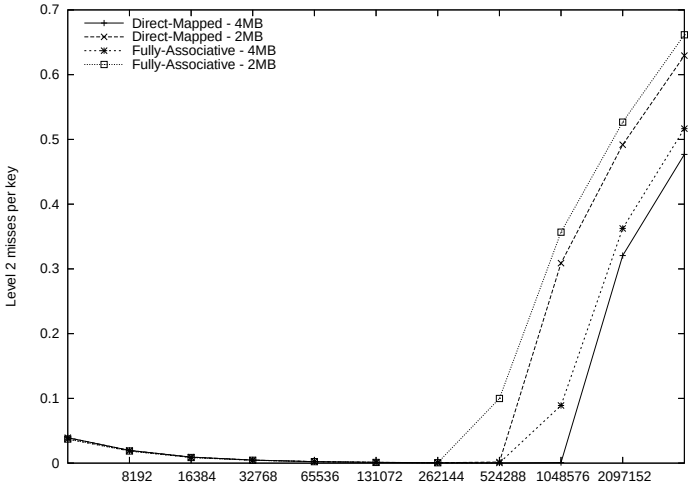
(a) Cycles per key



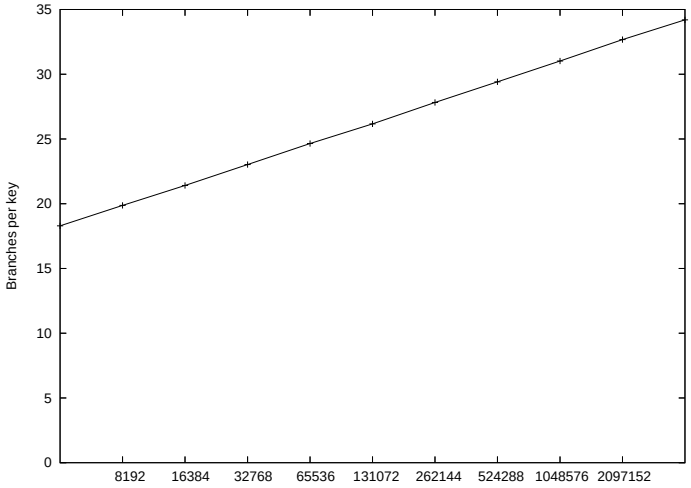
(b) Instructions per key



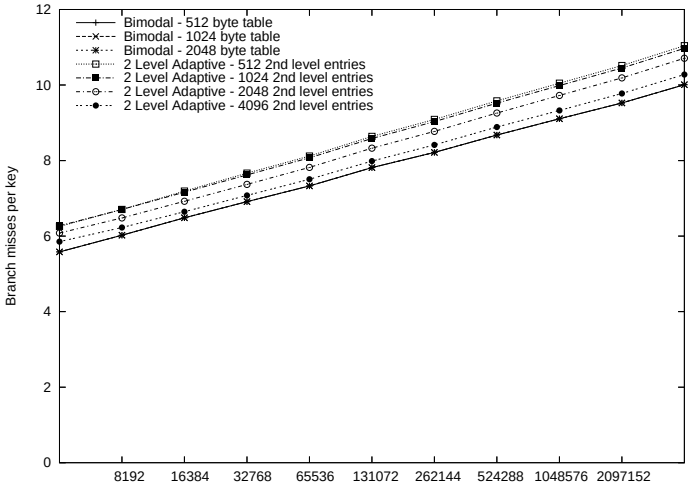
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

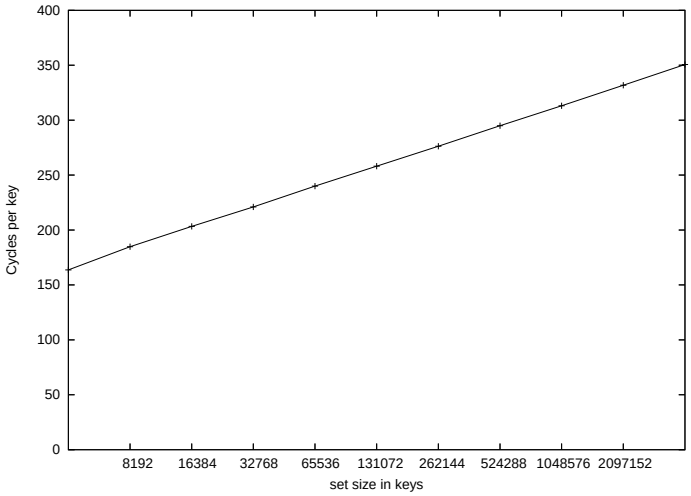


(e) Branches per key

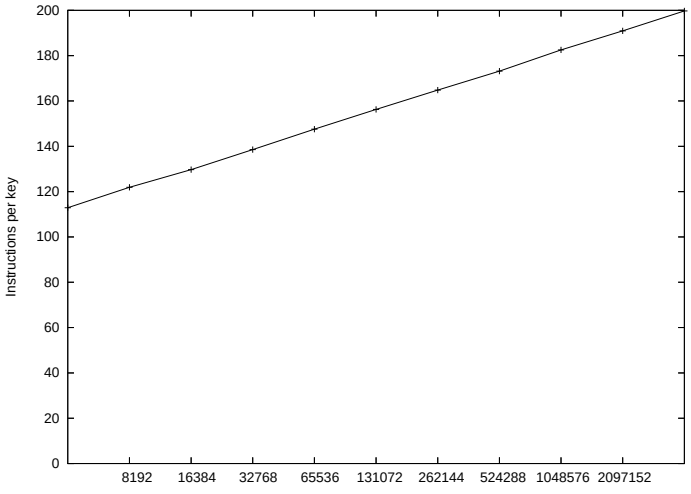


(f) Branch misses per key

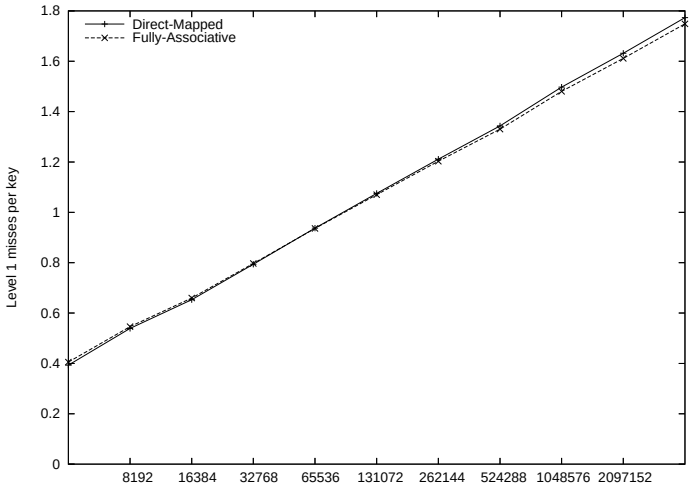
Figure A.20: Simulation results for base quicksort (pseudo-median-of-7)



