Fast Sorting on Modern Computers

Paul Biggar
B.A. (Mod.) Computer Science
Final Year Project April 2004
Supervisor: Dr. David Gregg

Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

____ July 9, 2004

Paul Biggar

Acknowledgements

I would like to thank my supervisor, Dr. David Gregg, for his help and advice during this project. I'd also like to thank those who read my first draft, and my friends and family for their support during the last year, and my entire time at college. Thank you.

Abstract

This project attempts to create and analyse sorts designed to take advantage of features of modern processors. The work on cache-conscious sorting by LaMarca and Ladner is the starting point; this is implemented and analysed using a hardware simulator. Results relating to a variety of cache and branch predictor configurations are obtained and analysed for a variety of sorts, beginning with insertion and selection sorts, bubblesort and shakersort, and continuing to heapsort, mergesort, quicksort and radixsort. Actual and expected results are compared together and to the results of LaMarca's research.

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 by discussing the cache implications of several algorithms, including sorts. This project attempts to investigate these claims and to test and analyse the high performance sorts based on them.

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

In addition to repeating LaMarca's work, this project 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 results are discovered during the course of the project. 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 mergesort, and removing a sentinel and bounds check from heapsort.

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 project.

Chapter 3 discusses the method and framework used in this project, 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 7 discuss base algorithms and cache-conscious improvements for heapsort, mergesort and quicksort. Results are presented and explained, and branch prediction implications of the changes are discussed.

Chapter 8 presents a radixsort implementation and discusses its performance in the areas of instruction count, cache-consciousness and branch prediction.

Chapter 9 presents conclusions and contributions developed over the course of this project.

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, 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 order across the sort. All the elementary sorts are stable, as is mergesort, heapsort and radixsort. Quicksort is 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 in this project is taken from [**?**]. 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. Items in this project were `unsigned`

```
#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

`integer`s, and 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 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

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

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, 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.

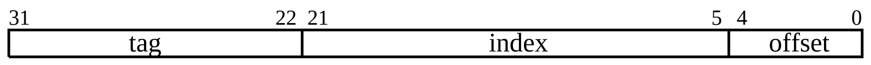


Figure 2.2: Addressing

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

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

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 optimisations 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 simulations in this project, 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 in this project 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 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

[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.

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.

For this project, I 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 project is only interested in dynamic predictors, due to their high levels of accuracy. The other types are none-the-less discussed below, as an introduction to dynamic predictors. A good discussion on predictor types is [**?**], 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 zero.

In the sequence T-T-T-N-T-T-T-N, a 2-bit predictor will only mispredict once. 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 slower 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.

An important thing to note is that the time the algorithms take to execute are proportion to their Order, 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 project. From the point of view of a Final Year Project, this is a description of what I did all year, what I spent my time on, what went right and wrong during the project, and what I learned. The first section discusses the steps involved in creating the project, programming the sorts and getting results. The second section discusses tools that I used or created in the course of the project.

3.1 Method

The following is the initial project description:

> 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. One weakness of classical sorting algorithms is that they are not designed for modern computers.
>
> Cache memories, pipelines, and branch prediction have greatly changed the relative strengths of different algorithms. For example, comparison based sorting algorithms (such as Quicksort) can be inefficient on modern machines, because they involve difficult to predict branches.
>
> The goal of this project is to design, implement and experiment with sorting algorithms designed to make the best use of modern architectural features. Several cache-conscious sorting algorithms have been proposed, and the first step would be to implement these algorithms and compare their performance with traditional algorithms.

> The next step is to consider modifications to the algorithms to make better use of the processor pipeline and multiple execution units and reduce branch mispredictions. The project will also involve a detailed study of the behaviour of the algorithms using a processor simulator or performance counters on a real machine.
>
> To do this project, you should have a reasonable knowledge of computer architecture and traditional sorting algorithms. You should also be enthusiastic about algorithm design, and making things go faster. All programming is in C or C++.

After discussion with my supervisor, the following project goals were agreed upon:

- Implement heapsort, mergesort, quicksort, radixsort, and the four elementary sorts: insertion sort, selection sort, bubblesort and shakersort.
- Simulate these sorts using the SimpleScalar architecture simulator, using a variety of branch predictors and cache configurations.
- Implement cache-conscious sorts from [?] and [?].
- Investigate and analyse the increase in speed, reduction in cache misses, and change in branch prediction accuracy as a result of these changes.

Several steps were necessary to prepare for the project. The first of these was to compile SimpleScalar. A guide to repeating this is included in Section D. A framework was required by which I could test sorts as they were developed. This is discussed in Section 3.2.3. It was also necessary to fully understand the technologies to be used in the project. Several texts were used for this: cache and branch predictors are both in [?]; branch predictors in [?], as well as in [?] and [?]; sorts are discussed in [?] and [?]; cache-conscious sorts are discussed [?], [?] and [?].

There were several steps involved in writing each iteration of the algorithms. The elementary sorts and radixsort had just one of these iterations; the other sorts had more than one[1]. A flowchart of these steps appears in Figure 3.1 on the following page.

The first step was to write the algorithm. [?] 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.

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

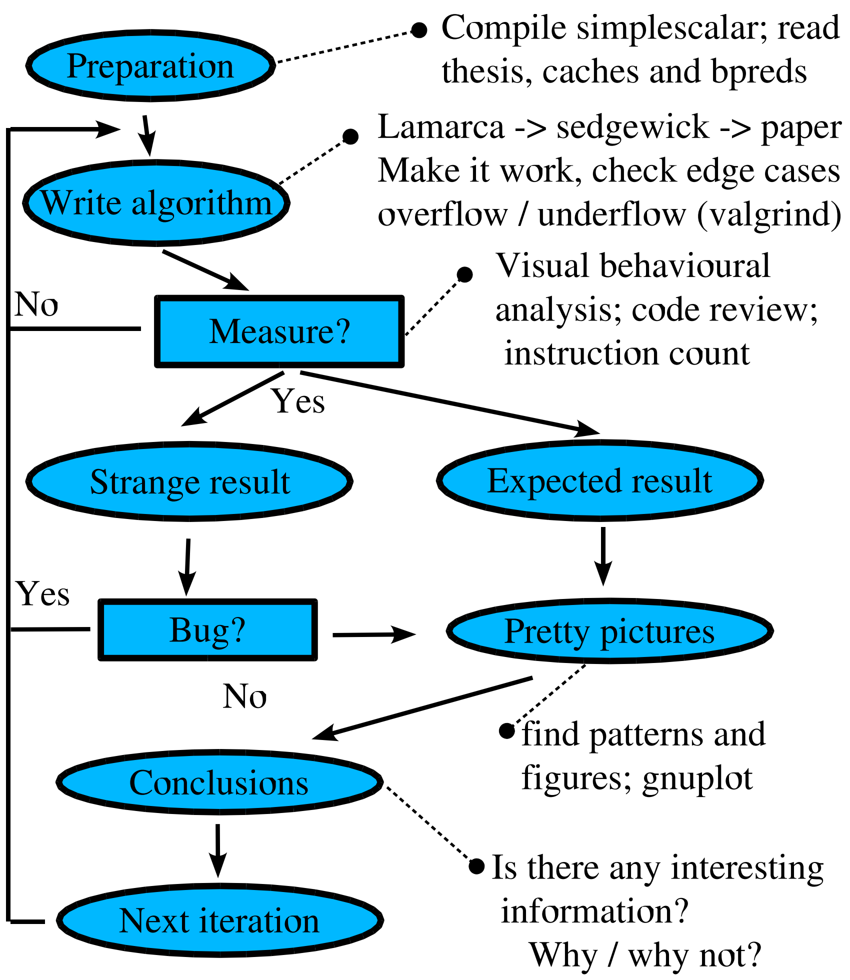

Preparation
Compile simplescalar; read thesis, caches and bpreds
Write algorithm
Lamarca -> sedgewick -> paper
Make it work, check edge cases
overflow / underflow (valgrind)
No
Measure?
Visual behavioural analysis; code review; instruction count
Yes
Strange result
Expected result
Yes
Bug?
Pretty pictures
No
find patterns and figures; gnuplot
Conclusions
Is there any interesting information?
Why / why not?
Next iteration

Figure 3.1: Flowchart describing the development of an algorithm

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

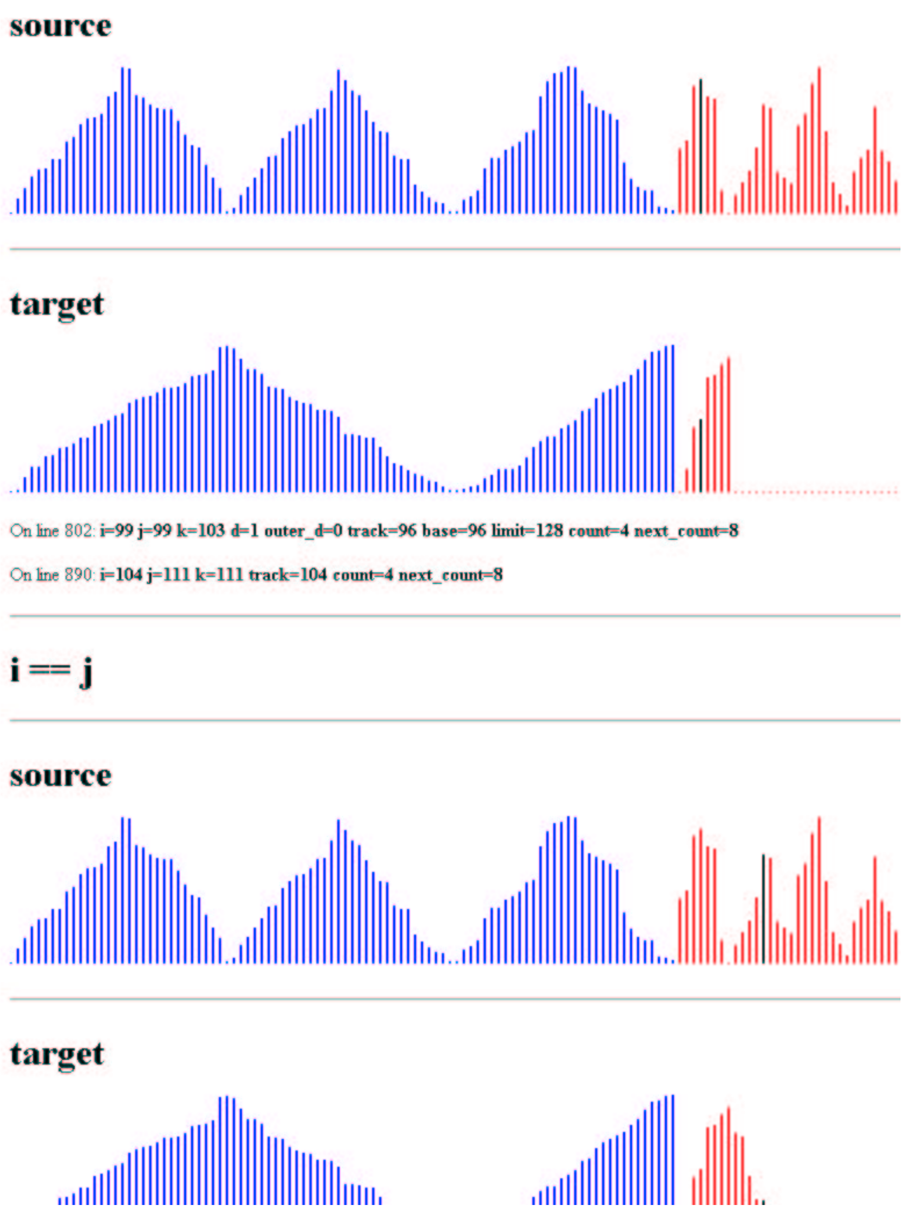
Early in the project, it was decided that exceeding the bounds of an array during a sort was to be forbidden. No serious sorting library or application could acceptably work in this manner, and any results obtained while allowing this were only of academic interest. It isn't possible, from a normal run 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. 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 (see Section B.1) 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



```
On line 802: i=99 j=99 k=103 d=1 outer_d=0 track=96 base=96 limit=128 count=4 next_count=8

On line 890: i=104 j=111 k=111 track=104 count=4 next_count=8
```

i == j

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

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.

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. SimpleScalar can be found at *http://www.simplescalar.com.* Documentation on how to use and modify SimpleScalar are [**?**], [**?**], [**?**], [**?**] and [**?**].

The installation of SimpleScalar can be difficult. A step by step description for installing it on an x86 linux machine is in Section D.

SimpleScalar works by compiling code into its own instruction set. It provides a compiler - a port of gcc 2.63 - with related tools such as a linker and assembler. Several simulators are then provided: `sim-fast` only counts instruction, quickly; `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 this project.

The output of SimpleScalar is human readable (see Section B.3 for an example) but is not conducive to collecting large amounts of data. To that end, a script was written to run each required set of simulations. The script was written in perl, and recorded data from the simulations into a single file for later retrieval. The source code to the script, `simple`, is in section B.1. A sample list of commands executed by `simple` are in section B.2. The data file produced was suitable to reading with a `diff` program.

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 cache with a 32 byte cache line. The remainder of the settings were taken from the DEC Alpha: 8KB level 1 data cache, 8KB level 1 instruction cache and shared instruction and data level 2 cache.

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. The source code to the script to do this is in Figure B.4. Once the data was converted, a script to run gnuplot (see Section B.5 for an example) was required. 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 project. Imaginatively called `main.c`, this program provided the means for testing and (roughly) timing each sort. Figure 3.3 on the following page contains this code.

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/*.

```
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(on2_insertsort, "O(N squared) Insertion");
time_sort(on2_insertsort, "O(N squared) Insertion");
```

Figure 3.3: Key sections of Main.c

3.2.5 perfex

`Perfex` is a tool used to access performance counters on a Pentium 4. By default, it prints out the number of cycles[2] which a program takes to execute. This was used to create the graphs comparing the running time of sorts. The gnuplot scripts for this were short and easily written.

Perfex was an alternative to using SimpleScalar. By using its hardware registers, actual real world results could have been measured. Using perfex instead of SimpleScalar was an option reviewed at the start, but the difficulty involved in setting up a machine to use it was considerable. Additionally, because perfex measures results on a real machine, only one type of cache and branch predictor could have been measured. Since access was provided late in the project, limited measurements were taken to supplement the SimpleScalar results.

3.2.6 Incidental Tools

The algorithms in this project were written in C using `vim`, a powerful text editor. `gcc` was used to compile the framework, when not using SimpleScalar. The report was written using LaTeX, and diagrams were written is `postscript`, or drawn using `OpenOffice.org`, an open-source office suite. Many files had to be edited in the same way, and this was accomplished either using `vim` macros, or by writing a short `perl` script.

3.3 Implementation of sorts

The `C` code written for this project was approximately 8000 lines long. Appendix B.6 lists many of the key parts of this code.

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 [?]. A flow control diagram and step by step guide is provided, but very little detail of how the sort actually works is included.

[2]The processor automatically maintains this figure in a 40-bit register called P4_TSC (Time Stamp Counter), and perfex prints this value after running the program

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 of the sort are described in assembly, and while they can be written in C, the result is a very low-level version. I attempted to rewrite the algorithm without `goto` statements and labels, but 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 optimisations 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 optimised 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 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. 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 8-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 tiled mergesort was written. This attempted to completely sort in the level 1 cache, then to completely sort in the level 2 cache. Double multi-mergesort was a combination of this and of multi-mergesort. The rewrite was based on the base mergesort, and was written far more cleanly, 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 6.5.3.

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.

Each of the three $O(NlogN)$ sorts was treated in this fashion. The mergesorts accounted for over 3000 lines of the 8000 lines of code in this project. Key portions of this code are shown in Section B.6.

3.4 Future work

SimpleScalar ran very slowly, with each simulation run taking at least $2\frac{1}{2}$ hours. Several steps could be taken to lower this time. Firstly, compiling SimpleScalar with the `-O3` flag, which performs a significant number of optimisations on the program. This was not done because of the difficulty of compiling the program in the first place.

`lib-cheetah` is a SimpleScalar library used to simulate multiple caches in the same simulation. Over half the simulation time was taken by four cache simulations, and running them concurrently might have significantly decreased the running time. This was not a significant worry during this project, but decreasing the running time would be necessary if the either the number of simulations or the number of sorts was to change dramatically.

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 mix 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 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.

As well as the number of cycles, Pentium 4 performance registers can also provide most of the same information as SimpleScalar, with different parameters. For example, the cache line of a Pentium 4 machine is 128 bytes whereas 32 byte cache lines were used in SimpleScalar. The addition of real world data would strengthen the results from this project.

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. They are insertion sort, selection sort, 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 on the next page.

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.

```
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

Expected Performance

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

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 is far 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 equally bad performance as a result. Since it is smaller than the array, the performance of the level 1 cache should simulate the performance of the level 2 cache.

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 array traversal has an average of $N/2$ comparisons, and approximately $log_2(N/2)$ of them should be taken, while the rest are not-taken. The taken predictions will be misses, while the rest should be hits, so it is expected that there will be $logN$ misses per key.

This number is not immediately obvious. Consider a single step of the selection sort. The first key to be sorted is, on average, larger than half the keys and smaller than the other half, so it will have to travel across half the array to find a key smaller than it, on average. Once it finds a smaller key, this will be larger than a quarter of the keys in the array, and so it will need to travel half of the remaining array, on average, before it finds it. The search will continue like this until the entire array has been covered. Comparing two keys and discovering that the new key is not smaller than the current smallest is the norm; every time a smaller key is found a branch misprediction will occur (if two or more keys in a row are successively smaller, this may not be the case). The length of the array to be considered in $N/2$ on average, and an average total of $Nlog_2(N/2)$ branch misses will occur. This assumes randomly sorted keys.

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

[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

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.

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 bubble sort 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 ended 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 $O(NlogN)$ misses, each of bubblesort's comparisons has the potential to be a miss, and not just in the pathological case. However, as the sort continues and the array becomes more sorted, the number of misses should decrease. The number of flow control misses should be very low.

4.4 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 in Figure 4.4 on the following page.

4.4.1 Testing

Expected Performance

The instruction count of shakersort is expected to be exactly the same as bubblesort. The version of shakersort used here, however, does not incorporate the improvement to end the sort early if the array is already sorted. This will probably slightly reduce the instruction count as the sort is unlikely to end early for

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

        // backwards
        for (int j = i ; j < k; j++)
            if (less(a[j+1], a[j])) exch(a[j+1], a[j]);
        k--;
    }
}
```

Figure 4.4: Shakersort code

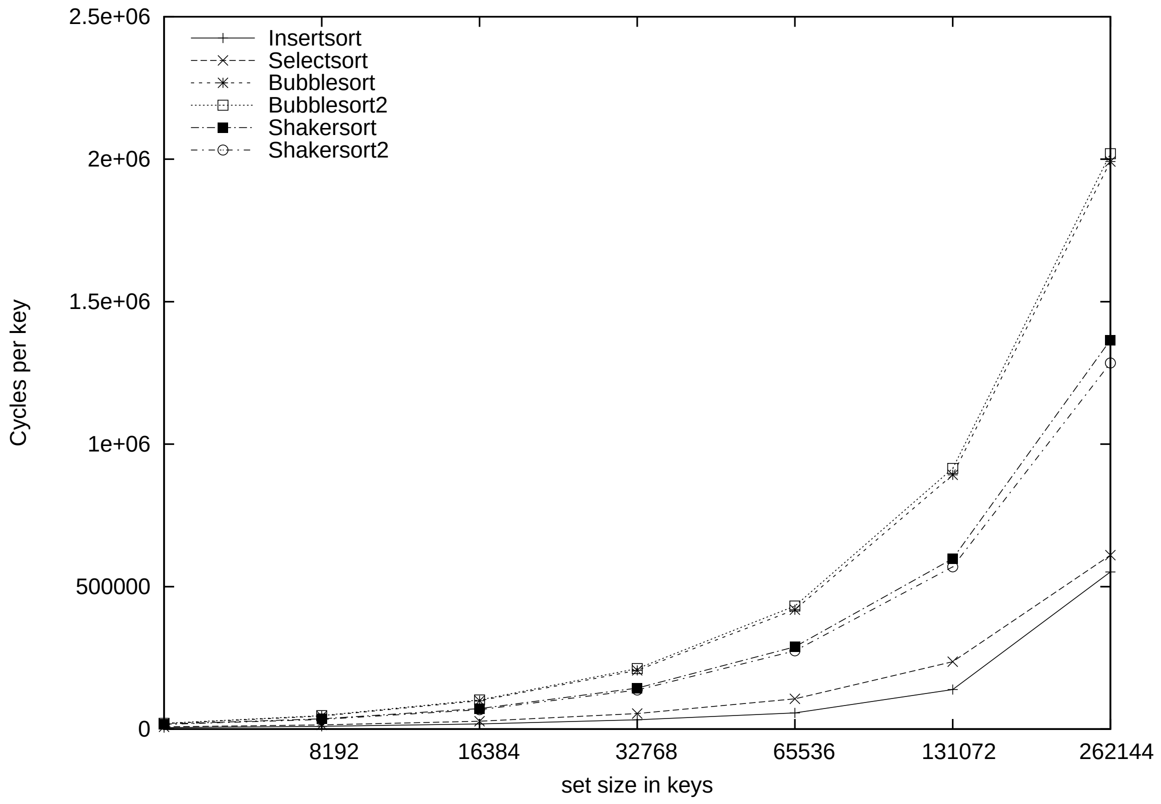
random keys. In addition, the worst case for bubblesort is not a worst case for shakersort and is less likely.

The cache performance of shakersort is expected to be superior to that of bubblesort. 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 array at all.

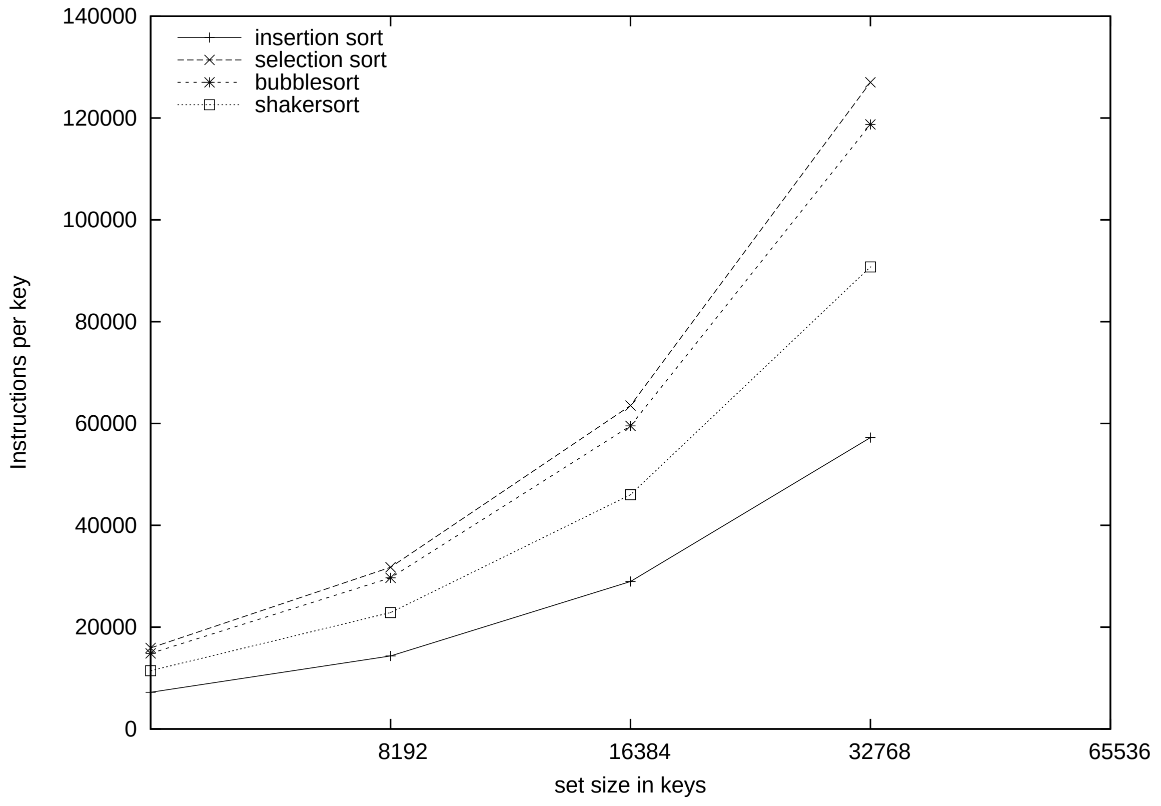
Finally, the branch prediction results are expected to be the same as in bubble sort, with one difference. Keys which must travel the entire way across the array are taken care of earlier in the sort. If the largest key is in the leftmost position, it must be moved the entire way right, which causes misses for every move in bubblesort, since each move is disjoint from the next. In shakersort, however, each in the correct direction will follow a move in the same direction, each of which should be 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.

4.5 Simulation Results

Figure 4.5(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

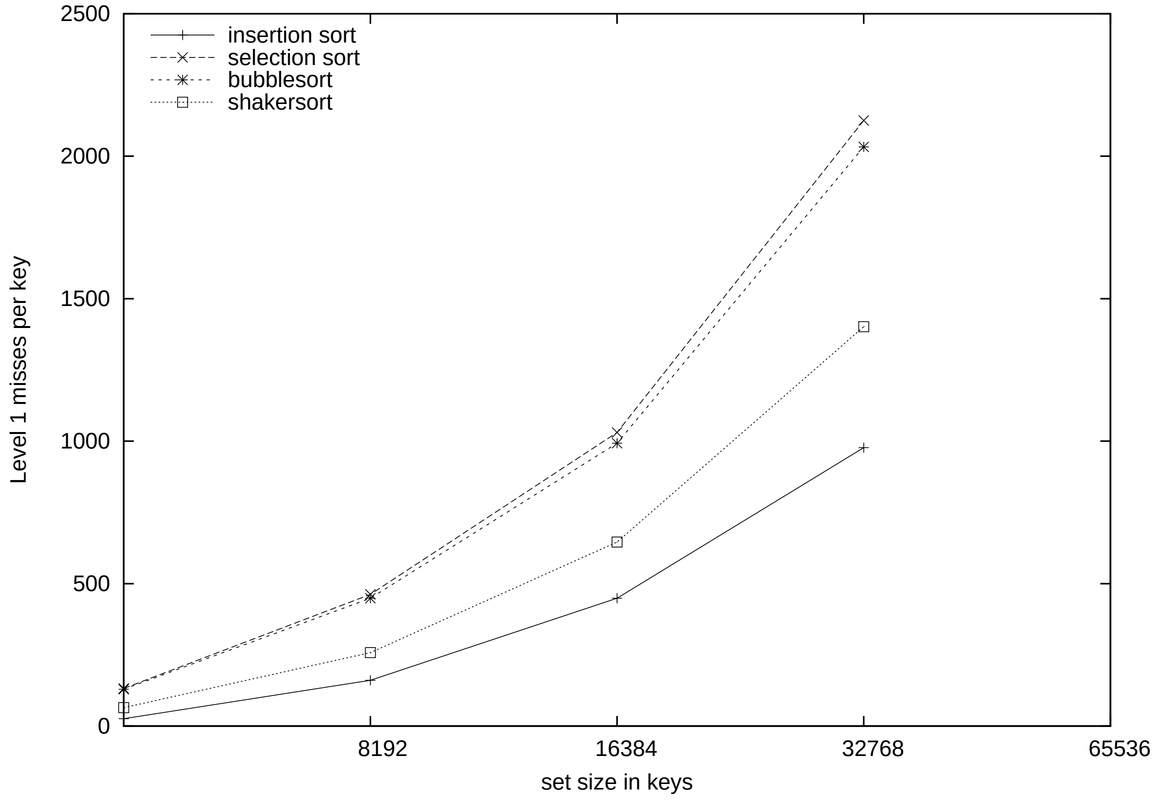


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

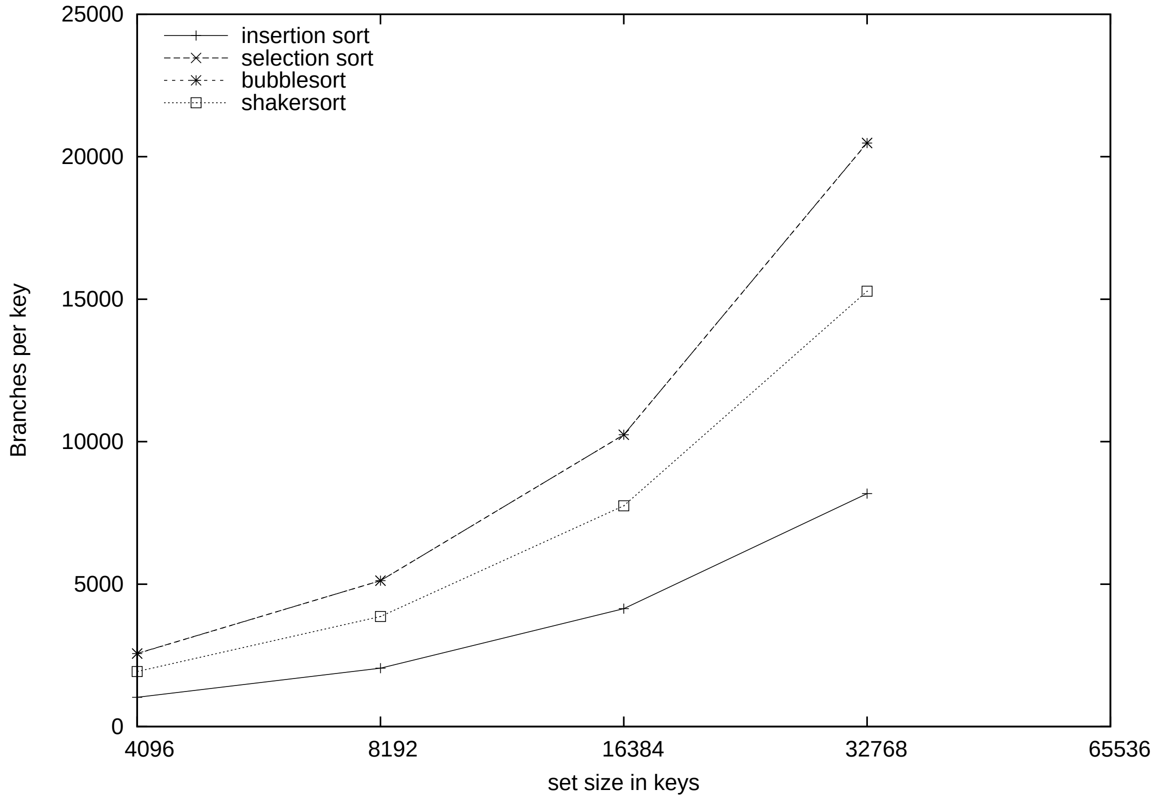


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

Figure 4.5: Simulated Instruction count and empiric cycle count for elementary sorts

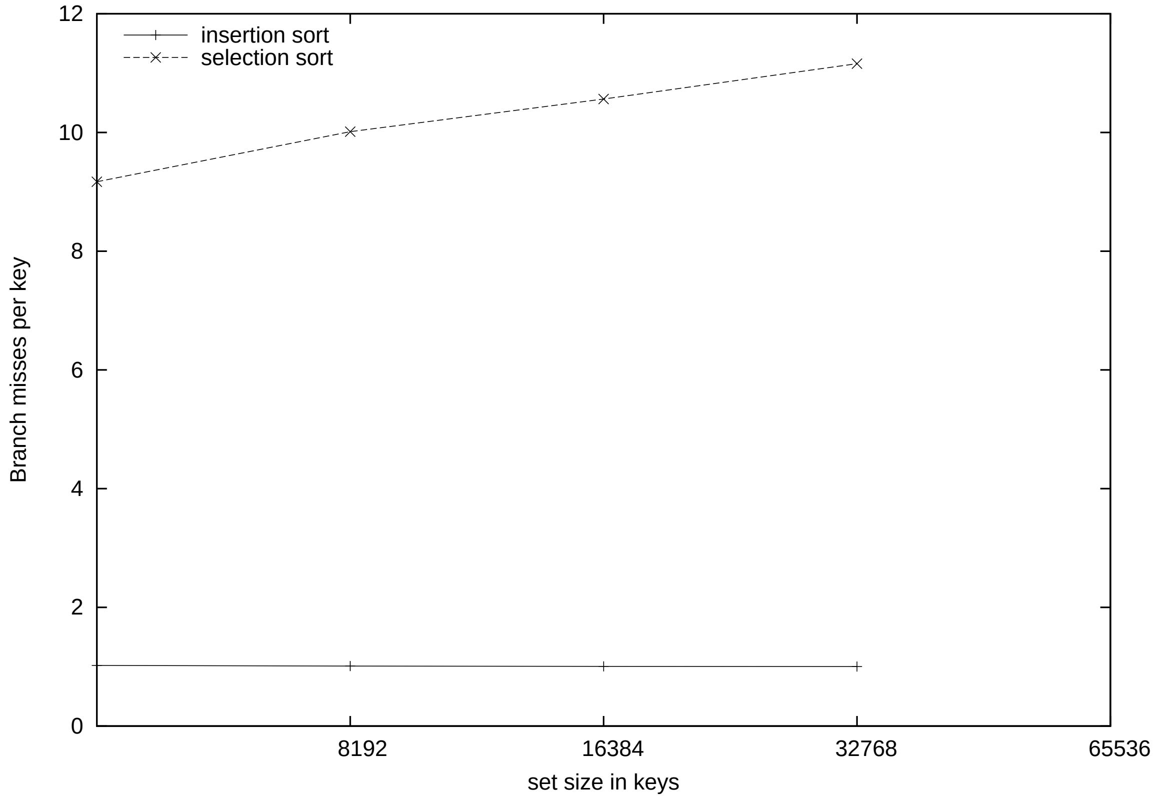


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32 byte cache line and separate instruction cache.

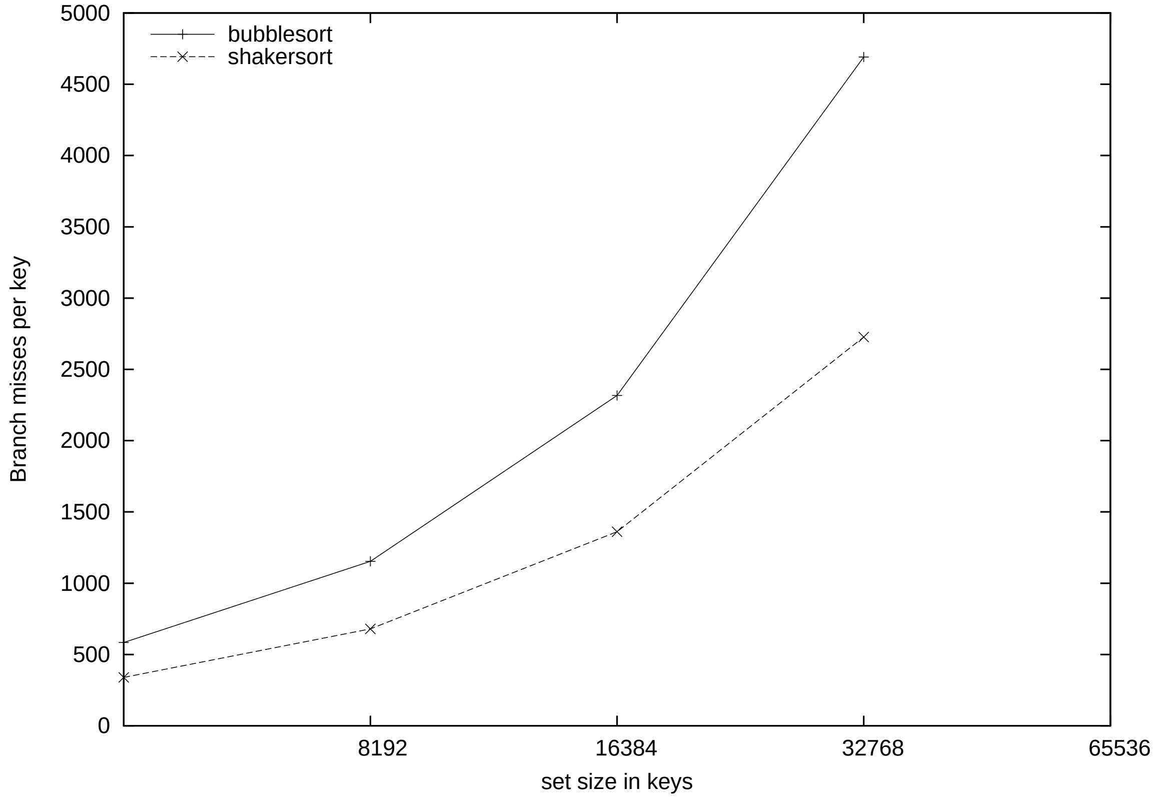


(b) Branches per key - this was simulated using `sim-bpred`.

Figure 4.6: 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.



(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 4.7: Branch Simulation results for elementary sorts

almost the same number of instructions as selection sort, but shakersort has a significant number less.

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 4.6(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 is the worst. 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 (see Section 2.2), reducing its number of misses.

Insertion sort performs best of the four 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 4.6(b) shows the number of branches per key of these sorts, while 4.7 shows the branch misses per key. Branch prediction performance is not surprising. Each sort has a similar number of branches, with insertion sort having half as many as selection and bubblesort, and shakersort being half-way between the two. The number of misses for bubblesort and shakersort are huge; nearly one in four comparisons are misses. As the data sets gets large, there are nearly 100,000 branch misses involved in sorting a single key.

Selection sort, meanwhile, has the same number of branches as bubblesort, but only $O(logN)$ 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.

Another interesting result comes from a comparison between shakersort and selection sort. The number of cycles to sort an array (shown in Figure 4.5(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 exchange-based 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.

4.6 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.

Shakersort doesn't have an opt-out-early clause like bubblesort, and as a result would perform very badly for sorted sets. Adding this enhancement is simple, and would give better results, which would also be more consistent with bubblesort.

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. 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 doing this 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*, which stores its largest key at the top, then removing keys from the top and placing them in their final position in the array.

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*. One node is called the *root*, and the tree branches out from this, 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 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 without any links to other nodes. 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 $N-1$, then 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 is slightly more difficult pointer arithmetic, which cannot be performed in some languages[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 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. For integers, this step should not be done.

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

[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*

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 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 staticly 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. This can be avoided relatively simply with no extra instructions. If the last key in the heap is left in the heap while its 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 optimisation used is to unroll the loop while destroying the heap. This has minimal affect, however.

