UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN
INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA

PSORT: AUTOMATED ESTIMATION OF HARDWARE PARAMETERS

Advisor: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
Co-Advisor: Dott. Marco Bressan

Candidate: Alberto Bedin

ANNO ACCADEMICO 2010 - 2011

ABSTRACT

This thesis describes the design and implementation of an automated hardware-detection environment for *psort*, a fast library for stable sorting of large datasets on external memory. Our goal was to create a tool that provides a complete set of estimated hardware parameters which will be used to auto-tune *psort* both at compiling and at run time. The entire detection system has been designed to be scalable and modular in order to simplify the addition of new tests, remaining as transparent as possible to the end user. Experiments prove that our code is high reliable and that there is a strict connection between hardware parameters and software performance, suggesting that *psort* should include our system among its tools.

SOMMARIO

Questa tesi descrive il design e l'implementazione di un apparato automatico in grado di rilevare l'hardware per *psort*, una libreria ad alte prestazioni per l'ordinamento stabile di grandi moli di dati su memoria esterna. Il nostro obiettivo è stato quello di creare uno strumento che fornisca un insieme completo di parametri hardware stimati che saranno utilizzati per ottimizzare automaticamente *psort*, sia al momento della compilazione, che in quello dell'esecuzione. L'intero sistema di rilevazione è stato creato per essere scalabile e modulare in modo da semplificare l'aggiunta di nuovi test, pur rimanendo il più trasparente possibile per l'utente finale. Gli esperimenti provano che il nostro codice è affidabile e che c'è una stretta connessione tra parametri hardware e prestazione del sofware, suggerendo che *psort* dovrebbe includere il nostro sistema tra i suoi strumenti.

CONTENTS

1 INTRODUCTION

This thesis describes the design and implementation of an automated system for the detection of hardware parameters. Although our system is general purpose, it has been designed for *psort*, a fast library for stable sorting of large datasets on external memory, that is highly tunable according to, amongst other things, the machine hardware. This chapter provides a brief summary of the external sorting problem (Section 1.1) and an overview of the structure of *psort* (Section 1.2) as an introduction to the tuning task.

psort *and hardware detection*

1.1 EXTERNAL SORTING

Sorting is one of the most classical computer science problems, that was as important in the last century as it is today. Although there exists a plethora of sorting algorithms which are optimal in theory (such as those matching the well-known $n \log n$[1] lower bound for comparison-based algorithms), a naive implementation hardly squeezes out more than half of a machine's computational power. Sorting algorithms may be divided in two classes according to the type of computer memories which they use. It is common to refer to the memories of a computer as a hierarchical structure [7] [9] where the levels are progressively faster, smaller, and more expensive. The fastest level is represented by the CPU registers, followed by the cache, the internal memory, and then the external memory, that is the slowest one. A well designed software should exploit all needed memory levels in accordance to the criteria of spatial and temporal locality [11]. Traditional sorting algorithms does not need to use external memory, that is instead the peculiarity of external sorting algorithms.

Memory hierarchic and sorting algorithms

Nowadays external sorting software finds its application in a wide range of sectors, from high-end industrial databases [15] to scientific research area, e.g. human genome classification. It basically allows to sort an amount of data that cannot fit the size of the internal memory of a machine. The classical example of external sort algorithm uses a *multi-way* merge sort [12] and can briefly be summarized in two steps. The first step divides the input, stored in the external memory, in blocks which can fit the size of the internal memory. Each block is loaded in this memory, sorted using a classical sorting algorithm, and then written back to the external memory. The second step merges the blocks reading the data from, and writing the output to the external

Classical external sort

1 It is the well-know lower-bound $\Omega(n \log n)$ in the worst case, where n is the number of elements to be sorted, as proved in [11].

memory. Finding the more efficient way to access and to sort the data in both steps is not a trivial problem and may cause huge performance differences.

Algorithms state of the art

Due to its importance, external sorting is a critical aspect evaluated by a lot of benchmarks such as the *Sort Benchmark* [6], a competition that annually awards the fastest state of the art sorting software in different categories.

External memory, that is commonly represented by hard disks, is hundred of times [11] slower than internal memory. To achieve the best result with all input typologies, it is not only sufficient to minimize external accesses: every memory level should be optimized and this is the primary goal of *psort*.

1.2 PSORT

psort *overview*

psort is a C++ software and library that allows to quickly sort large amount of data stored in the external memory. It can sort data according to an arbitrary comparison operator; but the library comes, by default, with several highly-optimized versions of the most common comparators – notably lexicographical and numerical. According to the *Sort Benchmark, psort* is the fastest desktop-based external sorting algorithm from 2008 to 2011 in the *PennySort* category[2]. It implements a high optimized version of the classical external sort algorithm described in Section 1.1. *psort* accepts as input a sequence of records stored in a file. Each record is composed by a fixed number of bytes divided in two group: the key bytes and the payload bytes. The key bytes (which should not necessarily be at the beginning of the record) represent the comparison portion of the record. The output file is stored as a unit on an external memory device and contains the sorted records according to their keys.

Tuning: the starting point

In *psort* the two steps described in Section 1.1 are respectively called *stage one* and *stage two*. Every stage has a huge number of configurable parameters, which affect its performance. Their are mainly influenced by the hardware configuration of the machine. Setting up manually these parameters would be a very difficult operation for a user and a time-waste task even for a capable developer. The developer could not know the particular architecture of the machine on which he is working or its software configuration. Therefore, to solve this problem, an auto-tuning structure looks like a natural solution. The first step of the auto-tuning is the automated estimation of hardware parameters. To deeper understand which tuning operations and hardware parameters are the most relevant, it is useful analyzing how stage one and stage two work (focusing on the first one, that is the more complex)

2 *PennySort* benchmarks the "*amount of data that can be sorted for a penny's worth of system time*". The original definition can be found in [13].

and then run some preliminary tests on the not-tuned version of *psort*. This last aspect is covered by Chapter 2.

1.2.1 Stage one

Initialization and I/O buffers

The first operation performed by stage one is the size estimation of each block that will be read from the disk and load in the main memory. This block is named in *psort* as a *run*. The size of a run is roughly the size of the available internal memory decreased by the amount of space reserved for the input/output buffers. This space is defined by the parameter *–s1-io-space*. In order to maintain the external memory as busy as possible, *psort* uses multiple input/output buffers and performs read/write operations with direct asynchronous I/O[3]. In this way data are read and written from and to the device while *psort* is still performing other CPU operations. A set of fine tune parameters allows to specify the number and size of both read and write buffers. These values should be carefully evaluated according to the bandwidth and access times of internal and external memories.

Key detach

If the length of a key is sufficient shorter than the length of the record, the key is detached from its payload and a pointer to the payload is attached to the key. The pair formed by the key and the pointer is called *extended key*. This division helps to works with shorter elements and reduces the number of moved bytes. It is also often possible and convenient in practical situations.

Microruns sorting

As shown on Figure 1, the run is then divided in *microruns* which are sorted exploiting the speed of the L2 cache in accordance with the spatial locality and the hierarchical model [7] described in 1.1. Known the size of the L2 cache, that is a hardware estimation problem, a microrun is composed by a number of records or by a number of extended keys that fits the size of this cache. The parameter which specifies the number of records or the number of extended keys for each microrun is *–s1-records-per-block*. The algorithm that sorts a microrun is a *quasi in place* merge sort that uses *1.25 times* the size of the its input although one of its variants that uses a *quasi in place wave* sorter has proved to be faster in particular situations. Actually we are not able to say *a priori* which situations are favorable to the second approach. For both implementation the base case of the algorithm is performed by a counting sort that works with a number of records (or extended keys) specified at compiling time by the parameter *chunk_size*. It is clear that the best choice of the sorting algorithm and of the chunk_size is hardware-related.

kmerger *sorting*

Continuing to follow Figure 1, the sorted microruns are then merged together in a single sorted run that is written back to the disk. The merging operation is performed by an object called `kmerger`, which is an *ad hoc* implementation of an algorithm similar to both a heap

3 For further information on Direct I/O see 2.1.1 and 3.4.1.

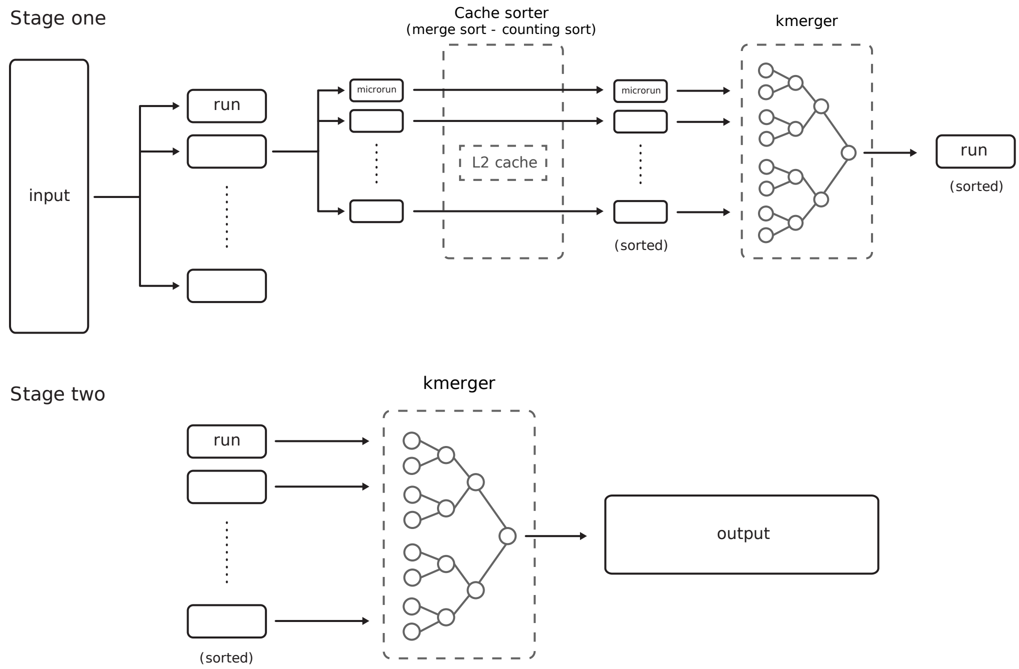


Figure 1: *psort* stage one and stage two overview. The first stage is the more complex and tries to exploit all memory levels.

merger (but stable) and a k-way merger. Until each microrun is empty, the record (or the extended key) with the smaller key among all the microruns is extracted and moved into a heap. The keys in the heap compete to reach the root from which they are moved to the external device. If it is necessary, the payload is reattached to the key using the pointer address. The disk writes can be almost entirely overlapped with the merge pass described above.

At the end of this process, all sorted runs are stored in the external memory.