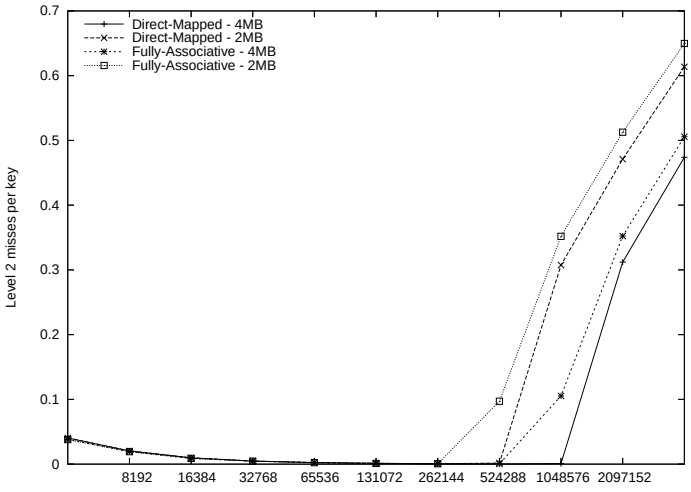
(a) Cycles per key



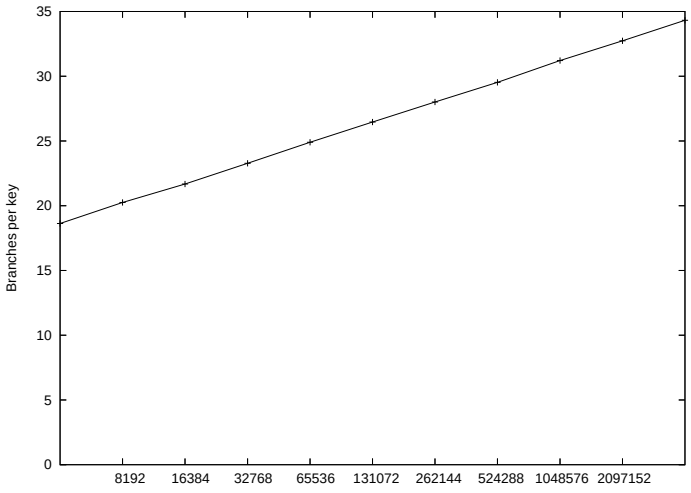
(b) Instructions per key



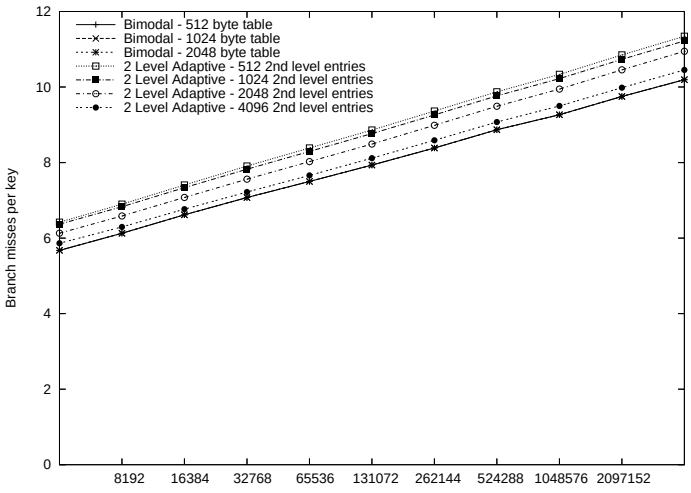
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

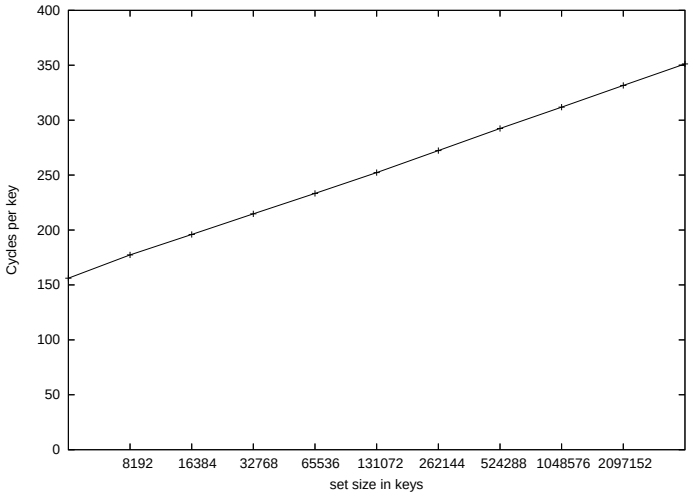


(e) Branches per key

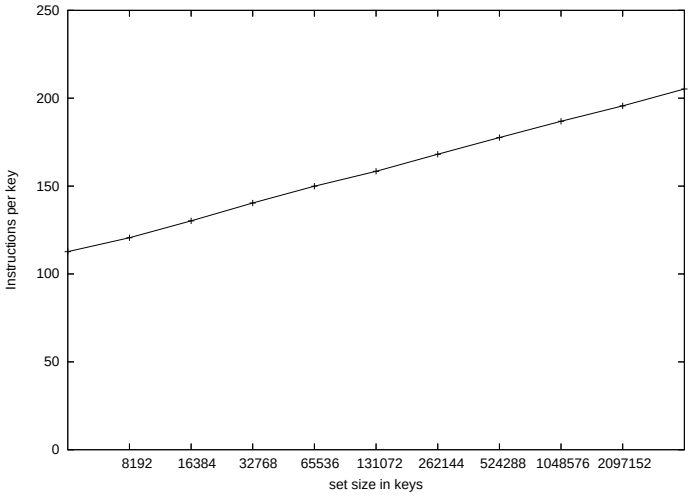


(f) Branch misses per key

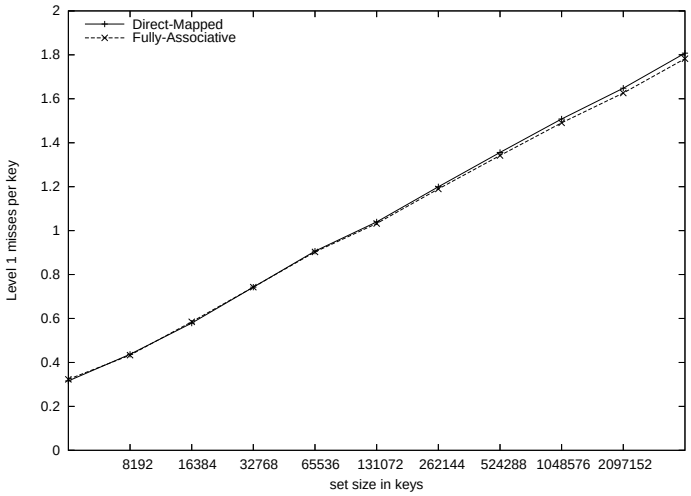
Figure A.21: Simulation results for base quicksort (pseudo-median-of-9)



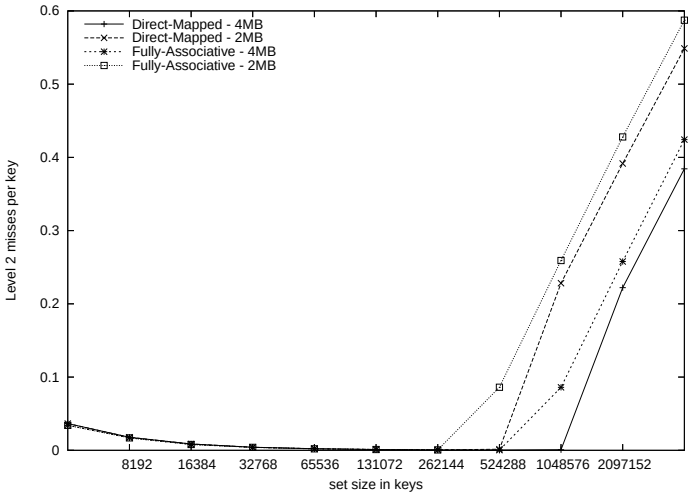
(a) Cycles per key



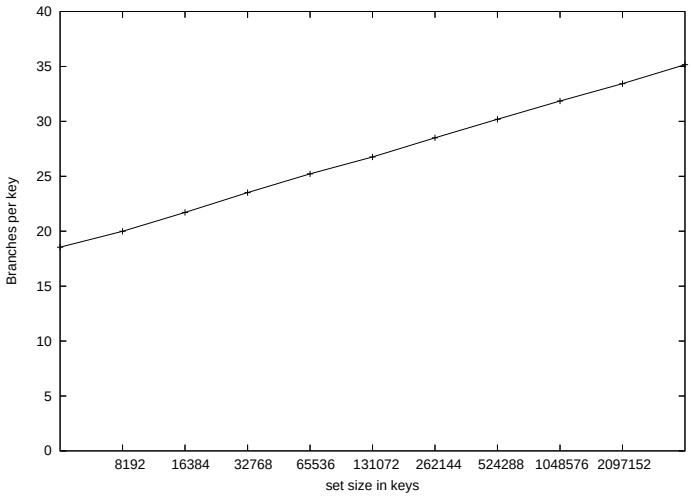
(b) Instructions per key



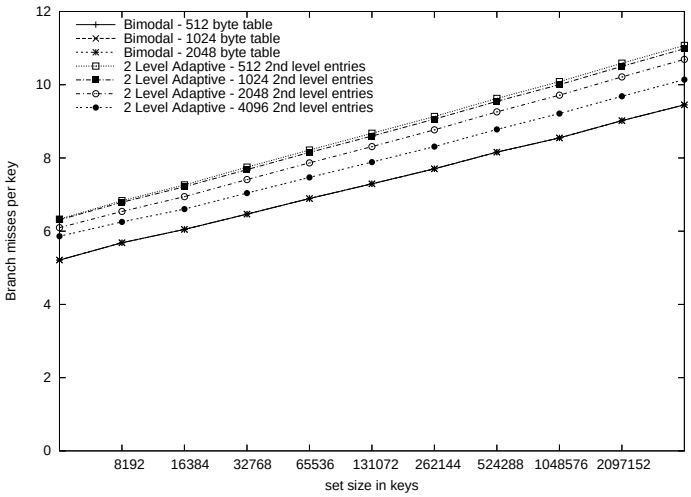
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