```
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

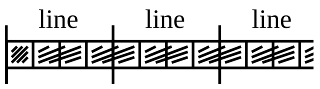
5.2 Cache Heapsort

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 keys being heapified are not related to the last keys that were 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.

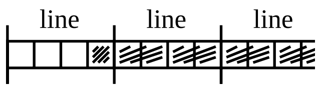
The Repeated-Adds algorithm, however, displays excellent temporal locality. A key has a 50% chance of having same parent, a 75% of the same grandparent and so on. The number of instructions, however, increases significantly, and the sort becomes out-of-place.

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 optimisations 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 half again the height of a 4-heap. 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 optimisation 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 in this project are performed on 32 byte cache lines. As a result, 8-heaps are 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. As a result, he used a 4-heap. To try and emulate his results, extra tests were done, using a 4-heap with a 32 byte cache line, and using a 4-heap using a 16 byte cache line, so that the entire cache line is used. `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 it's 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. However, the 4-heap should perform just as well with a 16 byte cache line.

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 are predictable. An 8-heap has seven comparisons between children, with the result that only an eighth of the branches are predictable.

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.

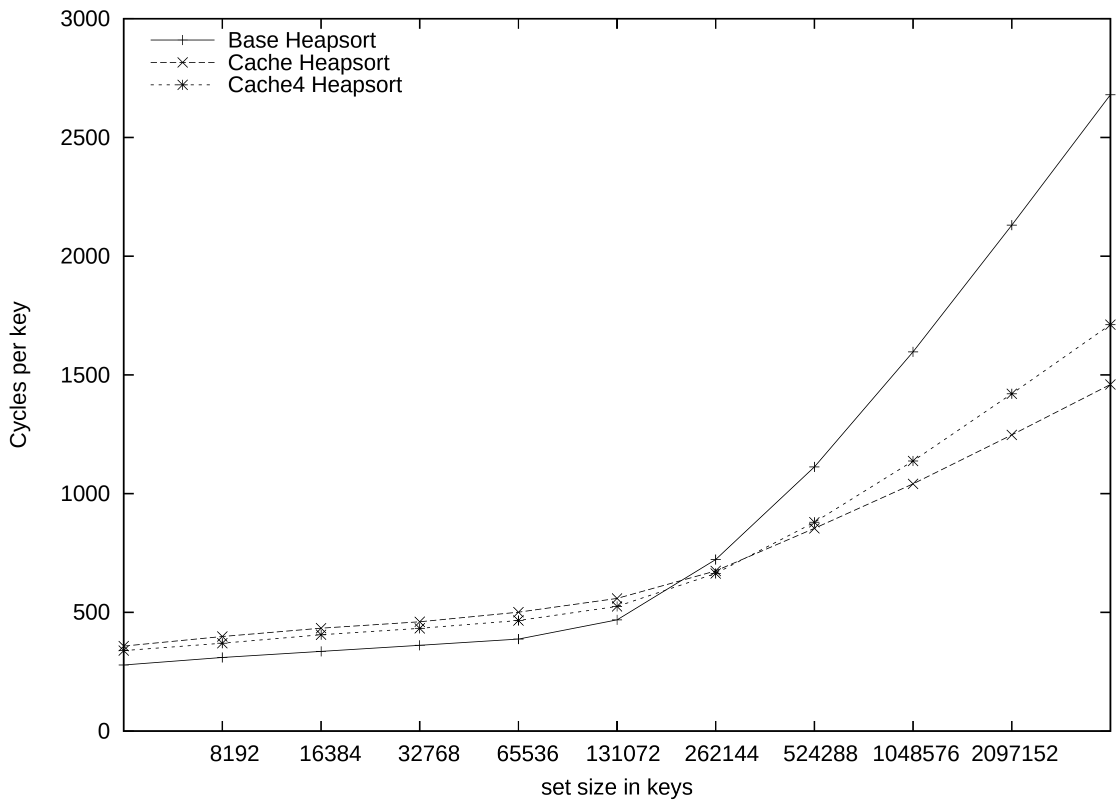
5.3.2 Simulation Results

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, despite the reduction in the height of the heap. This is shown in Figure 5.2(b).

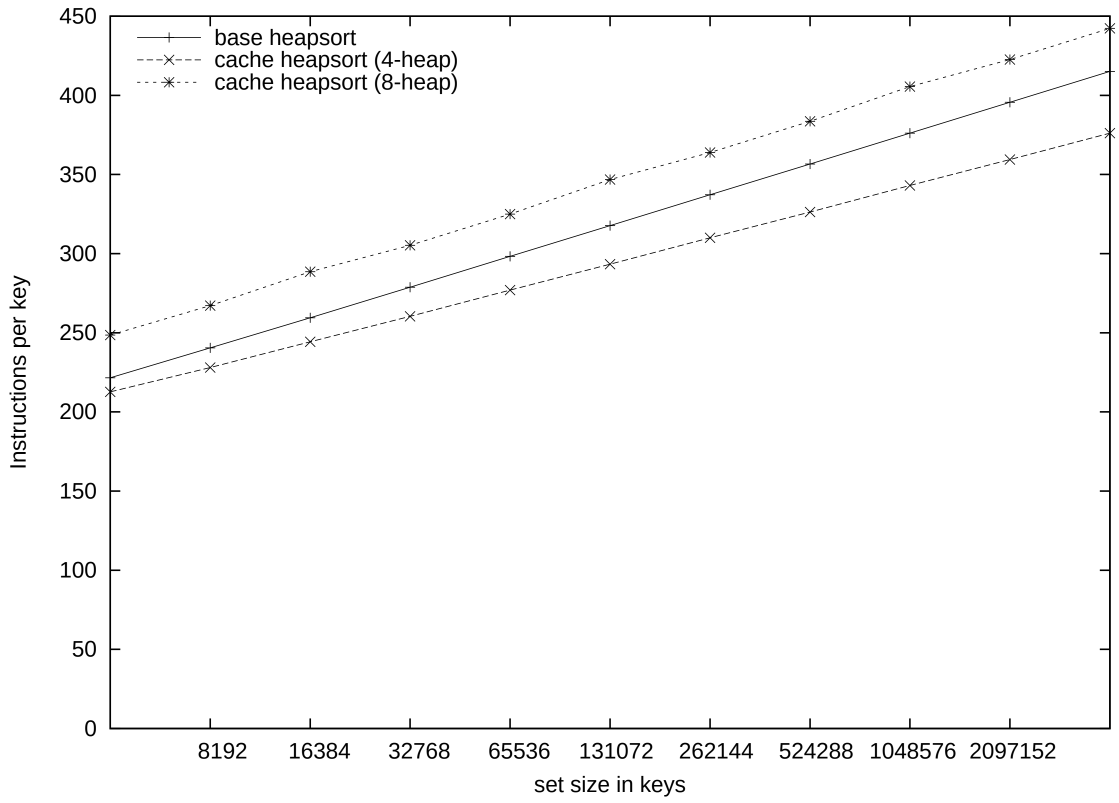
The level 1 cache results, shown in Figure 5.3(a), show the difference between the two types of heap. The slope of the base heapsort result is much greater than all the others, with the 8-heap having the fewest misses. This continues into the level 2 cache, where the flattest slope belongs to the 8-heap. The 4-heap with the 32-byte line has fewer misses than the 16-byte version, which is surprising since the 16-byte version can fit twice as many blocks in the cache, and uses each block fully. This may be due to the extra compulsory misses due to having twice as many blocks to load. The difference between fully associative and direct mapped is only minor.

In the graph, the base heapsort 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 half as many level 2 cache misses of the later, out-of-place sorts (See Figure 5.3(b)).

The branch prediction behaves similarly to the expected results, in that the

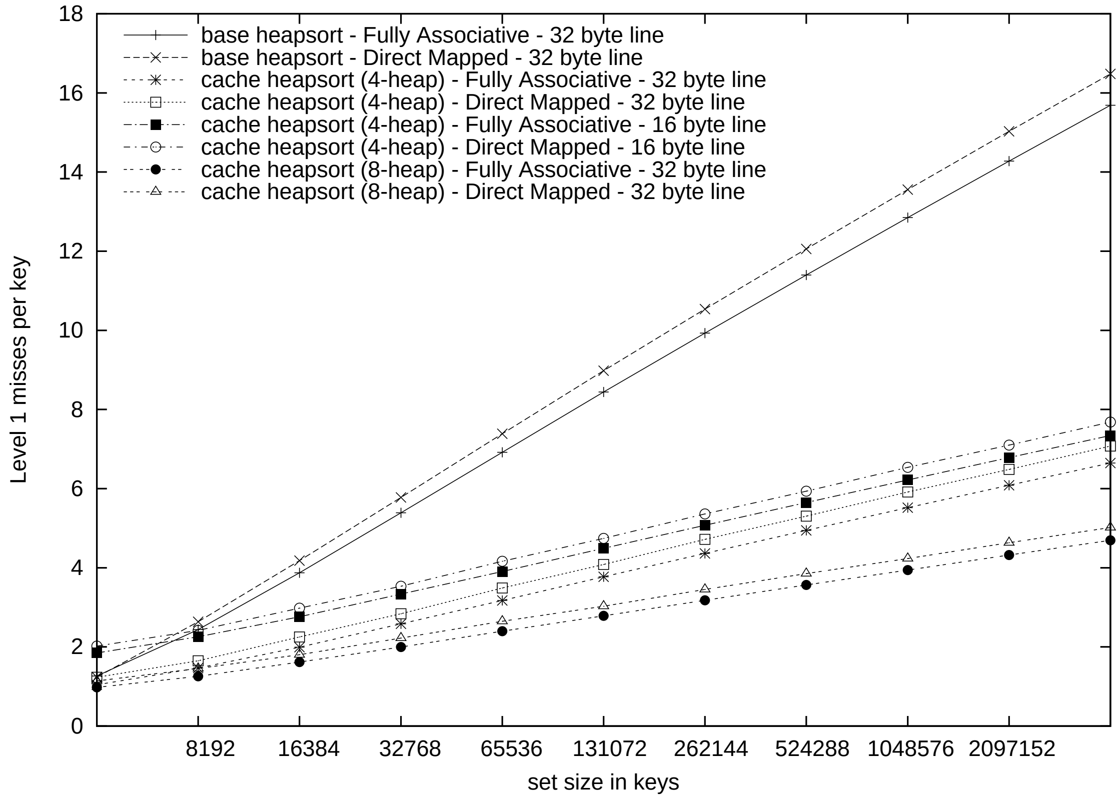


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

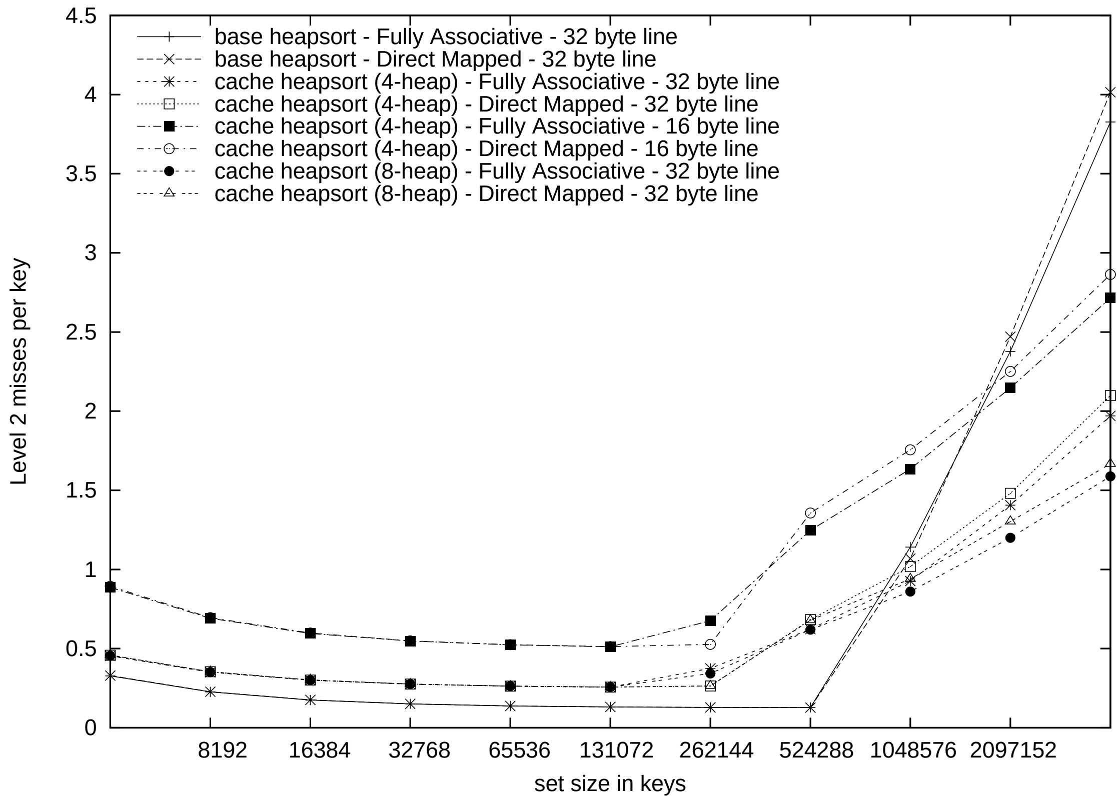


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

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

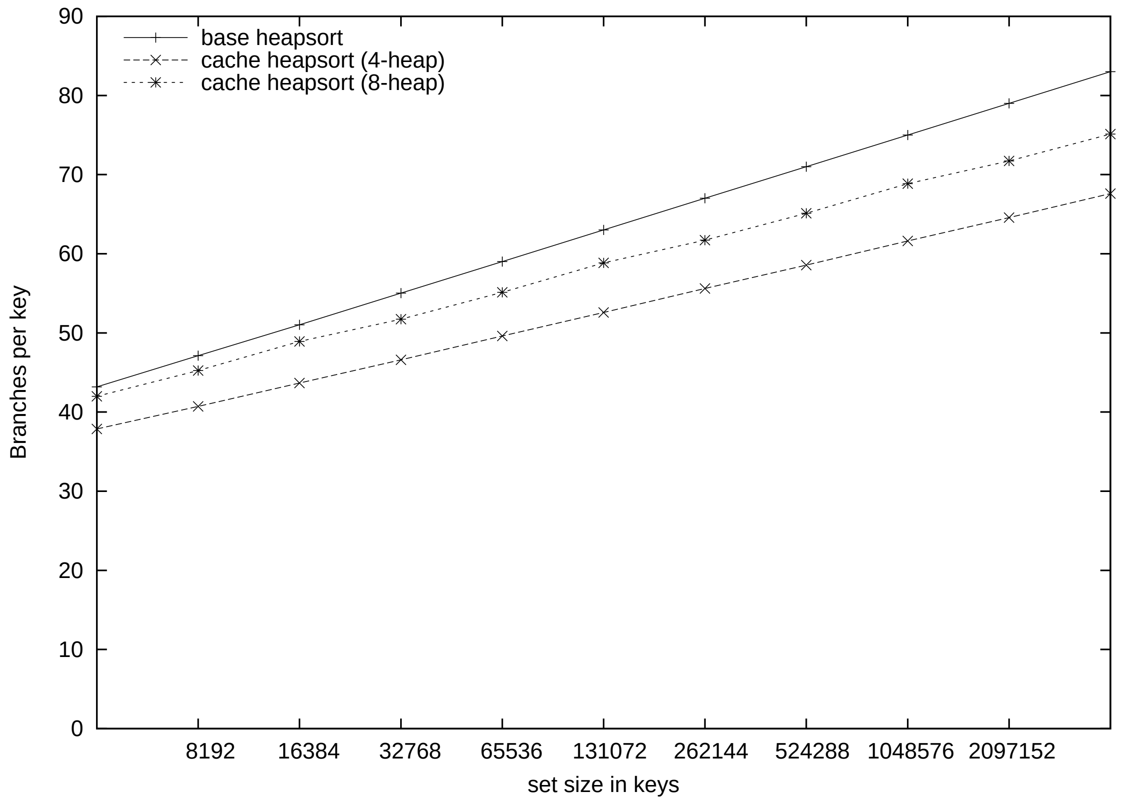


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32 byte cache line and separate instruction cache. Cache heapsort (4-heap) was also tested with a 16 byte cache line

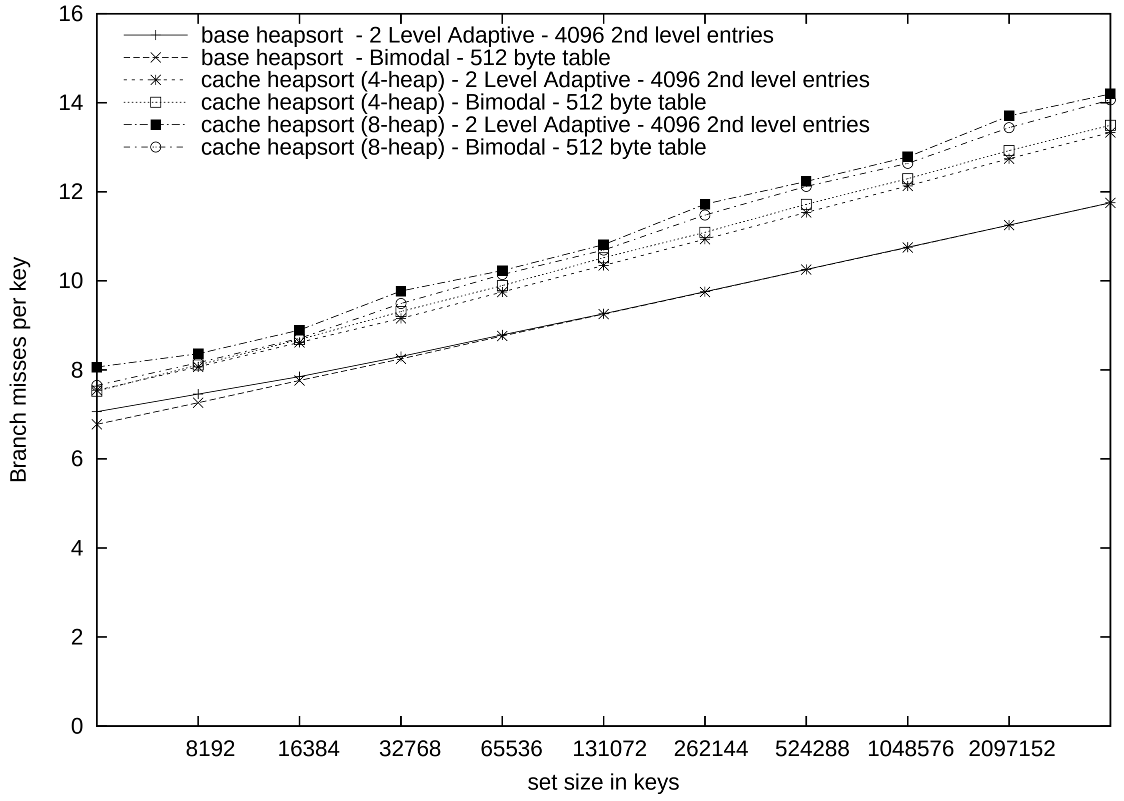


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32 byte cache line. Cache heapsort (4-heap) was also tested with a 16 byte cache line

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

prediction ratio gets worse as the fanout gets larger. However, the difference is minor between the 4-heap and 8-heap, 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 is visible in the base heapsort, though it's amortised for larger data sets. There is no difference between predictors for the 4-heap.

Figure 5.2(a) shows the actual time taken for heap sort on a Pentium 4 1.8GHz machine, which has an 8-way associative 256KB level 2 cache with a 64 byte cache line, a 4-way associative 8KB level 1 data cache, and a maximum branch misprediction delay of 20 cycles.[3] Its static branch prediction strategy is backward taken and forward not taken; its dynamic strategy is unspecified: though [**?**] mentions a branch history so it is likely a two-level adaptive predictor.

The figure shows that the base heapsort performs best while 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 eight 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, before 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, 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. This increase comes when the set size is twice that of cache mergesort when its sharp increase occurs. LaMarca's results do not show this.

Finally, though this is likely to be related the previous differences, base heapsort performs slightly better for small sets than the cache heapsorts do in the

[3]This is determined from the Pentium 4's 20 stage pipeline. This number is not exact, and is probably slightly lower, with the possibility of occasionally rising higher (in the case of an instruction cache miss, which is exceptionally unlikely for small routines like sorts).

results presented here. LaMarca's results indicate that the cache 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 unoptimised algorithm would not suffer as much.

5.4 Future Work

LaMarca later published heapsort code in [**?**]. 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.

Comparing the 4-heap with a 16 byte cache line to the 4-heap with a 32 byte cache line is not as interesting as it would be to compare it to an 8-heap on a 16 byte cache line.

Also, it would be interesting to use performance counters to find out whether the cache or the branch predictors are responsible for the base heapsort being faster than the 4-heap, on the Pentium 4.

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 smallest key on one 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, 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 result 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 [?]. 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 optimisations LaMarca describes on it.

6.1.2 Algorithm S

Knuth's algorithm S is a *straight* merge, and works as described above. It 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 optimisations 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 performs only slightly more slowly.

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 this project uses exact powers of two as set sizes, this problem does not affect the results. LaMarca, however, used set sizes such as 4096000, which at times would suffer from this type of problem.

6.1.3 Base Mergesort

Three optimisations are described by LaMarca for the base mergesort. Firstly, a bitonic sort, as described in [?], 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

[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.

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 19 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 4.5, 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.

Overall, these three optimisations reduced 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, which can be expensive for small arrays. The address of the auxiliary array is then offset such that $index(source) + index(auxiliary) = C$.

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

[2]Unrolling a loop means executing several steps of the loop between iterations, reducing the number of iterations, and lowering the loop overhead, as well as allowing for better scheduling of the instructions.

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 project address the cache by virtual address, and so this optimisation is effective in this case.

The next problem considered 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, the number of keys which fit in the cache 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.

6.3 Multi-mergesort

Multi-mergesort has two phases. The first is the same as tiled mergesort, and sorts the array into lists sized $C/2$. 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 lists grow big, 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, a priority queue is used. 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 key 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. 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

[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.

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 has excellent branch prediction properties, as shown earlier.

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 in the array, 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.

6.4 Double Tiled Mergesort / Double Multi-mergesort

While writing the cache mergesorts, I realized several opportunities for improvement. The first of these is 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.

The second improvement was to fully sort keys in the level 1 cache, before continuing to level 2 cache as in the tiled mergesort. The expected improvement is not as significant as in the case of the level 2 cache, since is the penalty for level 1 misses is not great.

Finally, the merge is rewritten in a more straightforward and readable manner. This was necessary due to the increased complexity of debugging the double mergesorts. During the process, different functions were used to process lists that were exactly the required size and lists which were not, which needed code to handle edge-cases. However, the instruction count improvement was in the double digits, and so these changes were removed in order to have lower instruction cache misses, as well as to reduce the fill time of the branch predictors and reduce the complexity and size of the algorithm.

6.5 Results

6.5.1 Test Parameters

The level 1 cache is 8, so 2048 keys can be sorted in the level 1 cache. The number of keys to be sorted in the level 2 cache is 262144. 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 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.

6.5.2 Expected Performance

Algorithm N and algorithm S are included in the results for reference, but are not discussed here.

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, as should its variant, double multi-mergesort.

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. Due to their design, the double mergesorts should have better performance than their counterparts.

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.

6.5.3 Simulation Results

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 6.2(b) are also as expected, although the multi-mergesort does not show the same improvement over tiled mergesort as in LaMarca's results. Double multi-mergesort does show an improvement, but not a great improvement to double tiled mergesort. It does, however, have a much flatter slope.

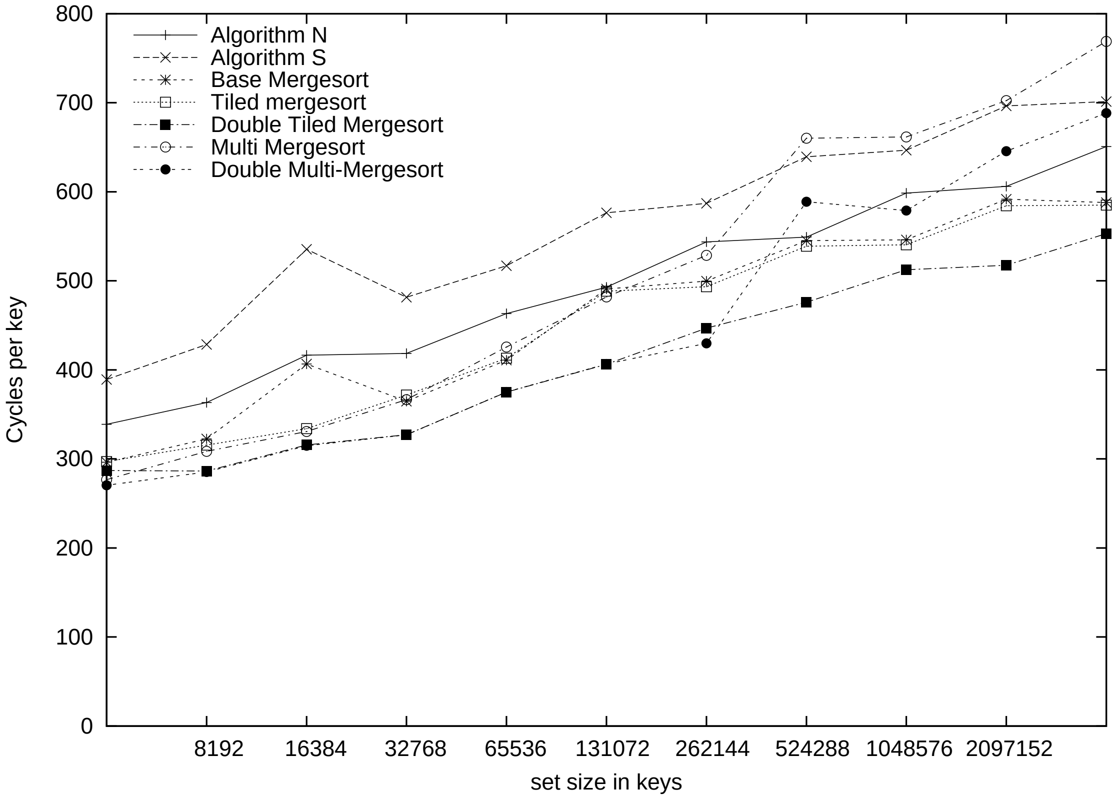
The prediction results in Figure 6.3 are as expected. Very little variation exists between the types of mergesort. It is noticeable, though, that the spike in branch misses due to the multi-mergesorts is considerably lessened by the two level adaptive predictor. The reason for this is not clear.

Despite the improvement in cache misses due to multi-mergesort, it still cannot account 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 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 optimised 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 results presented here differ from LaMarca's results. In the case of instruction count, both sets are nearly identical: the graphs are the same shape and have similar ratios between the base and multi-mergesorts. The cache results have a similar shape to LaMarca's, with a large increase in misses as soon as the arrays no longer fit in the cache, and similar ratios between the base and tiled versions. But while LaMarca reports a very small increase in level 2 misses once multi-mergesort's data set is larger than the cache, the results here show nearly the same jump as tiled mergesort. This results in half as many misses as the base mergesort, while LaMarca shows nearly 10 times fewer misses for his multi-mergesort.

As a result of this discrepancy, it is not a surprise that LaMarca's results indicate that multi-mergesort is far faster than base-mergesort, but the same results are not shown here. The increase in instruction count cripples multi-mergesort, and tiled mergesort emerges the victor.

This is not to say that LaMarca's results disagree with those here. In fact, it is



(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

noticeable that multi-mergesort's cache misses plateau at the edge of the graph, while other mergesorts' misses continue to climb. This is in keeping with both LaMarca's theory and results.

6.6 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.

The difference between the actual and expected results of the tiled and multi-mergesorts is another area to be investigated. While the results point to similar conclusions about cache misses between the base, tiled and multi-mergesorts, the relation these cache misses for direct mapped caches is not expected. Investigating, and possibly fixing this 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.

A bug exists in the double tiled and multi-mergesorts, which causes an extra merge step in some cases. More accurate results could be achieved by solving this problem. Also, to take advantage of the level 1 cache, the double sorts were sorted to 2048 keys first, then merged into larger sections. This number should have been half that. See section C.2 for more details.

To address one of the problems with multi-mergesort, that of the conflict misses on the k-way merge, [**?**] 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 [?] and [?], and later in more detail in [?]. Sedgewick did his PhD on quicksort, and several important improvements to quicksort are due to this. They are discussed in [?] and [?]. [?] discusses considerations for a fast C implementation. [?] 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 on the following page contains a simple implementation of the quicksort algorithm.