1.2.2 Stage two

kmerger *on stage two*

Stage two starts if and only if there is more than one run. `kmerger` manages the runs with a few differences from the algorithm applied to the microruns of the stage one. This time the input runs are in the external memory and therefore they are partially loaded into the main memory using a different buffer for each run. These buffers are sized as shown on Figure 2. Since the keys are usually uniformly distributed among the runs, filling all buffers with the same number of records would cause all buffers to be empty approximately at the same time. To avoid this problem, each buffer is filled with a different number of records according to a parameter called *–geometric-factor*. The buffers are refilled when their number of records becomes smaller than a

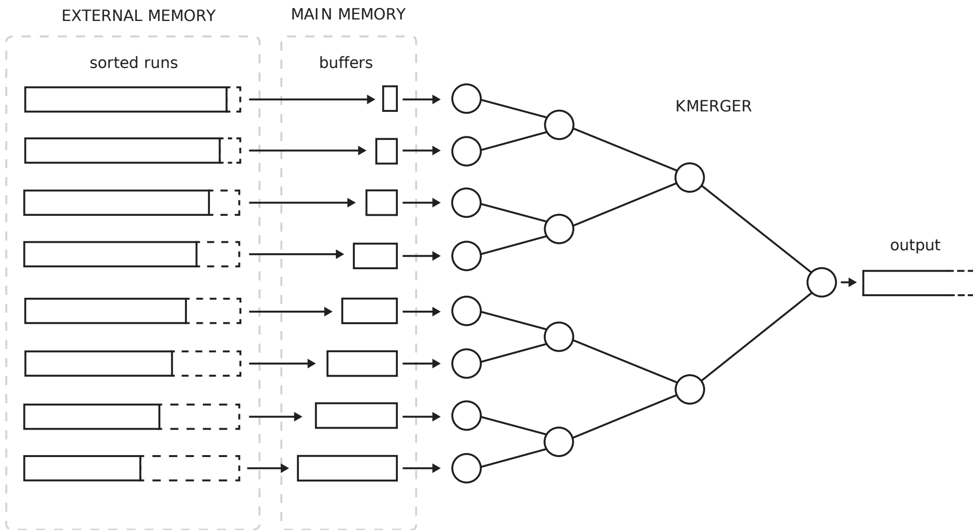


Figure 2: `kmerger` reads data from the external memory and place them into dynamic-size buffers according to the geometric factor value.

threshold specified by the parameter *–s2-read-threshold*. The bottleneck of this stage is the disks bandwidth, since the CPU and the main memory have a low load factor compared with the number of external device accesses.

Multi-pass stage two

Sometimes it is more convenient to merge first, in one pass, a subsets of the total amount of runs and then to merge these subsets together in a second pass. Usually one pass suffices to achieve the best performance but sometimes, especially sorting very large amount of data with a small internal memory, two or more passes are required in order to reduce the number of runs. For further details about the number of passes see [8].

2 HARDWARE AND CRITICAL PARAMETERS

Possible improvements

There are a lot of ways to achieve better performance in *psort*. Its base version can be improved by adding *multi-core* support, *multi-disk* support, and by tuning its parameters according to the machine hardware and to the input file. First of all we need to find which parameters affect the performance in a significant way and how they are related to the hardware. Section 2.1 shows the approach and the tools designed to discover these parameters. The following Section 2.2 is dedicated to the actual tuning structure of *psort* and to the hardware detection problem. Since stage one is the most CPU and RAM intensive, we focus our attention on it.[1]

2.1 FINDING CRITICAL PARAMETERS

Critical parameter discovering process

To discover critical parameters we need to run *psort* a large amount of times on the same input, changing execution and hardware parameters. Then we collect and compare the bandwidth of each execution and find which parameters give the best improvement according to a specific hardware configuration. For example Figure 3 on page 8 shows how the bandwidth changes with different choices of the parameter *–io-space* that is the total amount of main memory reserved for input/output buffers. The bandwidth is calculated as the ratio between the total input file size and the total execution time of stage one or stage two. The input file size may be measured in number of records or in bytes. It is important to achieve the highest possible bandwidth in both stages: we cannot choose optimal parameters which give high performance in stage one and low performance in stage two. This problem is discussed with the *–io-space* example in 2.1.4.

Data collection methods

In order to collect a large amount of data, we need to set up a complete, automatic, and efficient test system. Since each test could be run on many different machines, it should be able to be executed by remote via SSH. A versatile *test-script* in *bash* is prepared to achieve this result. It also saves execution values in a log file which is parsed by an *ad hoc* script called *psortInfoParser*. The parser generates another output file ready to be imported into plotting and high-level computational environments such as *Matlab*® [3] and GNU *Octave* [4]. Finally a script written in Matlab-language allows to quickly plot the data and

1 As shown in our tests *multi-disk* support, which is the primary alternative to RAID configuration, grants the best bandwidth improvement in stage two. According to this result, disks are the main bottleneck of stage two.

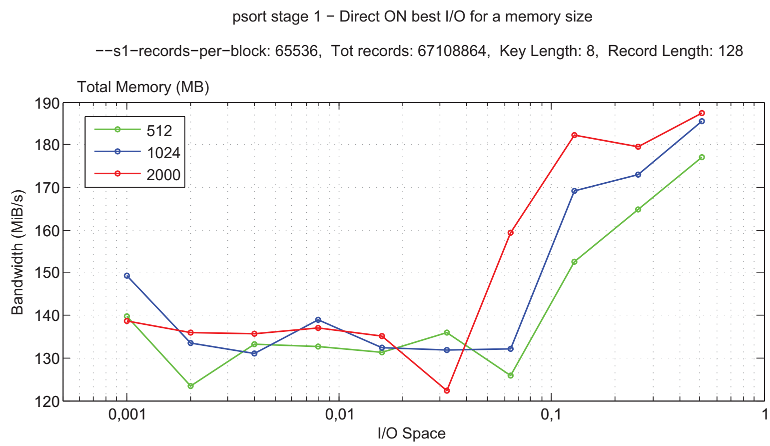


Figure 3: Different *psort* performance according to different values of the parameter *–io-space*. Choosing a bad value can cause a loss of bandwidth of more than 20% in stage one.

to compare different execution on the same figure. The entire process designed to discover critical parameters can be summarized as follow:

1. Execute a group of tests varying one or more parameters.
2. Collect, parse and plot data.
3. Analyze results and find critical parameters to be tuned.

The idea is to provide a collection of tools re-usable in the future by anyone who will need to test *psort*.[2] All scripts are actually stored in the *tuning-test* directory of *psort*. For future uses, we briefly describe how they work.

2.1.1 Bash script

Bash script overview

psortTestBash.sh can be used from a shell to start a test. There are a lot of parameters which can be set: we can choose to run only stage one, two or both, to enable *psort* Direct I/O support[3], to set the path of the input file to be sorted, and to set additional *psort* options. Plus we can check the sorted file to verify the presence of sorting errors and finally specify a loop of tests to be executed.

Every loop iteration changes the value of one parameter according to a chosen rule: the script requires to specify the start value, the end value, the incremental step, and the incremental method which can

2 More in general terms both *bash script* and *Matlab® script* can be used to test and benchmark not only *psort* but also *every* command-line software.

3 Direct I/O support allows software to read and write from and to the disk without using the O.S. cache. It increases input/output performance but required aligned operations. According to *psort* specifics to use Direct I/O "*record_length* · *block_size* must be a multiple of the boundary" (usually 512 KiB). For more information about Direct I/O see 3.4.1.

be sum or multiplication. The first means that the incremental step will be added to the start value until the reaching of the end value, while the latter means that the start value will be multiplied by the incremental step until the reaching of the final value. The syntax to properly configure the script can be found by adding the parameter *–help*.

Improvements to the bash script

First tests with Direct I/O *off* give strange results: the first execution is always slower than the other ones. The problem is that the O.S. stores a copy of the data into a fast cache used at the next execution on the same input. Since there is a considerable execution time gap, the script has to empty the cache after each execution. We find this code working on Linux with root access:

```
sync
echo 3 > /proc/sys/vm/drop_caches
```

A test can frequently last more than twelve hours and a crash during a single execution shouldn't stop the entire test. This is a fundamental aspect achieved by configuring the script to automatically check from time to time the status of the current test and to eventually start the next execution. Some tests require to change more than one parameter. This can be obtain by adding an additional *bash* script that contains loops which start the main *psortTestBash.sh* file with the desired parameters.

Log files generated by the bash script

The output is divided in two files:

- A log file that describes the operations performed by the *bash* script.
- A log file formatted according to *psort verbose-level one*[4] followed by used parameters and times. This file also contains executions which return an error, marked by a special symbol.

Times are calculated using built-in *psort* functions based on standard C library. Log files are formatted according to the specifics of the parser described in the next paragraph.

2.1.2 Parser

Parser overview

The parser is a really simple script that takes as input a log file from a bash test of *psort* and parses it into two formats: one is human readable while the other one is a common *csv* file that uses a vertical

4 *Verbose-level one* consists in a few but critical informations about the *psort* execution. All main parameter values are shown here.

slash (divider line "|") as field separator. This file can be imported very easily into *Matlab*® by an automatic script. The parser is also able to detect execution errors stored in the log file and to save them in a separated output file. Finally it can be used to set the proper decimal separator for double values (which can be a dot or a comma) according to the language of the importing software.

2.1.3 Matlab script

Matlab script overview

The plot script *Multiplotscript.m* is entirely written to test *psort* but is still a general purpose script. It is a complete tool to create complex figures in a short time. It reads as input a text file with a field separator value; the first row of the file may be the label. The tool is capable to plot only specific columns and eventually calculates new columns from the existing ones. This option is useful to obtain derived values, such as the bandwidth, starting from existing ones like the input size and the execution time.

All these (and much more) settings can be turned on by editing an existing set of variables and arrays at the beginning of the script. In order to obtain a script that can be used in the future, we add a complete set of comments which guide during the configuration of the script. We discover that is useful to have all the following functions ready to be used in the script:

1. Automatic import data from the input file.
2. Sort and remove columns.
3. Calculate additional columns as the sum, product, and ratio of existing ones.
4. Multi-plot different values on the same figure in different colors.
5. Create an additional plot that shows best values extracted from the multi-plot (e.g. see Figure 5 on page 12).
6. Automatic add titles, labels, legend, and grid.
7. Save the output image in different formats with an estimated coherent file name.
8. Export a matrix that contains only the filtered data.

Results extracted from the plots

Figure 4 on page 11 is created using this script and allows to quickly visualize the content of the test. There are dozens of plots like this. They show that there is a strict connection between the best value of a parameter and the hardware of the machine. The next paragraph analyzes the presented graphics to explain, with two examples, this relation.