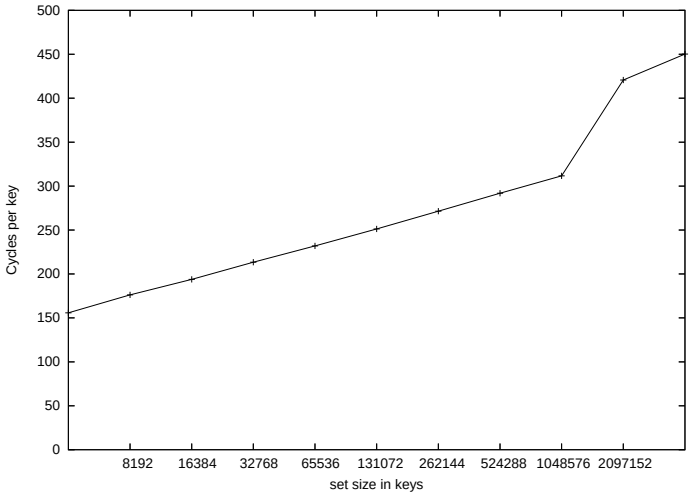


(e) Branches per key

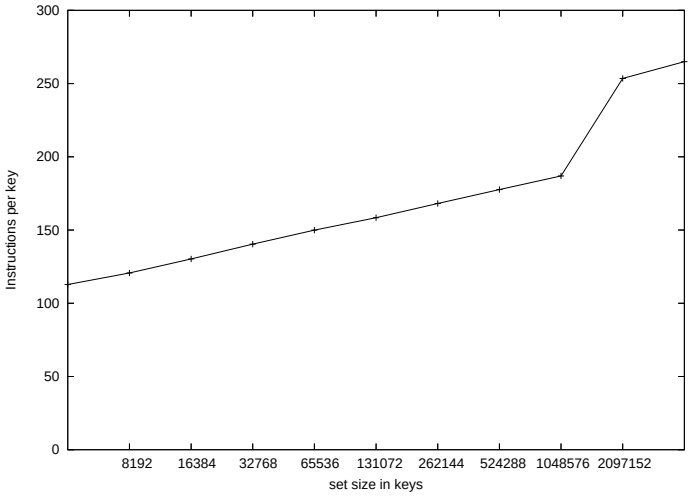


(f) Branch misses per key

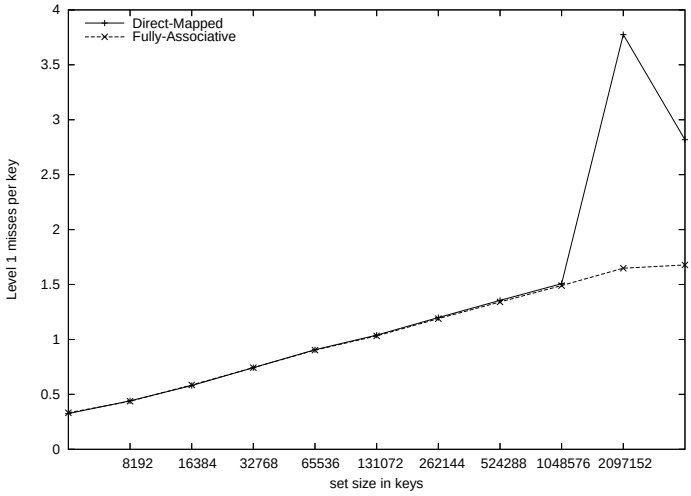
Figure A.22: Simulation results for memory-tuned quicksort



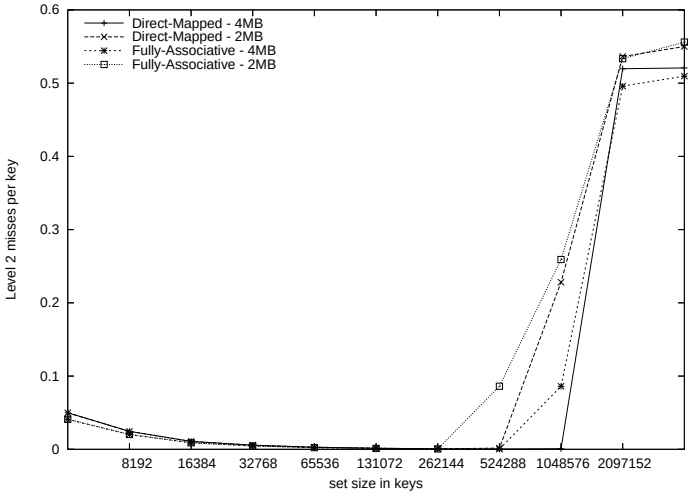
(a) Cycles per key



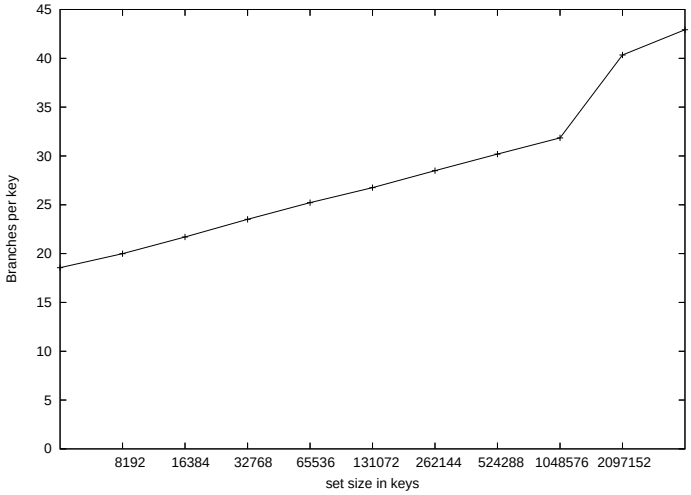
(b) Instructions per key



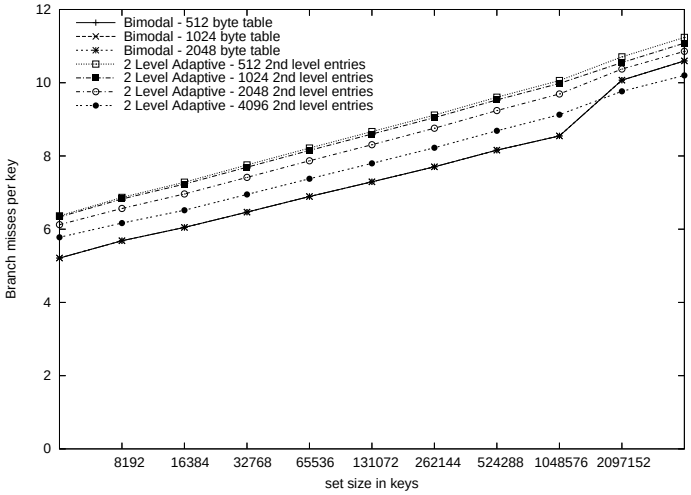
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

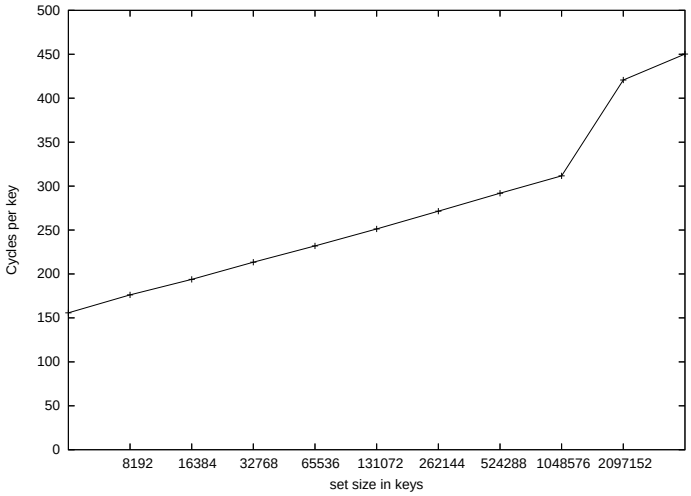


(e) Branches per key

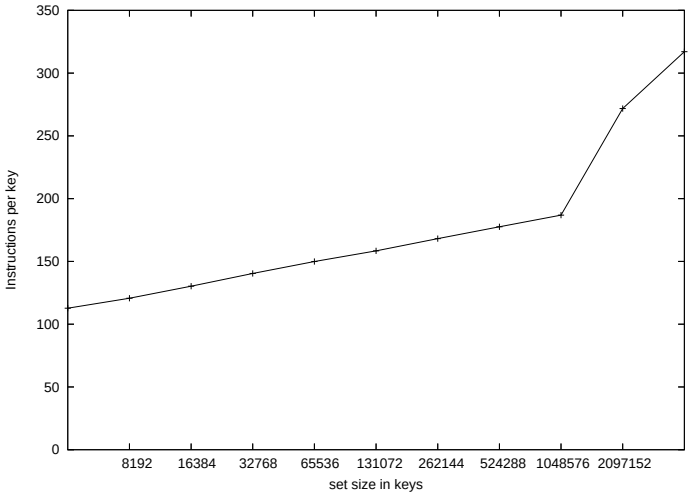


(f) Branch misses per key

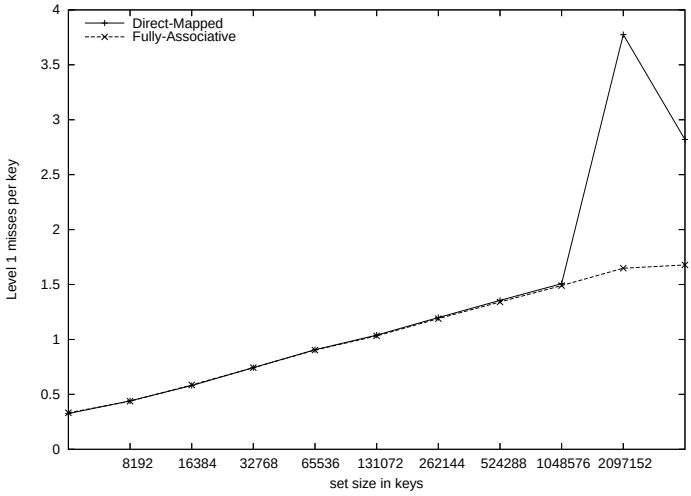
Figure A.23: Simulation results for multi-quicksort (binary search)