7.1 Base Quicksort

LaMarca uses the optimised quicksort described by Sedgewick for his base quicksort. This is an iterative quicksort which uses an explicit stack. In the worst

```
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

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 optimisation 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 subfiles. 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 5 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 the 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 three 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 [?], using a median of nine partition is discussed, and modern discussions on quicksort refer to the problem of equal keys, which is discussed there, as well as by Sedgewick in [?]. 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 would overwhelm the improvements LaMarca makes in the next phase of the algorithm. Secondly, it is not as important to write a brilliant version of quicksort as it is to analyse LaMarca's improvements.

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. However, 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 subfile if required. This check is expensive, but less so than either a bounds check or putting a sentinel into every subfile.

7.3 Multi-Quicksort

In the case that the array to be sorted fits in the cache, then quicksort has excellent cache properties. Once it no longer fits in 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[1].

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 5,000 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

[1]For a 2MB cache, this is 524,144.

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. The stack must therefore be increased in size before this happens. By pushing the last array copied onto the stack, an entire set of cache misses is avoided. Unfortunately, since a multi-partition is only done when the array is more than twice the size of the cache, this reduction in misses in not significant.

While emptying the containers back into the array, an opportunity is taken[2] 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

The threshold chosen was 10, simply as 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[3].

A 2MB cache was assumed, so the number of keys which fit in the cache for this in-place sort is 524,288. Multi-quicksort began once the number of keys was larger than 174,762.

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.

LaMarca notes that quicksort is very cache efficient already. 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.

[2]Results do not reflect this step. See C.2 on page 133.

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

The number of branches should increase in multi-quicksort along with the instruction count. When a binary search is used, the increase in misses 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 as predicted. The expected increases in instruction count for multi-quicksort was due to a test with a smaller data set and smaller constants. The actual results were not as high as the expected ones, but closer to 20% and 45% greater, for sequential and binary search versions, respectively.

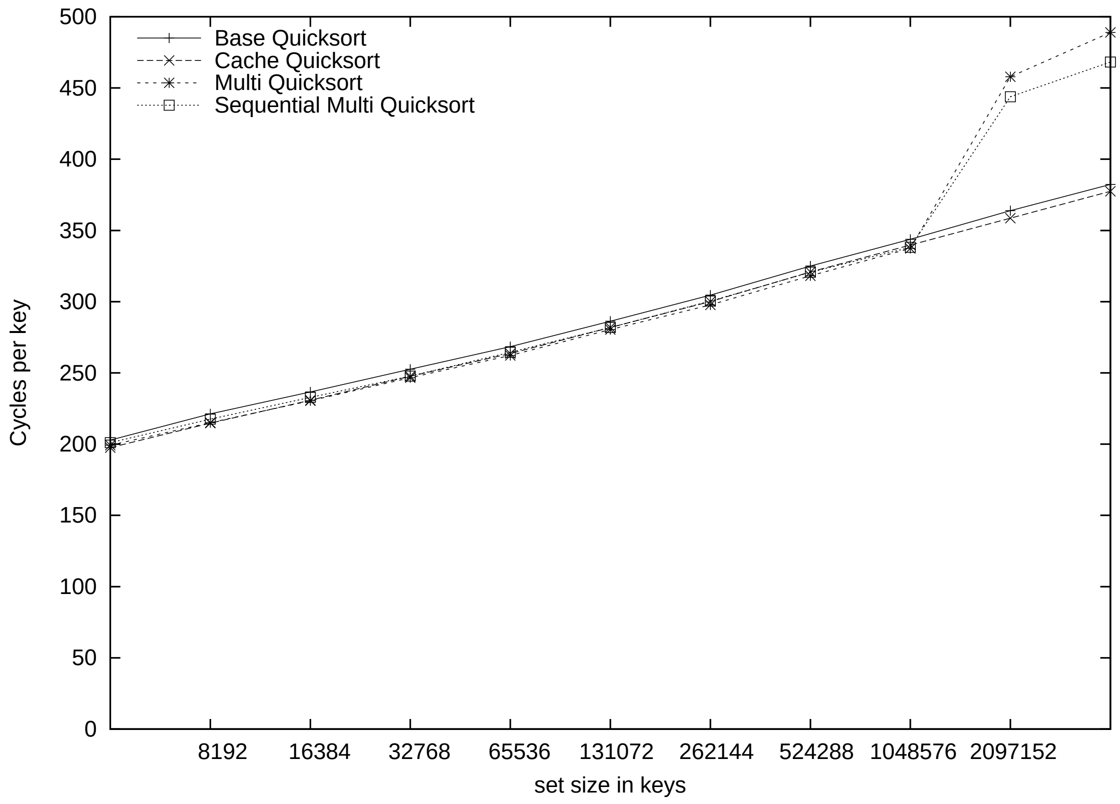
The cache results in Figure 7.3(b) were similar to predictions, except that the multi-quicksorts did not behave as expected, and performed worse than the memory-tuned version. However, at the edge of the graph their slopes are considerably less than the slopes of the other sorts, and may result in lower misses for larger data sets. The tuned version performed better than the base version, as predicted.

A surprising result is that direct mapped caches often perform better than fully associative ones. The reason for this is not clear, but it is obvious from the graph that this is true in the case of all the quicksorts.

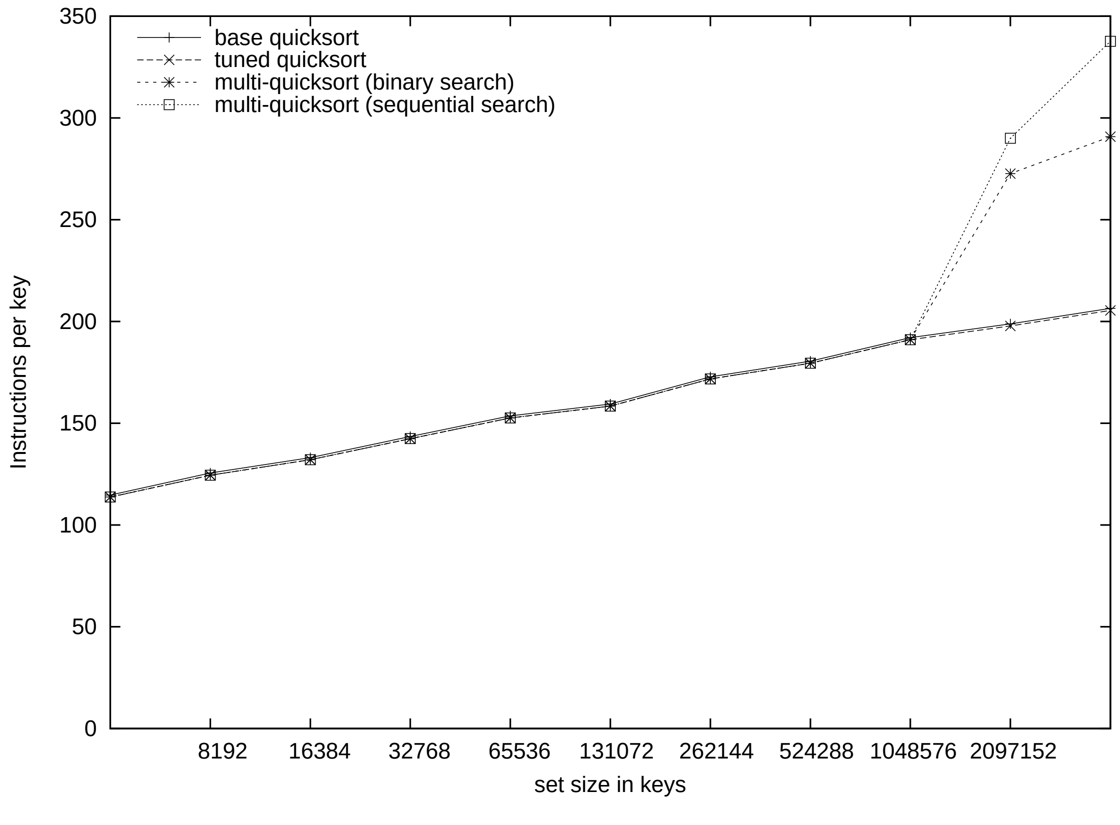
The branch prediction results in Figure 7.4(b) were exactly as expected. The binary search produced a dramatic increase in the branch misses, though not when the two-level adaptive predictor was used. A short spike was recorded in the bimodal results, and its not clear if this is an anomaly or not. If it is, then the two-level adaptive predictor performs worse than the bimodal, yet again.

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 and level 1 accesses, both of which were significantly reduced by using the binary search. The tuned quicksort, however, had the lower running time and even base quicksort out-ran the multi-quicksorts.

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 cache quicksort are the same in both sets of results, but the number of cache misses in multi-quicksort is much higher here than it is in LaMarca's findings. As with mergesort though, while the results are different, the shape is similar: multi-quicksort has a much flatter slope than the other sorts. This difference is likely to be due to the implementation of multi-quicksort; a slightly different ap-

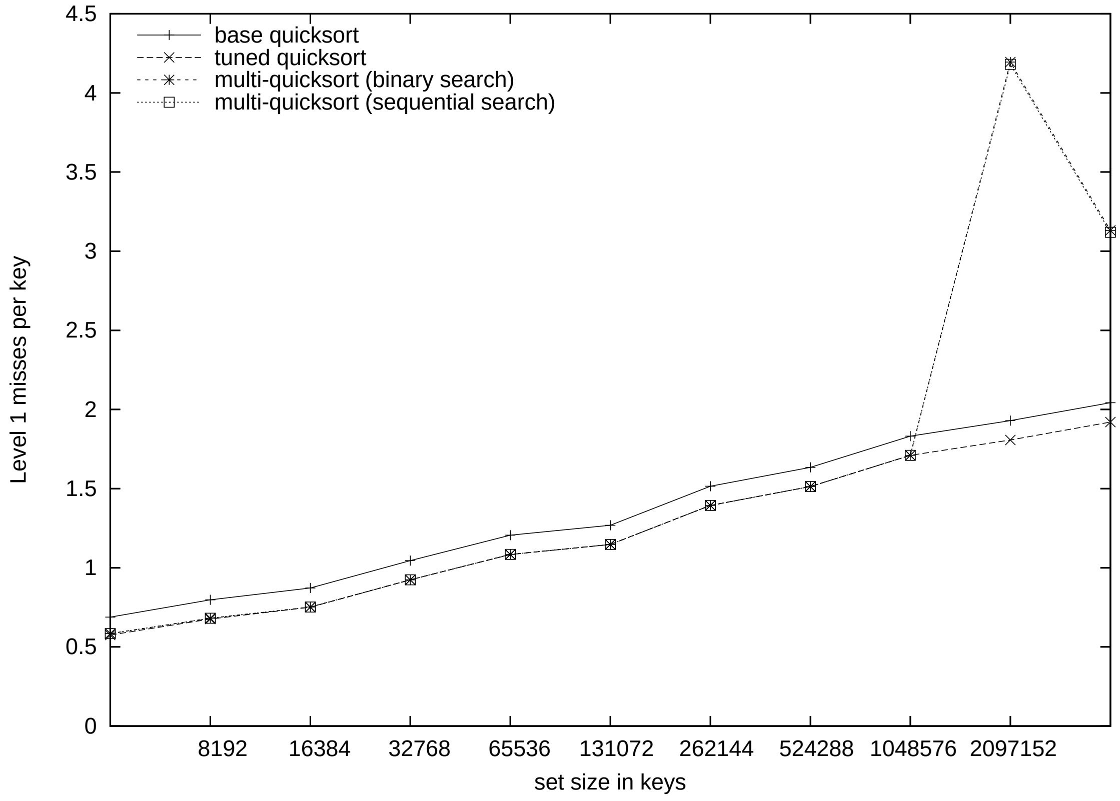


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

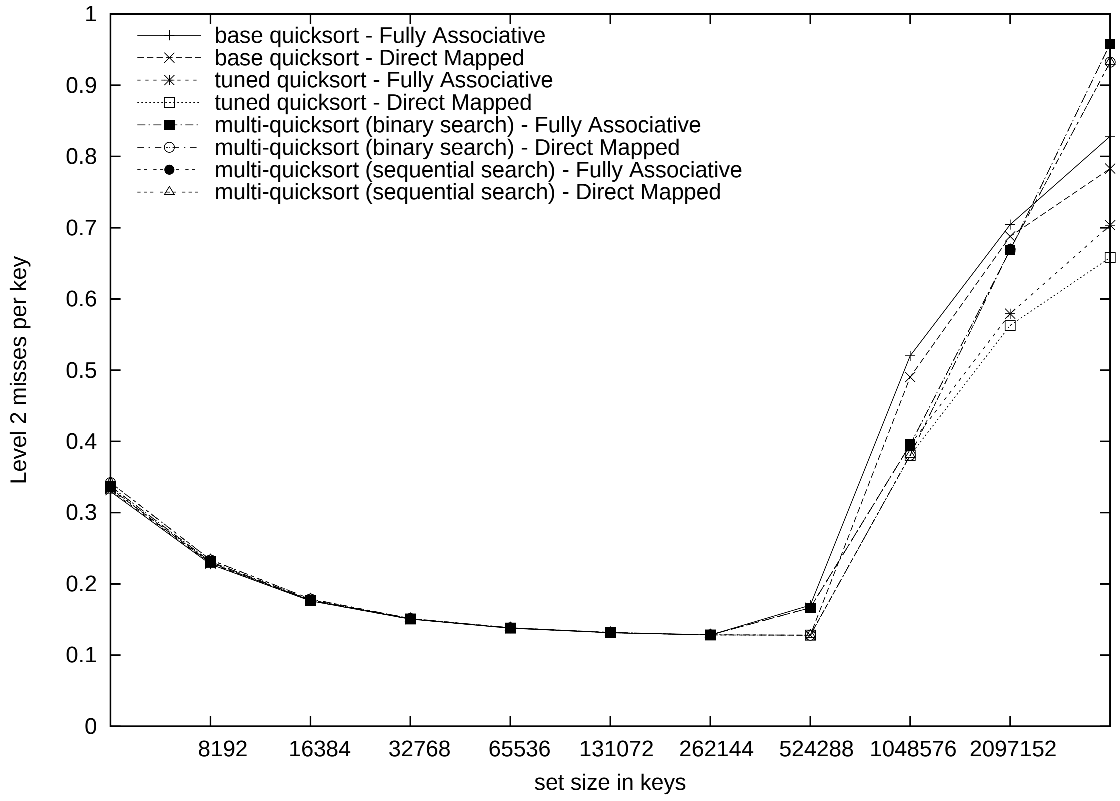


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

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

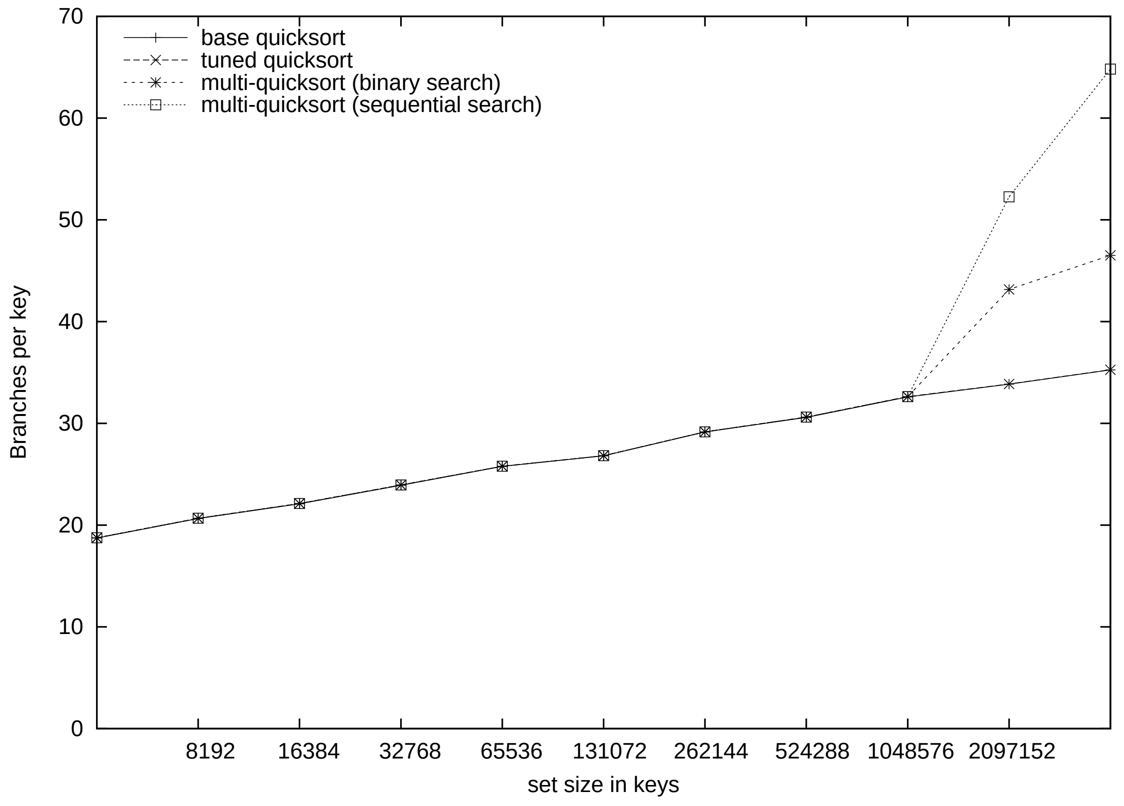


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32 byte cache line and separate instruction cache. Results shown are for a Direct Mapped cache.

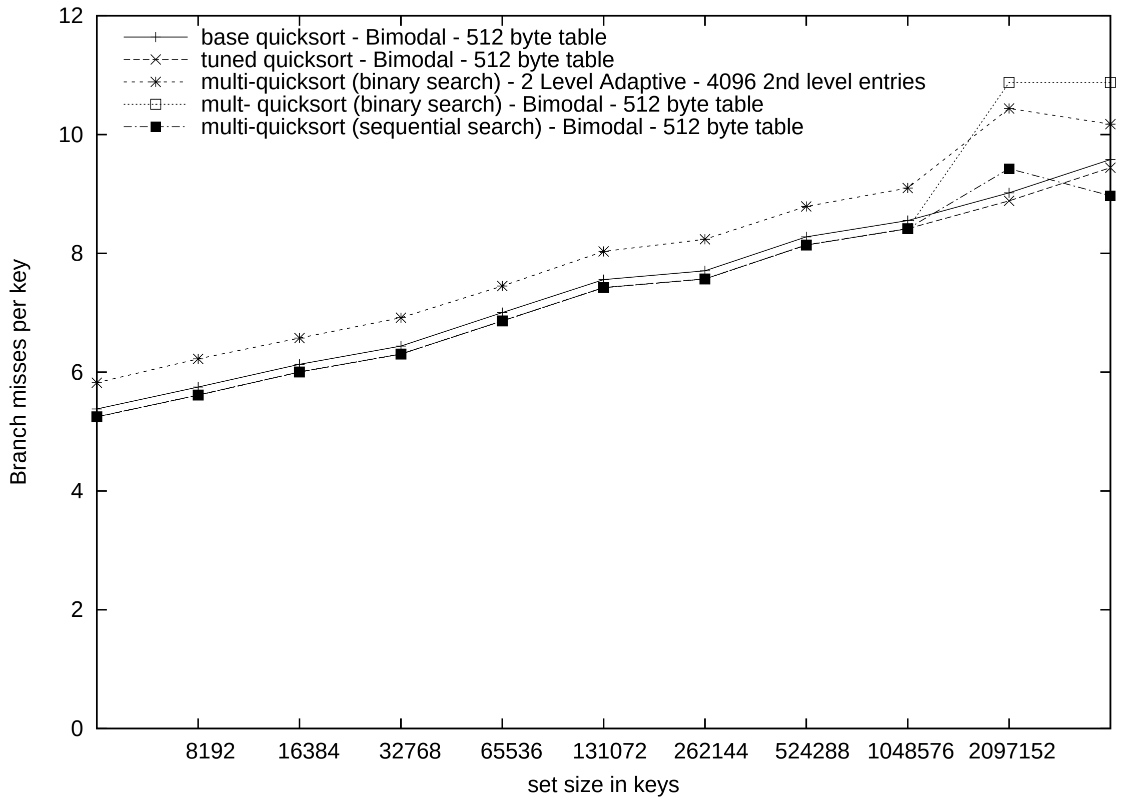


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32 byte cache line.

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

proach, which is described immediately below, may have the effect of changing the results sufficiently that they may agree with LaMarca's.

7.5 Future Work

In multi-quicksort, it may be possible to reduce the cache misses by sorting each containerful of keys as soon as it is put back into the array. This step would significantly increase the temporal locality, and may reduce the number of cache misses by half.

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.

Two areas of current research into quicksort are those of equal keys, and median-of-greater-than-3. The former affects the algorithmic properties, and some steps may need to be varied to avoid extra cache 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.