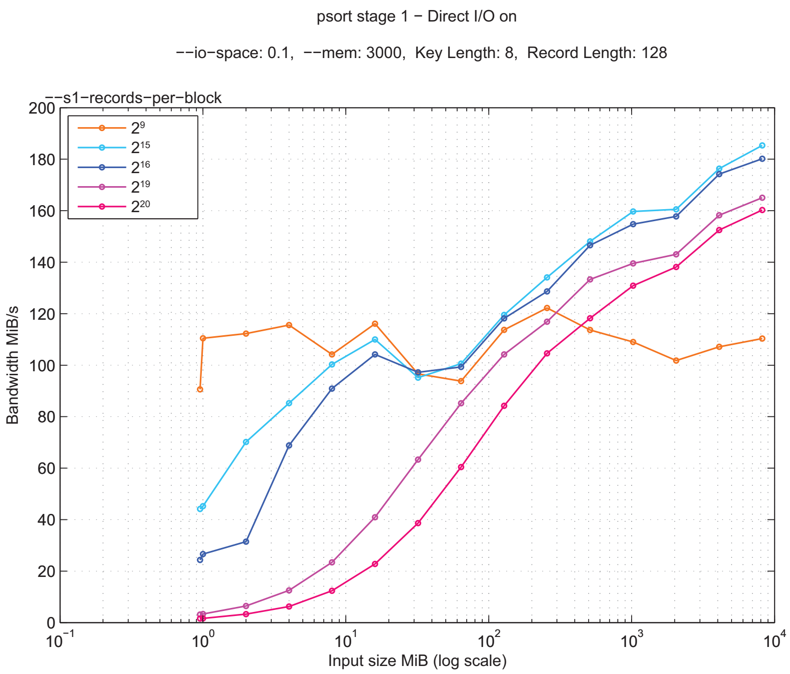


Figure 4: Comparison between different values of the parameter *–s1-records-per-block* on the same input. Choosing a bad value can cause a loss of bandwidth of more than 40% in stage one with a significant input file size.

2.1.4 Critical parameters analysis

There is a high amount of results which can be extracted from a few tests. Some of these are obvious while others are very interesting.

I/O space test

Figure 3 on page 8 shows that the larger the I/O space is, the faster stage one is. However starting from *1 GiB* of input file the difference of bandwidth between a large value of I/O space (such as 0.256 of the total main memory) and a relative small value (0.1 of the total main memory) is trivial. Now consider that using less main memory for the I/O space means to increment the total memory available for a run. In this way we can reduce the total number of runs to be merged in stage two granting a large amount of time. So we can conclude that the best choice for the I/O space is around 0.1 of the total main memory, for input size above *1 GiB*. The important thing to note is that this result is valid on the tested machine only (it has *3000 MiB* of memory dedicated for *psort* at the operating frequency of *800 MHz*) and not necessarily on every other hardware configuration. The best value of I/O space should not only consider the main memory specifics but also the disks specifics and the CPU specifics. This because CPU sorts the data read from the main memory and placed into the buffers which are directly affected by *–io-space*.

Stage one records-per-block

Analyzing the other plots we see that *–s1-records-per-block* is another

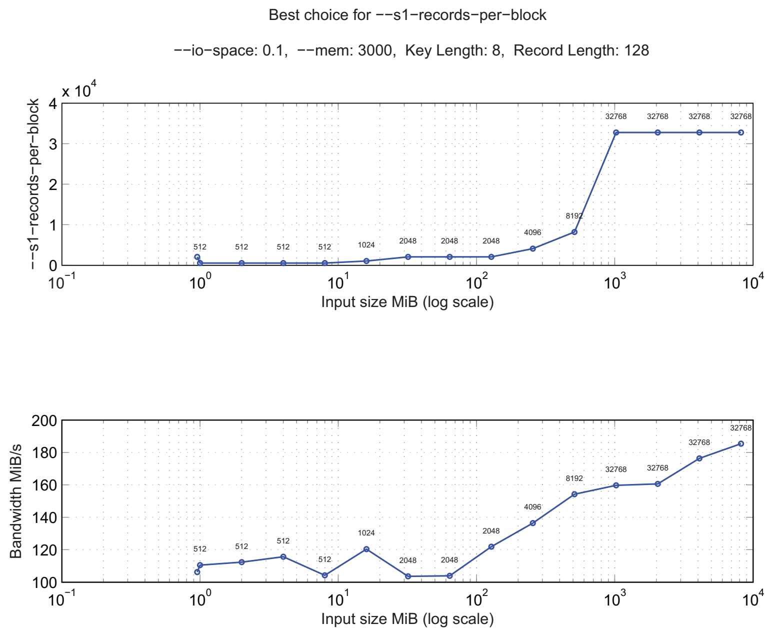


Figure 5: Best choice of *–records-per-block*.

critical parameter of stage one. Figure 4 on page 11 shows why this parameter must be considered in any further tuning operation. Starting from about *1 GiB* of input size, there is a particular value of it that achieves the best performance. Recalling that this parameter defines how many extended keys[5] must be sorted at once, it follows that their size must be equal to the size of the L2 cache of the machine. In fact the plotted test was performed on a machine with *1 MiB* of L2 cache with an extended key length of 16 byte (8 byte for the record key and 8 bytes for the payload address):

$$records\_per\_block \cdot extended\_key\_length = 2^{16} \cdot 16 = 1\ MiB$$

We can further improve performance by choosing as *–s1-records-per-block* a number of extended keys which size is a bit smaller of the L2 cache. In fact, Figure 4 on page 11 shows that the value 2^{15} gives a bit higher bandwidth. This could be due to the fact that in the cache there aren't only the extended keys but also some other (and maybe few) important values for the CPU current process: a miss is the cache will cause a slowdown which could be avoided by choosing a data set that can safely fit the cache size. This point will be discussed on Chapter 3.

Figure 5 on page 12 shows that the found value is a good value for all input above *1 GiB*. Why does this value does not achieve the

5 This parameter may also define the number of records in a microrun, if *psort* is set up to do not separate payloads from the keys. This also explains the origin of its name.

best performance with small input? We can suppose that it is because there is no reason to allocate a large amount of memory for in cache sorting, while there is no a large amount of data to sort. Whatever it is, the connection between the best value of this parameter and the L2 cache size is evident and confirmed by other tests. Therefore we need to correctly estimate the size of this cache and, more in general, all hardware parameters which can affect software performance.

2.2 TUNING AND HARDWARE DETECTION

Auto tuning

Usually we refer to auto tuning in a software as *"the capacity of optimizing internal running parameters in order to maximize or minimize the fulfillment of an objective function; typically the maximization of efficiency"[5].* However this is not its only meaning. With auto tuning we do not only optimize its internal parameters but also the source code before the compilation process. To achieve this result we preliminary need to estimate hardware parameters.

psort *tuning structure*

As summary, our intent is to modify *psort* in order to obtain a software that is able to auto detect hardware parameters, auto test its optimal source code for the current machine, and auto tune itself during the execution. At the end of this process it will be insert into an auto-configuring package. This section describes the actual design of the *psort* tuning structure focusing on the first step: the automatic estimation of hardware parameters.

2.2.1 *psort* tuning structure

We create a package that contains three directories which allow to execute:

1. Hardware detection.
2. Code tuning.
3. *psort* (runtime tuning).

There is also a *bash* file named *installTuned.sh* that starts the entire software auto installation. Default values are set in order to allow a user to start the installation process just by typing `bash installTuned.sh` in a shell. This installing script can also be used for the execution of isolated preliminary tests. The complete list of parameters and functions of this script can be read by adding the parameter *–help* at launch time. In order to execute code tuning the script requires the binary executable file of *CodeWorker*[6] [2] placed in the code-tuning directory. Additionally, to properly compile *psort*, it requires the installation of

6 This file is more than *30 MiB* in size and may not be included in every *psort* auto-tuning package. It this case, it must be downloaded and compiled apart.

CMake[7] [1]. Automatic estimation of hardware parameters can be executed without additional packages. More options can be configured by editing the *config* file in the hardware detection directory and by editing the *makefil*e of *psort*, eventually using *CMake-gui*.

Automatic estimation of hardware parameters

Once started, the installation process runs by default all hardware tests. These determinate a list of hardware-related parameters that *psort* and code tuning use (or will use) to boost the software. Some tests are very general purpose, e.g. the estimation of disks or main memory bandwidth, while other ones are very specific. When possible they try to obtain solid values working with the O.S. available functions and files, but more often they need to intensively test the hardware component, extracting the desired values or estimating which coding approach is more efficient. Tests are repeated more then once in order to minimize noise effects and therefore they may require a lot of time, according to the desired tuning level and hardware speed. During the installation process the user or the developer can see the current running test and the number of required iterations. There is also the possibility to set up a *custom* tuning level in which every test parameter can be configured. These parameters are all stored in *preprocessor* values so they can be modify by editing headers files in the CPU, disk, and memory subdirectories. It takes a few seconds to compile the tests files so there are no performance problems. Every test saves its output on a log file formatted as *csv*. Some of these values are immediately used by the code tuner.

Code tuning

psort code tuning is extensively discussed in [10]. Briefly it tries to estimate the source code that, once compiled, will grant the best performance on the machine hardware[8]. To do this, it automatically generates different versions of the same critical functions, compiles, and executes them. The function implementation that achieves the best bandwidth is chosen for *psort*. The actual code tuning covers *cache sorting* of stage one, tested with different *loop unroll factors.*

There is also another optimization: it compiles the best key comparison method according to the results of hardware detection described above. In particular the choice is between logical and bitwise comparisons. On different hardware configurations, one implementation can be better than the other one, proving that there is a strict connection between hardware detection and code tuning. Tests confirm that code tuning provides performance improvements in *psort*. This is, in fact, a well known speed-up approach in high-performance software[9].

Runtime tuning

Once *psort* is compiled, runtime tuning tries to adjust its parameters according to the hardware discovered during the automatic hardware detection and also according to input file specifics. Actually runtime

7 There is also the possibility to compile *psort* without using *CMake* by replacing the *Makefile* with the old file *Makefile.old*.

8 This is the so called *compiler based auto tuning*. There are other types of code tuning such as *analytical models*, *global empirical research* and *local research*.

9 For a focus on this topic see *ATLAS*, *FFTW*, *PhiPAC*

tuning fixes the value of the two critical parameters of the stage one *–s1-records-per-block* and *–s1-io-space*. The number of records per block is chosen starting from the size of L2 Cache as described in 2.1.4. Since the optimal value for the I/O space depends on the input (see Figure 3 on page 8), it is set at runtime according to the input size. Tests show also that this optimal value changes with the state of the Direct I/O flag.