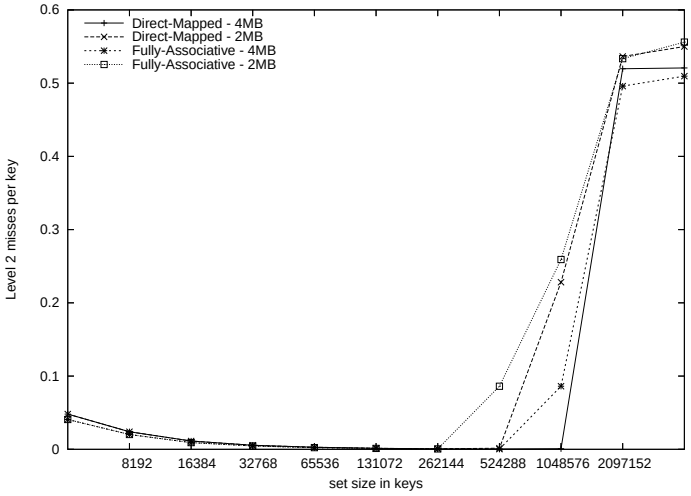
(a) Cycles per key



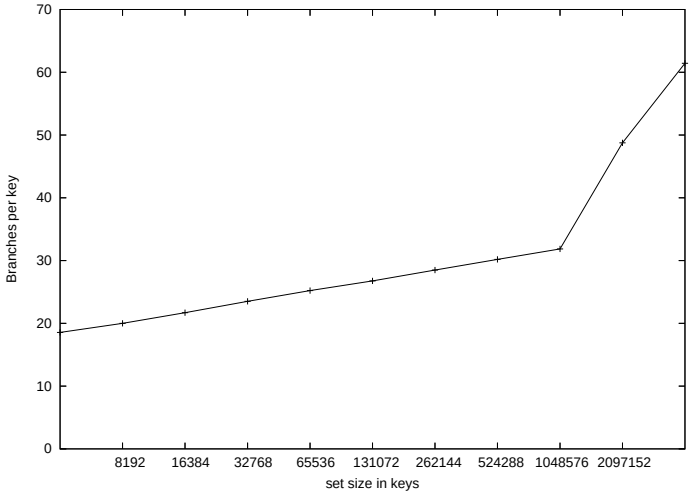
(b) Instructions per key



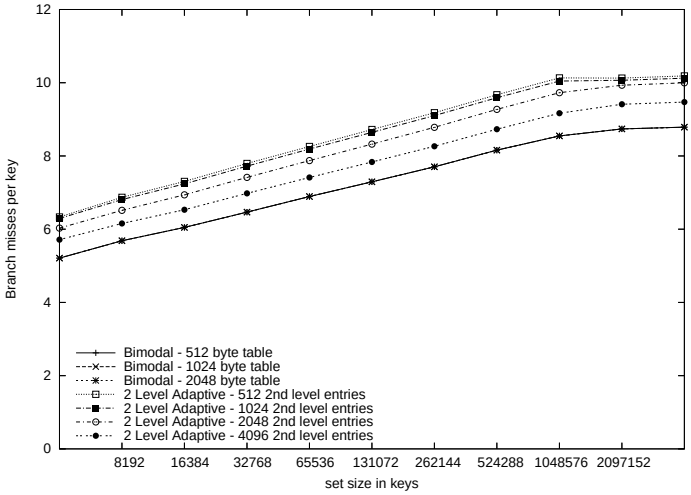
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

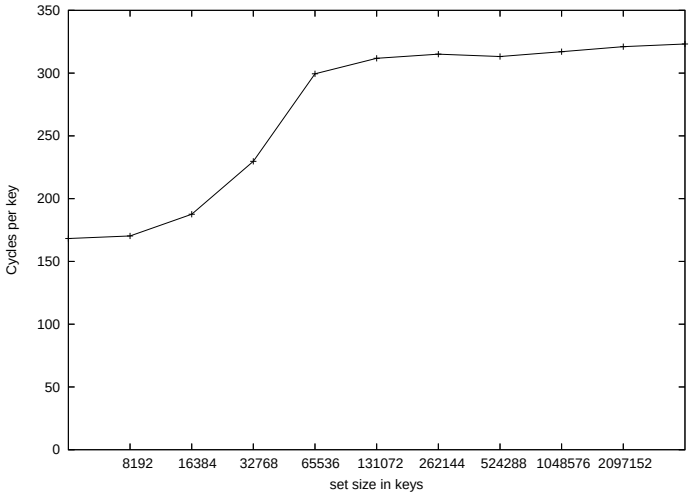


(e) Branches per key

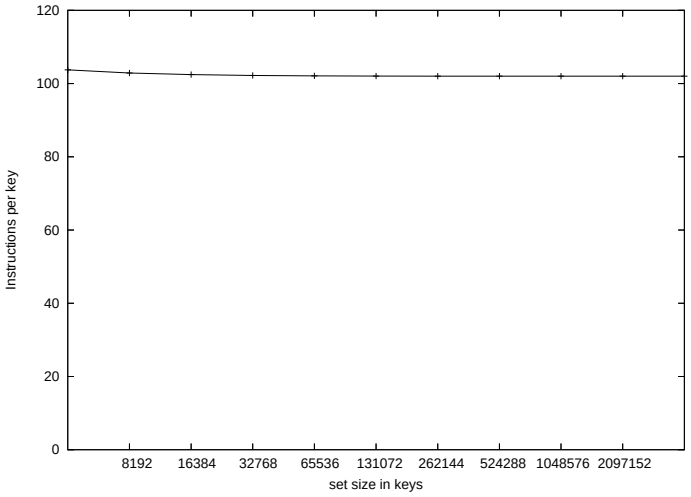


(f) Branch misses per key

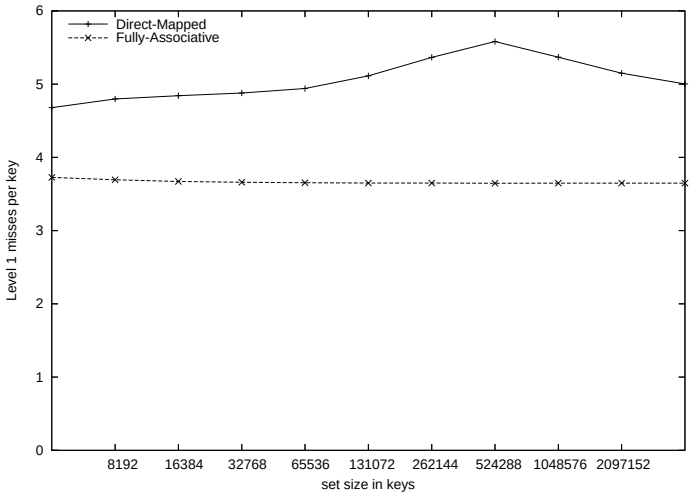
Figure A.24: Simulation results for multi-quicksort (sequential search)