The 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 $(N/THRESHOLD)$[4] times. 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 (ie, 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 3, 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 project 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 all 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 bits 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. Then the counters are summed in turn to store the cumulative count of keys that belong before them in the array. Now each byte can be put in place by accessing the counter indexed by its least significant byte.

These are written into an auxiliary array, which is written back before the next iteration 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.

This type of radixsort is called *Least Significant Digit* radixsort. Other implementations are possible, and several improvements also exist, but as radixsort is included here for comparative purposes, these are not implemented. They are, however, discussed in Section 8.2.

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

8.1 Results

8.1.1 Test parameters

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

8.1.2 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.

The cache performance of radixsort is not expected to be good, however. Quicksort and mergesort, for example, 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.

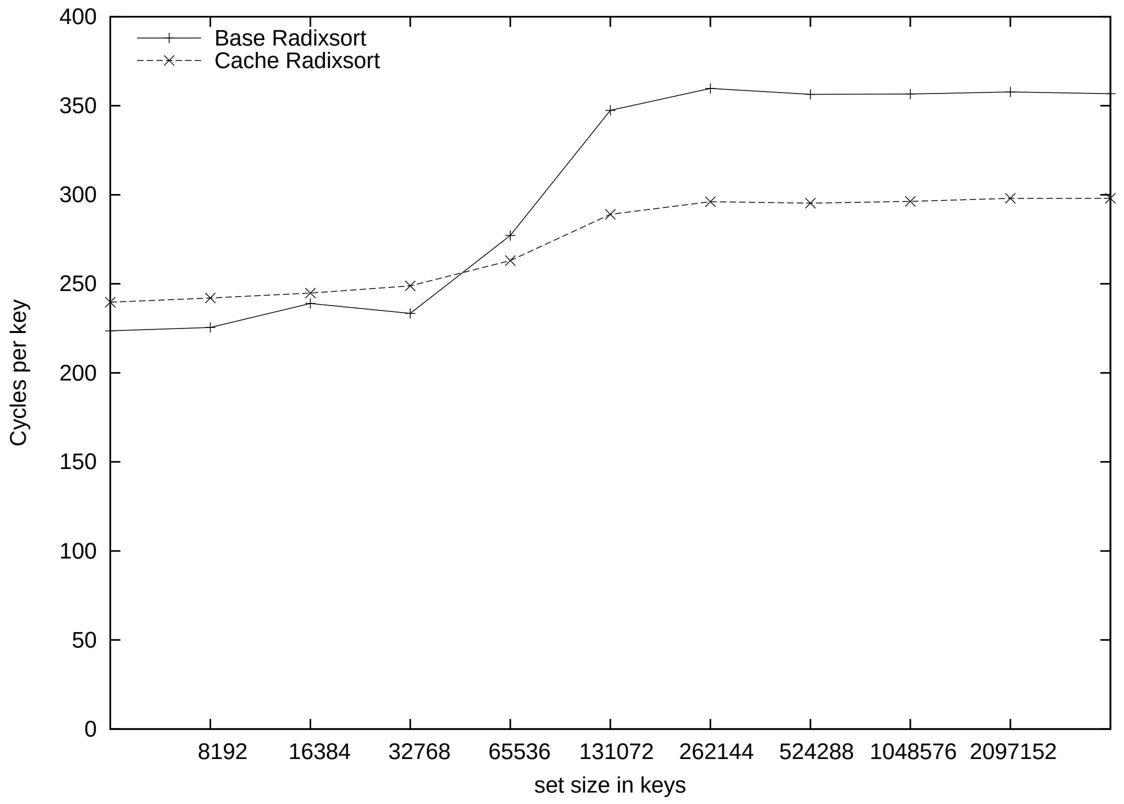
The number of iterations across the array is constant, however, proportional to $(bitsPerInt/log_2(Radix))$. For a 32 bit integer, this is four steps and each step has four passes over the array, for a total of 16 passes over the array. With eight keys per cache line, there should be about two misses 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, a branch due to a comparison can not be predicted except in exceptional circumstances[1]. As a result, only flow control branches occur in radixsort, and they are all predictable.

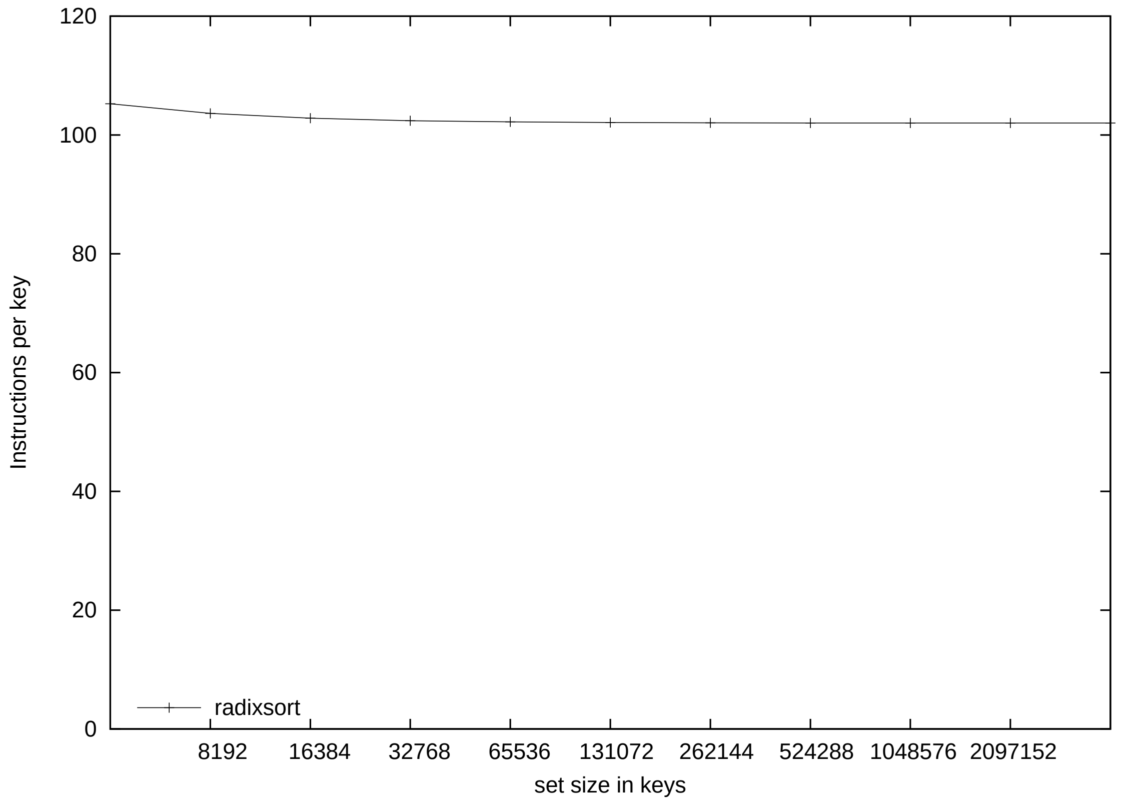
8.1.3 Simulation results

The instruction count of 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 amortisation of the loop overhead). The number of cache misses in Figure 8.2(b) is exactly the same, until the array no longer fits into the cache. The number

[1]See Section 4.5.

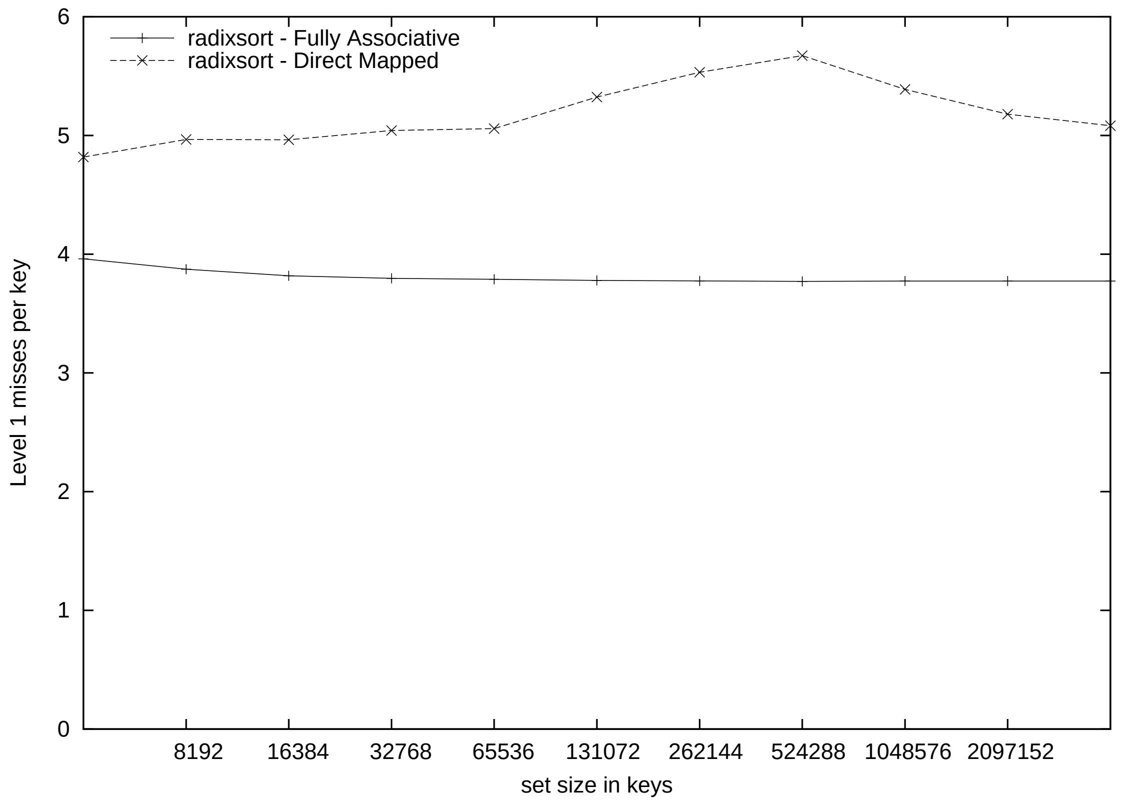


(a) Cycles per key - this was measured on a Pentium 4 using hardware performance counters.

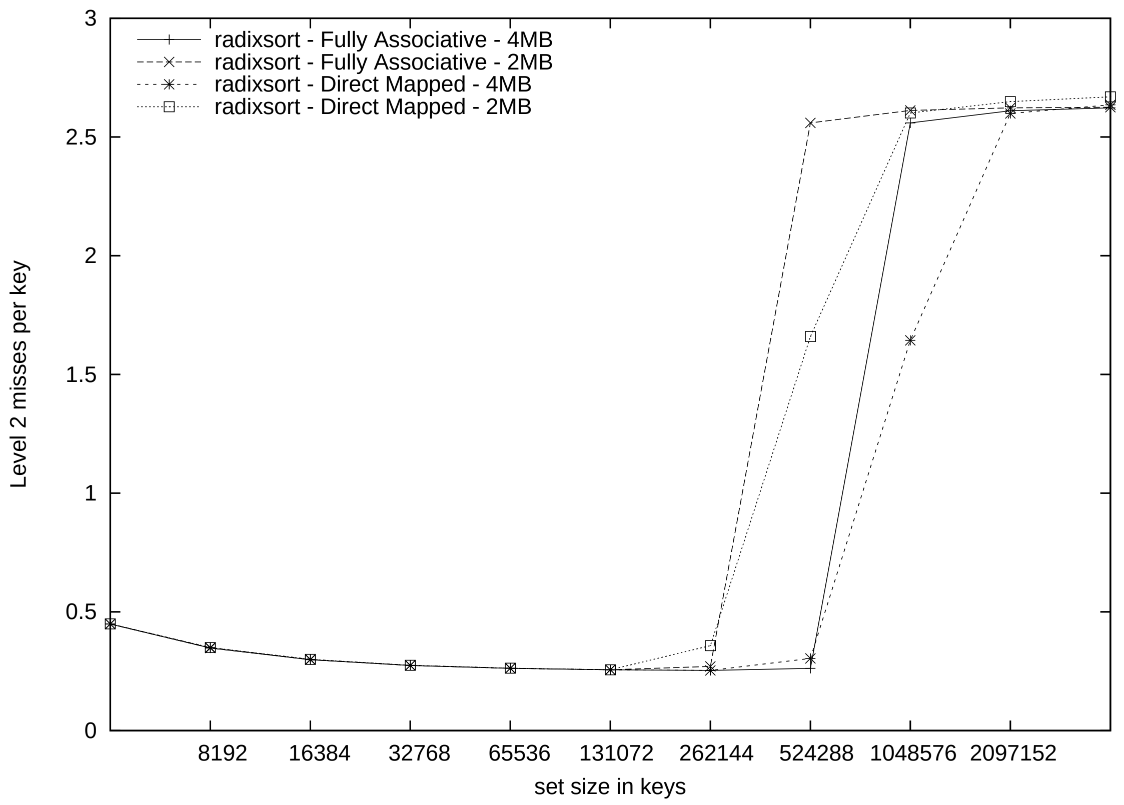


(b) Instructions per key - this was simulated using SimpleScalar `sim-cache`.

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

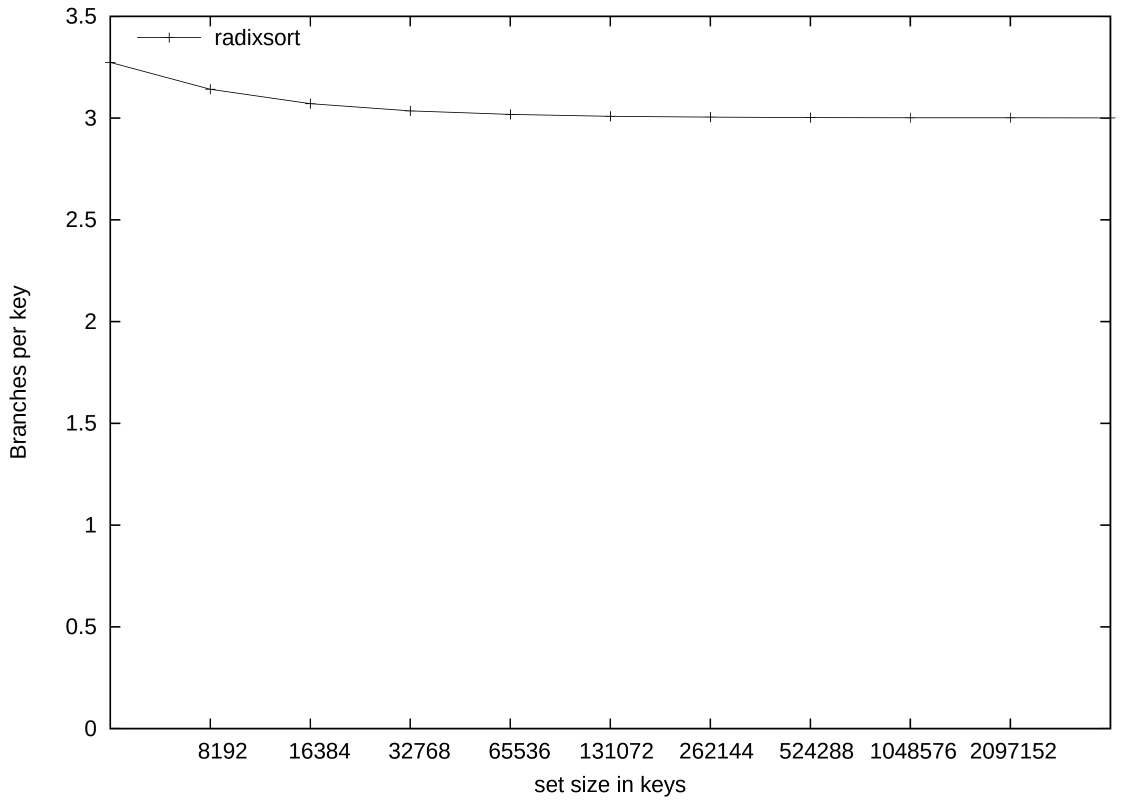


(a) Level 1 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating an 8KB data cache with a 32 byte cache line and separate instruction cache.

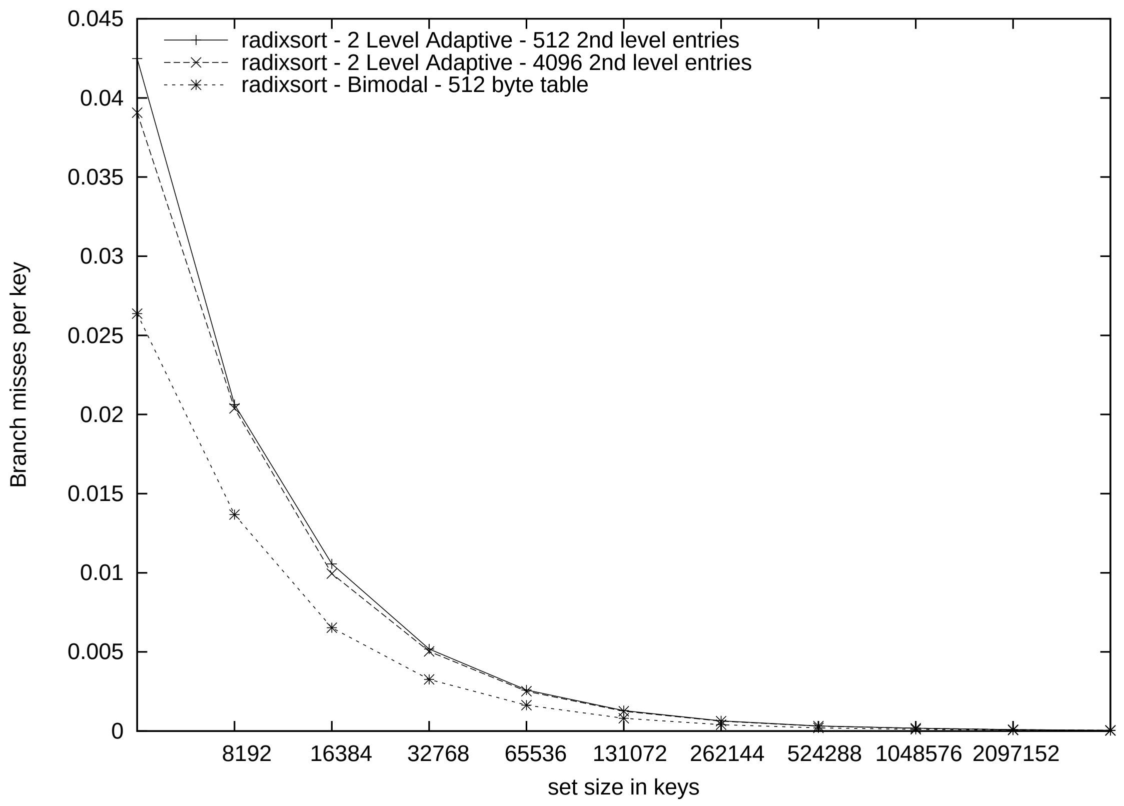


(b) Level 2 cache misses per key - this was simulated using SimpleScalar `sim-cache`, simulating a 2MB data and instruction cache and a 32 byte cache line.

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

of misses then leaps up to 2.5 per key, which is roughly what was predicted.

The number of branch misses, shown in Figure 8.3(b) was also extremely low. Despite 10 branches per key, there was less than .08 of a branch miss per key. The number of branches itself is quite low, as several of the sorts considered had nearly that number of misses. However, it is not unexpected.

With the low number of branch misses, a much better overall performance is expected than is achieved. When in the cache, it performs only slightly worse than quicksort, but once the data set grows larger, the gap widens. However, the results, shown in Figure 8.1(a), indicate that radixsort will never increase beyond 350 cycles per key, meaning that for larger sets it will perform significantly better than quicksort. The reason for the high number of cycles is that radixsort was not optimised. Possible optimisations are discussed below, and should significantly reduce the number of cache misses, the main cause of radixsorts poor performance.

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. A similar thing happens in both cache miss graphs: a sharp dip in LaMarca's results, and a flat-line with a slight slope in these results. Also, in both cases, once the array no longer fits in the cache, the number of misses jumps to around 2.5, where they stay.

The cycles per key in both cases are unsurprising based on instruction count and cache miss results: LaMarca's have a sharp fall before rising slightly once the data set no longer fits in the cache; the result presented here flat-line before rising slightly for the same reason. While both results agree for large data sets, they do not for smaller sets.

8.2 Future Work

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 suggested 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. At the moment, there is one step to count, one to copy, and one to write back (causing potentially two misses per write). LaMarca suggests counting for the next step in the previous count. In fact, all the counting can 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, but this is small compared to the main and auxiliary arrays.

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 potentially done with radixsort, without any increase in complexity. The number off steps in likely to be even, and is known at compile time, so the number of times the copy step is done could be reduced to zero. If the number of steps is odd, which is exceptionally unlikely when sorting integers, then one copy at the end would be required.

Combining other sorts with radixsort could also be possible. An optimisation to mergesort was to do a k-way merge at the end, but this dramatically increased the instruction count. Using a type of radixsort here would be complicated, but may be possible. Alternatively, sorting inside the cache using radixsort is far faster than merging. The k-way merge could then be performed at the end. In practice, though, radixsort is far faster than mergesort, has a lower instruction count, and less cache misses. While these steps could improve mergesort, they are more likely to slow radixsort, with no advantage.

Radixsort could also 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.

Additional performance improvements could possibly be gained using a different radix. This radix was chosen simply because Sedgewick used it. Another radix may lead to better results.

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 in 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

Conclusions

This section presents a summary of the results from previous chapters, lists important results and discoveries, and suggests contributions made to computer science by this project.

9.1 Cache Results