There are a lot of other parameters which can be tuned at run time, e.g. *–geometric-factor* looks like a critical one on stage two. The more information we acquire about *psort* working and hardware configuration, the easier runtime tuning will be.

2.2.2 Existing hardware-detection software

Existing hardware-detection programs

It looks difficult to us to find a complete open source software for Unix that is able to estimate all needed hardware parameters. *CPUID*[10] software could be a good starting point but unfortunately it only works on *Windows*® based systems. The diffuse tool `dmidecode`, which is already packaged in several Linux and BSD distributions, is only able to detect informations from the *BIOS* so it does not look very useful for us. It only shows cache informations but needs root permissions.

There are however some tools which could help to find particular hardware parameters such as disk bandwidth and cache size. For the first one we could use the free software `dd` that is able to easily estimate the disk bandwidth (sequential read and write) with both Direct I/O on and off. For the second one (caches size estimation) there are a lot of small tools which simply allocate an array and calculate access times. *JCache*[11] works in this way and also provides a small benchmark utility written in Java.

An ad hoc *implementation for* psort

We conclude that *psort* requires more specific tests, implemented *ad hoc* to discover how the hardware works with a really particular instance of a problem. Since *psort* has some small fragments of code which will be executed a huge amount of times, we definitively need to know which type of code implementation is faster on a particular CPU. Plus, our tests are implemented considering the actual *psort* source code and so they try to be as close as possible to it. In fact a lot of available tools are written in assembly with SSE1, SSE2, and SSE3 instructions set and so does not reflect a C++ compiler generated code.

Finally our tests are all written with the same style in order to be easily configured and modified. Next chapter analyzes each one of them discussing the design and implementation.

10 `http://www.cpuid.com`
11 `http://www.dei.unipd.it/ bertasi/jcache.html`

3 EXPERIMENTAL SETUP

Hardware detection overview

This chapter describes the design and the implementation of every single hardware-related test. It also provides an overview of the aim of each test.
In *psort*, hardware detection tests are divided in four main categories:

- CPU tests
- Cache tests
- Main memory tests
- Disks tests

We will occasionally refer to cache tests as a part of CPU tests or memory tests. This is because caches are placed in the CPU and they operate in strict relationship with the main memory.

Each test category is composed by a C++ file and two headers files. One header file contains the prototypes of functions used in the C++ file. This is the starting point to understand how the code works because every prototype is commented using a documentation style. The other header file contains configurable preprocessor values. They define the number of iterations and the size of each test. These values are divided into *three* tuning levels: normal, extreme, and custom. Finally the C++ file contains the implementation of each function and the `main()` function. The `main()` routine controls the output streams and the calls to the tests. Some common functions used by all tests are stored in a global file placed in the *misc* directory.

There are significant differences between each test category and so each one requires a deeper analysis.

3.1 CPU TESTS

CPU tests overview

There is a huge amount of CPU models and they can be implemented using a wide range of approaches. Every year several new models are released, therefore there is no way to know which code implementation achieves a better performance on a specific CPU without directly test it. In particular we suspect that there are three CPU aspects which affect *psort* performance. They are:

1. Pointers notation.
2. If-else against boolean statements.
3. Logical against bitwise evaluations.

One or more of these three aspects may be completely irrelevant on some hardware but they may be critical on other ones. For each of them we have to choose between two solutions. Both solutions are tested with the same input a sufficient amount of times in order to avoid noise effects. The implementation that gives the lowest execution time (or equivalently the highest bandwidth) is chosen for *psort*.

3.1.1 Pointers notation

Subscript notation and offset notation

This test tries to estimate which is the fastest way to increment a pointer in C++ according to the current hardware. It allocates an array of *elements_num* elements, starts the timer, and cycles through the array incrementing each element by one unit. At the end the timer is stopped. This entire operation is repeated more then once and the average time is considered as the final one. To give an idea, on normal tuning level *elements_num* is actually set to 2^{29} elements of type *int* and the test is repeated 3 times. The two approaches differs on how they increment each element.

The first approach uses the common *subscript notation* that is:

```
element_type *array = (element_type *) calloc( elements_
    num, sizeof(element_type) );
for (unsigned long i = 0; i < elements_num; i++)
   array[i]++;
```

While the second approach uses the *offset notation*:

```
for (unsigned long i = 0; i < elements_num; i++)
   (*(array++))++;
```

3.1.2 If-else against boolean statements

If-else and boolean statements

This test tries to estimate which is the fastest way to compare two `uint64_t`[1] and choose a branch according to the result of the comparison on the current hardware. It starts from two fixed large values (say a and b), compares them and if a is smaller than b, it increments a counter variable by one unit using one of the two approaches. After this step, a is incremented by a constant value (actually 10^4) and b

1 This is the current universal data type used by *psort* to manage almost all values. It is a 64 bit type that handles unsigned integers.

is calculated as `a xor b`. These two new values are used in the next iteration. On normal tuning level the test is repeated 3 times with both approaches and each test runs 10^9 comparisons. The test returns the average execution time.

The first approach uses the *if-else* statement to increment the counter:

```
for (unsigned long i = 0; i < single_test_length; i++) {
   a += 10000;
   b = a ^ b;
   if ( a < b )
      counter++;
}
```

While the second approach uses a *boolean* statement:

```
for (unsigned long i = 0; i < single_test_length; i++) {
   a += 10000;
   b = a ^ b;
   counter += ( a < b );
}
```

3.1.3 Logical against bitwise comparisons

Logical and bitwise evaluation

This test tries to estimate which is the fastest way to compare two keys *a* and *b*. A key is a sequence of bytes. The keys can be:

- Total equal keys: every bit of *key a* equals to the same bit of *key b*.
- Half equal keys: the first half bit of *key a* equals to the first half bit of *key b*, while the second half differs.
- Total different keys: each bit of *key a* differs from the corresponding bit of *key b*.

Keys are tested in pairs with different lengths from 4 bytes to 128 bytes, growing as two-powers. For each length total equal keys, half equal keys, and total different keys are compared.

A test starts allocating two memory areas for the two keys using `calloc` and a proper data type that can be `uint32_t` or `uint64_t`. Then `memset` is called to set the bytes values according to the specifics of the two keys (total equal, half equal, and total different). Finally a counter variable is incremented by one according to the result of the comparison between the two keys. The comparison is repeated a large amount of times. All bytes of key *b* are post-incremented after each comparison and pre-decremented before each comparison. This should convince the compiler that each comparison is different from the previous one and so it should avoid undesired code optimization by the compiler itself. The comparison is performed using one of the two approaches.

The first approach uses *logical* comparisons to increment the counter. Logical comparisons use the operator `and (&&)` and the operator `or (||)` which exploit the short-circuit evaluation. If the first argument of an *AND* comparison evaluates to false, then the entire function is false and therefore the second argument is not evaluated. If the first argument of an *OR* comparison evaluates to true, then the entire function is true and therefore the second argument is not evaluated. On normal tuning level 10^8 comparisons are performed. This is the code for 16 Bytes comparisons with half-equal keys:

```
uint64_t *a = (uint64_t *) calloc(2, sizeof(uint64_t));
uint64_t *b = (uint64_t *) calloc(2, sizeof(uint64_t));

memset(b + 1, UCHAR_MAX, sizeof(uint64_t) );

for ( uint64_t k = 0; k < total_iterations; k++ ) {
   counter += a[0] > b[0] || ( a[0] == b[0]++ && ( a[1]
       > b[1] || ( a[1] == b[1]++ ) ) );
   counter += a[0] > --b[0] || ( a[0] == b[0] && ( a[1]
       > --b[1] || ( a[1] == b[1] ) ) );
}
```

The second approach uses bitwise comparisons. The keys are compared bit to bit using the operators *bitwise* `and (&)` and *bitwise* `or (|)`. Every single bit of key *a* is compared with the corresponding bit of key *b*. Using this approach the evaluation is never stopped before the reaching of the end of the key. This approach is really fast on some hardware architecture. Test results may also show that this method is particularly efficient on specific key lengths. This is the code for 16 Bytes comparisons. Keys allocation is as above and it is not shown:

```
for ( uint64_t k = 0; k < total_iterations; k++ ) {
   counter += a[0] > b[0] | ( a[0] == b[0]++ & ( a[1] >
       b[1] | ( a[1] == b[1]++ ) ) );
   counter += a[0] > --b[0] | ( a[0] == b[0] & ( a[1] >
       --b[1] | ( a[1] == b[1] ) ) );
}
```

The length in characters of the evaluation code line grows exponentially on the length of the keys. An exponential regression curve applied to the available data set shows that the length of the code line grows as $20.3 \cdot 1.73^{\log_2 x - 1}$where x is the length of the key. For example, with a key of 64 Bytes, the evaluation code line is about 314 characters. Since it makes no sense to manually write these lines for long keys (such as 128 bytes), an *ad hoc* function is written to perform this operation.

3.2 CACHE TESTS

Cache size estimation

These tests try to estimate the size of L1, L2, and L3 caches. Sometimes L3 cache may not exist and sometime L2 and/or L3 may be shared along multi-core. The tests use a single core so the entire cache size should be estimated, even if it is shared. Caches sizes are evaluated using three approaches, two of them are O.S. based.

sysconf *approach*

The first approach checks if the values `_SC_LEVEL2_CACHE_SIZE` and `_SC_LEVEL3_CACHE_SIZE` are defined and eventually it calls the function `sysconf` to retrieve the size of L2 and L3 caches. It safely works on every *Linux* based system and returns the size of the cache in KB.

/proc/cpuinfo *approach*

The second approach is also strictly *Linux* based and tries to extract the cache size from the file `/proc/cpuinfo`. This file report for each *processor* (physical or logical) the attribute *cache size*. There is no way to know if this is the size of L1, L2 or L3 cache. However this value is still useful to be compared to the value reported by the other tests.

Direct cache estimation

The third approach measures the bandwidth of read operations from the main memory. It starts by reading a few KBytes which are surely copied into the L1 cache. Each successive iteration reads a larger amount of bytes and after some steps the total read amount does not longer fit in L1 cache. Increasing the size of the input, the same occurs for the L2 and eventually the L3 cache. Since data are read faster from smaller caches we can estimate the sizes of the caches by finding larger bandwidth gaps. For example if L2 cache size is *256 KiB* and we try to read *512 KiB*, the time spent will be relatively larger that the time needed to read *256 KiB*.