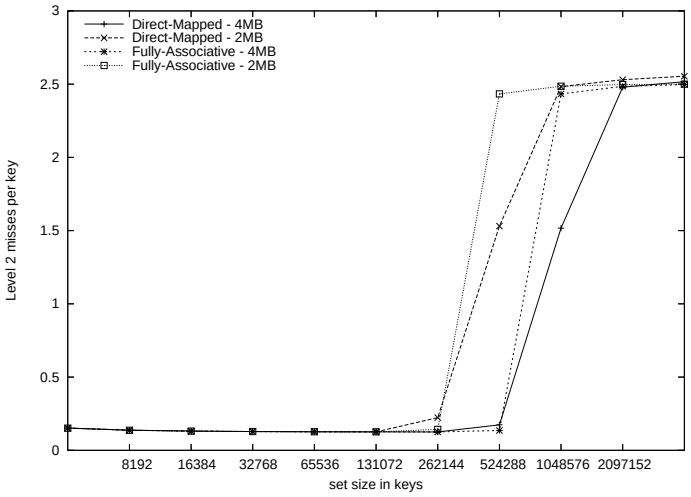
(a) Cycles per key



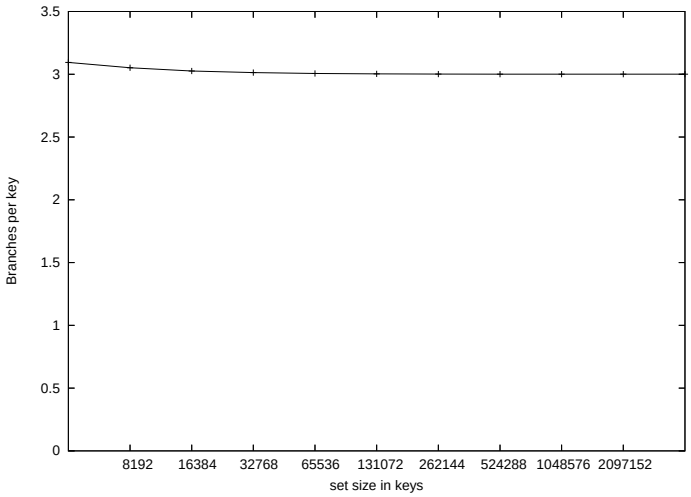
(b) Instructions per key



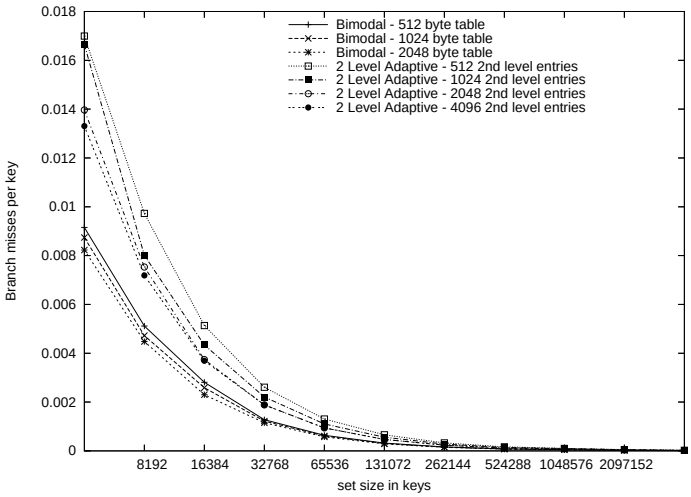
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

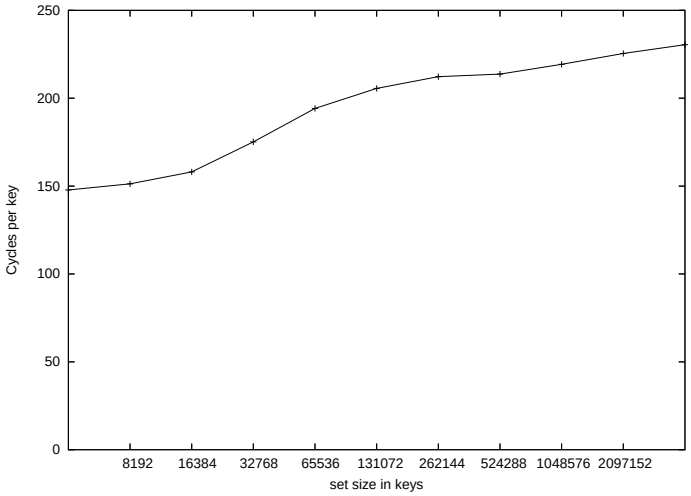


(e) Branches per key

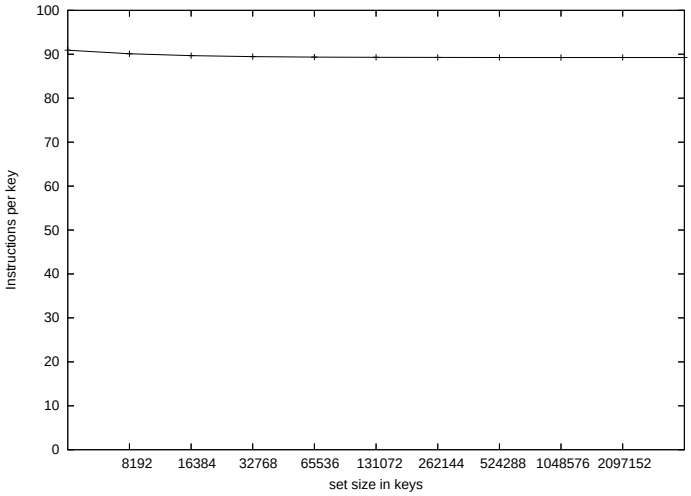


(f) Branch misses per key

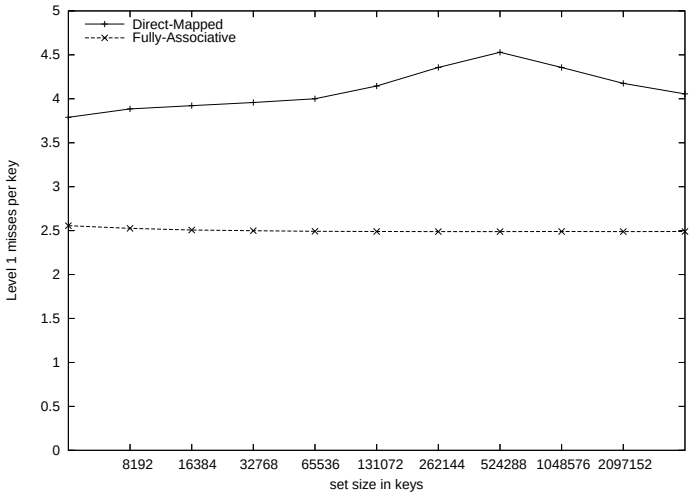
Figure A.25: Simulation results for base radixsort



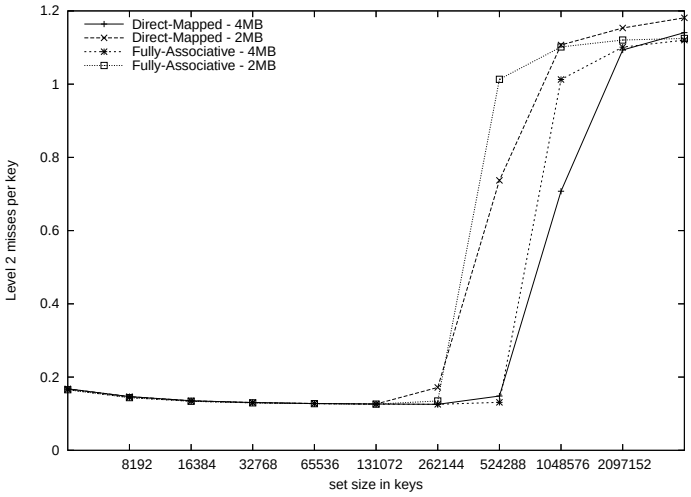
(a) Cycles per key



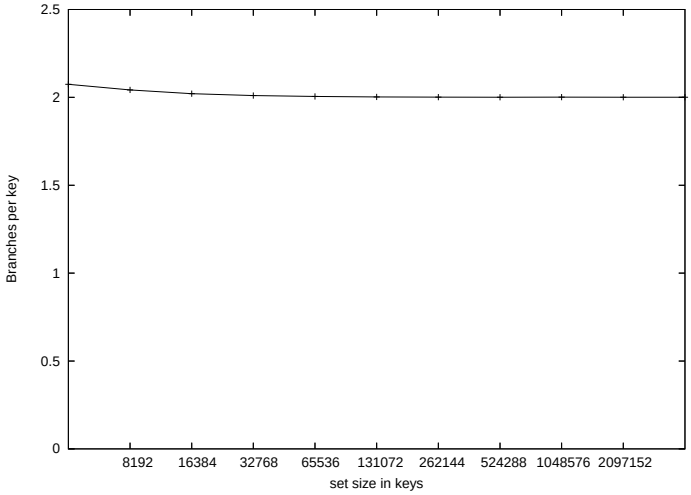
(b) Instructions per key



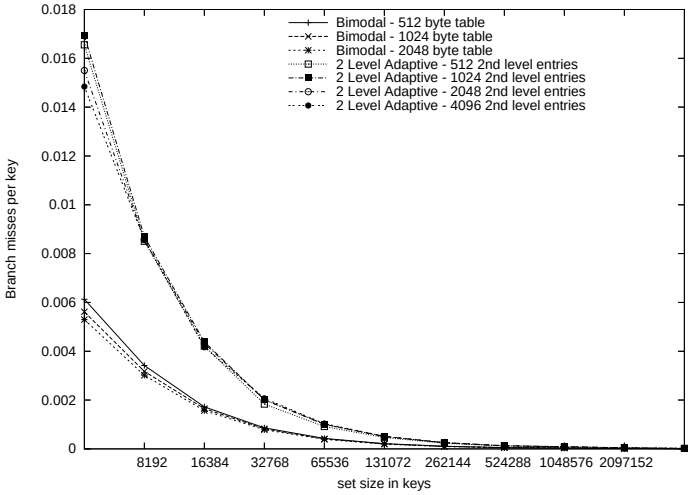
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