The test results in this project mostly agreed with LaMarca's results. The heapsort graphs had nearly exactly the same shape as LaMarca's, and the ratios between the base and cache-conscious versions were nearly identical. The multi-mergesort results were slightly different: instruction count was identical, but while the cache results of the base and tiled mergesorts were identical, multi-mergesort results were not the same. However, the basis for those results - a sharp increase in cache misses followed by a flat-lined graph - were in both cases.

The quicksort results were also very similar, though the multi-quicksort results had a slightly different shape to LaMarca's. However, the instruction count results for each quicksorts agreed with LaMarca's results, as did cache miss results of the base and cache-conscious versions.

The two sets of radixsort results were slightly similar. The rise and plateau of the curve to the right of the graphs was the same in both cases, though the left hand side of the graphs did not agree.

Another interesting observation can be made: the results of the simulations on the Pentium 4 were not always influenced by the cache misses. Though heapsort and radixsort began to perform poorly once the data set no longer fit in the cache, neither quicksort nor mergesort were affected. The reason why radixsort was affected is obvious - there is no temporal reuse; in heapsort, the full cache line is not being used, and spatial locality suffers as a result. It is interesting that the other two sorts were unaffected, even though their cache miss graphs indicate a significant increase in misses.

Finally, it was shown that in some cases a direct-mapped cache performed better than a fully-associative one. The reasons for this were not clear.

9.2 Branch Prediction Results

Insertion sort has only one branch misprediction per key. This shows that it is possible to predict comparative branches correctly, though the instruction cost of such an algorithm may be high. This is also shown in multi-quicksort: though the instruction count and memory reference count are both doubled, it is still faster to use a sequential search than a binary search across small lists.

Selection sort has a similar number of instructions and cache misses as bubblesort and a larger instruction and cache miss count than shakersort, yet it still considerably outperforms both of these. This shows the high cost of branch mispredictions on an algorithm.

Radixsort has almost no branch mispredictions at all. The only mispredictions are due to its flow control, but it escapes comparative mispredictions, and this factor allows it to perform as fast as, and in some cases faster than quicksort, despite its high number of cache misses.

The heapsort results show the disadvantage of using unsorted lists. Even though there are less branches in the 4-heap version, the number of branch mispredictions are higher than in a standard 2-heap.

It is also shown that bimodal branch predictors generally out-perform two-level adaptive predictors when resolving comparative misses. This is shown by the results of insertion sort, bubblesort, shakersort, heapsort and quicksort. In the cases where the two-level adaptive predictor does perform as well, the table required is much larger than that required by the bimodal predictors to get the same results.

9.3 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 9. This shows the cycles per key of each of the major sorts and their derivatives, measured using Pentium 4 performance counters. Neither the elementary sorts nor heapsorts are on the chart, as their results are significantly worse than the other results, which makes it difficult to accurately see those results.

The best stable sort is radixsort. The best comparison-based stable sort is double tiled mergesort. The best in-place sort is quicksort. The best in-place stable sort is, sadly, 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 radixsort. The best overall sort was quicksort, though radixsort is very close, and the radixsort considered here was not optimised. It is expected that an optimised radixsort would beat quicksort for smaller sets.

9.4 Contributions

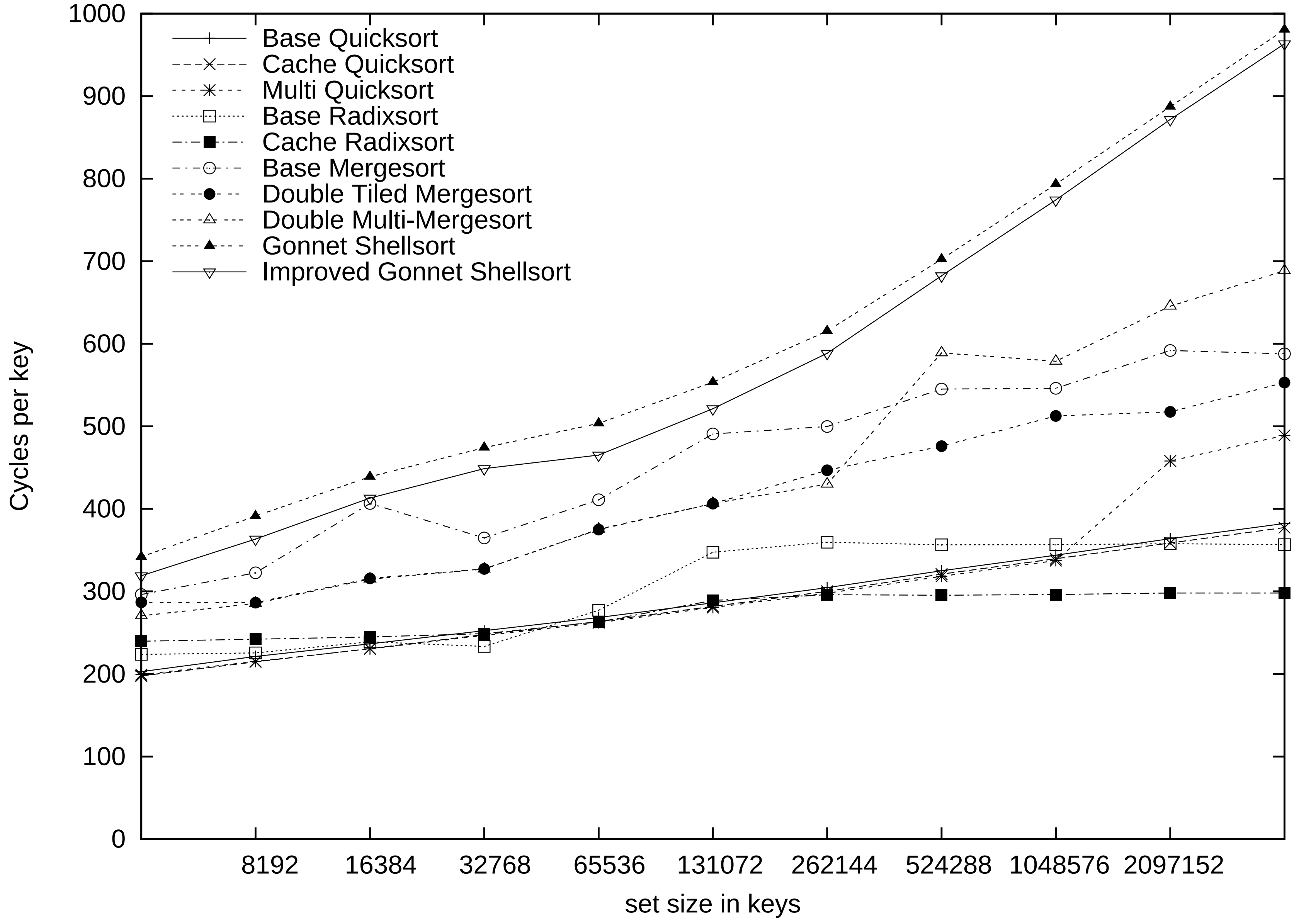
This section lists the contributions of this project. Each of these is novel and was discovered or developed during the course of this project.

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

Several discoveries were made and analysed: insertion sort has one misprediction per key; selection sort has $log_2(N/2)$ branch mispredictions per key - an explanation of this was also provided; tiling mergesort can be done in two steps, reducing the cache miss count; 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.

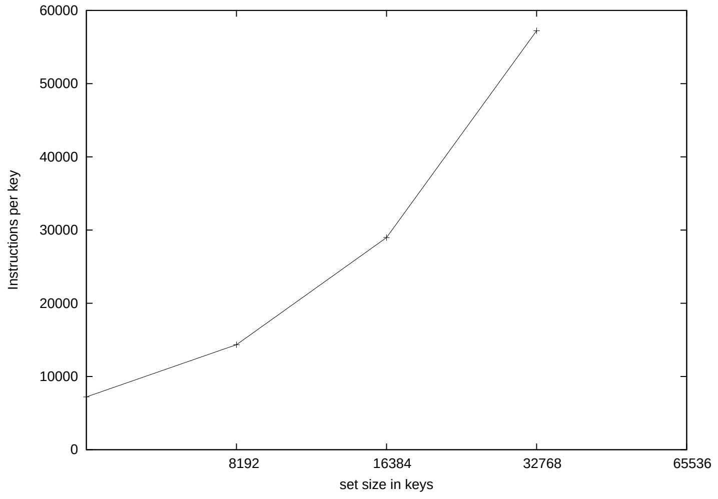
In addition, several ideas were conceived although not implemented, which relate to sorting algorithms: it should be possible to reduce the conflict misses of tiled mergesort without padding, by slightly reducing the size of the array segments sorted in the first sorting phase; it may be possible to reduce the cost of an insertion sort or selection sort by considering more than one key at a time; it may be possible to reduce the number of partitions - and hence the increase in instruction count - of multi-quicksort by choosing from a larger selection of pivots. Finally, it should be possible to reduce by half the number of cache misses in radixsort, by counting earlier and by copying back and forth between two arrays.

Figure 9: Cycles per key of several major sorts and their variations - this was measured on a Pentium 4 using hardware performance counters.

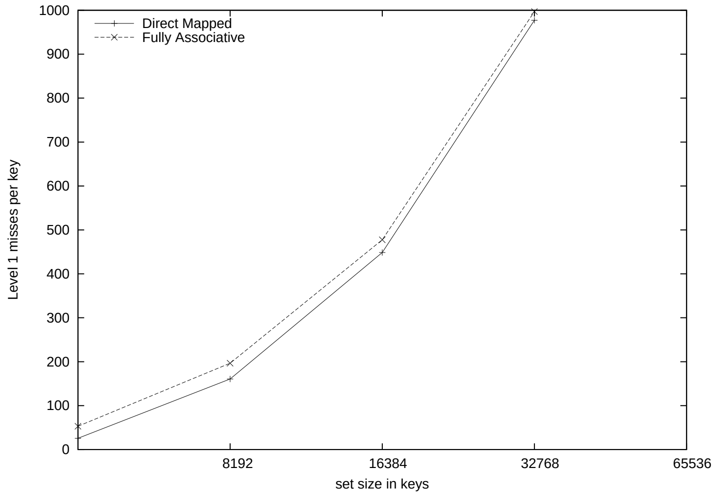


Appendix A

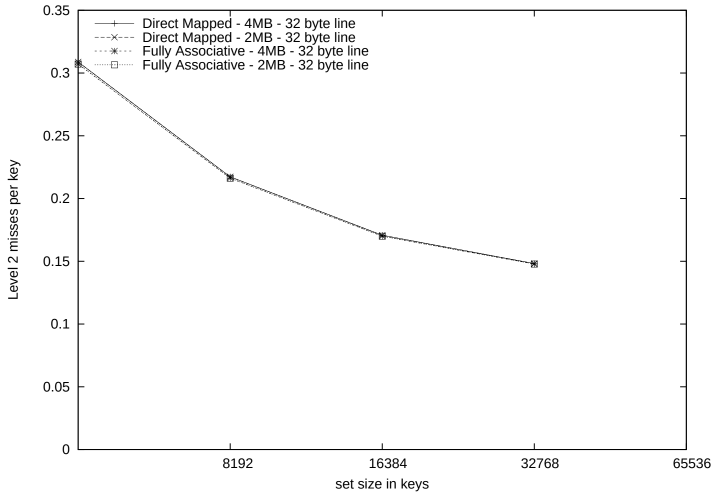
Simulation Result Listing



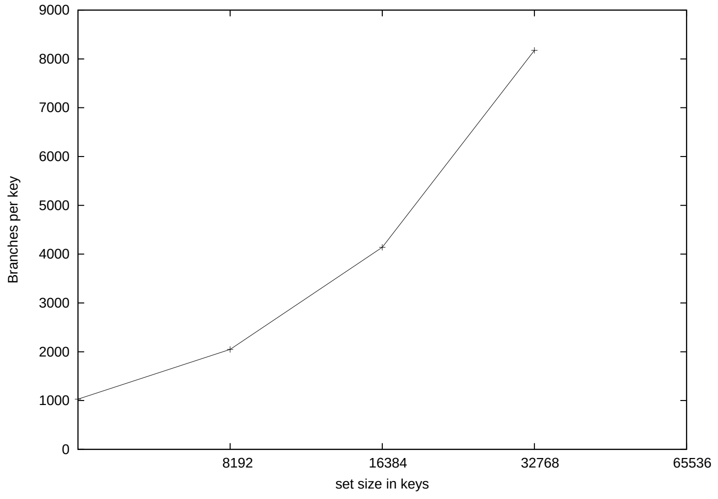
(a) Instructions per key



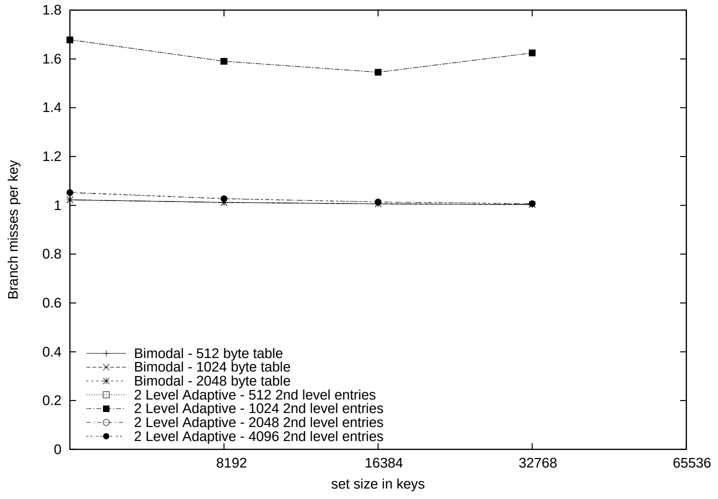
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

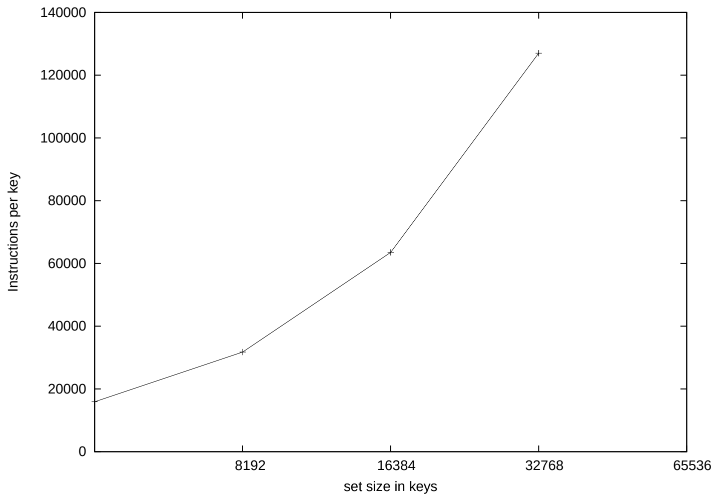


(d) Branches per key

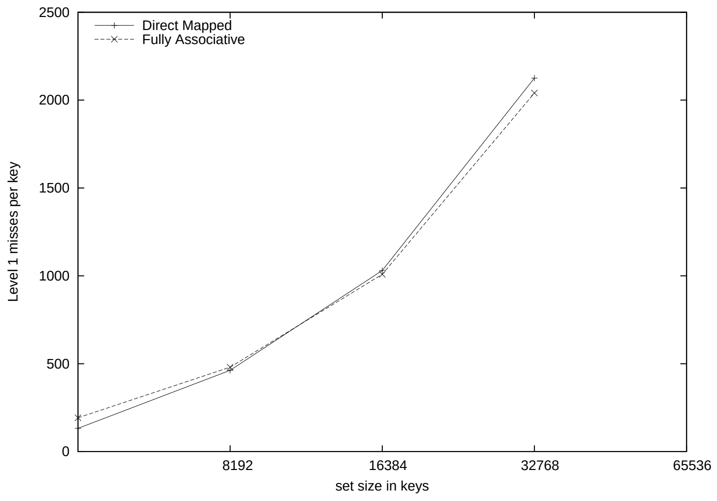


(e) Branch misses per key

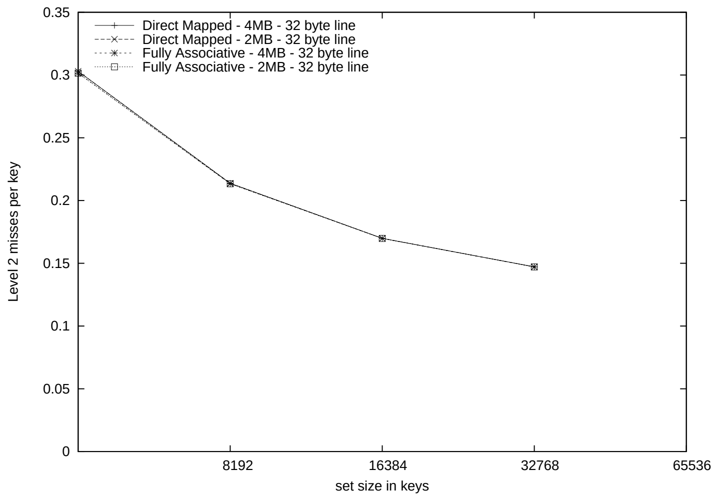
Figure A.1: Simulation results for insertion sort



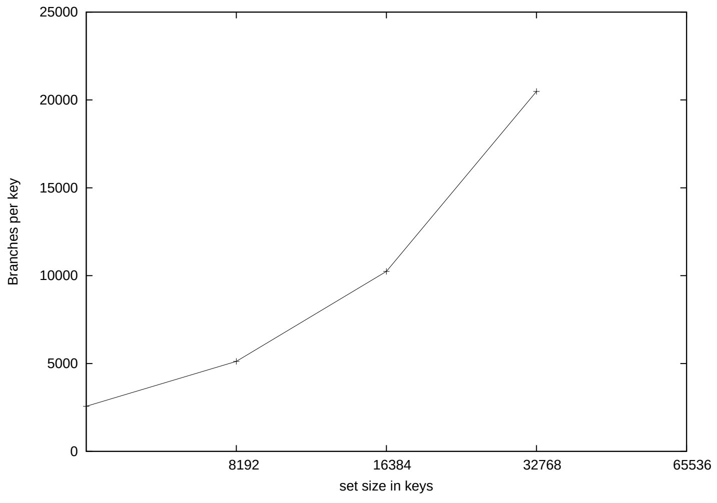
(a) Instructions per key



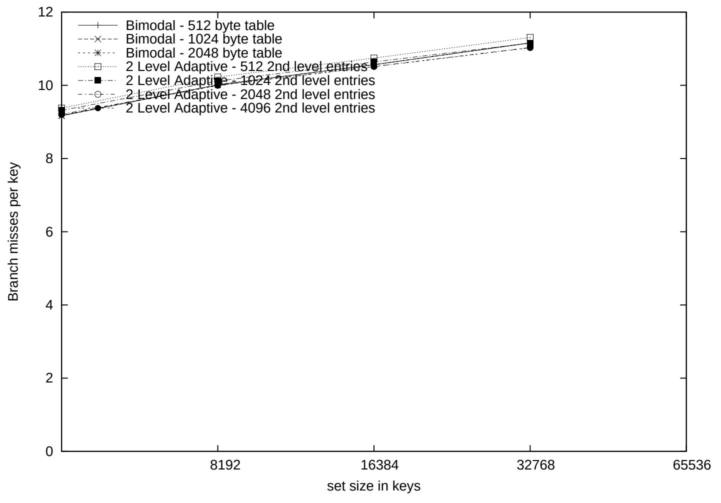
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

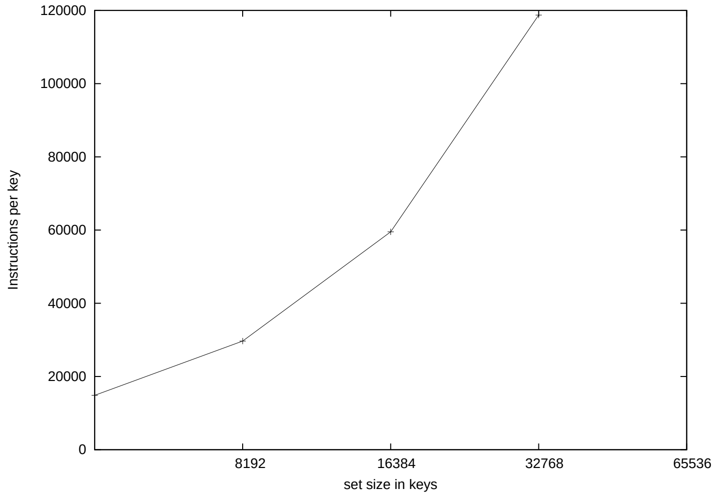


(d) Branches per key

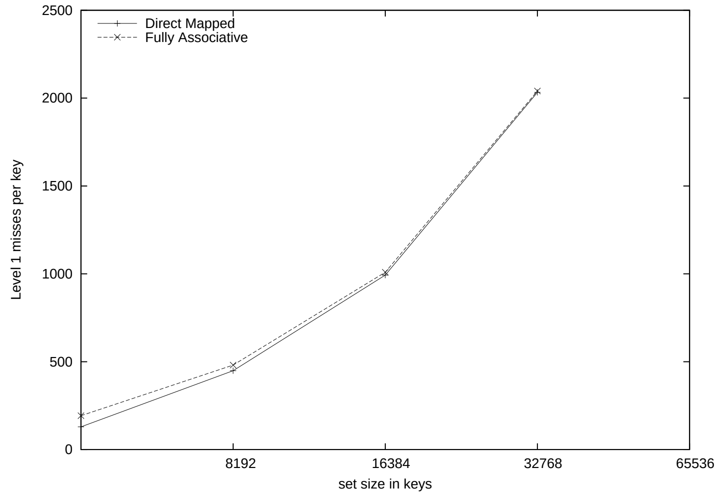


(e) Branch misses per key

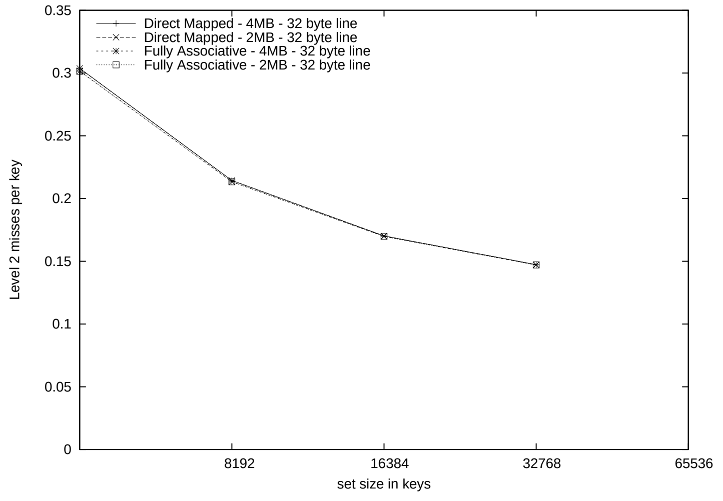
Figure A.2: Simulation results for selection sort



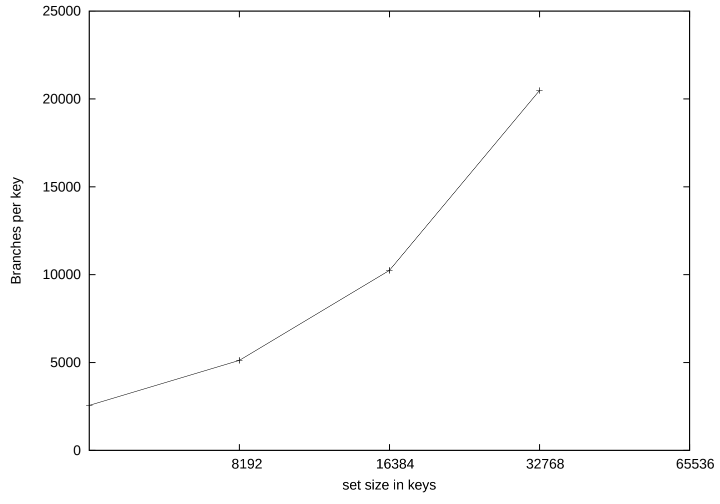
(a) Instructions per key



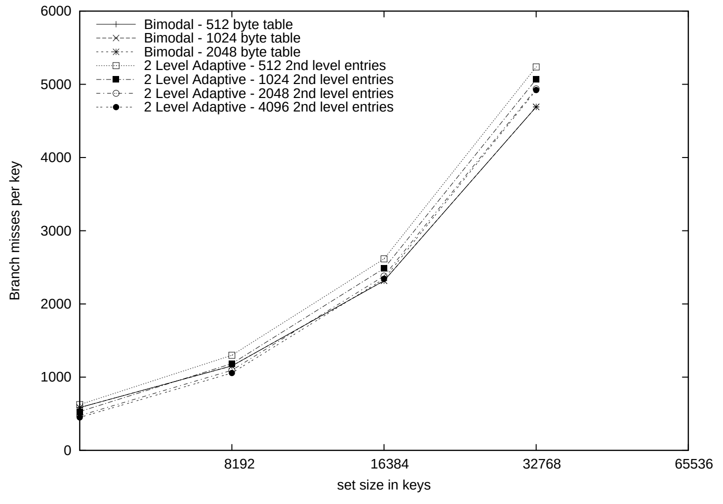
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