The test starts with the allocation of the vector in the main memory. The array data type is `elem_t` and contains a configurable number of `uint64_t` (actually *32*). The allocated memory is then initialized to random values. Now the timer starts and a loop reads an entire `elem_t` for each iteration, adding its values to the *checksum* variable. The elements are not read in sequence. The loop jumps inside the array using a quite large prime number *STEP* and the `mod` operation on the length of the array.

```
for ( uint64_t i = 0; i < NUM_ACCESSES; i += 1 ) {
   for ( uint64_t k = 0; k < STRUCT_SIZE; k++ )
     checksum += v[ v_pos ].content[ k ];
     v_pos = ( v_pos + STEP ) % n_elem;
}
```

Actually on normal tuning level `NUM_ACCESSES` is 2^{23}. Since `STRUCT_SIZE` is 32, an iteration reads a total of

$$\begin{aligned} num\_accesses \cdot struct\_size \cdot elem\_size &= total\_size \\ 2^{23} \cdot 32 \cdot 8 &= 2048\,\text{MiB} \end{aligned}$$

The timer stops at the end of the external loop and returns the calculated bandwidth. The measurement starts with a minimum array size of *8 KiB* and ends with a maximum array size of *48 MiB*. If x is the size of an iteration input, the next iteration has the size $x + \frac{x}{2}$. Now we have all bandwidth values in the interval *8 Kib - 48 MiB* and we can try to guess the size of the *two* most relevant caches in term of bandwidth. Table 1 shows the result of a bandwidth test as a function of the input size on an *Intel Core i7*.

Size *(KiB)*	Bandwidth *(MiB/s)*	Size *(KiB)*	Bandwidth *(MiB/s)*
8	20087	768	15920
12	20095	1024	15865
16	20155	1536	15851
24	20119	2048	15846
32	20244	3072	14045
48	19630	4096	12138
64	19622	6144	10188
96	19595	8192	7590
128	18169	12288	6025
192	19587	16384	5973
256	17972	24576	59401
384	16415	32768	5914
512	15968	49152	5851

Table 1: Cache bandwidth on *Intel Core i7 920*.

Cache size estimation algorithm

These data are plotted on Figure 7 on page 33. We return two cache size values which are two-powers[2], estimated as follows:

2 This is not a limitation for *psort*, since it works only with two-power values.

1. Create a list A that contains all sizes which are not two-powers (see the table above. Chosen values are 12, 24, 48, 96, and so on).

2. Consider all pairs of two contiguous values x and y from the list A, where $x < y$.

3. For each pair calculate the relative bandwidth variation between x and y:
$$variation = \left|\frac{bandwidth(x) - bandiwdth(y)}{bandwidth(x)}\right|$$

4. Extract the largest bandwidth variation and its corresponding pair a.

5. Extract the second largest bandwidth variation and its corresponding pair b. Each element of pair b must not be an element of pair a.

6. For both pairs a and b, return the two-power that is larger than x and smaller than y, where x and y are the elements of the pair.

The condition on point 5, assures that the algorithm does not consider two bandwidth gaps caused by the same cache. On the example shown in the table above, a is (192, 384) and b is (6144, 12288) so the estimated cache sizes are *256 KiB* and *8192 KiB*.

3.3 MAIN MEMORY TESTS

Read and write bandwidths of the main memory

These tests try to estimate the read and write bandwidth of the main memory. The design and implementation is really similar to the third approach of the cache size test. In particular the read test works exactly in the same way of the cache test. The only difference is that it usually works with larger inputs: its upper-bound is the total amount of available memory. Write test does not increment a *checksum* variable but writes a pseudo-random value (chosen as the loop counter value) in the memory area that is accessed. However a *checksum* variable is created to convince the compiler that the values written in the memory will be used:

```
checksum = v[v_pos].content[v_pos & (STRUCT_SIZE - 1)];
```

Next *checksum* is evaluated by an if-statement. This approach should force a compiler that uses a high optimization level to compile the entire source code as wanted. This aspect, briefly summarized here, will be analyzed at the end of this chapter.

3.4 DISK TESTS

Disks tests overview

These tests try to estimate the read and write bandwidth from/to a disk or from/to a RAID configuration. The first test estimates the two bandwidths during sequential read and write operations performed using the filesystem, while the second test estimates bandwidth (or better the access time) during read operations from the physical device.

3.4.1 Sequential read and write

Caches workarounds

There are two important caches which can affect the data collected by this test:

- Kernel cache
- Disk cache

The kernel cache is managed by the O.S. and may copy read and written data from/to the disk in a fast accessible location. This would invalidate all bandwidth calculation in repeated tests. It is bypassed by managing files with the flag `O_DIRECT` that is widely supported by *Linux* since version *2.4.10* and *FreeBSD* 4[3]. This flag allows direct read and write operation from the user's buffers space to the device without passing from the kernel cache. It may also be used in *psort* by compiling it with the appropriate flag. Unfortunately Direct I/O does not assure kernel-bypass and does not allow the management of all input/output operations. Usually on *Linux 2.6* or greater a *512-byte* alignment is required, while on elder versions there are additional boundaries on the transfer size and on the alignment of the user's buffer. While Direct I/O has been strongly criticized in the past (Torvald [14]), it is widely used in the database and high performance applications and looks like an excellent solution for our problem since we can work with values which are multiples of the *512-byte* boundary.

The disk cache is a hardware component of the disk itself and may cause the same problems of the kernel cache. The only way to avoid the effects of this cache is to load, before each execution, trivial data which differ from the data used in the next execution. This result is achieved by arranging the order of the executions in a strategic way.

Sequential read and write test setup

Read and write tests are performed alternatively starting from small input/output sizes which increase at each iteration. The tests stop when the bandwidth gap of two consecutive input/output sizes is smaller than a defined relative value or when the maximum input/output file size is reached. Actually on normal tuning level the relative bandwidth gap threshold is 5% and maximum file size is *1 GiB*. The tests run these operations:

3 For more information on this topic see Linux man-pages. Available: 3.27.

1. Write *inputFile* to the disk. The size of this file is the maximum file size.
2. Start from the smallest input/output size *test_size* and:
   a) Write *test_size* bytes to *outputFile.*
   b) Read *test_size* bytes from *inputFile.*
3. Calculate bandwidth for both tests.
4. If the bandwidth variation is larger than the threshold, repeat step 2 with a larger input/output size (typically the double of the previous one).
5. Print the two bandwidths.

Step 2 is repeated more then once with the same input/output size and the average bandwidth is considered in order to reduce noise effects. In addition, consecutive tests always works on different files (once on *inputFile* and once on *outputFile*) to minimize disk cache effects.

Files are managed using functions from `fcntl.h` and `unistd.h`. In particular the `open` function on write operations is called as follows:

```
open(pathname, O_CREAT|O_TRUNC|O_WRONLY|O_DIRECT,
    S_IRWXU);
```

Data are always read and written *entirely* from and to a buffer allocated with `posix_memalign`. *psort* uses buffers which usually have the same size of the values tested here.

3.4.2 Random read

Access time for a read operation

This test estimates the average access time needed for a random read operation from a physical device. Since it involves quite low level functions, it requires root privileges on the tested machine. The Listing 1 describes the test in pseudo-code. The pseudo-function `get_number_of_blocks` uses the function `ioctl(file, BLKGETSIZE, &numberOfBlocks)` that is dedicated to the control of devices attributes. `change_disk_reading_position` uses the function `lseek64(file, mini blocksize * offset, SEEK_SET)` that moves the offset of a 64-bit read/write file. The number of iterations is calculated starting from the number of bytes which are totally read by the test. Since usually *block_size* is *512 B,* the number of iterations is calculated as:

$$iterations = \frac{total\ read\ bytes}{block\_size} = \frac{total\ read\ bytes}{512\,\mathrm{B}}$$

On normal tuning level *total_read_bytes* is *512 KiB.*

Listing 1 Random reads algorithm

```
file = open ( device, read_only );
get_number_of_blocks ( file, number_of_blocks );

start_timer();

for each iteration {
   offset = number_of_blocks * random (0..1);
   change_disk_reading_position ( file, offset *
       block_size );
   read ( file, block_size );
}

stop_timer();

average_access_time = total_elapsed_time / iterations;
```

AVOID UNDESIRED COMPILER OPTIMIZATIONS

Be aware about compiler optimizations

To be as close as possible to *psort,* all hardware tests are compiled using *g++* and optimization level *3*. There are also other optimizations performed during the compiling process, in particular the following compiler flags are declared: *-funroll-loops, -funsafe-loop-optimizations, -march=native, -mtune=native.* They unroll loops and try to optimize the code according to the hardware architecture. Even if they are not so powerful as an *ad hoc* tuning, they significantly contribute to increase performance. However we should watch out for optimization side effects.

Our tests allocate variables or large memory areas and perform on them a lot of operations calculating the elapsed time or the bandwidth. Compilers try to track values and arrays which are initialized, modified but never accessed in the future: never printed, never used in a comparison, and so on. Then they may decide to simply remove from the code the operations performed on these values and arrays. This fact could cause the evaluation of totally low and wrong times.

To avoid such a problem, we implement functions which always perform a trivial operation on the data used during the test. The operation should be able to produce an output. In this way the compiler cannot discard any single line of code. The safer way to achieve this result is to compare a *checksum* or *counter* variable to an integer and eventually print a small output.

```
if ( checksum == 0 ) {
   printf("Test ended. Checksum value is zero."); }
```

Checksum and counter variables contain a trivial value obtained from the test, such as the sum of all accessed memory locations or the sum of all key comparison results in which the first key is smaller than the second key.

4 RESULTS

We test the hardware detection code on different machines. Some of them contain medium-end and high-end hardware components while others are ordinary machines which are used every day as personal computers. Even if the code is designed for *psort* we would test if it could be used also on low-end machines with different purposes. We test two different O.S. and both 32-bit / 64-bit architectures. The four main tested machines are:

Tested machines

Model	CPU	Main Memory	Tested disk
Desktop Ubuntu 11.04	Intel® Core i7 920 @ 3.8 GHz L1: 32 KiB, L2: 256 KiB, L3: 8192 KiB sh.	6 GiB DDR3 @ 1666 MHz	`ST3500320AS` 7200.11 SATA 3Gb/s 500-GB
Desktop Deb Linux 6.0	AMD® Phenom™ II X4 945 L1: 128 KiB, L2: 512 KiB, L3: 6144 KiB	8 GiB DDR3 @ 1066 MHz	`HDS721010CLA332` 7200 RPM 500 GB 5-disk RAID
Notebook Ubuntu 11.04	Intel® Core 2 Duo P8400 L1: 32 KiB, L2: 3072 KiB	4 GiB DDR2 @ 667 MHz	`WD3200BEVT` 5400 RPM SATA 3Gb/s 320-GB
Macbook Mac OS X 10.6	Intel® Core i5 @ 1.7 GHz L2: 256 KiB, L3: 3072 KiB shared	4 GiB DDR3 @ 1333 MHz	-

Table 2: Tested hardware configurations.

Appendix B contains the execution log of the entire installer package on the *Intel Core i7* machine.