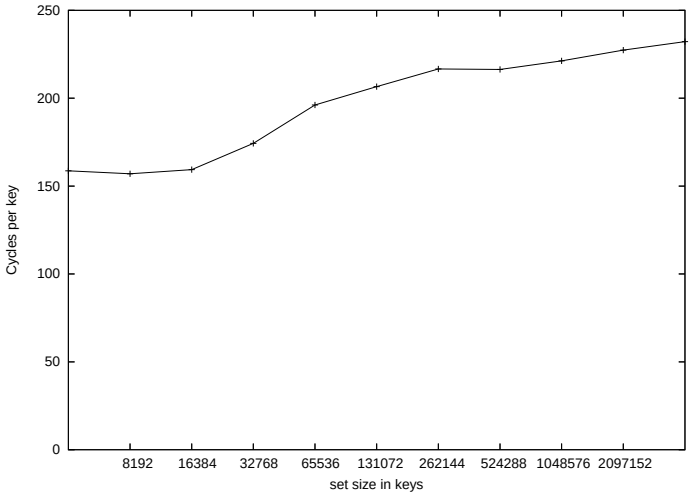


(e) Branches per key

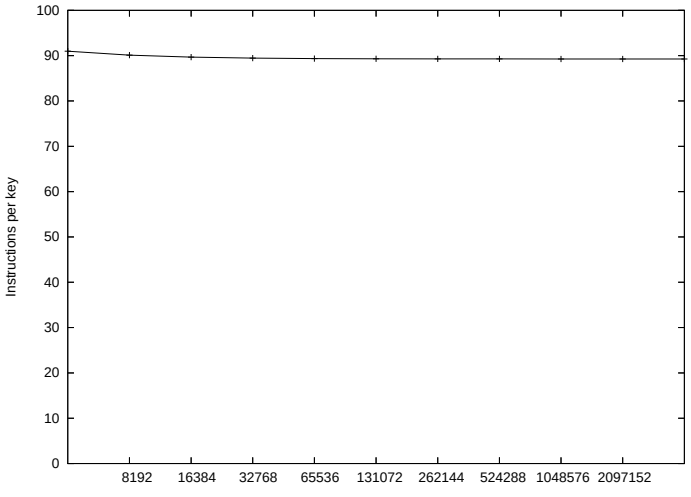


(f) Branch misses per key

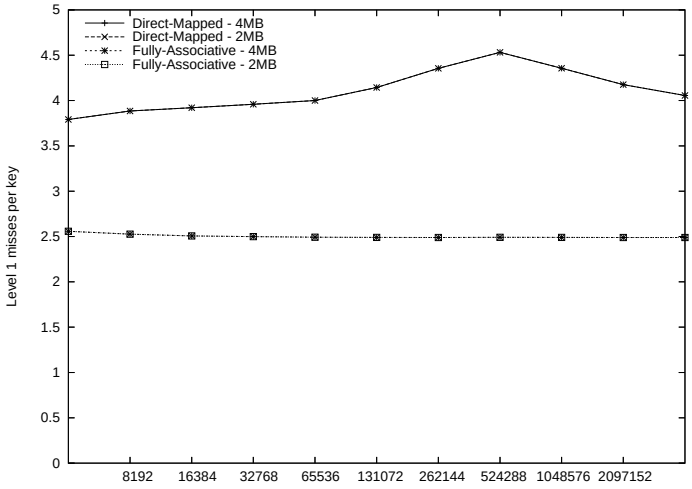
Figure A.26: Simulation results for memory-tuned radixsort



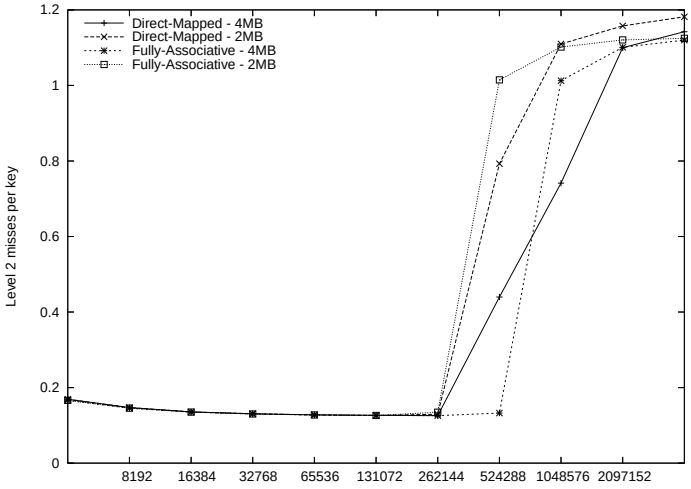
(a) Cycles per key



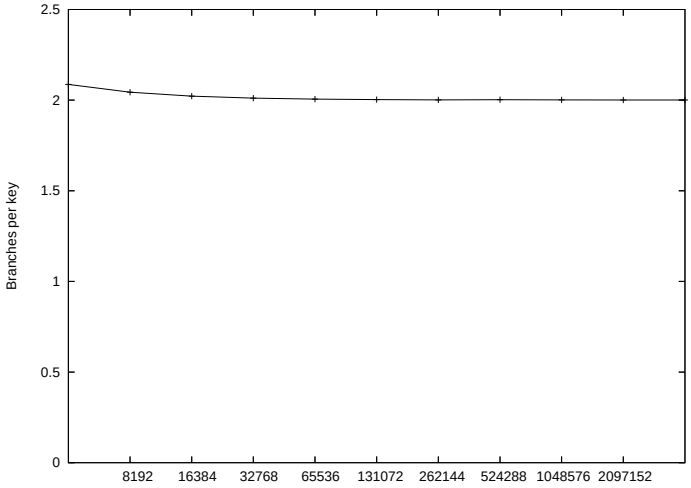
(b) Instructions per key



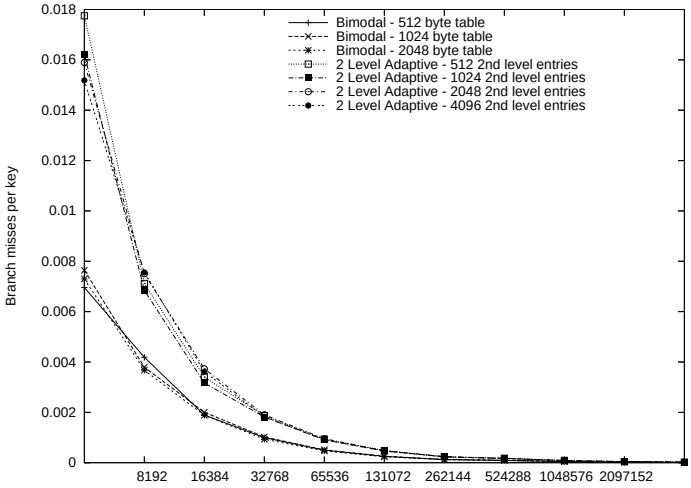
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