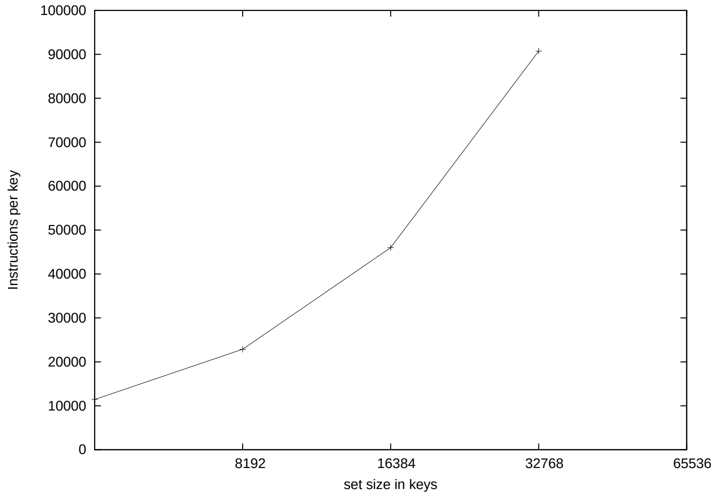


(d) Branches per key

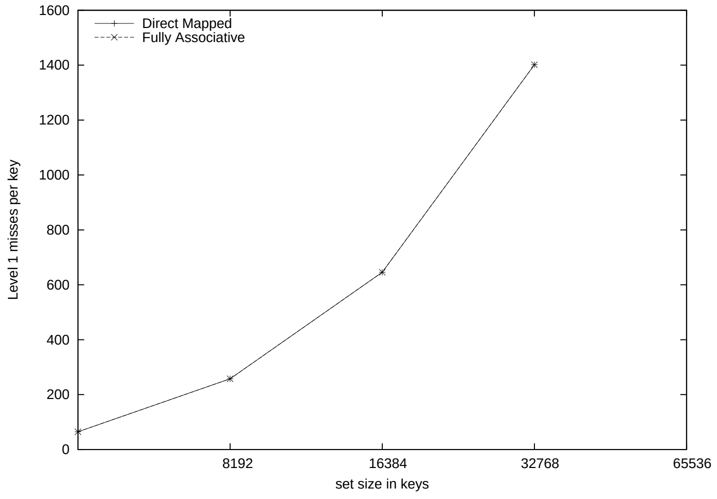


(e) Branch misses per key

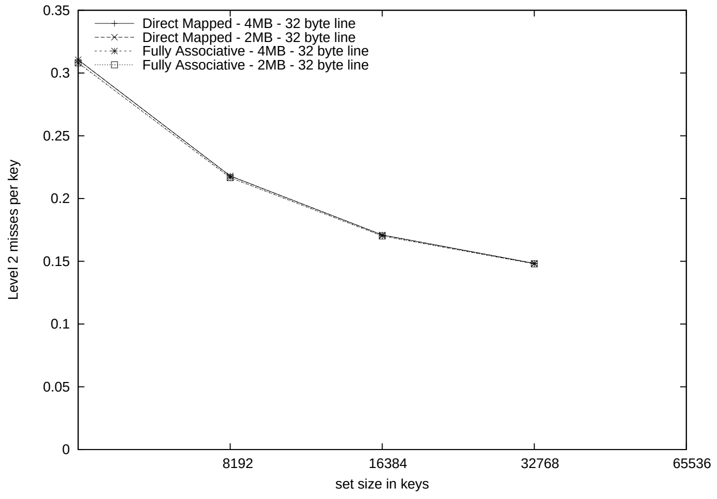
Figure A.3: Simulation results for bubblesort



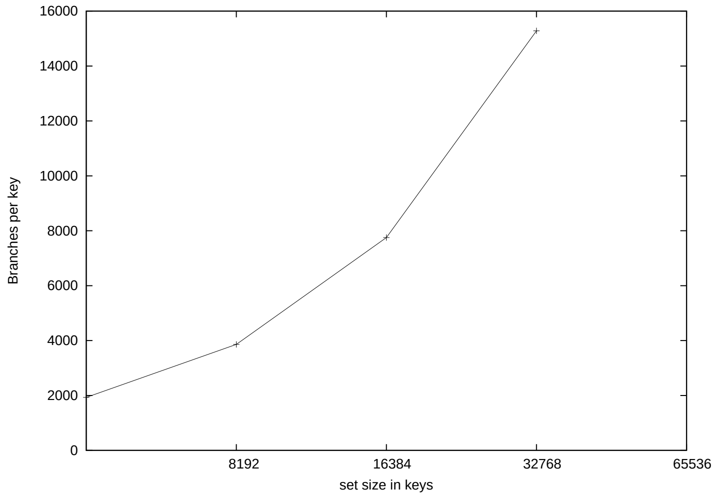
(a) Instructions per key



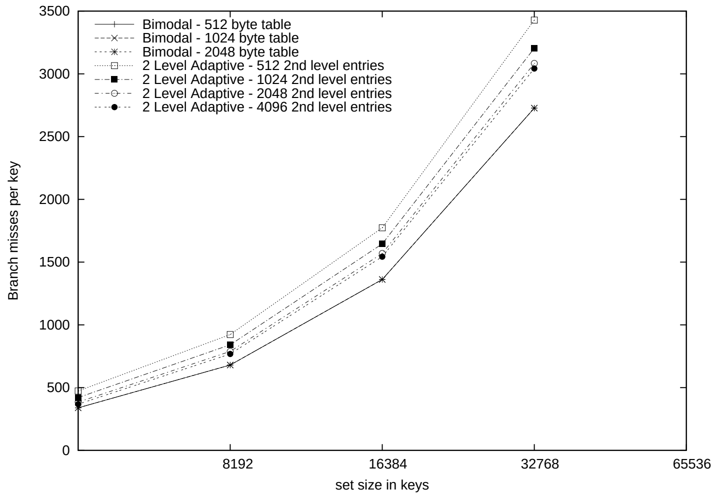
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

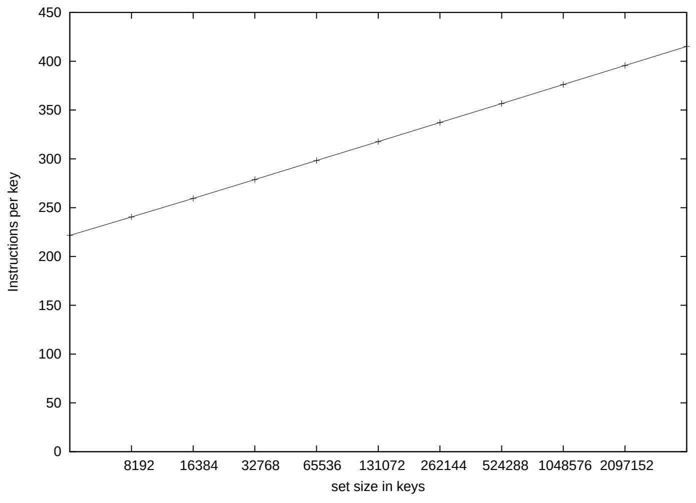


(d) Branches per key

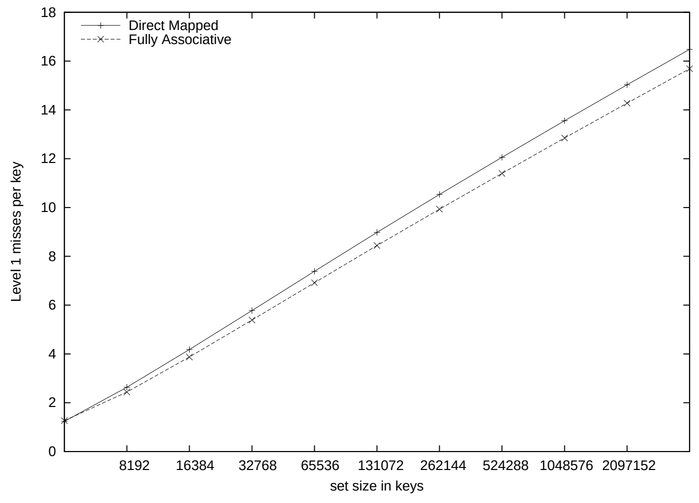


(e) Branch misses per key

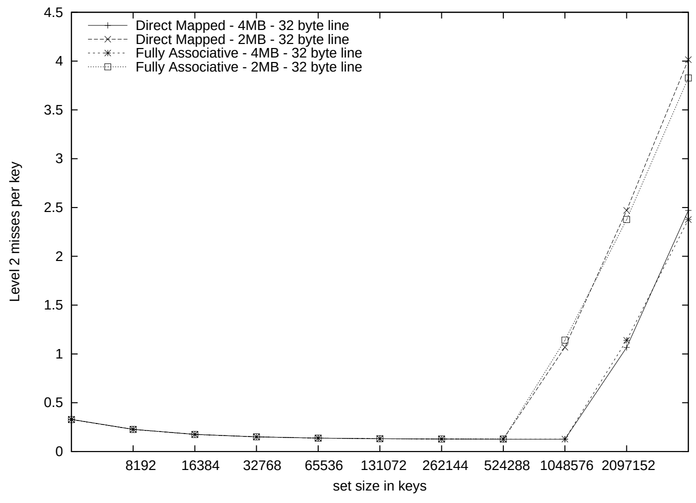
Figure A.4: Simulation results for shakersort



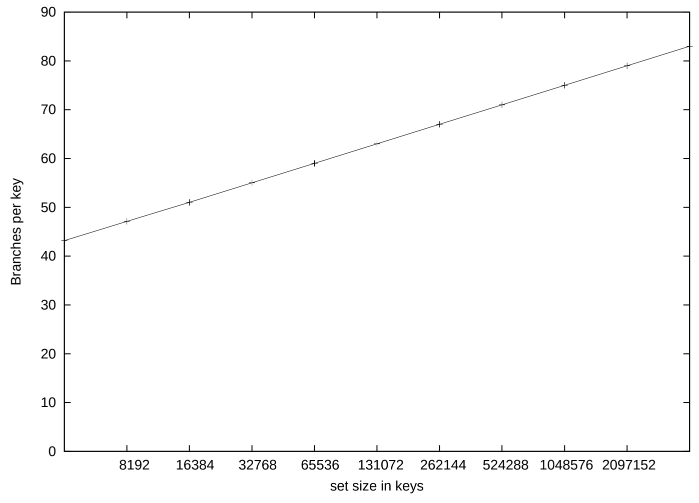
(a) Instructions per key



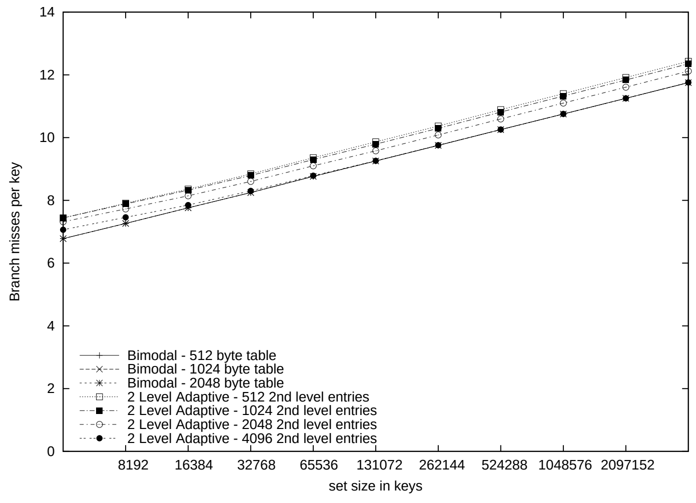
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

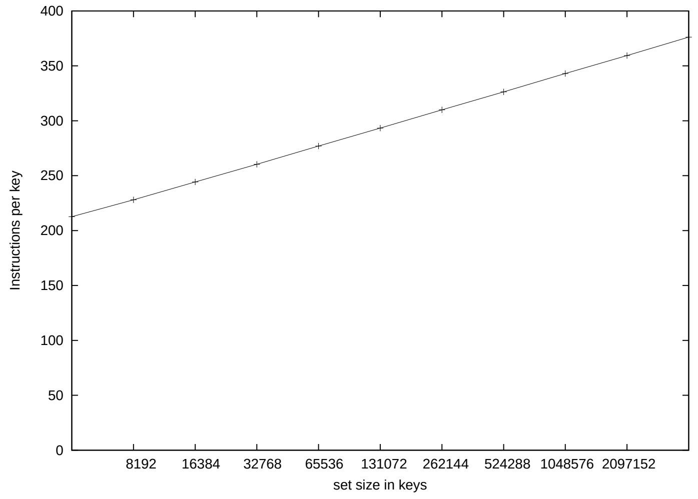


(d) Branches per key

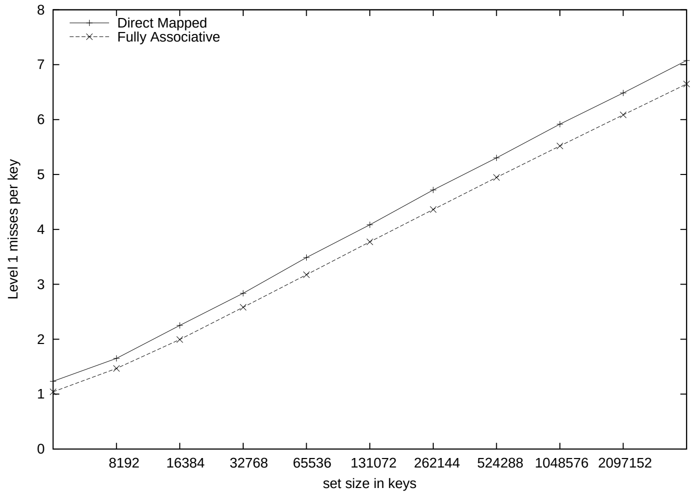


(e) Branch misses per key

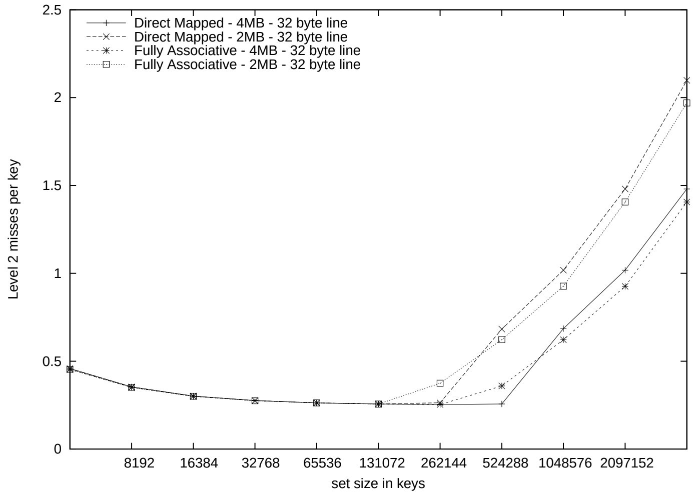
Figure A.5: Simulation results for base heapsort



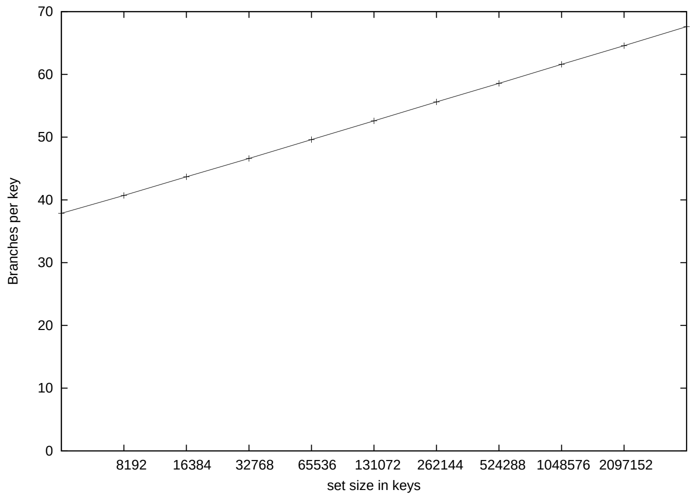
(a) Instructions per key



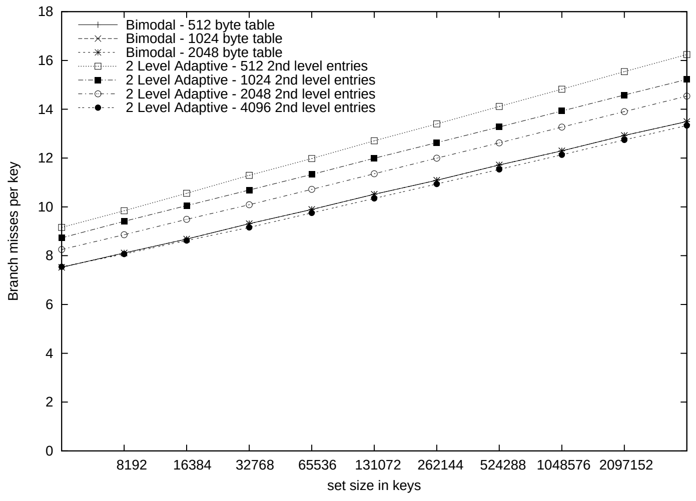
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

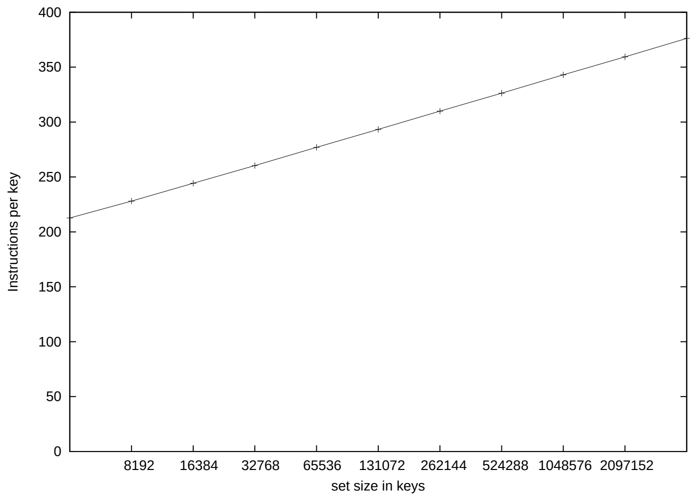


(d) Branches per key

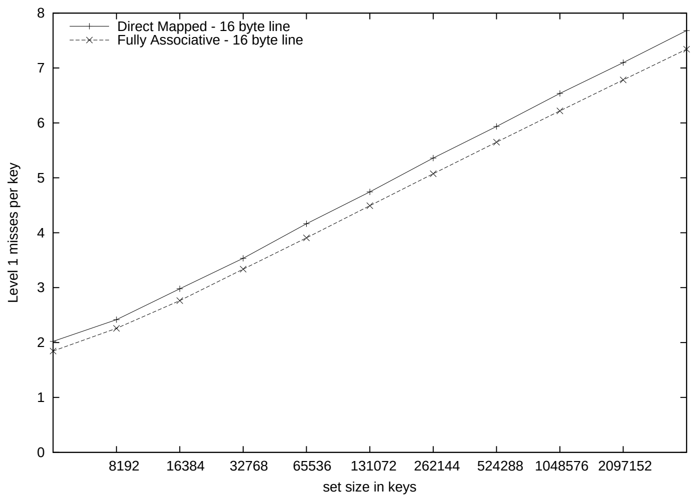


(e) Branch misses per key

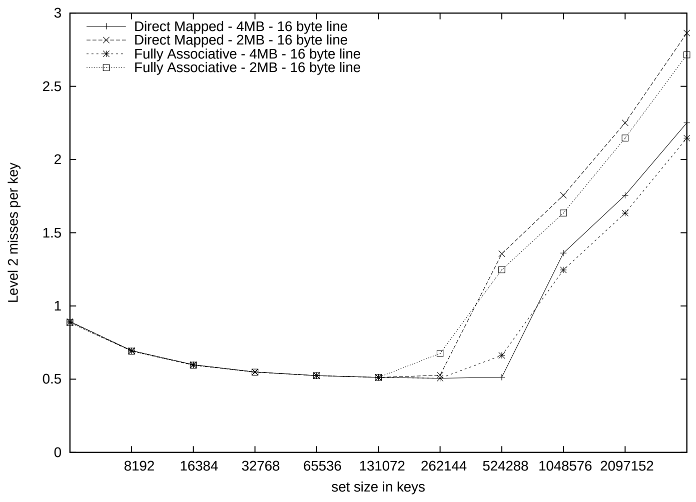
Figure A.6: Simulation results for cache heapsort (4-heap)



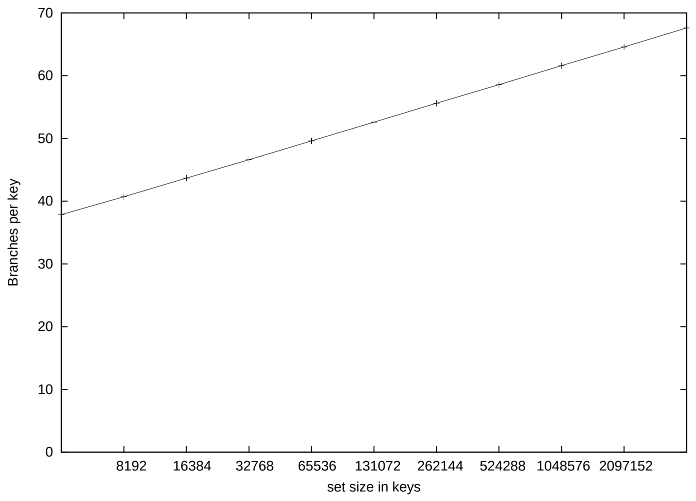
(a) Instructions per key



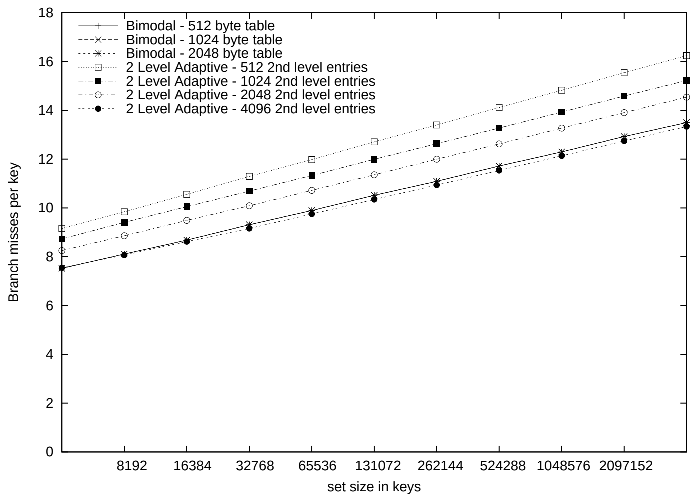
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

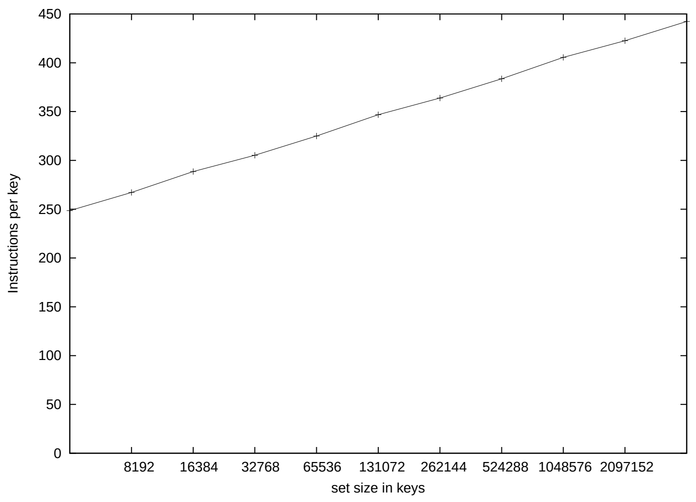


(d) Branches per key

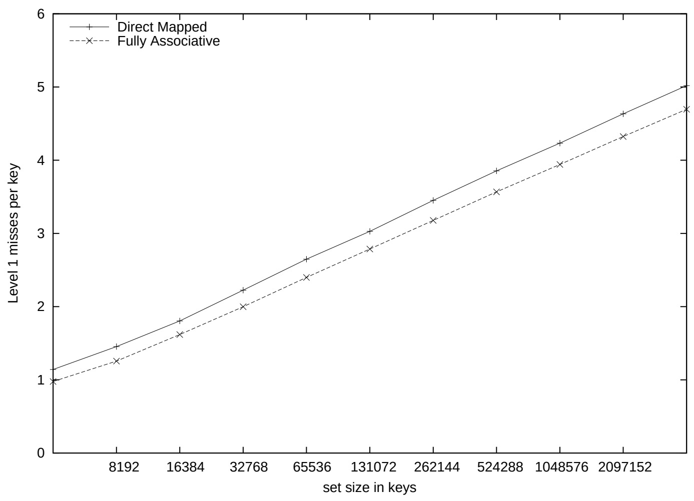


(e) Branch misses per key

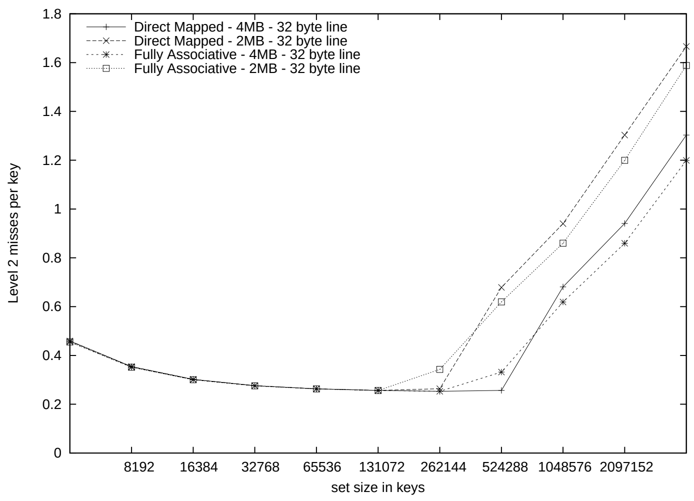
Figure A.7: Simulation results for cache heapsort (4-heap) (with 16-byte cache line)



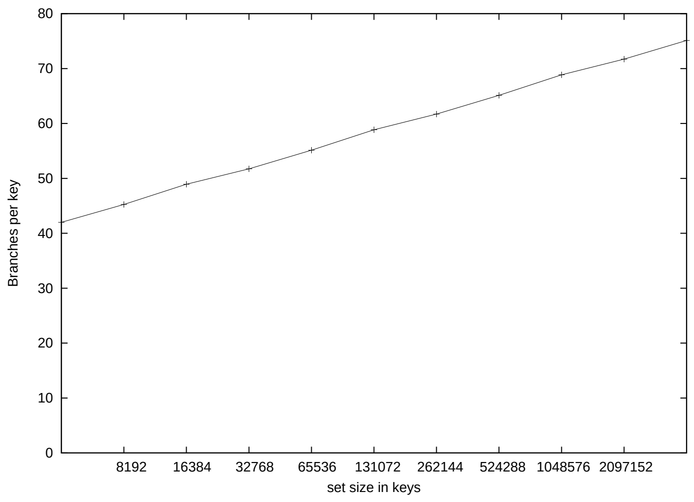
(a) Instructions per key



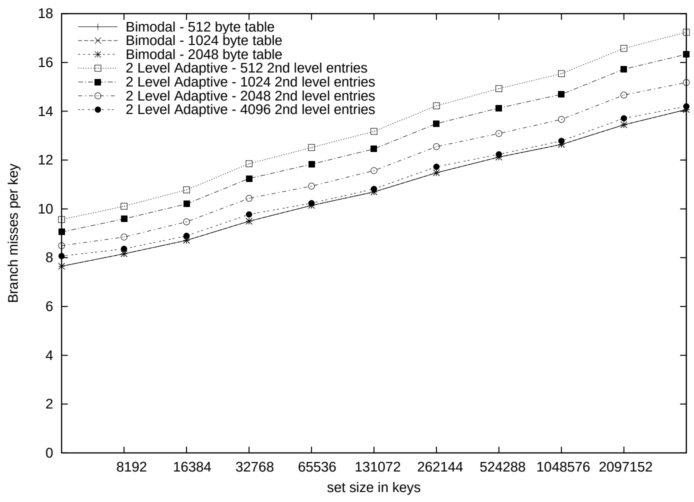
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

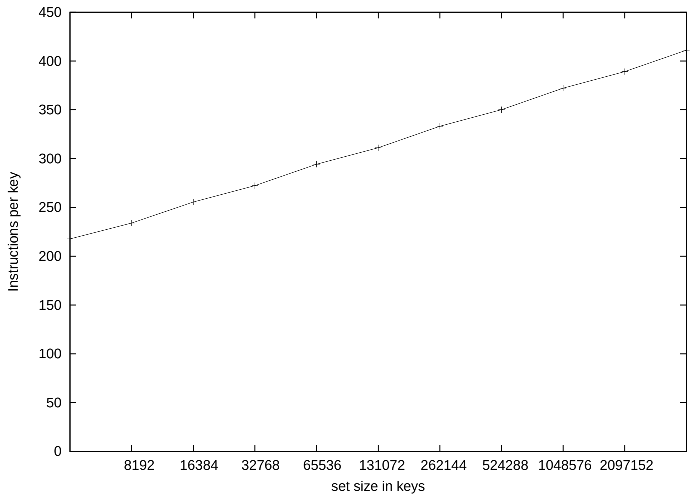


(d) Branches per key

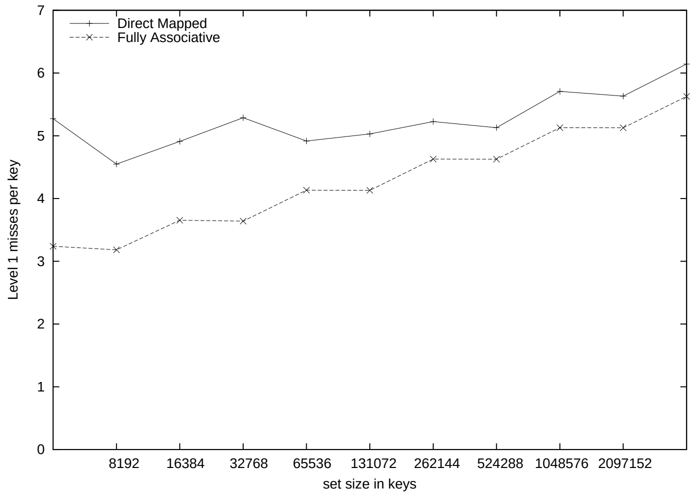


(e) Branch misses per key

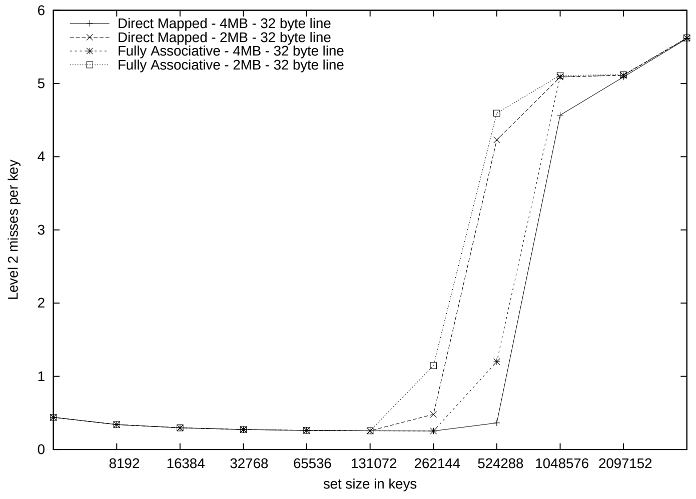
Figure A.8: Simulation results for cache heapsort (8-heap)



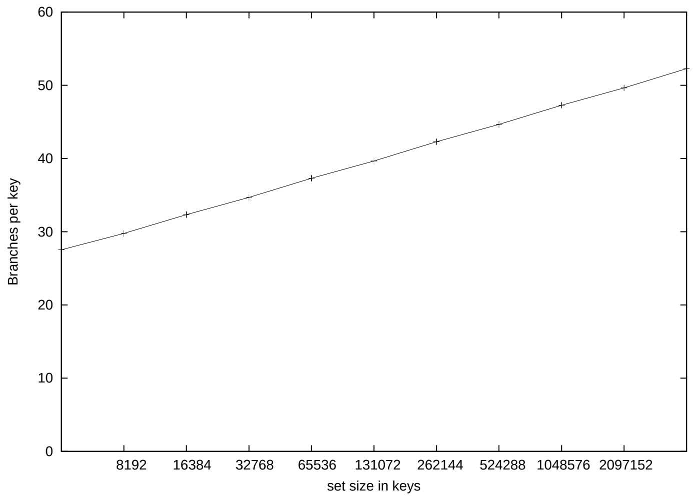
(a) Instructions per key



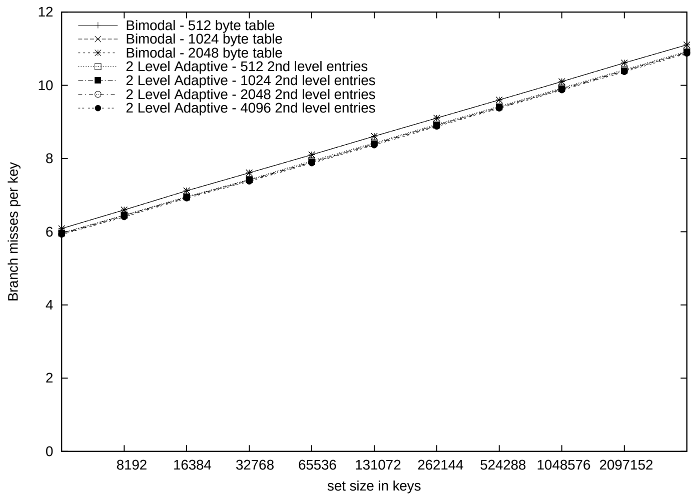
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

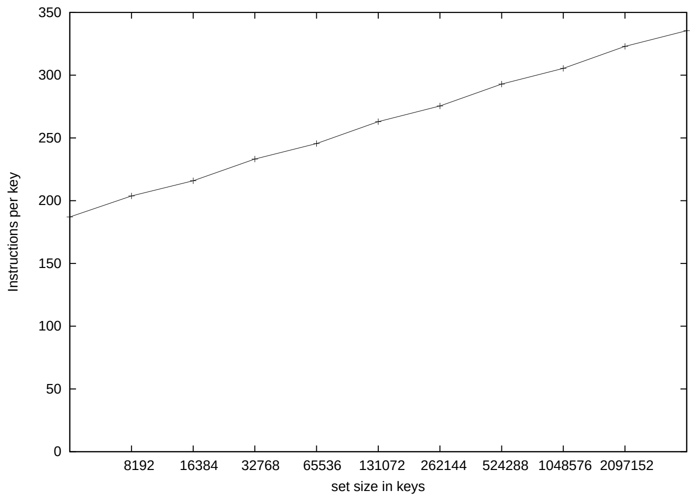


(d) Branches per key

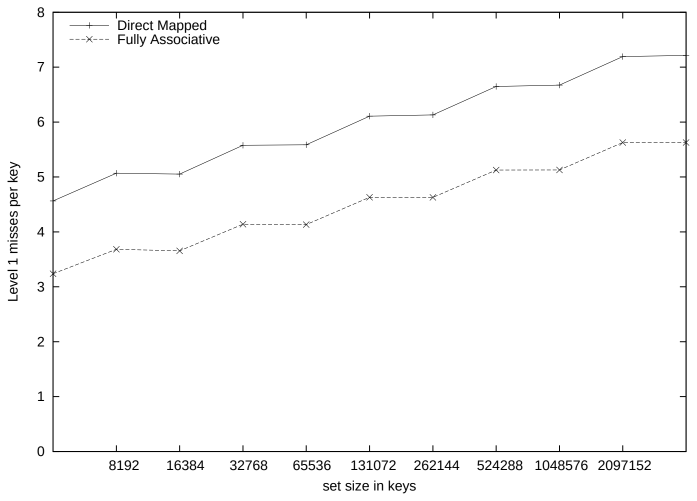


(e) Branch misses per key

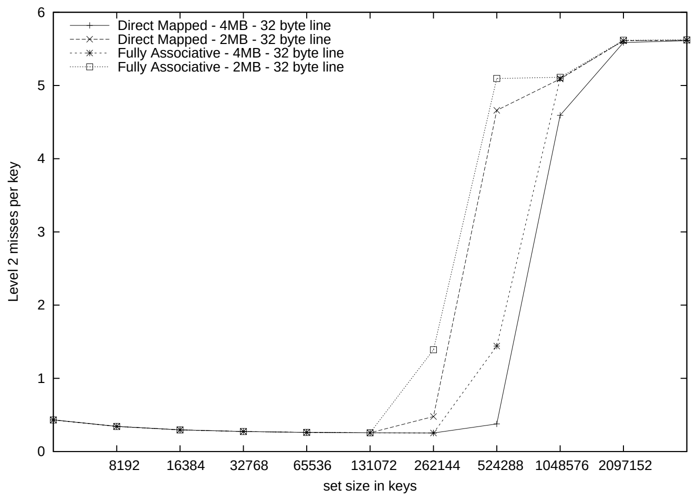
Figure A.9: Simulation results for algorithm N



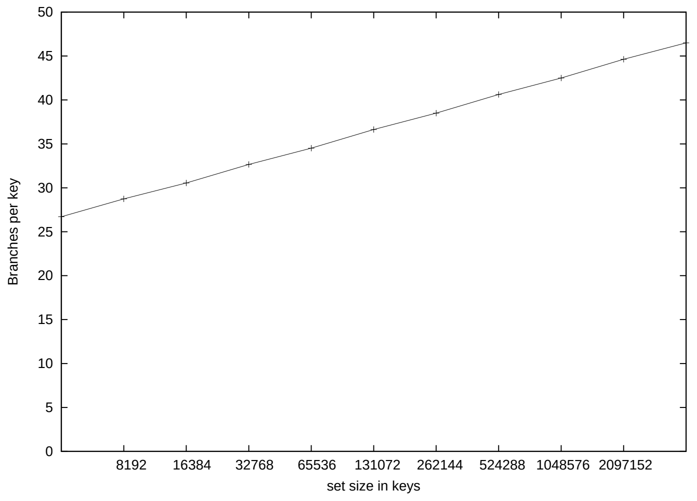
(a) Instructions per key



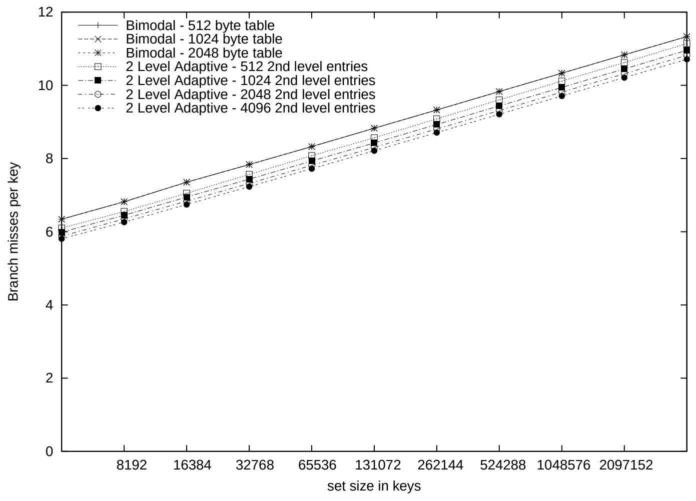
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

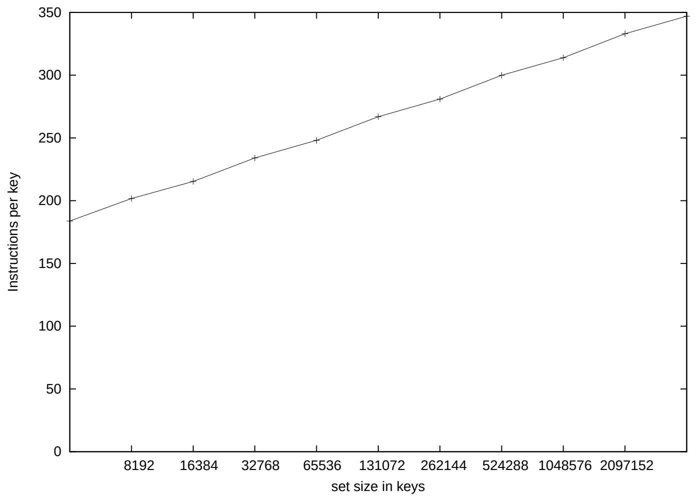


(d) Branches per key

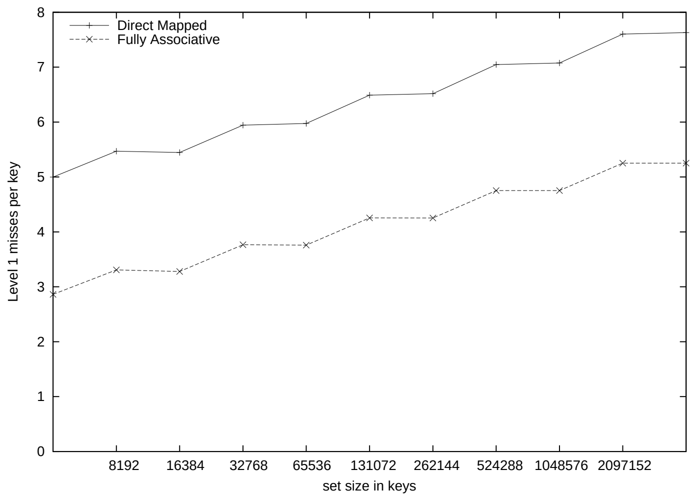


(e) Branch misses per key

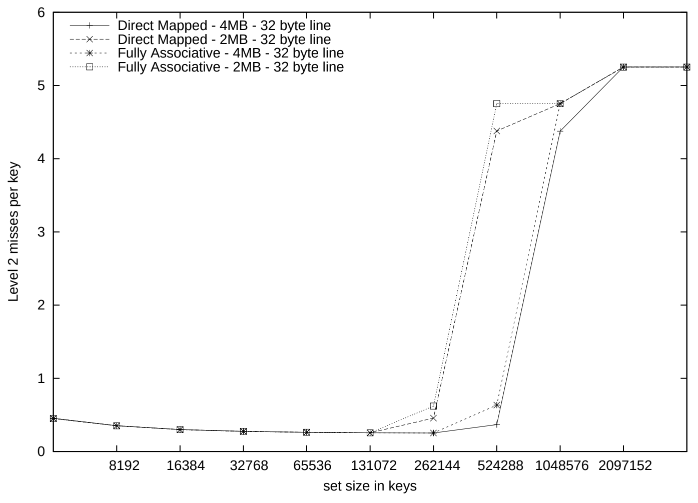
Figure A.10: Simulation results for algorithm S



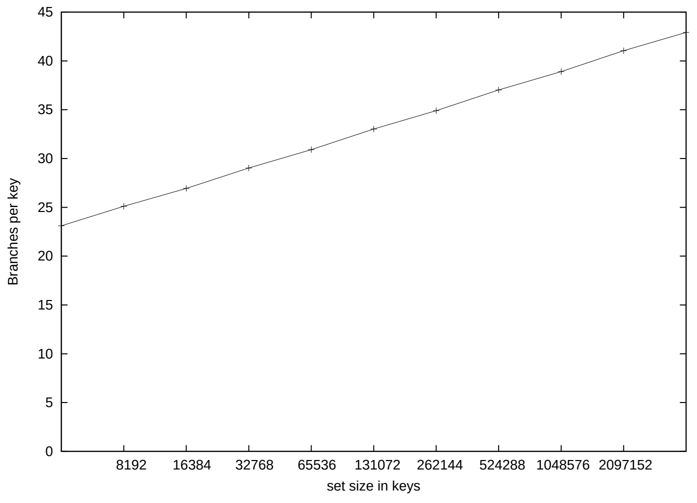
(a) Instructions per key



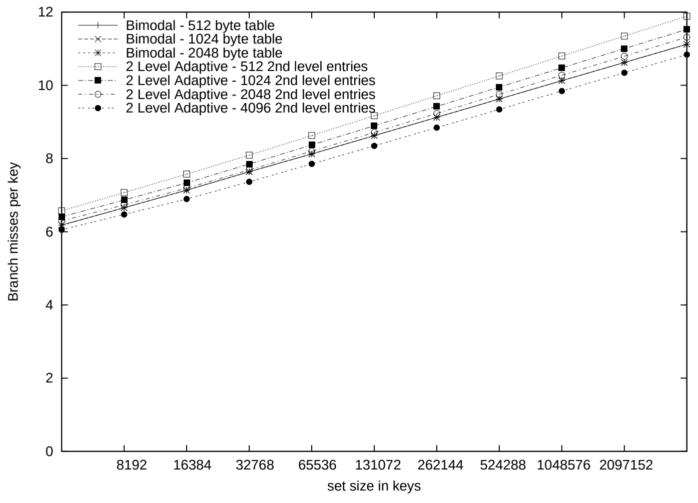
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

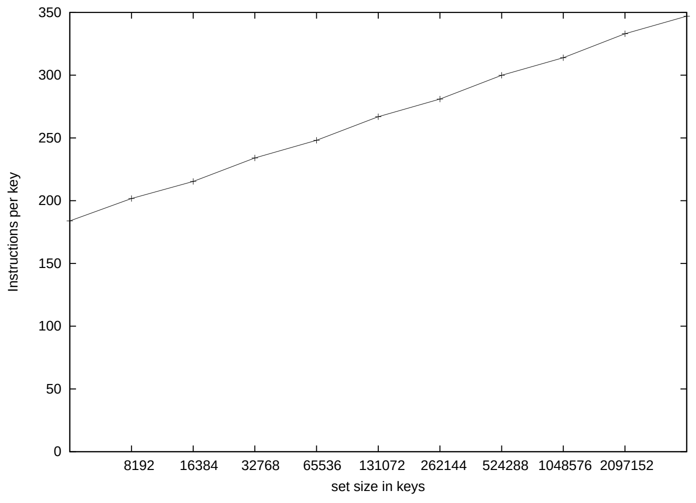


(d) Branches per key

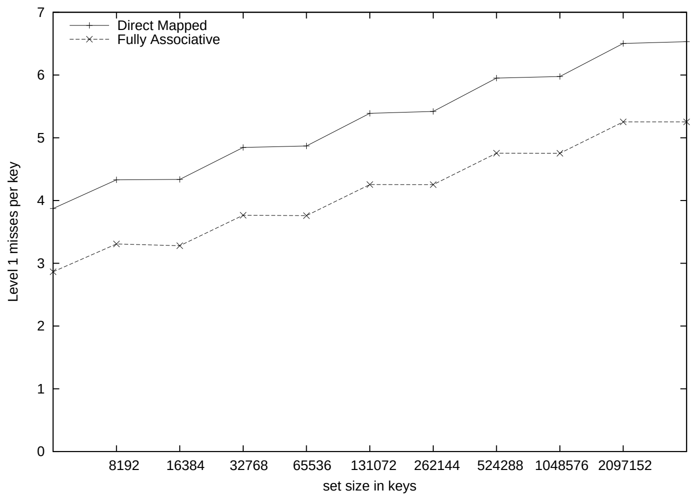


(e) Branch misses per key

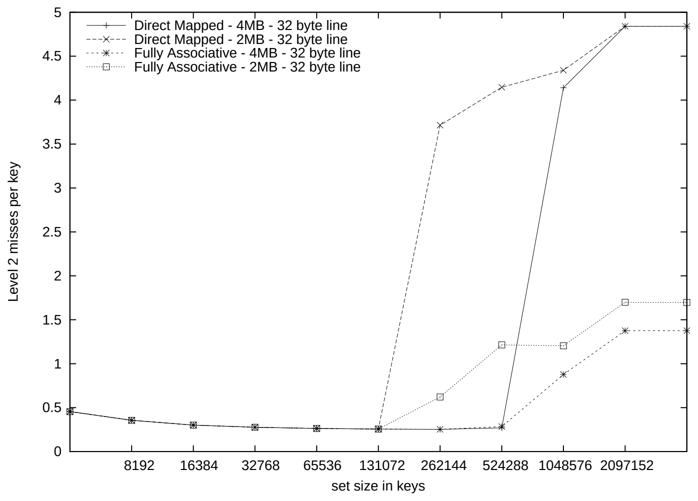
Figure A.11: Simulation results for base mergesort



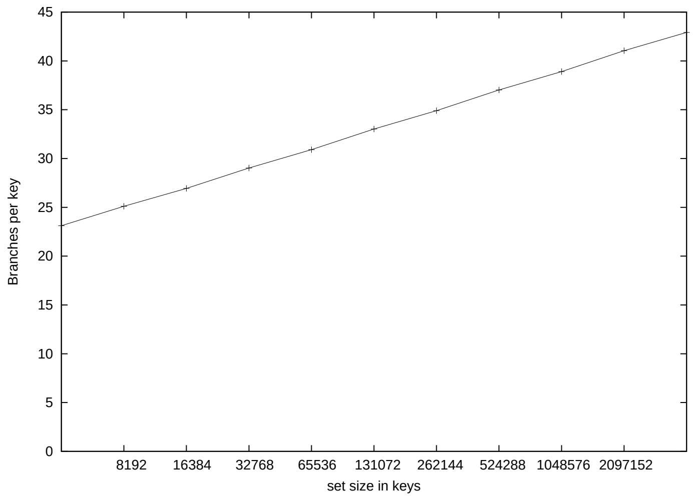
(a) Instructions per key



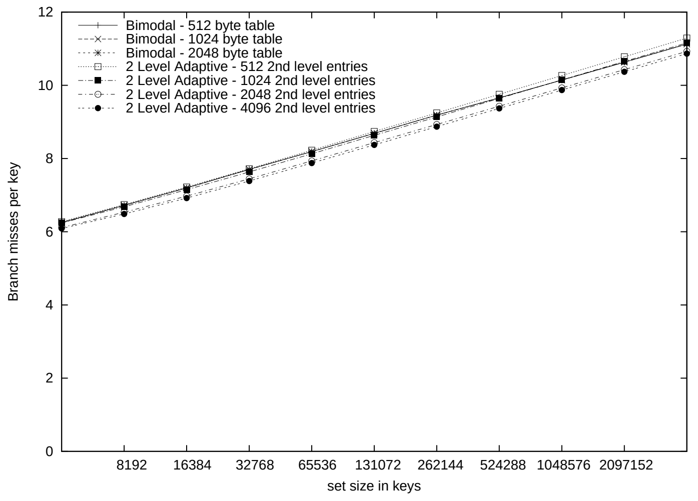
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

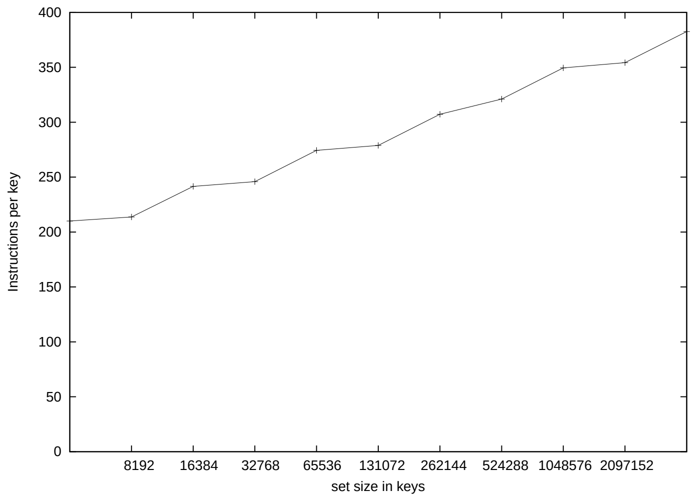


(d) Branches per key

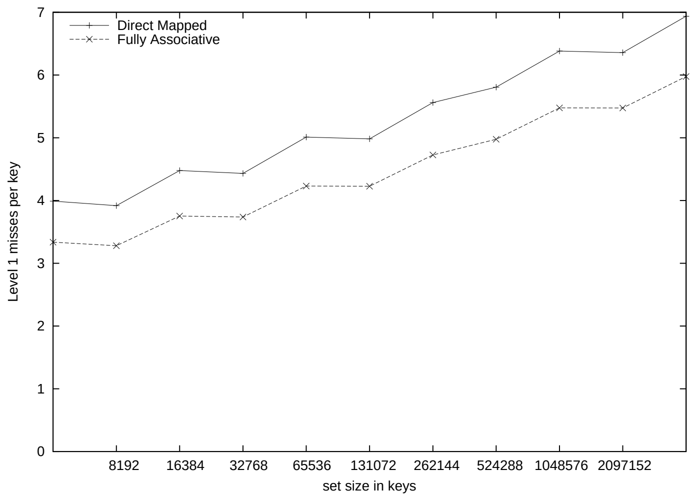


(e) Branch misses per key

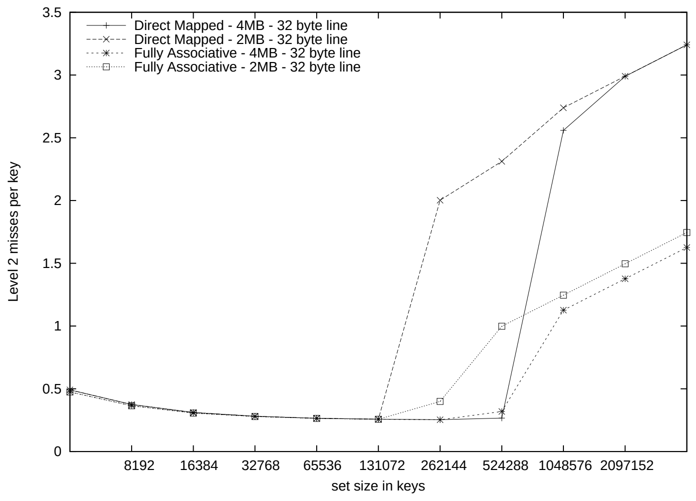
Figure A.12: Simulation results for tiled mergesort



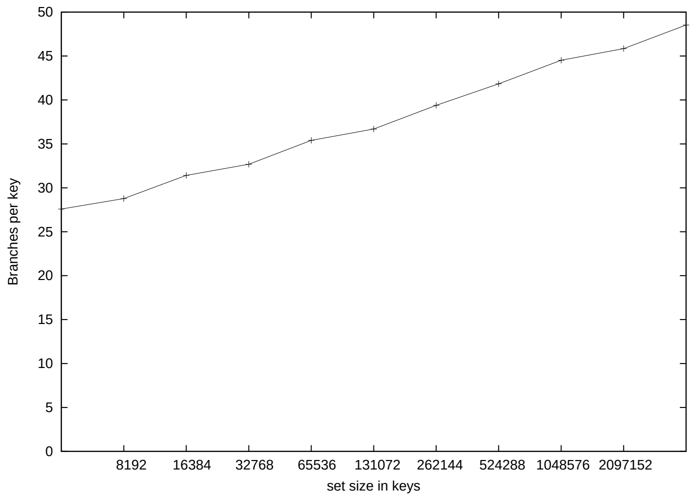
(a) Instructions per key



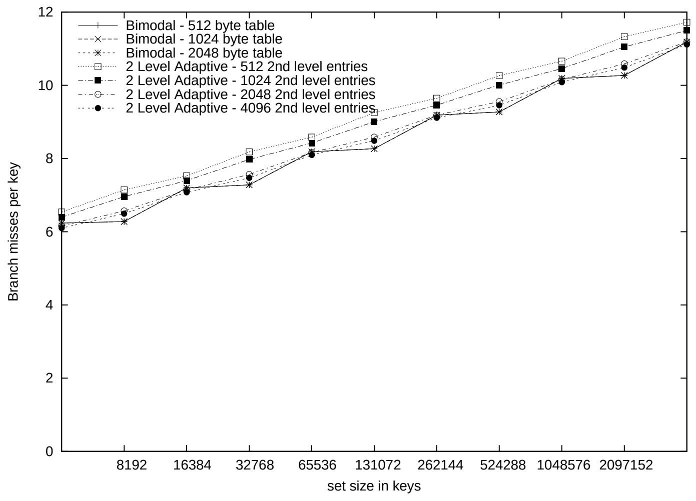
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key



(d) Branches per key



(e) Branch misses per key

Figure A.13: Simulation results for double tiled mergesort

(a) Instructions per key

(b) Level 1 cache misses per key

(c) Level 2 cache misses per key

(d) Branches per key

(e) Branch misses per key

Figure A.14: Simulation results for multi-mergesort

(a) Instructions per key

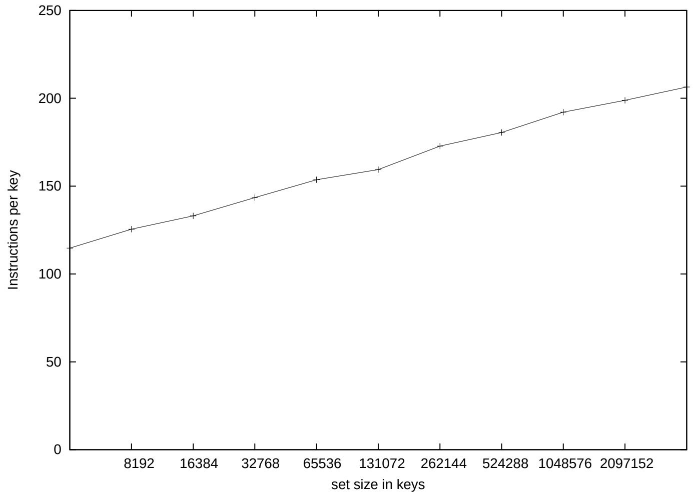
(b) Level 1 cache misses per key

(c) Level 2 cache misses per key

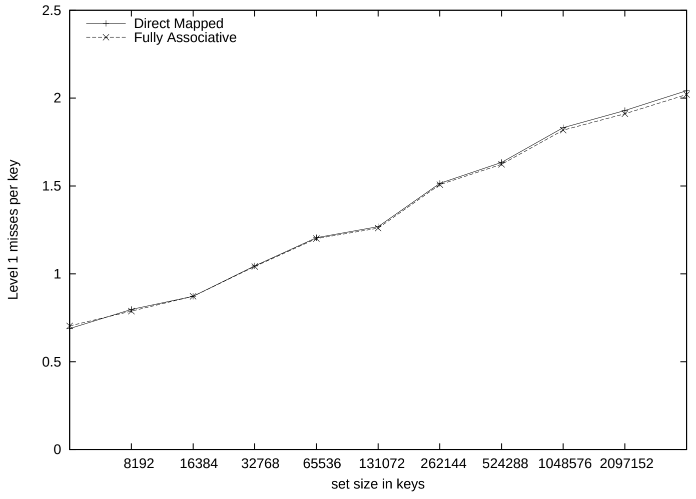
(d) Branches per key

(e) Branch misses per key

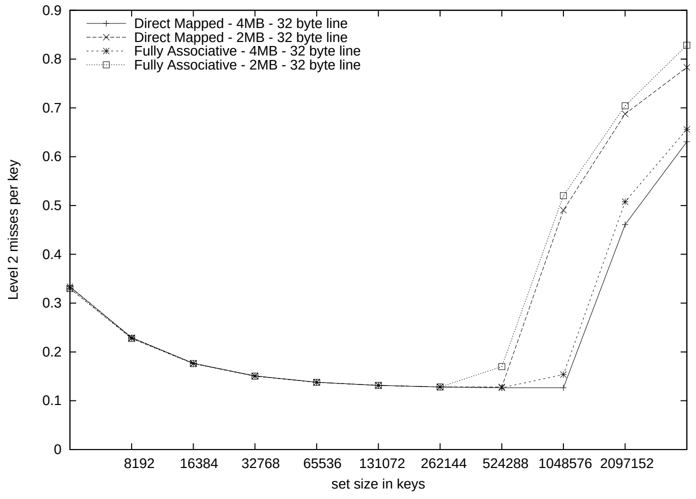
Figure A.15: Simulation results for double multi-mergesort



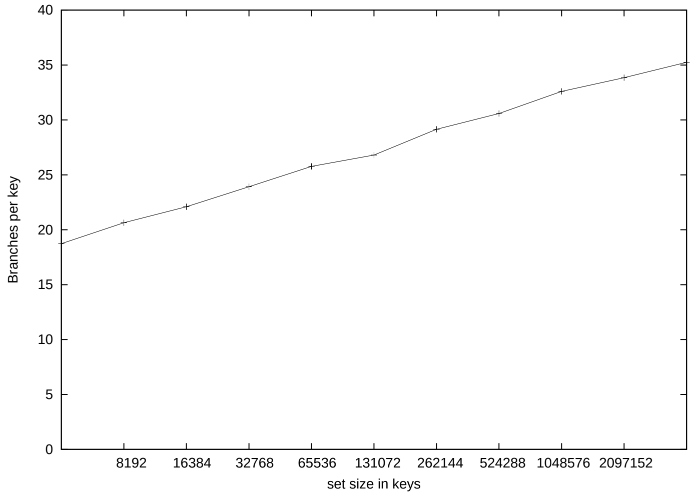
(a) Instructions per key



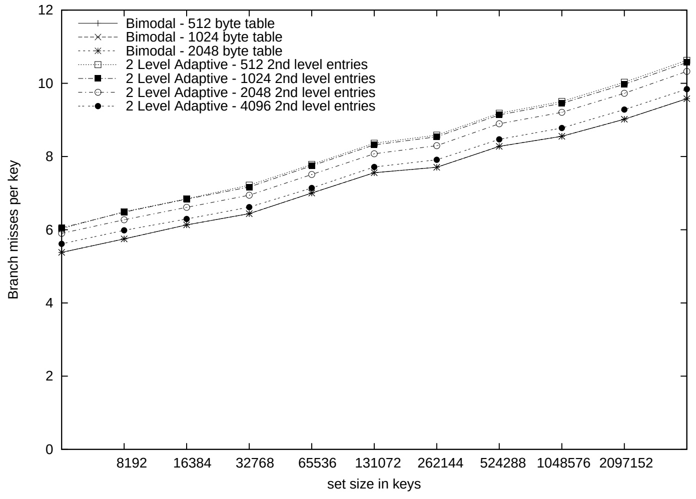
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

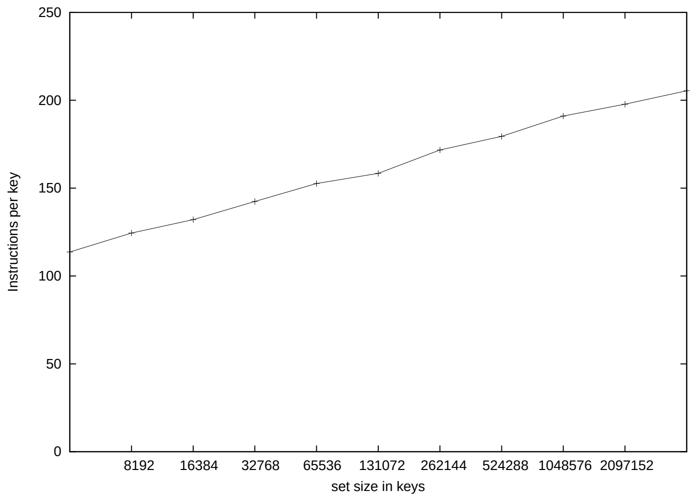


(d) Branches per key

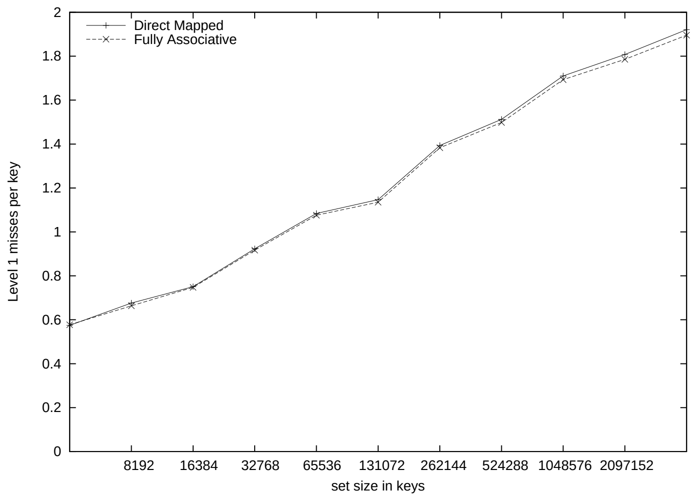


(e) Branch misses per key

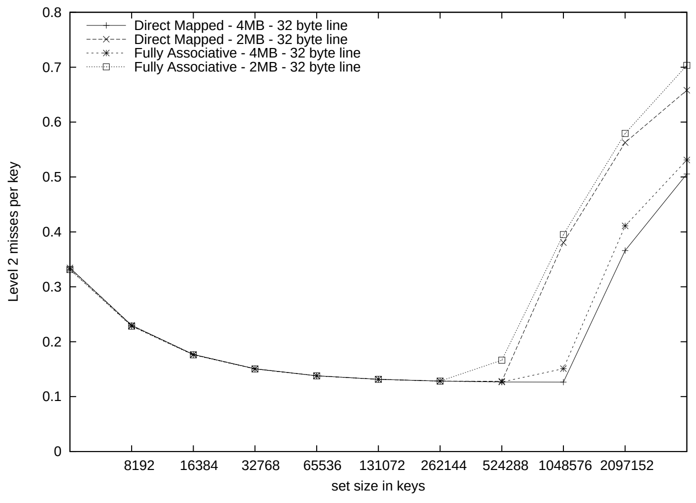
Figure A.16: Simulation results for base quicksort



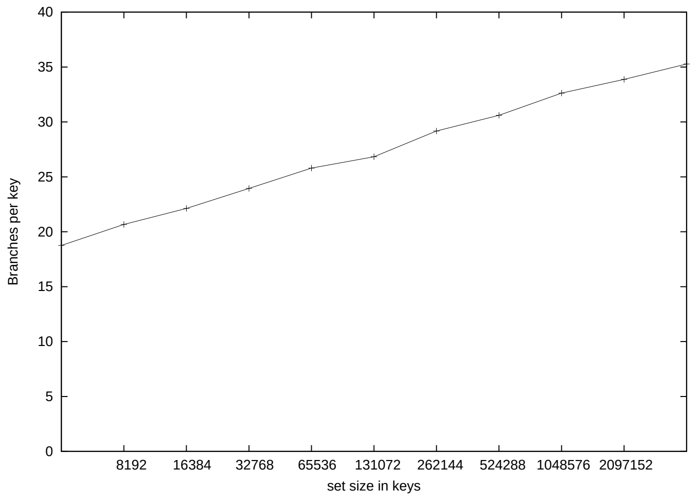
(a) Instructions per key



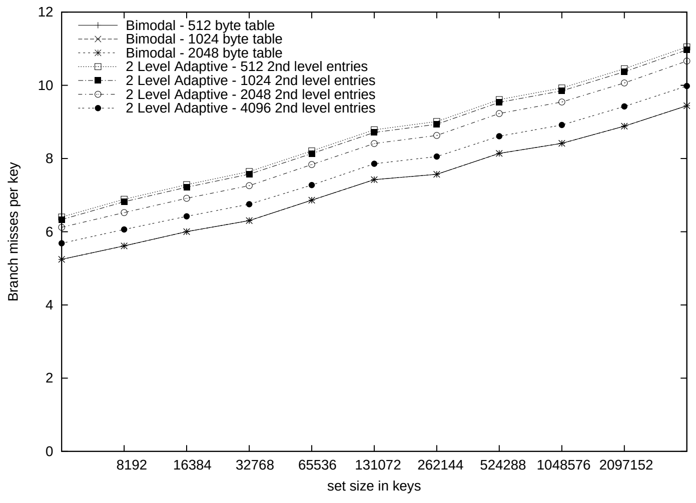
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

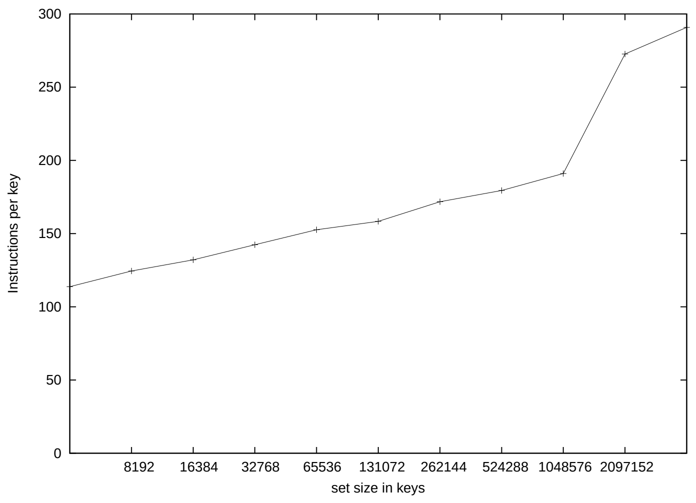


(d) Branches per key

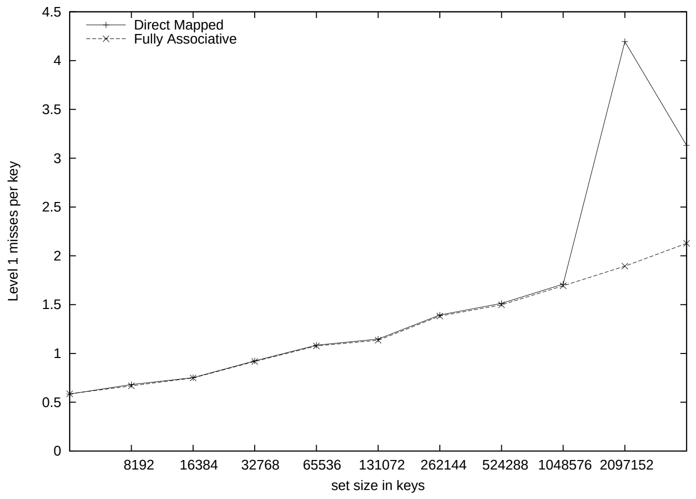


(e) Branch misses per key

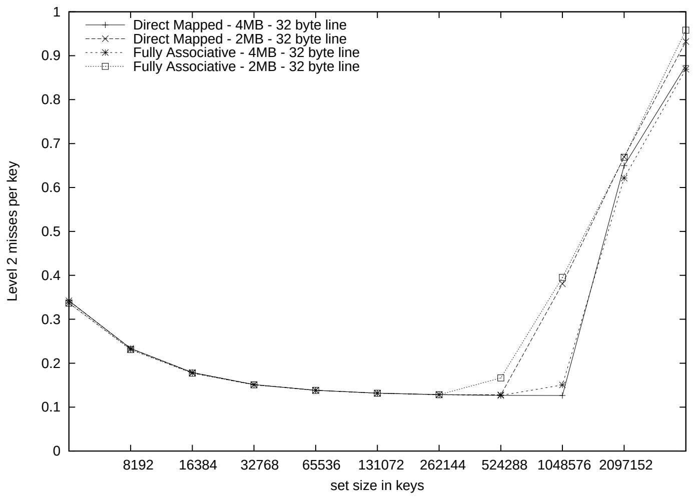
Figure A.17: Simulation results for memory-tuned quicksort



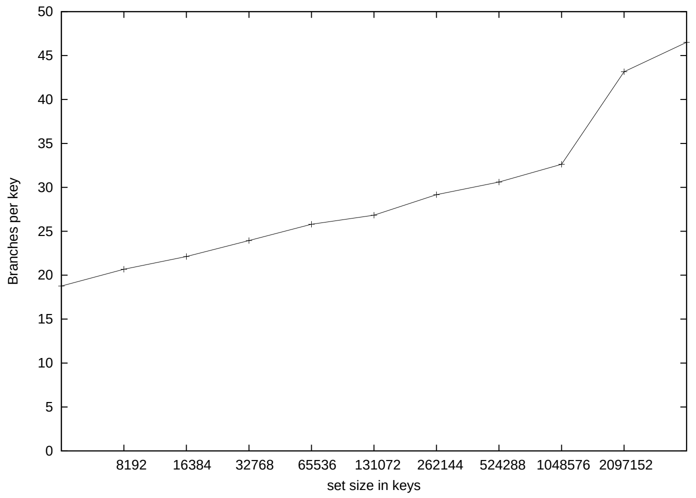
(a) Instructions per key



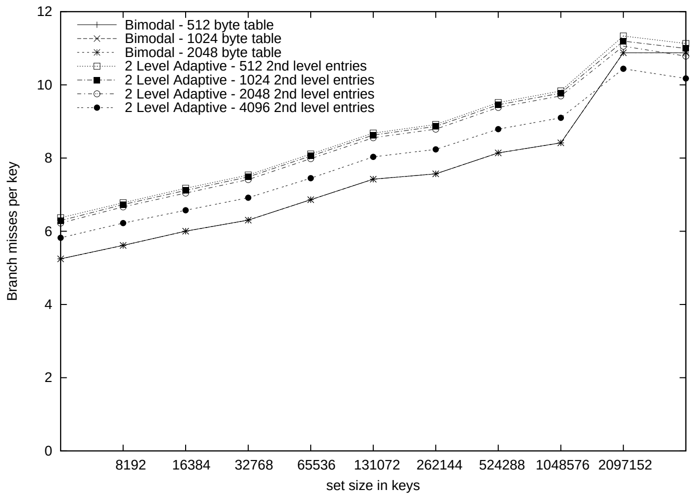
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

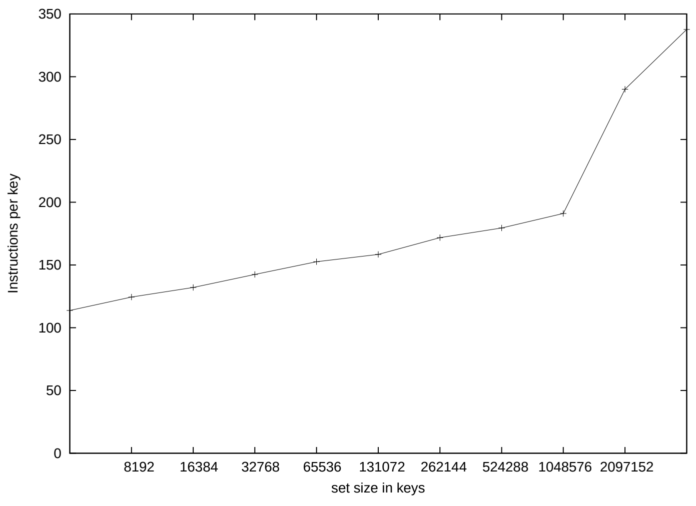


(d) Branches per key

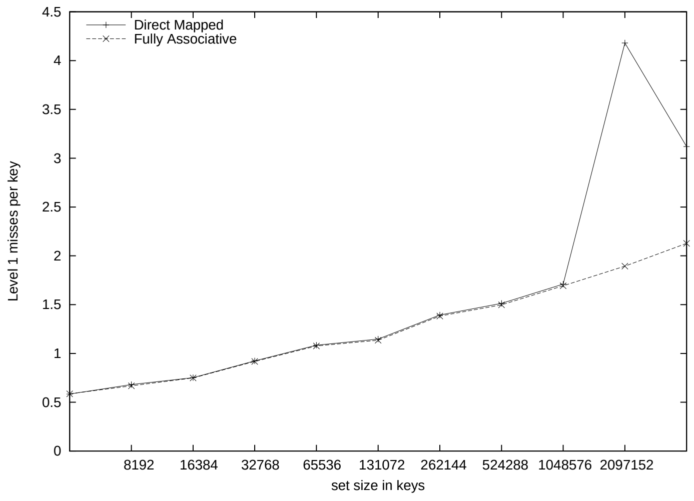


(e) Branch misses per key

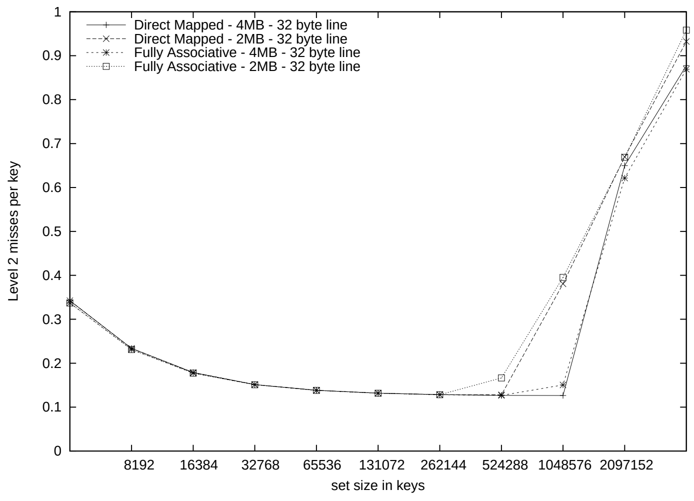
Figure A.18: Simulation results for multi-quicksort (binary search)



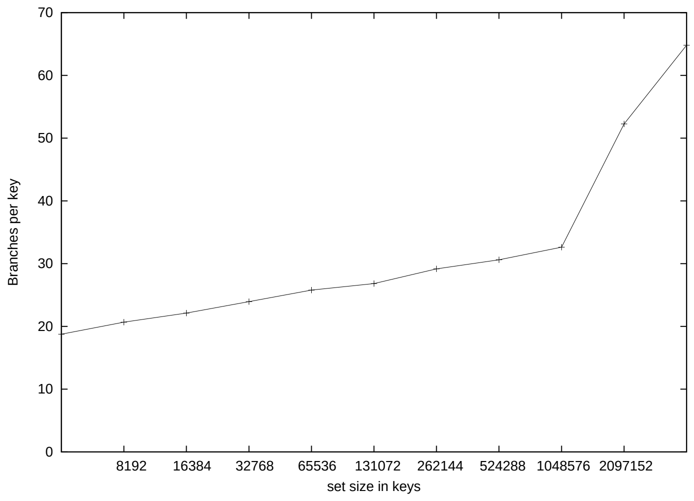
(a) Instructions per key



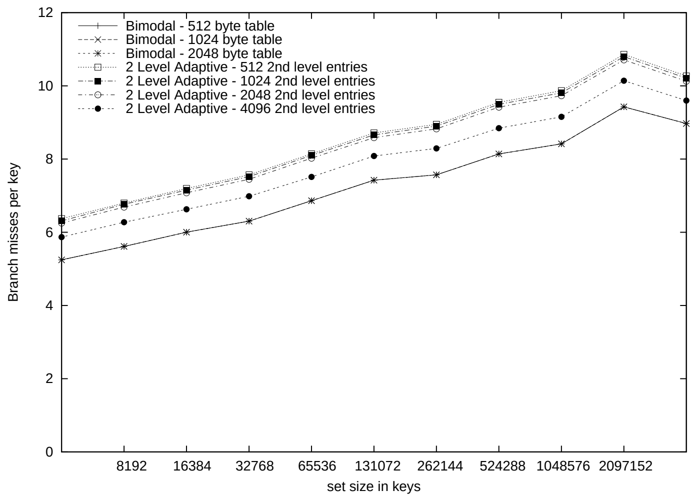
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key

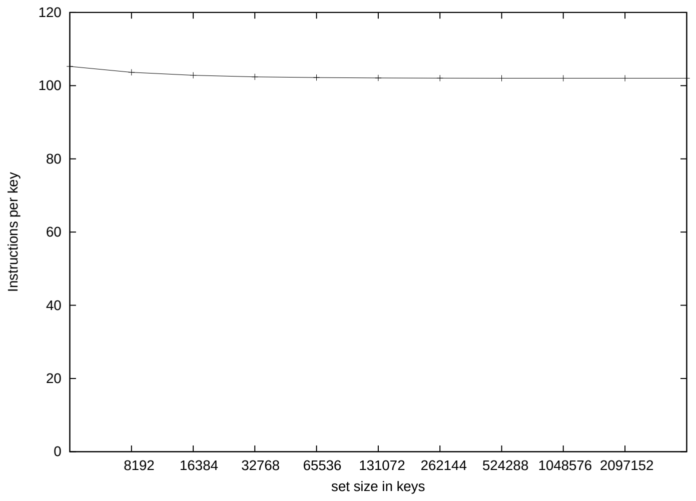


(d) Branches per key

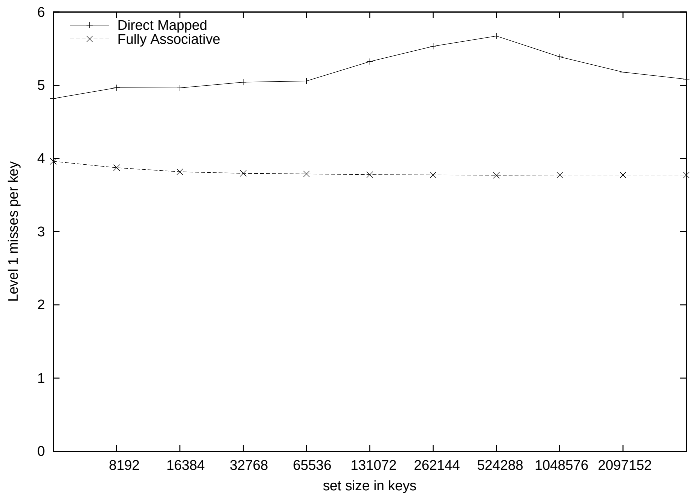


(e) Branch misses per key

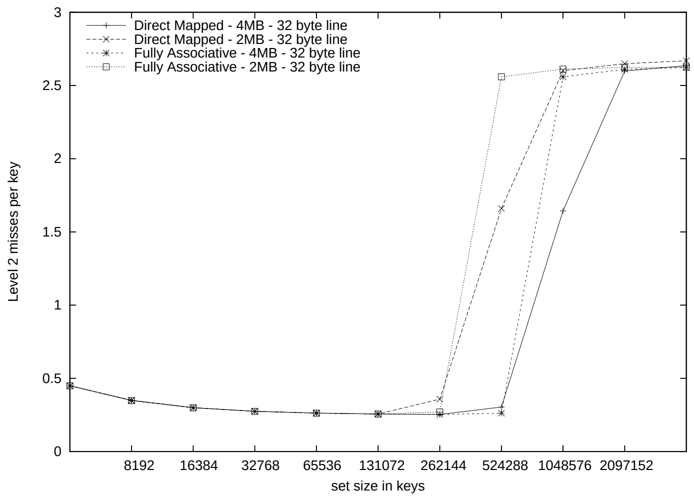
Figure A.19: Simulation results for multi-quicksort (sequential search)



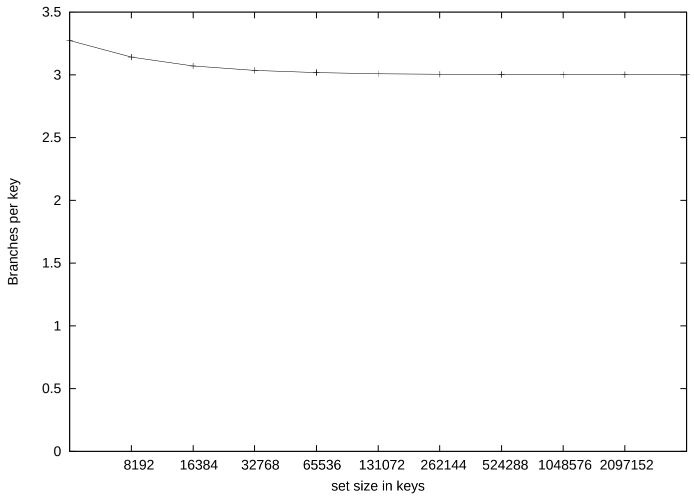
(a) Instructions per key



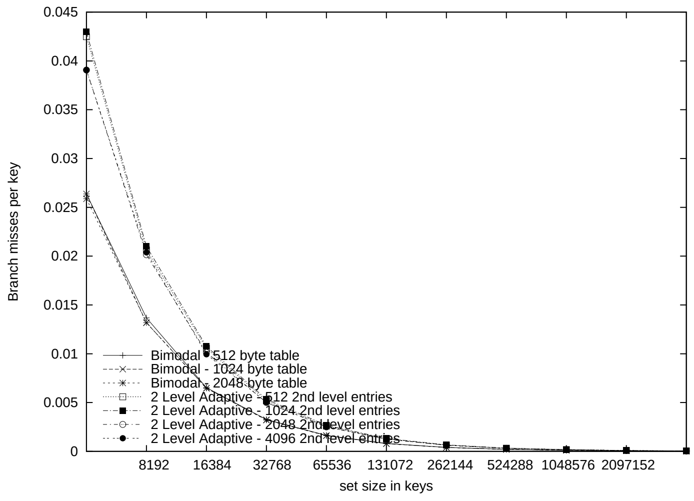
(b) Level 1 cache misses per key



(c) Level 2 cache misses per key



(d) Branches per key



(e) Branch misses per key

Figure A.20: Simulation results for radixsort

Appendix B

Code Listing

B.1 Simple Script

Below is the code for the script used to run the SimpleScalar simulations.