Now we compare the results extracted from the collected data on different hardware configurations and try to evaluate the reliability of each test.

4.1 CPU TESTS

The first test is about pointers notation. On all tested machine *offset* notation is a bit slower than *subscript* notation. However the difference between the two implementations is so small that *psort* should not be optimized to take an advantage from them. The larger delta between the two collected times is on *MBA* and it is about *0.7%*. On *phenom* it is *0.6%* and it is even smaller on the other configurations. This is not noise because repeating the test brings to the same result: *offset* notation is always slower than *subscript*. The difference may be more consistent on other machines. Operations with pointers are so common that this difference could become a relevant factor of *psort* performance.

Pointers notation results

The second test analyzes branch evaluations. The results shows that the *if-else* approach is faster on some hardware configurations, the *boolean* in others, and there is no difference at all on some CPUs. Table 3 shows the collected times.

If-else against boolean statements results

CPU	Normal tuning *Execution time (s)*	Extreme tuning *Execution time (s)*
Intel Core i7	2.26 *if else*	18.11
	1.94 *boolean*	16.40
Phenom	1.99	15.93
	2.24	17.92
Core 2 Duo P8400	10.60	-
	12.60	-
Intel Core i5	3.49	-
	3.50	-

Table 3: Statement test results. The first rows refers to the *if-else* approach, while the second one to the *boolean* approach.

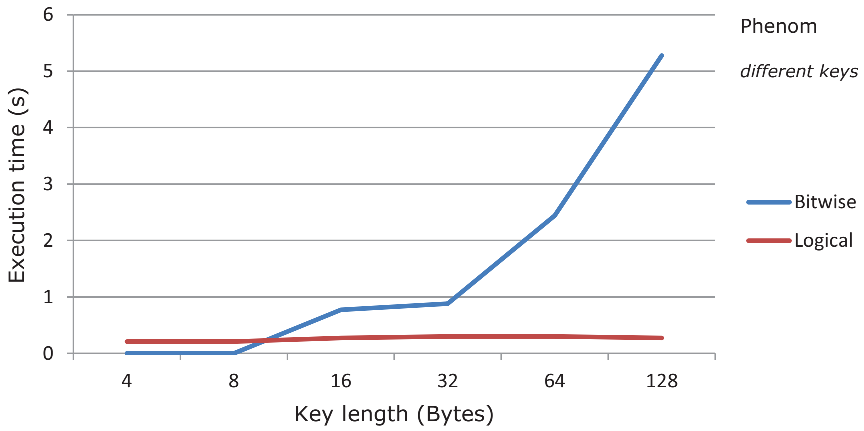
Intel Core i7 achieves better performance using the *boolean* approach with a time boost of *10-15%*. *AMD Phenom* works better with *if-else* approach that is about *12%* faster. The P8400 CPU is really faster using the *if-else* approach (*18%*) but it is absolutely the slowest CPU. It makes sense because it is also the eldest one. On *MBA* with *Core i5* there is no difference between the two approaches. Since *psort* widely uses branch evaluation, it should implement both and choose the fastest one according to the result of this test. The choice may be performed both at compiling or at run time but in order to produce a cleaner code, the implementation at compiling time looks better. The significant variation on the performance suggests that this is a critical feature to be add in *psort*. *Merge sort* uses these typologies of evaluations and its optimization could assure that CPU will not be a bottleneck for stage one.

Logical against bitwise comparisons results

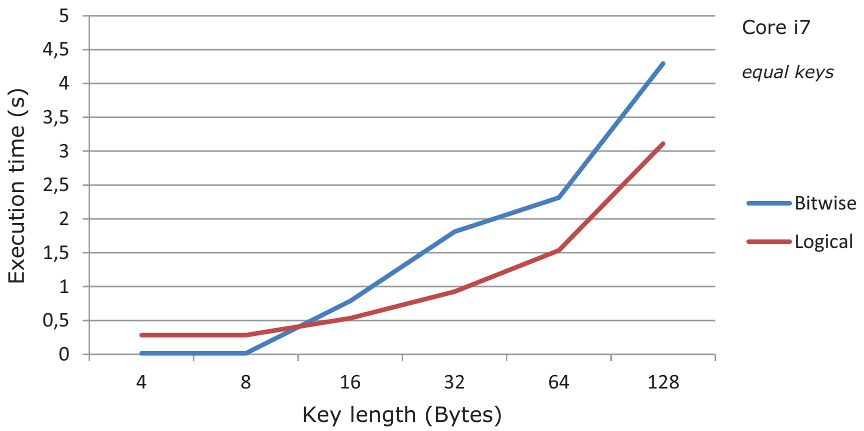
The third test is about key comparisons: logical and bitwise. A complete log of this test can be found in the appendix B. Figure 6 on the next page shows some the most relevant results, which are:

1. Bitwise evaluations are always faster comparing keys which are 8-byte or shorter, it does not matter if equal or different keys are compared. The gain is significant: from 4 times faster on *P8400* to hundred times faster on *Phenom.*

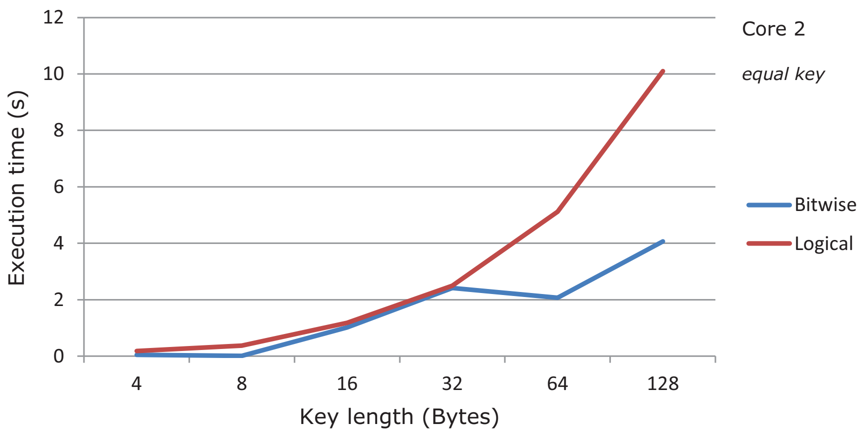
2. Comparing different keys which are longer then 8-byte can be performed using logical evaluations in constant time, independently on how long the key is. Logical approach, in fact, stops the evaluation after the first mismatch. Sometime the keys are constituted by random characters and so the probability that



(a) Comparing total different keys on *Phenom* requires a trivial time with the logical approach, and an exponential time on key length with the bitwise approach.



(b) On *Intel Core i7* the logical approach is faster working with large keys. This is not a general result, as shown in (c).



(c) On *Core 2 Duo* the bitwise approach is faster working with large keys. This is not a general result, as shown in (b).

Figure 6: Key comparison methods

the first n characters equal to the first n characters of another key is very small, for a sufficient large value of n. Since logical approach is faster than boolean only with keys which are longer than *8* characters[1], suppose that n is *8*. Then the probability that the first *8* characters equal to the first *8* char of another key is $(1/256)^8 \backsim 0$ for each key, assuming that each character is equally probable and that there are *256* different characters. Therefore for random keys longer than 8-byte logical evaluations should be used. A different situation happens with equal keys.

3. Comparing equal keys longer then 16-byte is faster using logical approach on some CPUs, such as *Intel Core i7*, while is faster using bitwise approach on other CPUs, such as *Core 2 Duo*.

There are different situation and possible combinations to be considered. Some *psort* users may want to sort incremental keys which are equal or half-equal. This is a common situation in databases environments, working with IDs. In order to answer to this requirement, the *installTuned.sh* file allows to choose, before the beginning of the installation, what typology of keys (random or incremental) will be sort more frequently. Then the result of this test is automatically load as input in the Code Tuner that compiles *psort* with the faster approach.

4.2 CACHE TESTS

O.S. cache tests results

Cache tests are divided in two groups. Tests in the first group try to retrieve cache values from the O.S. They provide the correct value but they are not guaranteed to work on every software configuration. In addition, sometimes they return indefinite values, such as the command `cat /proc/cpuinfo` that returns on *Intel Core i7*:

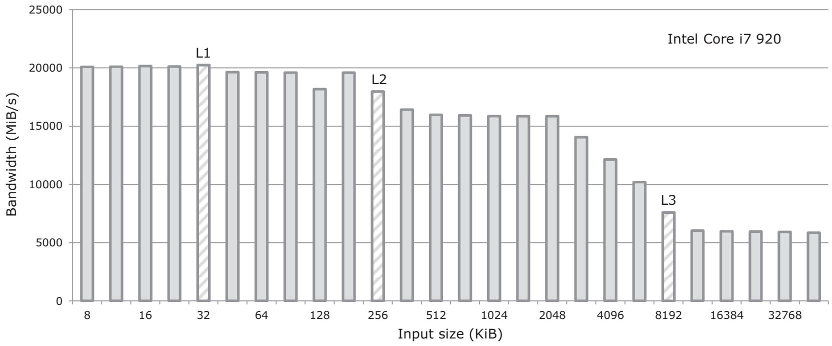
```
model name  : Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
stepping    : 5
cpu MHz     : 1600.000
cache size  : 8192 KB
```

Is this the size of L1, L2 or L3 cache? It is not specified. Moreover the value *CPU MHz*, even if it does not actually interest us, is wrong and may convince us to doubt about the other values returned by this command. However `cache size` is still useful to perform comparisons with other results.

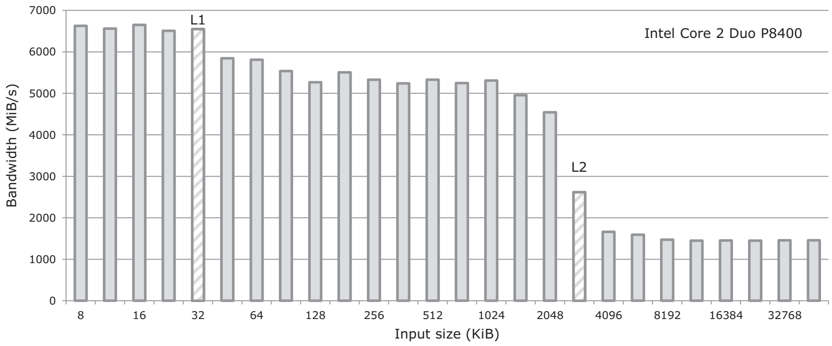
Direct estimation cache test result

The second group is composed by only one test that directly tries to

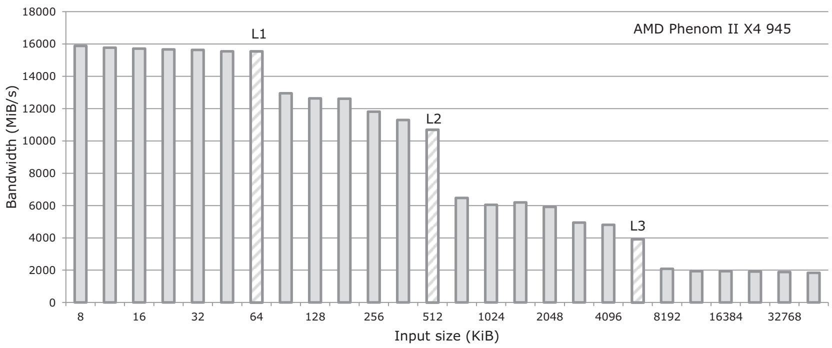
1 We are assuming that 1 character is 1-byte, that is a common (but not the only one) situation for keys in text files.