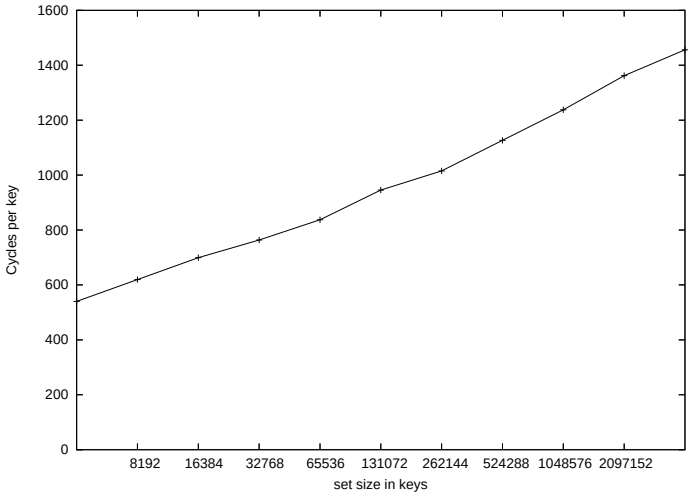


(e) Branches per key

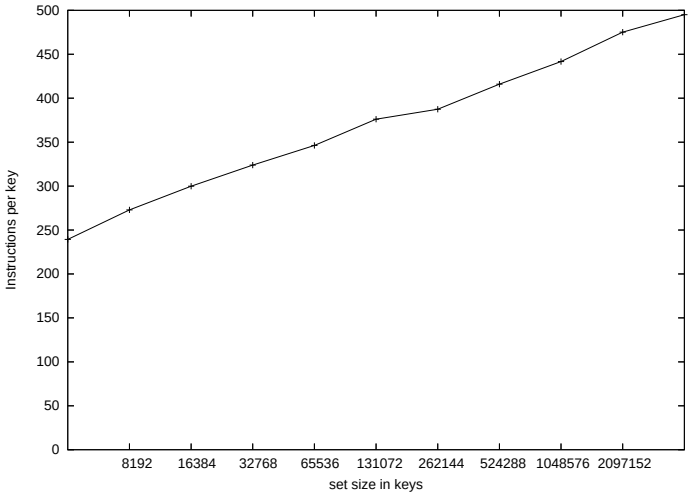


(f) Branch misses per key

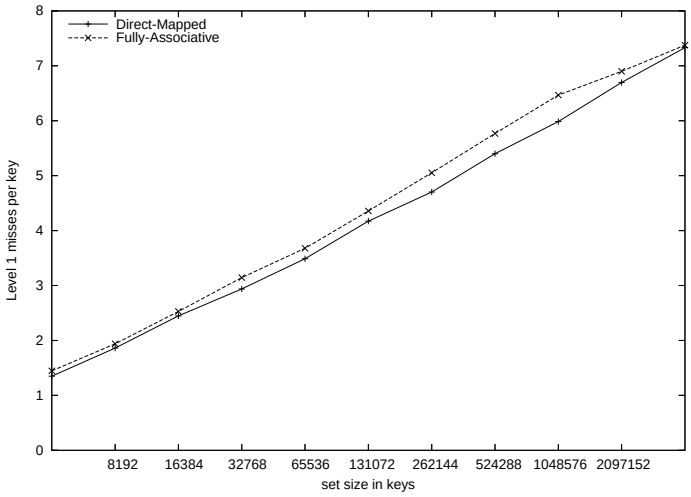
Figure A.27: Simulation results for aligned memory-tuned radixsort



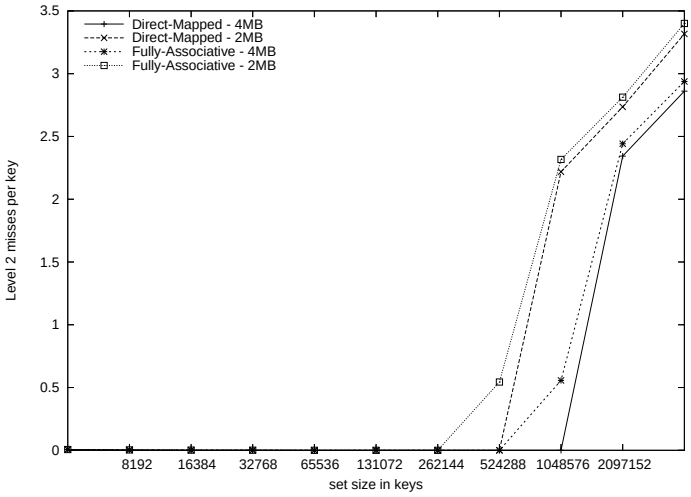
(a) Cycles per key



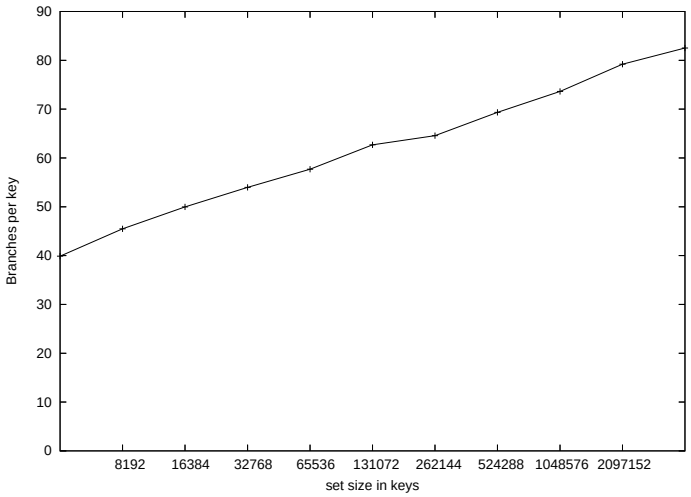
(b) Instructions per key



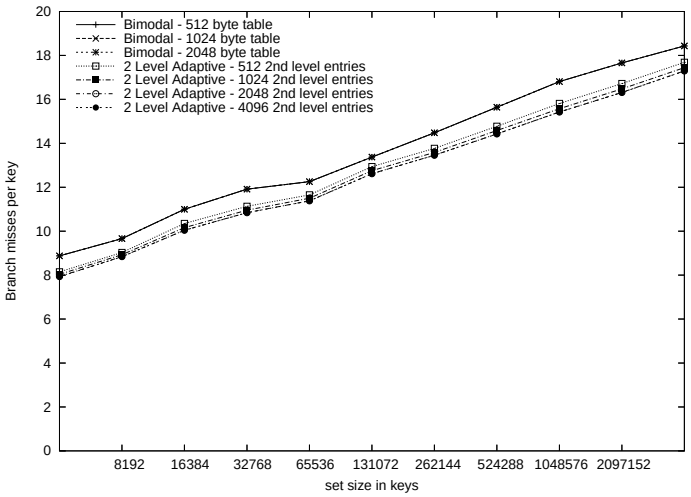
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key

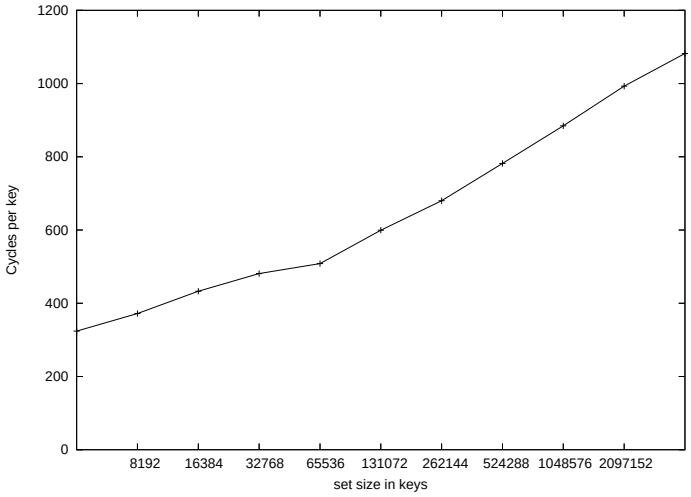


(e) Branches per key

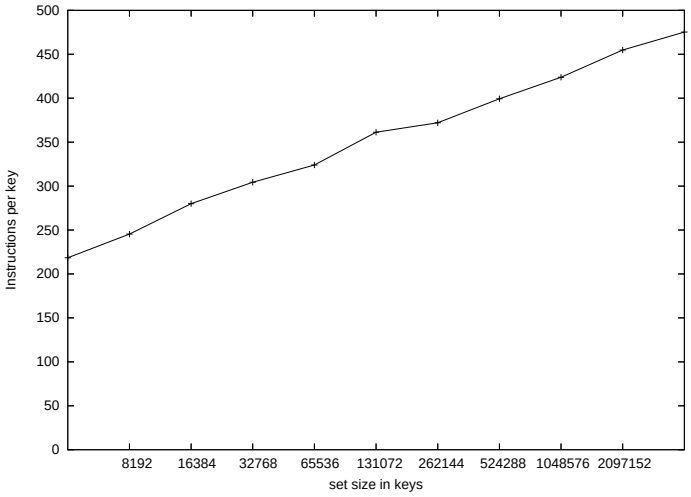


(f) Branch misses per key

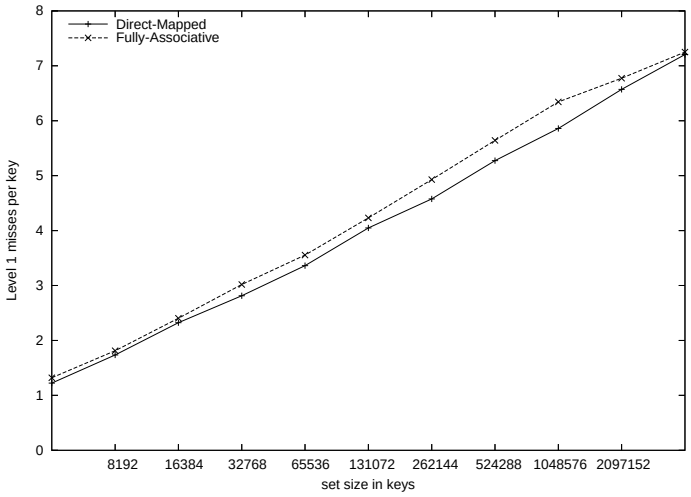
Figure A.28: Simulation results for shellsort