```
#!/usr/bin/perl -w
use Getopt::Std;
$command_line = $0;
foreach (@ARGV) { $command_line .= " $_"; }

# ——————————————————————————-
#                                   Do the options
# ——————————————————————————-

my %options=();
getopts("c:L:ihlm:M:s",\%options);

if (defined($options{"h"}) || (scalar(@ARGV) == 0))
{
      die ("simple [options] sort\noptions are:\n\t
             -c comment - comment with which to name log\n\t
             -l list commands simple will execute\n\t
             -s do a short run\n\t
             -h this help message\n\t
             -L length - cache line length\n\t
             -i just count instructions\n\t
             -m - specify minimum sort size\n\t
             -M - specify maximum sort size\n");
}

# i means just get an instruction count, nothing else
$just_instruction = 0;
if (defined($options{"i"})) { $just_instruction = 1; }
```

```
#redfine the max and minimum number of items
$min_num_items = 4096;
$max_num_items = 4194304;

if (defined($options{"m"})) { $min_num_items = $options{"m"}; }
if (defined($options{"M"})) { $max_num_items = $options{"M"}; }

$line_length = 32; # the length of a cache line
if (defined($options{"L"})) { $line_length = $options{"L"}; }

# check if a short run is desired
$short_run = 0;
if (defined($options{"s"})) { $short_run = 1; }

# check if just the list of commands is wanted
$just_list = 0;
if (defined($options{"l"})) { $just_list = 1; }

# set the comment
$comment = localtime();
if (defined($options{"c"})) { $comment = $options{"c"}; }
$comment =~ s/[\s:]/_/g;

# ------------------------------------------------------------------------------
#                                        set the constants
# ------------------------------------------------------------------------------

# the simulator details
@simulators = ( "sim-cache", "sim-bpred" ) ;

# put in the keywords
$keywords{"sim-cache"} =
        [
                "sim_num_insn", "sim_num_refs", "dl1.accesses", "dl1.hits",
                "dl1.misses", "dl1.replacements", "dl1.writebacks", "dl1.miss_rate",
                "dl1.repl_rate", "dl2.accesses", "dl2.hits", "dl2.misses",
                "dl2.replacements", "dl2.writebacks", "dl2.miss_rate", "dl2.repl_rate",
                "mem.page_mem"
        ];
$keywords{"sim-bpred"} =
        [
                "sim_num_branches", "sim_IPB", "bpred_bimod.lookups",
                "bpred_bimod.updates", "bpred_bimod.addr_hits", "bpred_bimod.dir_hits",
                "bpred_bimod.misses", "bpred_bimod.bpred_addr_rate",
                "bpred_bimod.bpred_dir_rate", "bpred_2lev.lookups",
                "bpred_2lev.updates", "bpred_2lev.addr_hits", "bpred_2lev.dir_hits",
                "bpred_2lev.misses", "bpred_2lev.bpred_addr_rate",
                "bpred_2lev.bpred_dir_rate"
        ];

#<name>:<nsets>:<bsize>:<assoc>:<repl>
# cache sizes are 2 and 4 MB, with a line size of 32 Bytes.
# This mean either the associativity or #sets is 65536/131072 and l1 is 256.
# we make l1 FA

$size_4_meg = 4 * 1024 * 1024 / $line_length;
$size_2_meg = 2 * 1024 * 1024 / $line_length;
```

```
$size_8_k = 8 * 1024 / $line_length;

$params{"sim-cache"}[0] = "-cache:dl1 dl1:$size_8_k:$line_length:1:l
            -cache:dl2 dl2:$size_4_meg:$line_length:1:l
            -cache:il1 il1:$size_8_k:$line_length:1:l -cache:il2 dl2";

$params{"sim-cache"}[1] = "-cache:dl1 dl1:$size_8_k:$line_length:1:l
            -cache:dl2 dl2:$size_2_meg:$line_length:1:l
            -cache:il1 il1:$size_8_k:$line_length:1:l -cache:il2 dl2";
$params{"sim-cache"}[2] = "-cache:dl1 dl1:1:$line_length:$size_8_k:l
            -cache:dl2 dl2:1:$line_length:$size_4_meg:l
            -cache:il1 il1:1:$line_length:$size_8_k:l -cache:il2 dl2";
$params{"sim-cache"}[3] = "-cache:dl1 dl1:1:$line_length:$size_8_k:l
            -cache:dl2 dl2:1:$line_length:$size_2_meg:l
            -cache:il1 il1:1:$line_length:$size_8_k:l -cache:il2 dl2";

$params{"sim-bpred"}[0] = "-bpred bimod -bpred:bimod 512";
$params{"sim-bpred"}[1] = "-bpred bimod -bpred:bimod 1024";
$params{"sim-bpred"}[2] = "-bpred bimod -bpred:bimod 2048";

$params{"sim-bpred"}[3] = "-bpred 2lev -bpred:2lev 1 512 10 1";
$params{"sim-bpred"}[4] = "-bpred 2lev -bpred:2lev 1 1024 10 1";
$params{"sim-bpred"}[5] = "-bpred 2lev -bpred:2lev 1 2048 10 1";
$params{"sim-bpred"}[6] = "-bpred 2lev -bpred:2lev 1 4096 10 1";

if ($short_run)
{
      # reduce the number of items
      $max_num_items = 32768; # this is caled down from (4096 * 1024) ie by 128

      # now simulate a smaller cache

      $params{"sim-cache"}[0] = "-cache:dl1 dl1:16:32:1:l
                  -cache:dl2 dl2:256:32:1:l -cache:il1 il1:16:32:1:l
                  -cache:il2 il2:256:32:1:l";
      $params{"sim-cache"}[1] = "-cache:dl1 dl1:16:32:1:l
                  -cache:dl2 dl2:512:32:1:l -cache:il1 il1:16:32:1:l
                  -cache:il2 il2:512:32:1:l";
      $params{"sim-cache"}[2] = "-cache:dl1 dl1:1:32:16:l
                  -cache:dl2 dl2:1:32:256:l -cache:il1 il1:16:32:1:l
                  -cache:il2 il2:1:32:256:l";
      $params{"sim-cache"}[3] = "-cache:dl1 dl1:1:32:16:l
                  -cache:dl2 dl2:1:32:512:l -cache:il1 il1:16:32:1:l
                  -cache:il2 il2:1:32:512:l";

}

$number_of_items = $min_num_items;

while($number_of_items < $max_num_items)
{
      push(@ns, $number_of_items);
      $number_of_items *= 2;
}
push(@ns, $max_num_items);

$log_directory = "./logs/";
```

```
# compiler settings
$CC = "sslittle-na-sstrix-gcc";
$CFLAGS = "-Wall -O3 -funroll-loops -finline-functions";
$SIMPLEFLAGS = ""; #-lm

select STDOUT;

$dashes = "-----------------------------------------------------\n";
# each sort, do a cache for each size, then a bpred for each size


if ($just_instruction)
{
        $sortname = $ARGV[0];
        @seeds = ( "0", "20", "1234", "63345", "23" );
        # we max out at 32768 unless you specify more with the -m.
        # you can specify less with the -M
        if ($max_num_items >= 32768)
        {
                $max_num_items = 32768;
        }
        if ($min_num_items > $max_num_items)
        {
                $max_num_items = $min_num_items;
        }

        foreach $seed (@seeds)
        {
                @instruction_count = ();
                $comp_command = "$CC $CFLAGS $SIMPLEFLAGS -funroll-loops
                -finline-functions -DRANDOM_SIZE=$max_num_items -DSEED=$seed
                -DSORT=$sortname $sortname.c simpleMain.c -o special_file";
                # run the command
                # capture the compiler errors (STDERR)
                $comp_output = `$comp_command 2>&1 1>/dev/null`;

                # if it doesnt compile stop and print out errors
                if ($comp_output =~ /./)
                {
                        die("This command returned errors:
                                \"$comp_command\"\n$comp_output");
                }
                $sim_command = "sim-fast special_file";
                @sim_output = `$sim_command 2>&1 1>/dev/null`;

                @instruction_count = grep(/sim_num_insn/, @sim_output);
                print $instruction_count[0];
        }

        system("rm special_file");
        exit;
}


SORT: foreach $sortname (@ARGV)
{
```

```
$sortfile = $sortname.int(rand(2147483648)).".c";
$object_file = $sortname.int(rand(2147483648));
system("cp $sortname.c $sortfile");


# make a temp file
$n_count = 0;
foreach $n (@ns) # ther should be num_sim_runs * count($n) columns.
{
        $comp_command = "$CC $CFLAGS $SIMPLEFLAGS -DRANDOM_SIZE=$n -DSEED=0
                -DSORT=$sortname $sortfile simpleMain.c -o $object_file";

        print $comp_command."\n";
        push (@command_list, $comp_command);

        # run the command
        # capture the compiler errors (STDERR)
        $comp_output = `$comp_command 2>&1 1>/dev/null`;

        # if it doesnt compile stop and print out errors
        if ($comp_output =~ /./)
        {
                print $comp_output;
                next SORT;
        }

        foreach $simulator (@simulators) # run each of the simulators
        {
                $order_count = 0;
                $run_count = 0;
                foreach $param (@{$params{$simulator}})
                {
                        $sim_command = "$simulator $param $object_file";
                        push (@command_list, $sim_command);
                        print $sim_command."\n";
                        if ($just_list) { next; }

                        @sim_output = `nice -19 $sim_command 2>&1 1>/dev/null`;

                        # ignore the headers we dont want
                        while($line = shift(@sim_output))
                        {
                                if ($line =~ /sim: \*\* simulation statistics \*\*/)
                                        { last; }
                                if ($line =~ /fatal/)
                                        { die("Fatal error with
                                                command $sim_command"); }
                        }

                        # fill the array with data in the correct order
                        while(shift(@sim_output) =~ /(\S+)\s+(\S+).*/)
                        {
                                # once for each simulator
                                if (!($results{$simulator}{$1}))
                                {
                                        $order{$simulator}[$order_count++] = $1;
                                }
                @{ $resuℓts{$simuℓator}{$1} }[$n_count *
                    scalar(@{$params{$simuℓator}}) + $run_count]
                    = $2;
            }
            $run_count++;
        }
    }
    $n_count++;
    system("rm -f $object_file");
}

if ($just_list)
{
    system("rm $sortfile");
    exit;
}

push (@my_output, "\n$dashes Results for $sortname\n$dashes");
foreach $simuℓator (@simulators)
{
    @output = ();
    for ($i = 0; $i < scalar(@{$order{$simuℓator}}); $i++)
    {
        $keyword = $order{$simuℓator}[$i];
        $output_string = sprintf("%-36s ", $keyword);
        @result_array = @{ $resuℓts{$simuℓator}{$keyword} };
        foreach $resuℓt (@result_array)
        {
            if (defined($resuℓt))
                { $output_string .= sprintf("%13s |", $resuℓt);}
            else { $output_string .= "xxxxxxxxxxxxx |"; }
        }
        $output_string .= "\n";
        push (@output, $output_string);
    }

    # do headers here
    $header = "$dashes";
    $header .= sprintf("%-36s ", $simuℓator);
    foreach $n (@ns)
    {
        foreach $param (@{$params{$simuℓator}})
        {
            $header .= sprintf("%13s |", $n);
        }
    }
    foreach $param (@{$params{$simuℓator}})
    {
        $header .= "\n$param";
    }
    $header .= "\n$dashes";

    push (@log_output, $header);
    push (@log_output, @output);

    $keyword_string = join ("|", @{$keywords{$simuℓator}});
    @temp = grep { /$keyword_string/; } @output;
```

```
        push (@my_output, $header);
        push (@my_output, @temp);

    }


    # we store the complete log
    $ℓogname = "$sortname\_$comment";
    open(LOG, ">".$ℓog_directory.$ℓogname);
    print LOG "".localtime()."\n";
    print LOG "Run as: \"$command_line\"\n";
    print LOG @log_output;

    print LOG "\nCommand list: \n";
    foreach $command (@command_list)
    {
        print LOG "\t".$command."\n";
    }

    close(LOG);

    # remove the temp
    system("rm $sortfile");

}
print @my_output;
```

B.2 Sample Simulation Run

The instructions listed below form a sample simulation run of SimpleScalar. Each of the commands is run in turn and its results stored by `simple`.

```
sslittle−na−sstrix−gcc −Wall −O3 −funroll−loops −finline−functions
        −DRANDOM_SIZE=4096 −DSEED=0 −DSORT=base_heapsort
        base_heapsort.c simpleMain.c −o base_heapsort
sim−cache −cache:dl1 dl1:256:32:1:l −cache:dl2 dl2:131072:32:1:l
                      −cache:il1 il1:256:32:1:l −cache:il2 dl2 base_heapsort
sim−cache −cache:dl1 dl1:256:32:1:l −cache:dl2 dl2:65536:32:1:l
                      −cache:il1 il1:256:32:1:l −cache:il2 dl2 base_heapsort
sim−cache −cache:dl1 dl1:1:32:256:l −cache:dl2 dl2:1:32:131072:l
                      −cache:il1 il1:1:32:256:l −cache:il2 dl2 base_heapsort
sim−cache −cache:dl1 dl1:1:32:256:l −cache:dl2 dl2:1:32:65536:l
                      −cache:il1 il1:1:32:256:l −cache:il2 dl2 base_heapsort
sim−bpred −bpred bimod −bpred:bimod 512 base_heapsort
sim−bpred −bpred bimod −bpred:bimod 1024 base_heapsort
sim−bpred −bpred bimod −bpred:bimod 2048 base_heapsort
sim−bpred −bpred 2ℓev −bpred:2ℓev 1 512 10 1 base_heapsort
sim−bpred −bpred 2ℓev −bpred:2ℓev 1 1024 10 1 base_heapsort
sim−bpred −bpred 2ℓev −bpred:2ℓev 1 2048 10 1 base_heapsort
sim−bpred −bpred 2ℓev −bpred:2ℓev 1 4096 10 1 base_heapsort
```

```
                                   .
                                   .
                                   .
                                   .
                                   .
                                   .

sslittle-na-sstrix-gcc -Wall -O3 -funroll-loops -finline-functions
        -DRANDOM_SIZE=4194304 -DSEED=0 -DSORT=base_heapsort
        base_heapsort.c simpleMain.c -o base_heapsort
sim-cache -cache:dl1 dl1:256:32:1:l -cache:dl2 dl2:131072:32:1:l
                        -cache:il1 il1:256:32:1:l -cache:il2 dl2 base_heapsort
sim-cache -cache:dl1 dl1:256:32:1:l -cache:dl2 dl2:65536:32:1:l
                        -cache:il1 il1:256:32:1:l -cache:il2 dl2 base_heapsort
sim-cache -cache:dl1 dl1:1:32:256:l -cache:dl2 dl2:1:32:131072:l
                        -cache:il1 il1:1:32:256:l -cache:il2 dl2 base_heapsort
sim-cache -cache:dl1 dl1:1:32:256:l -cache:dl2 dl2:1:32:65536:l
                        -cache:il1 il1:1:32:256:l -cache:il2 dl2 base_heapsort
sim-bpred -bpred bimod -bpred:bimod 512 base_heapsort
sim-bpred -bpred bimod -bpred:bimod 1024 base_heapsort
sim-bpred -bpred bimod -bpred:bimod 2048 base_heapsort
sim-bpred -bpred 2ℓev -bpred:2ℓev 1 512 10 1 base_heapsort
sim-bpred -bpred 2ℓev -bpred:2ℓev 1 1024 10 1 base_heapsort
sim-bpred -bpred 2ℓev -bpred:2ℓev 1 2048 10 1 base_heapsort
sim-bpred -bpred 2ℓev -bpred:2ℓev 1 4096 10 1 base_heapsort
```

B.3 Sample Simulation

The text below is an output of a simgle run of `sim-bpred`.

```
sim-bpred: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use. No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

sim: command line: sim-bpred base_heapsort454574856

sim: simulation started @ Wed Apr 28 04:07:08 2004, options follow:

sim-bpred: This simulator implements a branch predictor analyzer.

# -config                    # load configuration from a file
# -dumpconfig                # dump configuration to a file
# -h                   false # print help message
# -v                   false # verbose operation
# -d                   false # enable debug message
# -i                   false # start in Dlite debugger
-seed                      1 # random number generator seed (0 for timer seed)
# -q                   false # initialize and terminate immediately
# -chkpt              <null> # restore EIO trace execution from <fname>
# -redir:sim           <null> # redirect simulator output to file (non-interactive only)
# -redir:prog          <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst                   0 # maximum number of inst's to execute
-bpred                  bimod # branch predictor type {nottaken|taken|bimod|2lev|comb}
-bpred:bimod 2048 # bimodal predictor config (<table size>)
-bpred:2lev  1 1024 8 0 # 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:comb 1024 # combining predictor config (<meta_table_size>)
-bpred:ras                  8 # return address stack size (0 for no return stack)
-bpred:btb   512 4 # BTB config (<num_sets> <associativity>)

  Branch predictor configuration examples for 2-level predictor:
    Configurations: N, M, W, X
      N  # entries in first level (# of shift register(s))
      W  width of shift register(s)
      M  # entries in 2nd level (# of counters, or other FSM)
      X  (yes-1/no-0) xor history and address for 2nd level index
    Sample predictors:
      GAg   : 1, W, 2^W, 0
      GAp   : 1, W, M (M > 2^W), 0
      PAg   : N, W, 2^W, 0
      PAp   : N, W, M (M == 2^(N+W)), 0
      gshare : 1, W, 2^W, 1
  Predictor 'comb' combines a bimodal and a 2-level predictor.



sim: ** starting functional simulation w/ predictors **

sim: ** simulation statistics **
sim_num_insn           1045897 # total number of instructions executed
sim_num_refs            264304 # total number of loads and stores executed
sim_elapsed_time             1 # total simulation time in seconds
sim_inst_rate     1045897.0000 # simulation speed (in insts/sec)
sim_num_branches        167628 # total number of branches executed
sim_IPB                 6.2394 # instruction per branch
bpred_bimod.lookups     167628 # total number of bpred lookups
bpred_bimod.updates     167628 # total number of updates
bpred_bimod.addr_hits   139898 # total number of address-predicted hits
bpred_bimod.dir_hits    140006 # total number of direction-predicted hits
bpred_bimod.misses       27622 # total number of misses
bpred_bimod.jr_hits       8551 # total number of address-predicted hits for JRs
bpred_bimod.jr_seen       8552 # total number of JR's seen
bpred_bimod.jr_non_ras_hits.PP 0 # total number of address-predicted hits for non-RAS JRs
bpred_bimod.jr_non_ras_seen.PP 0 # total number of non-RAS JR's seen
bpred_bimod.bpred_addr_rate 0.8346 # branch address-prediction rate
bpred_bimod.bpred_dir_rate 0.8352 # branch direction-prediction rate
bpred_bimod.bpred_jr_rate 0.9999 # JR address-prediction rate
bpred_bimod.bpred_jr_non_ras_rate.PP <error: divide by 0> # non-RAS JR addr-pred rate
bpred_bimod.retstack_pushes     8554 # total number of address pushed onto ret-addr stack
bpred_bimod.retstack_pops     8552 # total number of address popped off of ret-addr stack
bpred_bimod.used_ras.PP     8552 # total number of RAS predictions used
bpred_bimod.ras_hits.PP      8551 # total number of RAS hits
bpred_bimod.ras_rate.PP 0.9999 # RAS prediction rate (i.e., RAS hits/used RAS)
```

B.4 Convert Script

The following is the source code to the script used to convert data from the stored format of the `simple` script to that required by gnuplot.