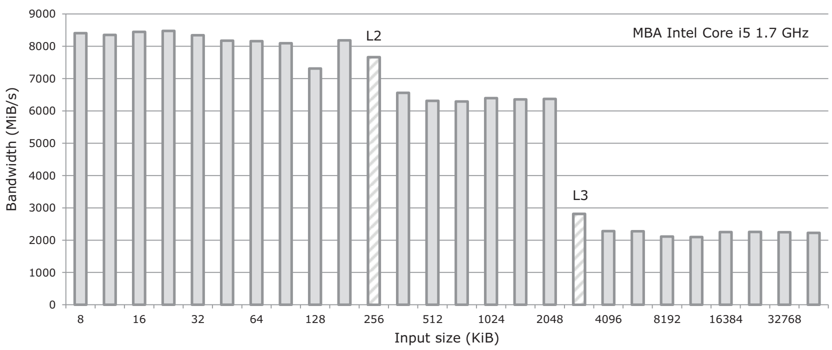
(a) Cache size test on Intel Core i7 920



(b) Cache size test on Intel Core 2 Duo P8400



(c) Cache size test on AMD Phenom II X4 945



(d) Cache size test on MBA Intel Core i5 1.7 GHz

Figure 7: Cache detection test performed on different CPUs. Dashed bars corresponds to the values declared by the vendors.

estimate the cache size as described in 3.2. Figure 7 on the preceding page shows all collected bandwidths which are used to estimate the sizes of the caches. The bars filled with diagonal dashed lines represent the nominal cache values of the CPU. The algorithm correctly estimate the two most significant caches for each CPU. If a CPU has a cache size that is not a two-power, the nearest two power is returned.

Intel Core i7 *cache result*

Sub-figure (a) refers to *Intel Core i7* and is divided in three levels of bandwidths. The first level, that corresponds to the nominal level of the L1 cache (*32 KiB*), is difficult to be detected. The reason could be related to the really small size of this cache. However, focusing on the values, there is a visible bandwidth gap between *32 KiB* and *48 KiB* and it is about 3%: the other gaps near *32 KiB* are all smaller. Finding the other two caches is easier because the gap between the value immediately before and immediately after a cache size, is very significant: 19% for the L2 cache and 70% for the L3 cache. Visually L2 and L3 caches have bandwidths which are at the halfway between the average bandwidth of the two adjacent levels. The algorithm detects correctly the L2 and L3 cache values.

Intel Core 2 Duo *cache result*

Sub-figure (b) refers to *Intel Core 2 Duo* that has two caches, both correctly identified (the L2 as the closest smaller two-power). The gaps are really visible and the relative bandwidth variation between the previous and the following values of the L2 cache reaches the value of 172%.

AMD Phenom *cache result*

Sub-figure (c) refers to *Phenom* and shows the most linear result: there are three steps and each one ends with the nominal size of a cache. For each step, the average value of the bandwidth is minimally affected by noise and so has a small variance. In some CPUs the cache size corresponds to the first value that gives a smaller bandwidth compared with smaller sizes. In this case it corresponds to the last value that has a bandwidth in average with smaller sizes. This could be related to the particular hardware architecture or to the particular software configuration, e.g. number of active processes during the test.

Intel Core i5 *cache result*

Sub-figure (d) refers to *MBA Intel Core i5*. It is curious to note that there is not nominal L1 cache size for this CPU and in fact the test confirms this particularity. Yet another time, the two largest gaps identify the two caches. However this time L3 cache size is the first value to give a small bandwidth compared to the previous one.

The test does not only identify the caches, but also proves that they significantly affect performance. Therefore we must work with data sizes which fit the caches. Actually in this package *psort* is designed to run *cache sorter* using the size of the L2 cache.

4.3 MAIN MEMORY TESTS

RAM test results

Tests on main memory collect data which can be useful to understand if the main memory is a bottleneck for stage one. They simply calculate the bandwidth of reading and writing operations from and to the RAM. First of all we can compare the bandwidth of different memories. This could help us to evaluate how much a high-end memory is faster then a low or medium-end memory and therefore we can understand if a faster memory may improve *psort* stage one performance or if the disks limit the entire process. Table 4 compares different hardware configurations.

Memory	Read bandwidth *(MiB/s)*	Write bandwidth *(MiB/s)*
`CMT6GX3M3A1866C9` 6 GiB DDR3 @ 1666 MHz on Intel Core i7 920	6430	7803
8 GiB DDR3 @ 1066 MHz on AMD Phenom 945	3077	1803
4 GiB DDR2 @ 667 MHz on Intel Core 2 Duo	1619	1378
4 GiB DDR3 @ 1333 MHz on Intel Core i5	2613	5238

Table 4: Read and write bandwidths of the main memory on different hardware configurations.

It is interesting to note that in some memories read operations are faster than write operations while in other ones the vice versa is true. Furthermore these data are collected accessing 256-byte atomically. The estimated bandwidth significantly changes accessing a different amount of bytes, e.g. on *Intel Core i7* (see Table 5).

`CMT6GX3M3A1866C9` Bytes atomically accessed	Read bandwidth *(MiB/s)*	Write bandwidth *(MiB/s)*
128	4979	7240
256	6430	7803
512	5952	8060

Table 5: Read and write bandwidths change with the number of bytes atomically accessed.

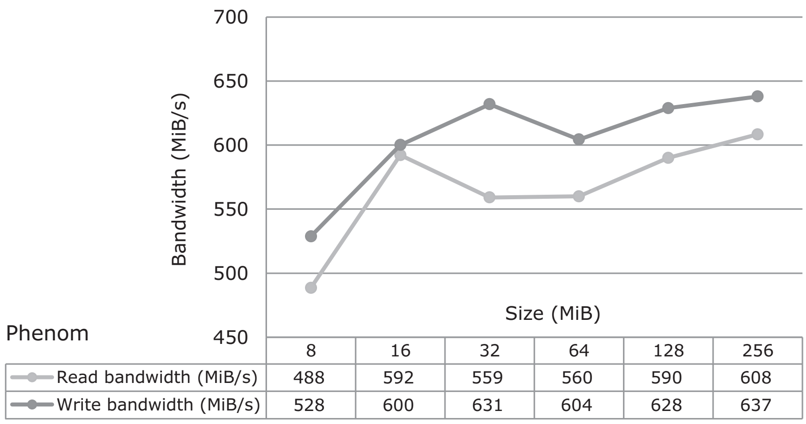


Figure 8: Disk bandwidth estimation on *Phenom*

4.4 DISK TESTS

Disk tests results: read and write bandwidths

The only tested machine with a relevant disks configuration is *Phenom* that has a RAID array divided in three partitions (slow, medium, and fast) according to the rotation speed of the disk. In fact "*transfer time is lower for data logically closer to the beginning of the array, corresponding physically to the area of the disk closer to the outer rim*" as stated in [8]. The estimated bandwidth for the fast section of this array is *608 MiB/s* (read) and *637 MiB/s* (write) as shown in Figure 8. The test ends as designed with an input/output size of *256 MiB* because, considering the previous execution, the delta of bandwidth is less then 4% both for reading and writing operations.

The other machines use a single disk with a low-end read bandwidth of 75 MiB/s (*Intel Core i7* machine) and 27 MiB/s (*Intel Core 2 Duo* machine).

Disk tests results: seek time

The second test is about access time/seek time for a single device without passing for the filesystem. The data collected are slightly different from the nominal values reported by the vendors as shown on Table 6.

Device	Nominal seek time *(ms)*	Estimated seek time *(ms)*
`ST3500320AS` 7200.11 SATA 3Gb/s 500-GB	8.5	12.71
`WD3200BEVT` 5400 RPM SATA 3Gb/s 320-GB	12.00	13.05

Table 6: Hard disk seek times do not always match the nominal value.

This could happen because every disk is unique: buying multiple copies of the same disk model, there could be high differences in performance.

We finally recall that *psort*, according to the Sort Benchmark specifics [6], manages data using the filesystem and not directly from/to the device.

5 CONCLUSIONS

The actual role of hardware detection

The test environment described in this work allows us to conclude that choosing optimal value for critical *psort* parameters drastically increases its performance. Furthermore the optimal values of these parameters are strictly related to the hardware configuration and therefore automatic estimation of hardware parameters plays an important role in the tuning process of *psort*.

Nowadays there is a large variety of possible hardware/software combinations and consequently it is unsafe to retrieve all hardware parameters only through O.S. based functions. The specifics of a hardware model may also differ from a particular hardware component to another. These aspects should convince that the data must be collected with different approach, including the direct test of the hardware component. In fact, a hardware component that is declared to work according to certain specifics, may vary its performance in relation to its interaction with the other hardware components. Hardware architectures are so complex that it is impossible to theoretically estimate an implementation that gives a better performance without testing it.

Further developments

Even if the tests are mainly general purpose and independent from *psort*, our work would be a starting point to insert *psort* in the universe of the automatically tuned software. Further developments are widely possible and may move into two directions: the design and implementations of new tests, and the addition of some code portion in *psort* capable to take advantage of the tests results.

There is an infinity of possible new tests but writing the current hardware detection code bring us to identify some tests, which can be more interesting and more useful for *psort*:

1. Add another OS-dependent approach to estimate cache sizes and other cache values. Actually we are collecting only the size of the L1, L2 and L3 caches. It would be useful to collect also other values such as the ways of associativity and cache line size. The command `grep . /sys/devices/system/cpu/cpu0/cache/index*/*` is a good starting point. Cache line size may also be calculated.

2. Test *psort* using *Callgrind*, a component of *Vallgrind* and analyze the results with *kcachegrind.* These tools should allow to deeper understand how *psort* exploits the caches according to their sizes.

3. Test disk performance using alternative solutions to Direct I/O such as `madvise` and `posix_fadvise`.

The current test structure is designed to easily allow the addition of new tests using a modular approach.