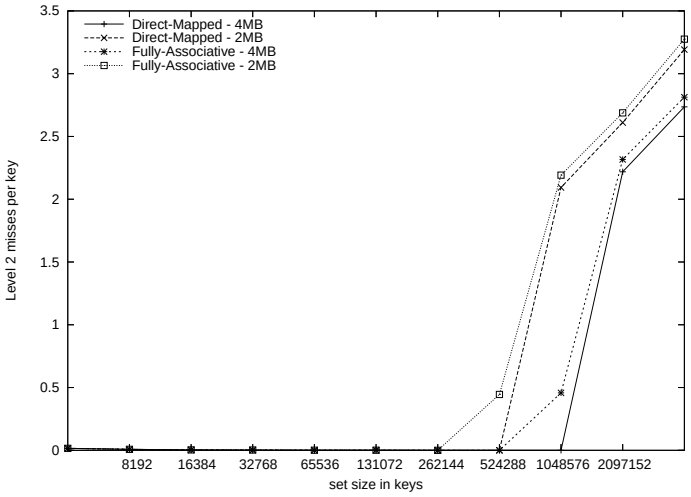
(a) Cycles per key



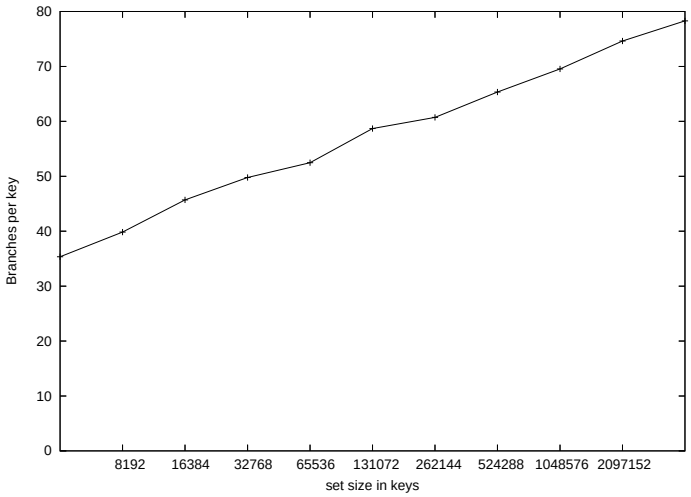
(b) Instructions per key



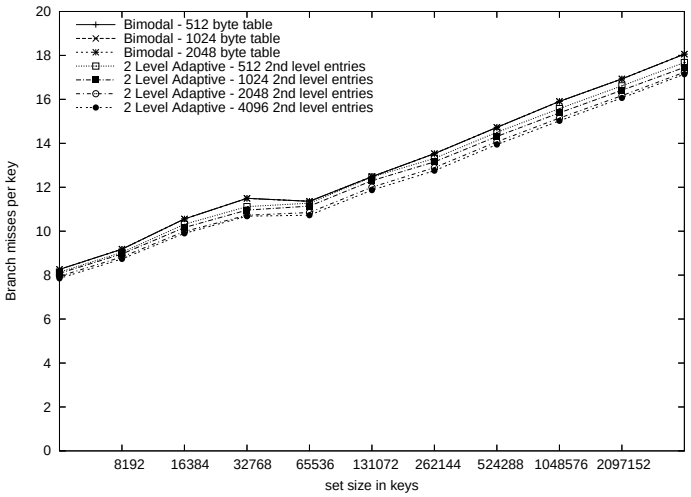
(c) Level 1 cache misses per key



(d) Level 2 cache misses per key



(e) Branches per key



(f) Branch misses per key

Figure A.29: Simulation results for improved shellsort

Appendix B

Bug List

B.1 Results

To reduce the costs of creating the array of random numbers, this time was measured and subtracted from the results. While this works as expected for instruction count, the effects on cache misses is less clear. Subtracting level 2 cache misses incurred during filling removes compulsory misses from the equation. For shorter lists, the compulsory misses are completely removed. For longer lists, those greater than the size of the cache, compulsory misses come back, since the keys at the start of the array are flushed from the cache by the time they begin to be sorted. LaMarca's work centers around reducing conflict misses, however, so this is not necessarily a problem, just something to be aware of.

Bibliography

[Austin 97] Todd Austin & Doug Burger. *SimpleScalar Tutorial (for tool set release 2.0)*, 1997.

[Austin 01] Todd Austin, Dan Ernst, Eric Larson, Chris Weaver, Ramadass Nagarajan Raj Desikan, Jaehyuk Huh, Bill Yoder, Doug Burger & Steve Keckler. *SimpleScalar Tutorial (for release 4.0)*, 2001.

[Austin 02] Todd Austin, Eric Larson & Dan Ernst. *SimpleScalar: An Infrastructure for Computer System Modeling*. Computer, vol. 35, no. 2, pages 59–67, 2002.

[Bentley 93] Jon L. Bentley & M. Douglas McIlroy. *Engineering a sort function*. Softw. Pract. Exper., vol. 23, no. 11, pages 1249–1265, 1993.

[Burger 97a] Doug Burger. *SimpleScalar version 2.0 installation guide*, 1997.

[Burger 97b] Doug Burger & Todd M. Austin. *The SimpleScalar tool set, version 2.0*. SIGARCH Comput. Archit. News, vol. 25, no. 3, pages 13–25, 1997.

[Burger 97c] Doug Burger & Todd M. Austin. *The SimpleScalar Tool Set, Version 2.0*, 1997.

[Friend 56a] Edward H. Friend. *Sorting on Electronic Computer Systems*. J. ACM, vol. 3, no. 3, page 152, 1956.

[Friend 56b] Edward H. Friend. *Sorting on Electronic Computer Systems*. J. ACM, vol. 3, no. 3, pages 134–168, 1956.

[Hoare 61a] C. A. R. Hoare. *Algorithm 63: partition*. Commun. ACM, vol. 4, no. 7, page 321, 1961.

[Hoare 61b] C. A. R. Hoare. *Algorithm 64: Quicksort*. Commun. ACM, vol. 4, no. 7, page 321, 1961.

[Hoare 62] C. A. R. Hoare. *Quicksort*. Computer Journal, vol. 5, no. 1, pages 10–15, 1962.

[Intel 01] Intel. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Rapport technique, 2001.

[Intel 04] Intel. *IA-32 Intel Architecture Optimization - Reference Manual*. Rapport technique, 2004.

[Knuth 97] Donald E. Knuth. The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[Knuth 98] Donald E. Knuth. The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., 1998.

[LaMarca 96a] Anthony LaMarca & Richard Ladner. *The influence of caches on the performance of heaps*. J. Exp. Algorithmics, vol. 1, page 4, 1996.

[LaMarca 96b] Anthony G. LaMarca & Richard E. Ladner. *Caches and algorithms*. PhD thesis, University of Washington, 1996.

[LaMarca 97] Anthony LaMarca & Richard E. Ladner. *The influence of caches on the performance of sorting*. In Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, pages 370–379. Society for Industrial and Applied Mathematics, 1997.

[Lowney 93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell & John Ruttenberg. *The multiflow trace scheduling compiler*. J. Supercomput., vol. 7, no. 1-2, pages 51–142, 1993.

[McFarling 93] Scott McFarling. *Combining Branch Predictors*. Rapport technique TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[Patterson 90] David A. Patterson & John L. Hennessy. Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc., 1990.

[Sedgewick 78] Robert Sedgewick. *Implementing Quicksort programs*. Commun. ACM, vol. 21, no. 10, pages 847–857, 1978.

[Sedgewick 96] Robert Sedgewick. *Analysis of Shellsort and Related Algorithms*. In Proceedings of the Fourth Annual European Symposium on Algorithms, pages 1–11, London, UK, 1996. Springer-Verlag.

[Sedgewick 02a] Robert Sedgewick. Algorithms in c. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Sedgewick 02b] Robert Sedgewick & Jon Bentley. *Quicksort is Optimal*. 2002.

[Shell 59] D. L. Shell. *A high-speed sorting procedure*. Commun. ACM, vol. 2, no. 7, pages 30–32, 1959.

[Smith 81] James E. Smith. *A study of branch prediction strategies*. In Proceedings of the 8th annual symposium on Computer Architecture, pages 135–148. IEEE Computer Society Press, 1981.

[Uht 97] Augustus K. Uht, Vijay Sindagi & Sajee Somanathan. *Branch Effect Reduction Techniques*. Computer, vol. 30, no. 5, pages 71–81, 1997.

[Wolfe 89] M. Wolfe. *More iteration space tiling*. In Proceedings of the 1989 ACM/IEEE conference on Supercomputing, pages 655–664. ACM Press, 1989.

[Xiao 00] Li Xiao, Xiaodong Zhang & Stefan A. Kubricht. *Improving memory performance of sorting algorithms*. J. Exp. Algorithmics, vol. 5, page 3, 2000.