```
#!/usr/bin/perl -w
use Getopt::Std;

my %options=();
getopts("hso",\%options);

if (defined($options{"h"}) || (scalar(@ARGV) == 0))
{
	die("convert [options] filename\noptions are:\n\t-h - this help message\n
		-s - scale numeric results by the number of keys \n
		-o - offset everything from do_nothing\n");
}

$scale = 0;
if (defined($options{"s"})) { $scale = 1; }

$offset = 0;
if (defined($options{"o"})) { $offset = 1; }

if ($offset) # read in the offset results
{
	$argument = "logs/do_nothing_";
	unless (-e $argument) { die("file '$argument' does not exist\n"); }

	#get the actual filename
	$filename = "do_nothing";

	# parse the log
	open(LOG, $argument);
	@lines = <LOG>;
	chomp @lines;
	close(LOG);

	# skip date
	shift(@lines);

	# skip "run as" and line of dashes
	shift(@lines);
	shift(@lines);

	@simulators = ();
	while(@lines)
	{

		# find out all the sizes and the simulator name
		$simulator_sizes = shift(@lines);
		if ($simulator_sizes eq "Command list: ") { last;} # the end
		$simulator_sizes =~ /(\S+)\s+(.*)/;
		$simulator = $1;
		push @simulators, $simulator;
```

```
                @sizes = split(/\|/, $2);
                foreach $size (@sizes) {$size =~ s/\s//g;} # remove whitespace
                %seen = ();
                @sizes = grep { ! $seen{$_} ++ } @sizes; # uniq it

                # get the parameters
                @local_params = ();
                while($_ = shift(@lines))
                {
                        if ($_ =~ /\-\-/) {last;}

                        push @local_params, $_;
                }
                @{$params{$simulator}} = @local_params;

                # get the results
                @local_keywords = ();
                while($_ = shift(@lines))
                {
                        if ($_ =~ /\-\-/) {last;} # seperator is ---------- etc
                        $_ =~ /(\S+)\s+(.*)/;
                        $keyword = $1;
                        push @local_keywords, $keyword; # save it for later
                        @local_results = split(/\|/, $2);

                        # now put them into the big associative array
                        foreach $n (@sizes)
                        {
                                foreach $param (@local_params)
                                {
                                        $result = shift(@local_results);
                                        if (defined($result))
                                        {
                                                $result =~ s/\s//g; # remove whitespace
                                                if (!($result eq "xxxxxxxxxxxxx"))
                                                {
                                                        $nothing{$simulator}{$param}{$n}
                                                                {$keyword} = $result;
                                                }
                                        }
                                }
                        }
                }
                @{$keywords{$simulator}} = @local_keywords;
        }

}

# start to actually do thing
foreach $argument (@ARGV)
{
        unless (-e $argument) { die("file '$argument' does not exist\n"); }

        #get the actual filename
        @path = split(/\//, $argument);
        $filename = pop(@path);
```

```
# make a directory for the logs
mkdir "./gnuplot_logs/$filename";

# parse the log
open(LOG, $argument);
@lines = <LOG>;
chomp @lines;
close(LOG);

# skip date
shift(@lines);

# skip "run as" and line of dashes
shift(@lines);
shift(@lines);

@simulators = ();
while(@lines)
{

        # find out all the sizes and the simulator name
        $simuℓator_sizes = shift(@lines);
        if ($simuℓator_sizes eq "Command list: ") { last;} # the end
        $simuℓator_sizes =~ /(\S+)\s+(.*)/;
        $simuℓator = $1;
        push @simulators, $simuℓator;

        @sizes = split(/\|/, $2);
        foreach $size (@sizes) {$size =~ s/\s//g;} # remove whitespace
        %seen = ();
        @sizes = grep { ! $seen{$_} ++ } @sizes; # uniq it

        # get the parameters
        @local_params = ();
        while($_ = shift(@lines))
        {
                if ($_ =~ /\-\-/) {last;}

                push @local_params, $_;
        }
        @{$params{$simuℓator}} = @local_params;

        # get the results
        @local_keywords = ();
        while($_ = shift(@lines))
        {
                if ($_ =~ /\-\-/) {last;} # seperator is ————- etc
                $_ =~ /(\S+)\s+(.*)/;
                $keyword = $1;
                push @local_keywords, $keyword; # save it for later
                @local_results = split(/\|/, $2);

                # now put them into the big associative array
                foreach $n (@sizes)
                {
                        foreach $param (@local_params)
```

```
                                {
                                        $result = shift(@local_results);
                                        if (defined($result))
                                        {
                                                $result =~ s/\s//g; # remove whitespace
                                                if (!($result eq "xxxxxxxxxxxxx"))
                                                {
                                                        $results{$simulator}{$param}{$n}
                                                                {$keyword} = $result;
                                                }
                                        }
                                }
                        }
                }
                @{$keywords{$simulator}} = @local_keywords;
        }
        foreach $simulator (@simulators)
        {
#               print "simulator = $simulator\n";
                foreach $param (@{$params{$simulator}})
                {
#                       print "param = $param\n";
                        $comment = "#set_size_in_keys";
                        $output = "";
                        foreach $n (@sizes)
                        {
#                               print "n = $n\n";
                                $output .= "$n ";
                                foreach $keyword (@{$keywords{$simulator}})
                                {
#                                       print "keyword = $keyword\n";
                                        $result = $results{$simulator}{$param}
                                                {$n}{$keyword};
                                        if (defined($result))
                                        {
                                                if ($n == $sizes[0]) # only do once
                                                {
                                                        $comment .= " $keyword";
                                                }
                                                if ($offset && (!($result =~ m/\D/))
                                                        && (!$argument =~ /do_nothing/))
                                                {
                                                        $result -= $nothing{$simulator}{$param}
                                                                        {$n}{$keyword};
                                                }
                                                if ($scale && (!($result =~ m/\D/)))
                                                {
                                                        $output .= ($result/$n)." ";
                                                }
                                                else
                                                {
                                                        $output .= "$result ";
                                                }

                                        }
                                }
                                $output .= "\n";
                        }
                        $param =~ s/\ /_/g;
                        $fullname = "$filename\_$simulator\_$param";
                        open(OUT, "> ./gnuplot_logs/$filename/$fullname");
                        print OUT $comment."\n";
                        print OUT $output;
                        close(OUT);
                }
            }
    }
```

B.5 Sample Gnuplot Script

The following script is used to make plot results. The image made by this file is in Figure 9 on page 82.

```
set key top left Left reverse
set xrange [4096:4194304]
set yrange [0:]
set xtics (8192,16384,32768,65536,131072,262144,524288,1048576,2097152)
set format x "%8.0f"
set logscale axis
set data style linespoints
set term postscript eps
set output '../docs/plots/all_cycles.eps'
set xlabel 'set size in keys'
set ylabel 'Cycles per key'
plot \
'processed_cycles_data' using 1:5 title 'base quicksort',\
'processed_cycles_data' using 1:6 title 'cache quicksort',\
'processed_cycles_data' using 1:7 title 'multi quicksort',\
'processed_cycles_data' using 1:10 title 'base mergesort',\
'processed_cycles_data' using 1:14 title 'double tiled mergesort',\
'processed_cycles_data' using 1:16 title 'double multi-mergesort',\
'processed_cycles_data' using 1:9 title 'base radixsort'
```

B.6 Sort listings

Included below is a code listing of the sorts used in this project. Note that the code is not exact. It is edited for brevity and clarity, and may not even compile. In addition, loops are rolled, edge-cases are removed and variable declarations are moved or removed. The actual code used is contained on the CD accompanying this project.

Base heapsort is listed in Figure 5.1.
Cache Heapsort:

```
#define HEAP_SIZE 8
fix_up(Item heap[ ], int child)
{
        Item v = heap[child];
        while(less(v, heap[int parent = ((child−1)/HEAP_SIZE)]))
        {
                heap[child] = heap[parent];
                heap[parent] = v;
                child = parent;
        }
}

fix_down(Item heap[ ], int parent, int N)
{
        Item v = heap[parent];
        while(int parent*HEAP_SIZE+1 < N)
        {
                int child = parent*HEAP_SIZE + 1;
                int child2 = child;

                if (less(heap[child2 + 0], heap[child])) child = child2 + 0;
                if (less(heap[child2 + 1], heap[child])) child = child2 + 1;
                if (less(heap[child2 + 2], heap[child])) child = child2 + 2;
                if (less(heap[child2 + 3], heap[child])) child = child2 + 3;

#if (HEAP_SIZE == 8)
                if (less(heap[child2 + 4], heap[child])) child = child2 + 4;
                if (less(heap[child2 + 5], heap[child])) child = child2 + 5;
                if (less(heap[child2 + 6], heap[child])) child = child2 + 6;
                if (less(heap[child2 + 7], heap[child])) child = child2 + 7;
#endif
                /* stop when the larger child is les than the parent */
                if (!less(heap[child], v)) break;

                /* move down */
                heap[parent] = heap[child];
                parent = child;
        }
        heap[parent] = v;
}

void cache_heapsort(Item a[ ], int N)
{
        int start_padding = HEAP_SIZE − 1;
        int length = 0;
        int sentinel_count = (HEAP_SIZE) − (N % HEAP_SIZE) + 1;
        if (sentinel_count == HEAP_SIZE) sentinel_count = 0;

        /* the size is N + padding_at_the_start + padding_at_end,
         * rounded up to next HEAPSIZE
         * */
        int size_to_allocate = ((N + start_padding + sentinel_count − 1)
                        & ˜(HEAP_SIZE−1)) + HEAP_SIZE;

        Item* heap_data = memalign(HEAP_SIZE, size_to_allocate * sizeof(Item));
        Item* heap = &heap_data[start_padding];
        /* first build the heap */
        for(i = 0; i < N; i++)
        {
                heap[length] = a[i];
                fix_up(heap, length);
                length++;
        }

        /* add sentinels for fixdown */
        for(int i = 0; i < sentinel_count; i++) heap[N + i] = UINT_MAX;

        int i = 0;
        while(length > 1)
        {
                a[i++] = heap[0];
                heap[0] = heap[length−1];
                heap[length−1] = UINT_MAX;
                fix_down(heap, 0, −−length);
        }

        a[i] = heap[0];
        free(heap_data);
}
```

Base Mergesort:

```
void base_mergesort(Item a[ ], int N)
{
        Item * aux = malloc(N * sizeof(Item)); /* make it twice the size to use the notation */
        unsigned int* source = a; unsigned int* target = aux;

        /* pre-sort */
        for(int i = 0; i <= N; i+=8)
        {
                int j = i+1;
                if (less(a[i+2], a[j])) j = i+2;
                if (less(a[i+3], a[j])) j = i+3;
                compexch(a[i], a[j]);

                j = i+2;
                if (less(a[i+3], a[j])) j = i+3;
                compexch(a[i+1], a[j]);

                compexch(a[i+2], a[i+3]);

                j = i+5;

                if (less(a[j], a[i+6])) j = i+6;
                if (less(a[j], a[i+7])) j = i+7;
                compexch(a[j], a[i+4]);

                j = i+6;
                if (less(a[j], a[i+7])) j = i+7;
                compexch(a[j], a[i+5]);
                compexch(a[i+7], a[i+6]);
        }

        /* now beging merging */
        int track = 0; int next_count = 8;
        int i = 0; int j = next_count − 1;
        int k = 0; int d = 1;

        while(1)
        {
                /* iterate through the left list */
                while(1)
                {
                        if (source[i] >= source[j]) break;
                        target[k] = source[i++];
                        k += d;
                }

                /* iterate through the right list */
                while(1)
                {
                        if (source[i] <= source[j]) break;
                        target[k] = source[j−−];
                        k += d;
                }

                /* check if we're in the middle */
                if(i == j)
                {
                        target[k] = source[i];
                        i = track + next_count;

                        /* check whether we have left the building */
                        if (i >= N)
                        {
                                if (next_count >= N) break;

                                Item* temp_pointer = source;
                                source = target;
                                target = temp_pointer;

                                /* the next iteration will be oevr bigger lists */
                                next_count <<= 1;
                                i = 0;
                                j = next_count − 1;
                                if (j >= N) j = N−1;
                                track = 0;
                                k = 0;
                                d = 1;
                                continue;
                        }

                        j = i + next_count − 1;
                        if (j >= N) j = N−1;

                        track = i;
```

```
            /* setup k for the next one */
            if (d == 1)
            {
                d = −1;
                k = j;
            }
            else
            {
                d = 1;
                k = i;
            }
        }
        else if (source[i] == source[j])
        {
            target[k] = source[i++];
            k += d;
        }
    }

    if (target != a)
        memcpy(a, target, N * sizeof(Item));

    free(aux);
}
```

Tiled Mergesort is very similar to double-tiled mergesort, and so is not included. Merge and presort functions are also not included, as they are very similar to those of base mergesort.

Double Tiled Mergesort:

```
#define LIMIT OUT_OF_PLACE_LIMIT
#define LIMIT_BITS OUT_OF_PLACE_LIMIT_BITS

#define ODD_COUNT 8
#define EVEN_COUNT 4

void
double_tiled_mergesort(Item a[], int N)
{
    /* get the address we need*/
    int minusA = ((1 << BLOCK_BITS) − get_index(a));
    Item *aux_data = memalign(ALIGNMENT, (N + 2*LIMIT) * sizeof(unsigned int));

    Item* aux = (unsigned int*)(((unsigned int)aux_data
                & (~BLOCK_AND_LINE_MASK)) | (minusA << LINE_BITS));

    if (aux < aux_data) /* then the new index is less than the old one */
        aux = (unsigned int*)((unsigned int)aux + (1 << (BLOCK_AND_LINE_BITS)));

    /* the number of standard LIMIT sized passes */
    int level2_count = N / LIMIT;
    /* the number of standard 8k sized passes */
    int level1_count = LIMIT / 2048;
```

```
Item* level2_start = a;
Item* level2_aux_start = aux;

/* odd means it should end up in aux. and the final merge will get it into a */
/* even means it should end up in a, the final merge will do an even number of steps */
if (get_count(N) & 1)
{
        level1_finish = &level2_start;
        level1_other = &level2_aux_start;
        set_presort_count(ODD_COUNT);
        odd = 1;
}
else
{
        level1_finish = &level2_aux_start;
        level1_other = &level2_start;
        set_presort_count(EVEN_COUNT);
        odd = 0;
}

for(i = 0; i < level2_count−1; i+=2) /* sort it level 2 */
{
        unsigned int* level1_start = level2_start;
        unsigned int* level1_aux_start = level2_aux_start;
        for(j = 0; j < level1_count; j+=2) /* merge the level 1 cache first */
        {
                presort(level1_start, 2048);
                merge(level1_start, 2048, presort_count, level1_aux_start);
                level1_start += 2048;
                level1_aux_start += 2048;

                /* now reverse it */
                presort(level1_start, 2048);
                merge_reverse(level1_start, 2048, presort_count, level1_aux_start);
                level1_start += 2048;
                level1_aux_start += 2048;
        }

        /* merge them all into LIMIT sized bits */
        merge(*level1_finish, LIMIT, 2048, *level1_other);

        level2_start += LIMIT;
        level2_aux_start += LIMIT;


        /* now do it in reverse */
        for(j = 0; j < level1_count; j+=2)
        {
                presort(level1_start, 2048);
                merge(level1_start, 2048, presort_count, level1_aux_start);
                level1_start += 2048;
                level1_aux_start += 2048;

                /* now reverse it */
                presort(level1_start, 2048);
                merge_reverse(level1_start, 2048, presort_count, level1_aux_start);
                level1_start += 2048;
                level1_aux_start += 2048;
                /* these end up in aux */
            }

            /* merge them all into LIMIT sized bits */
            merge_reverse(*level1_finish, LIMIT, 2048, *level1_other);

            level2_start += LIMIT;
            level2_aux_start += LIMIT;
        }

        /* now merge everything together */
        if (N > LIMIT)
        {
            if (odd) merge(aux, N, LIMIT, a);
            else merge(a, N, LIMIT, aux);
        }
        free(aux_data);
}
```

Double Multi-Mergesort:

```
void double_multi_mergesort(Item a[], int N)
{
        /* assume that code is merged into cache sized segments at this point.
         * This is now merged in one step, by the following code
         */
        int K = (((N−1) >> (LIMIT_BITS))+1);

        int heap_length = 0;
        /* this changes to an even number (rounding up) */
        indices = memalign(HEAP_SIZE, ((K+1)^1) * sizeof(int));
        heap_data = memalign(HEAP_SIZE, (K+1) * HEAP_SIZE * sizeof(Item));
        heap = heap_data + (HEAP_SIZE − 1);
        memset(heap_data, 0xff, (K+1) * HEAP_SIZE * sizeof(Item));
        last_added_data = memalign(HEAP_SIZE, (K+1) * sizeof(Item));
        last_added_indices = memalign(HEAP_SIZE, (K) * sizeof(Item));

        last_added = &last_added_data[1];
        last_added_data[0] = 0;

        /* we add 8 from each to the Q */
        for(i = 0; i < K; i++) /* we do the last one ourselves */
        {
            /* this may cause k misses in the bimodal predictor.
             * but k is so small thats its not worth the effort */
            if (i & 1) /* this is a reverse list */
            {
                if (i == K−1) indices[i] = N−1; /* last mini-list */
                else indices[i] = ((i+1) << LIMIT_BITS) − 1;

                add_heap_line(&aux[indices[i]], heap, −1, heap_length, HEAP_SIZE);
                indices[i] −= HEAP_SIZE;

                last_added[i] = aux[indices[i]+1];
                last_added_indices[i] = i;

        }
        else /* foward list */
        {
                indices[i] = i << LIMIT_BITS;
                if (i == K−1)
                {
                        int space_to_end = HEAP_SIZE;
                        if (N − indices[i] < HEAP_SIZE ) space_to_end = N − indices[i];

                        /* this makes a sort of sentinel ,which simplifies the logic below */
                        indices[i+1] = N−1;
                        add_heap_line(&aux[indices[i]], heap, 1, heap_length, space_to_end);
                        indices[i] += space_to_end;
                }
                else
                {
                        add_heap_line(&aux[indices[i]], heap, 1, heap_length, HEAP_SIZE);
                        indices[i] += HEAP_SIZE;
                }

                last_added[i] = aux[indices[i]−1];
                last_added_indices[i] = i;
        }
        heap_length += HEAP_SIZE;
}
sort_last_index(last_added, last_added_indices, K);

/* do the merge */
i = 0;
while(i < N)
{
        a[i] = heap[0];
        heap[0] = heap[heap_length−1];
        heap[heap_length−1] = UINT_MAX;
        fix_down(heap, 0, −−heap_length);

        if (a[i] == last_added[0])
        {
                int last_added_index = last_added_indices[0];

                if (last_added_index & 1)
                {
                        /* we add 1 here because both indices are on an
                         * item that hasnt yet been added */
                        add_heap_line(&aux[indices[last_added_index]],
                                        heap, −1, heap_length, HEAP_SIZE);
                        indices[last_added_index] −= HEAP_SIZE;
                        last_added[0] = aux[indices[last_added_index]+1];
                        heap_length += HEAP_SIZE;
                }
                else
                {
                        /* the will have a buddy on the right */
                        add_heap_line(&aux[indices[last_added_index]],
                                        heap, 1, heap_length, HEAP_SIZE);
                        indices[last_added_index] += HEAP_SIZE;
                        last_added[0] = aux[indices[last_added_index]−1];
                        heap_length += HEAP_SIZE;
                }

                sort_last_index(last_added, last_added_indices, K);
            }
            i++;
    }
    free(indices); free(heap_data); free(last_added_data);free(last_added_indices);
}
```

Base Quicksort:

```
#define THRESHHOLD 10

static int* stack = NULL;
static int stack_index = 0;

static void
stackinit(unsigned int N)
{

    int lg2N = 32; /* im hard coding this and i dont care */
    unsigned int msbN = 1 << (lg2N − 1);
    while(msbN > N)
    {
        lg2N−−;
        msbN = msbN >> 1;
    }
    stack = malloc(((lg2N + 2) << 2) * sizeof(int));
}

inline static void push(int valueA, int valueB)
{
    stack[stack_index++] = valueA;
    stack[stack_index++] = valueB;
}

inline static int pop()
{
    stack_index−−;
    return stack[stack_index];
}

inline static void stackclear()
{
    if (stack != NULL) free(stack);
    stack = NULL;
}

inline static int stackempty() { return (stack_index==0); }

static int partition(Item a[], int l, int r)
{
        Item v = a[r];
        int i = l − 1;
        int j = r;

        for(;;)
        {
                while (less(a[++i], v))
                        ;
                while (less(v, a[−−j]))
                        ;

                if (i >= j) break;
                exch(a[i], a[j]);
        }
        exch(a[i], a[r]);

        return i;
}

base_quicksort(unsigned int a[], int N)
{
        stackinit(N);
        int r = N−1;
        int l = 0;

        while(1)
        {
                if (r − l <= THRESHHOLD)
                {
                        if (stackempty()) break;

                        l = pop(); r = pop();
                        continue;
                }

                /* Median of 3 partitioning*/
                m = (l+r)/2;

                exch(a[m], a[r−1]);
                compexch(a[l], a[r−1]);
                compexch(a[l], a[r]);
                compexch(a[r−1], a[r]);

                int i = partition(a,l+1,r−1);

                if (i−l > r−i)
                {
                        push(i−1,l);
                        l = i+1;
                }
                else
                {
                        push(r,i+1);
                        r = i−1;
                }
        }
        stackclear();
        /* the +1 isnt immediately obvious.
         * its because THRESHHOLD is the difference between l and r up above */
        if (THRESHHOLD + 1 > N) insertion_sentinel(a,N);
        else insertion_sentinel(a,THRESHHOLD+1);

        insertion(a, N);
}
```

Cache Quicksort:

```
cache_quicksort(unsigned int a[], int N)
{
        stackinit(N);
        int r = N−1;
        int l = 0;

        while(1)
        {
                if (r − l <= THRESHHOLD)
                {
                        if (l == 0) insertion_sentinel(a, r);

                        insertion(&a[l], r−l);
                        if (stackempty()) break;

                        l = pop();
                        r = pop();
                        continue;
                }

                /* Median of 3 partitioning*/
                int m = (l+r)/2;

                exch(a[m], a[r−1]);
                compexch(a[l], a[r−1]);
                compexch(a[l], a[r]);
                compexch(a[r−1], a[r]);

                int i = partition(a,l+1,r−1);

                if (i−l > r−i)
                {
                        push(i−1,l);
                        l = i+1;
                }
                else
                {
                        push(r,i+1);
                        r = i−1;
                }
        }
        stackclear();
}
```

Multi-Quicksort:

```
#define KEYS_PER_LIST 1018
struct List
{
        unsigned int blocks[KEYS_PER_LIST];
        struct List* previous;
        int count;
};

static struct List **lists = NULL;
static int list_count = 0;
static void* list_memory = NULL;
static struct List* next_available_list = NULL;

static void listinit(int number_of_pivots, int N)
{
        int i;
        long num_lists_required = number_of_pivots + (N / KEYS_PER_LIST);
        lists = malloc(sizeof(struct List*) * number_of_pivots);
        list_memory = malloc(sizeof(struct List) * num_lists_required);
        next_available_list = list_memory;

        for(i = 0; i < number_of_pivots; i++)
        {
                lists[i] = next_available_list++;
                lists[i]->count = 0;
                lists[i]->previous = lists[i];
        }
        list_count = number_of_pivots;
}

static void listclear()
{
        if (lists == NULL) return;

        free(list_memory);
        free(lists);
        lists = NULL;
}

/* these functinos are not symetric. you fill it first.
 * then you pop it. nothing more is required, or supported */
static inline void
add_to_list(unsigned int list_index, unsigned int key)
{
        struct List* list = lists[list_index];

/*      printf("adding key %d to list %d\n", key, list_index); */
        if (list->count == KEYS_PER_LIST) /* if we're full up */
        {
                struct List* old_list = list;
                list = next_available_list++;

                list->previous = old_list;

                list->count = 0;
                lists[list_index] = list;
        }

        list->blocks[list->count] = key;
        list->count++;

}

/* an empty list has count 0. that is all anybody needs to know */
static inline unsigned int
pop_list(unsigned int list_index)
{
        struct List* list = lists[list_index];
        unsigned int result;

        list->count--;
        result = list->blocks[list->count];

        /* if we've run on */
        if (list->count == 0)
        {
                list = list->previous;
                if (list != NULL) lists[list_index] = list;
        }

        return result;
}



void
multi_quicksort(unsigned int a[], int N)
{
        int pivot_count = 0;
        unsigned int* pivots = NULL;
        unsigned int* pivots_memory = NULL;

        stackinit(N);
        if (N > 2 * C)
        {
                /* this number is not accurate.
                 * but it is consistant, and so removes that bug */
                /* the -1 removes a bug where N is an exact number of C_DIV_3s */
                int pivot_count = (N-1) / C_DIV_3;

                /* allocateenough emmory for sentinels at either side*/
                Item* pivots_memory = malloc(sizeof(Item) * (pivot_count + 2));
                Item* pivots = &pivots_memory[1];

                listinit(pivot_count + 1, N);

                /* choose the pivots */
                pivots[-1] = 0;
                for(i = 0; i < pivot_count; i++) pivots[i] = a[(i+1) * C_DIV_3];
                pivots[pivot_count] = UINT_MAX;

                /* sort the pivots - theres already sentinels */
```

```
insertion(pivots, pivot_count−1);

/* fill the lists, one pivot group at a time */
for(i = 0; i < N; i++)
{
        if ((i % (C_DIV_3) == 0) && (i != 0)) continue;

        Item *v = a[i];
        int l = 0;
        int r = pivot_count+1;

        while(1)
        {
                k = (l+r) >> 1;

                if (v > pivots_memory[k+1])
                {
                        l = k+1;
                        continue;
                }

                if (v < pivots_memory[k])
                {
                        r = k;
                        continue;
                }

                add_to_list(k, v);
                break;
        }
}

j = 0;

min = 0;
temp = 0;
v = UINT_MAX;
while(lists[0]−>count > 0)
{
        a[j] = pop_list(0);
        if (a[j] < v) /* find the smallest item */
        {
                min = j;
                v = a[j];
        }
        j++;
}
a[j++] = pivots[0];
if (a[j] < a[min])
        min = j;

exch(a[0], a[min]);

/* the pivot is in its final place */
push(j−2,temp);
temp = j;
```

```
        for(i = 1; i < pivot_count; i++)
        {
                while(lists[i]−>count > 0)
                {
                        a[j++] = pop_list(i);
                }
                a[j++] = pivots[i];

                /* the pivot is in its final place */
                push(j−2,temp);
                temp = j;
        }

        if (temp < N−1)
                push(N−1,temp);

        /* add the ones to the right of the last pivot */
        while(lists[i]−>count > 0)
        {
                a[j++] = pop_list(i);
        }
        free(pivots_memory);
        l = pop();
        r = pop();
    }
    else
    {
        r = N−1;
        l = 0;
    }

    /* as cache quicksort from here */
}
```

Sequential Multi-Quicksort:

```
/* fill the lists, one pivot group at a time */
for(i = 0; i < N; i++)
{
    if ((i % (C_DIV_3) == 0) && (i != 0)) continue;

    v = a[i];
    l = 0;
    r = pivot_count+1;

    for(k = 0; k < pivot_count+1; k++)
    {
        if(v < pivots[k])
        {
            add_to_list(k, v);
            break;
        }
    }
}
```

Radixsort:

```
#include "base_sorts.h"

#define THRESHHOLD 10

#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define Radix (1 << bitsbyte)

#define digit(A, B) (((A) >> (bitsword-((B)+1)*bitsbyte)) & (Radix-1))

unsigned int* radix_aux;

void
radixLSD(unsigned int a[], int l, int r)
{
        /* R is the size of a digit, hence the number of bins */
        int i, j, w, count[Radix+1];

        /* this is from lsb. 3 == lsByte, 0 == msByte */
        for(w = bytesword - 1; w >= 0; w--)
        {
                /* set the counts to 0 */
                for(j = 0; j < Radix; j++) count[j] = 0;
                for(i = l; i <= r; i++) /* count each digit */
                {
                        count[digit(a[i], w) + 1]++;
                }
                /* make the count cumulatative */
                for(j = 1; j < Radix; j++)
                {
                        count[j] += count[j-1];
                }
                /* when we look up the array, we now have its position */
                for(i = l; i <= r; i++)
                {
                        radix_aux[count[digit(a[i], w)]++] = a[i];
                }
                for(i = l; i <= r; i++) /* copy it back */
                {
                        a[i] = radix_aux[i];
                }
        }
}

void
base_radixsort(unsigned int a[], int N)
{
        radix_aux = malloc(N * sizeof(unsigned int));
        radixLSD(a, 0, N-1);
        free(radix_aux);
}
```

Appendix C

Bug List

C.1 Bugs

This section lists some of the bugs that exist in the remaining code. In a project of limited timescale, it is not possible to remove every possible bug from the code. These bugs are listed here in order that it is possible to continue on this work with minimal difficutly.

C.2 Algorithmic

In multi-quicksort, the results include the sentinel check which should have been removed before the results were measured. This adds one branch per `N / THRESHOLD` keys , and the same number of extra instructions. In the same algorithm, the old number of 1018 is the number of keys in a block of a list. The effects of this are not significant. Also, the check for the sentinel was not removed, though the effect of this is minor.

In several of the mergesorts, it is possible that an extra merge step is being taken. This has not been properly analysed, due to time restrictions. It is, however, obvious from the level 2 miss graph of all the merges, that this does not affect base mergesort, but it does affect both double mergesorts, and the tiled mergesort. It is uncertain if multi mergesort is affected. The problem stems from the end condition of the merge. It should end once the size of the next merge is equal the size of the array, rather than greater than. However, the algorithms do not properly merge when this is the case.

Double tiled mergesort and double multi-mergesort were optimized to sort in the level 1 cache first. The number of keys that fit into the cache was fixed at 2048; this number should have been half that, since the sort is out of place. This probably resulted in thrashing, especially since the arrays were aligned. Removing this would improve the sorts.

C.3 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.

Appendix D

SimpleScalar installation

This section details how to install SimpleScalar on a Pentium 4 running RedHat Linux 7. The reason for its inclusion is that the installation was difficult, and solutions to problems encountered were not easily available on the SimpleScalar website, nor were emails answered by the authors.

The primary guide this is based on is [**?**]. However, that guide is more general than this.

D.1 Download

Go to *http://www.simplescalar.com/tools*. The simulations in this project used the standard packages provided on this page. Links to other packages here were not used. Download simplesim-3v0d.tgz, simplesim-3v0d.tgz and simplesim-3v0d.tgz. Unzip the tools. The following directories should be present: binutils-2.5.2, f2c-1994.09.27, gcc-2.6.3, glibc-1.09, simplesim-3.0, ssbig-na-sstrix and sslittle-na-sstrix.

D.2 Compile

Run the following commands:

```
cd binutils-2.5.2
./configure --host=i386-redhat-linux --target=sslittle-na-sstrix
--with-gnu-as --with-gnu-ld --prefix=/home/ciaran/simple
```

```
make
make install
cd ../simplesim-3.0
make
cd ../gcc-2.6.3
./configure --host=i386-redhat-linux --target=sslittle-na-sstrix
--with-gnu-as --with-gnu-ld --prefix=/home/ciaran/simple
make LANGUAGES=c
```

The following error may occur at this point:

```
cccp.c:194: conflicting types for ‘sys_errlist’
/usr/include/stdio.h:554: previous declaration of ‘sys_errlist’
```

Comment out line 194 of `cccp.c`.

This error then occurs:

```
gcc -c -DCROSS_COMPILE -DIN_GCC -DPOSIX
-g -I. -I. -I./config sdbout.c
sdbout.c:57:18: syms.h: No such file or directory
```

This can be resolved[1] by changing line 57 of `sdbout.c` from `#include <syms.h>` to `#include "gsyms.h"`.

```
bc-typecd.def: In function ‘bc_emit_instruction’:
bc-typecd.def:17: ‘SFtype’ is promoted to ‘double’
when passed through ‘...’
bc-typecd.def:17: (so you should pass ‘double’ not
‘SFtype’ to ‘va_arg’)
```

In line 17 of `bc-typecd.def`, change `SFtype` to `DFtype`.

```
gcc.c:172: conflicting types for ‘sys_errlist’
/usr/include/stdio.h:554: previous declaration of ‘sys_errlist’
```

Comment out line 172 of `gcc.c`

The tools should now be compiled. For speed, the architecture type could be changed to `i686` and the `-O3` flag could be set.

[1]Credit for this is due to http://moss.csc.ncsu.edu/ mueller/rt/rt02/g1/sim2install.txt