Regarding the implementation in *psort* of new code that exploits test results, the collected and analyzed data of Chapter 4 suggests that the first step should be the adding of both *if-else* and *boolean* evaluations during *k-way merge sort* operations.

Code reliability and usability

This work already implements the exploit of some hardware-related values together with Code-tuning and *psort* itself. It provides further developers with a complete test environment and the basic structure to add new tests. The test system has proved to be reliable even if run using a fast tuning level that takes only a few minutes on modern hardware configuration. It is performed only one time, during the installation process. Since the results show that the correct choice of a solution according to the hardware, speed-up different *psort* operations and since we do not find any obstacles, we propose to add the automatic estimation of hardware parameters to the current version of *psort*.

A SOURCE CODE

A.1 CPU TESTS SOURCE CODE

Listing 2: Extract from CPU tests source code

```
#include "pre-tuner_cpu.h"

int main( int argc, const char *argv[] ) {

bool run_test_one = TEST_ONE_RUN;
bool run_test_two = TEST_TWO_RUN;
bool run_test_three = TEST_THREE_RUN;

printf("\n### Starting CPU TESTS ###\n");

createOutputFiles();

/* TEST 1 (pointers) */

if ( run_test_one ) {

  printf("\nStarting test 1 (pointers). It will be
      repeated %d time(s). Every test works with an
      array of size %lu MiB.\n\n", TEST_ONE_NUM, ((
      unsigned long) TEST_ONE_ARRAY_ELEMENTS) * sizeof(
      TEST_ONE_ARRAY_ELEMENTS_TYPE) / (1<<20) );

  runFoolOperations(TEST_ONE_FOOL_OPERATION_AMOUNT_MB);
  double subscript_total_time = runTestOneSubscript(
      TEST_ONE_NUM, TEST_ONE_ARRAY_ELEMENTS );
  runFoolOperations(TEST_ONE_FOOL_OPERATION_AMOUNT_MB);
  double offset_total_time = runTestOneOffset(
      TEST_ONE_NUM, TEST_ONE_ARRAY_ELEMENTS );

  printf("Subscript notation total time: %f\n",
      subscript_total_time );
  printf("Offset notation total time: %f\n",
      offset_total_time );

  //cut: save results code here
}

/* TEST 2 (branch-merge) */

```

```
if ( run_test_two ) {

  printf("\nStarting test 2 (branch−merge). It will be
      repeated %d time(s). Every test performs %lu
      comparisons.\n\n", TEST_TWO_NUM,
      TEST_TWO_SINGLE_TEST_LENGTH);

  runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
  double ifelse_total_time = runTestTwoIfElse(
      TEST_TWO_NUM, TEST_TWO_SINGLE_TEST_LENGTH);
  runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
  double boolean_total_time = runTestTwoBoolean(
      TEST_TWO_NUM, TEST_TWO_SINGLE_TEST_LENGTH);

  printf("If−else approach total time: %f\n",
      ifelse_total_time);
  printf("Boolean approach total time: %f\n",
      boolean_total_time);

  //cut: save results code here;
}


/* TEST 3 (logical against bitwise comparisons) */

if ( run_test_three ) {

  printf("\nStarting test 3 (logical−bitwise). Every
      test performs %lu comparisons.\n\n",
      TEST_THREE_SINGLE_TEST_LENGTH);

  double B4_logical_time;
  double B4_bitwise_time;
  double B8_logical_time;
  double B8_bitwise_time;
  double B16_logical_time;
  double B16_bitwise_time;
  double B32_logical_time;
  double B32_bitwise_time;
  double B64_logical_time;
  double B64_bitwise_time;
  double B128_logical_time;
  double B128_bitwise_time;

  if ( TEST_THREE_RUN_KEY_LEVEL_ZERO ) {

   runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
   B4_logical_time = runTestThreeLogical_4B(
       TEST_THREE_SINGLE_TEST_LENGTH, 0 );
   runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
   B4_bitwise_time = runTestThreeBitwise_4B(
       TEST_THREE_SINGLE_TEST_LENGTH, 0 );
```

```
      printf("4 Byte (equal key) Logical time: %f\n",
          B4_logical_time);
      printf("4 Byte (equal key) Bitwise time: %f\n",
          B4_bitwise_time);

      //cut: duplicated code for larger keys here

      printf("\n");

      //cut: save results code here

      /* Export data for code-tuner */

      char outputString[256] = "equal keys:";
      if ( B8_bitwise_time < B8_logical_time )
        sprintf(outputString, "%s %d", outputString, 1);
      else
        sprintf(outputString, "%s %d", outputString, 2);
      if ( B16_bitwise_time < B16_logical_time )
        sprintf(outputString, "%s %d", outputString, 1);
      else
        sprintf(outputString, "%s %d", outputString, 2);
      if ( B32_bitwise_time < B32_logical_time )
        sprintf(outputString, "%s %d %d\n", outputString,
            1, 1);
      else
        sprintf(outputString, "%s %d %d\n", outputString,
            2, 2);
      writeStringToFile("codetuner", outputString);
    }

    if ( TEST_THREE_RUN_KEY_LEVEL_ONE ) {

      runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
      B4_logical_time = runTestThreeLogical_4B(
          TEST_THREE_SINGLE_TEST_LENGTH, 1 );
      runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
      B4_bitwise_time = runTestThreeBitwise_4B(
          TEST_THREE_SINGLE_TEST_LENGTH, 1 );
      printf("4 Byte (half equal key) Logical time: %f\n",
          B4_logical_time);
      printf("4 Byte (half equal key) Bitwise time: %f\n",
          B4_bitwise_time);

      //cut: duplicated code for larger keys here

      //cut: save results code here

      /* Export data for code-tuner */

      char outputString[256] = "half-equal keys:";
      if ( B8_bitwise_time < B8_logical_time )
```

```
        sprintf(outputString, "%s %d", outputString, 1);
      else
        sprintf(outputString, "%s %d", outputString, 2);
      if ( B16_bitwise_time < B16_logical_time )
        sprintf(outputString, "%s %d", outputString, 1);
      else
        sprintf(outputString, "%s %d", outputString, 2);
      if ( B32_bitwise_time < B32_logical_time )
        sprintf(outputString, "%s %d %d\n", outputString, 1, 1);
      else
        sprintf(outputString, "%s %d %d\n", outputString, 2, 2);
      writeStringToFile("codetuner", outputString);

    }

    if ( TEST_THREE_RUN_KEY_LEVEL_TWO ) {

     runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
     B4_logical_time = runTestThreeLogical_4B( TEST_THREE_SINGLE_TEST_LENGTH, 2 );
     runFoolOperations(TEST_TWO_FOOL_OPERATION_AMOUNT_MB);
     B4_bitwise_time = runTestThreeBitwise_4B( TEST_THREE_SINGLE_TEST_LENGTH, 2 );
     printf("4 Byte (total different key) Logical time: %f \n", B4_logical_time);
     printf("4 Byte (total different key) Bitwise time: %f \n", B4_bitwise_time);

      //cut: duplicated code for larger keys here

      //cut: save results code here

      /* Export data for code-tuner */

      //cut: save results code here

  }

  printf("\n");

  return 0;

  }

  double runTestOneSubscript( int input_test_num, unsigned long input_elements_num) {

    int tests_num = input_test_num;
    size_t elements_num = (size_t) input_elements_num;

```

```
  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for (int k = 0; k < tests_num; k++ ) {

    typedef TEST_ONE_ARRAY_ELEMENTS_TYPE element_type;

    element_type *array = (element_type *) calloc(
        elements_num, sizeof(element_type) );
    element_type *arrayPtr = array;

    for (unsigned long i = 0; i < elements_num; i++)
      array[i]++;

    srand( time(NULL) );
    if ( arrayPtr[ rand() % elements_num ] == rand() )
        //just try to avoid compiler's trick forcing it
        to use the array
      printf("Single exec of test one completed with a
          match.\n");
    free(arrayPtr);
  }

  gettimeofday( &end_time, NULL );

  return ( double ) ( ( end_time.tv_sec - start_time.
      tv_sec ) * 1000000 + ( end_time.tv_usec -
      start_time.tv_usec ) ) / 1000000;
}


double runTestOneOffset( int input_test_num, unsigned
    long input_elements_num) {

  int tests_num = input_test_num;
  size_t elements_num = (size_t) input_elements_num;

  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for (int k = 0; k < tests_num; k++ ) {

    typedef TEST_ONE_ARRAY_ELEMENTS_TYPE element_type;

    element_type *array = (element_type *) calloc(
        elements_num, sizeof(element_type) );
    element_type *arrayPtr = array;

```

```
      for (unsigned long i = 0; i < elements_num; i++)
        (*(array++))++;

      srand( time(NULL) );
      if ( arrayPtr[ rand() % elements_num ] == rand() )
          //just try to avoid compiler's trick forcing it
          to use the array
        printf("Single exec of test one completed with a
            match.\n");
      free(arrayPtr);
    }

    gettimeofday( &end_time, NULL );

    return ( double ) ( ( end_time.tv_sec - start_time.
        tv_sec ) * 1000000 + ( end_time.tv_usec -
        start_time.tv_usec ) ) / 1000000;
  }



  double runTestTwoIfElse(int input_test_num, unsigned
      long input_single_test_length) {

    unsigned long single_test_length =
        input_single_test_length;
    int tests_num = input_test_num;
    uint64_t a;
    uint64_t b;
    unsigned long counter = 0;

    struct timeval start_time;
    struct timeval end_time;

    gettimeofday( &start_time, NULL );

    for (int k = 0; k < tests_num; k++ ) {

      a = 1377923;
      b = 1029341;

      for (unsigned long i = 0; i < single_test_length; i
          ++) {
        a += 10000;
        b = a ^ b;
        if ( a < b )
          counter++;
      }
    }

    gettimeofday( &end_time, NULL );
```

```
  if ( counter == 0 ) // just try to avoid compiler's trick forcing it to use counter
    printf("Test completed with an entire counter's cycle.\n");

  return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestTwoBoolean(int input_test_num, unsigned long input_single_test_length) {

  unsigned long single_test_length = input_single_test_length;
  int tests_num = input_test_num;
  uint64_t a;
  uint64_t b;
  unsigned long counter = 0;

  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for (int k = 0; k < tests_num; k++ ) {

    a = 1377923;
    b = 1029341;

    for (unsigned long i = 0; i < single_test_length; i++) {
      a += 10000;
      b = a ^ b;
      counter += ( a < b );
    }
  }

  gettimeofday( &end_time, NULL );
  if ( counter == 0 ) // just try to avoid compiler's trick forcing it to use counter
    printf("Test completed with an entire counter's cycle.\n");

  return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeLogical_4B ( uint64_t input_total_iterations, int key_level ) {
```

```
const uint64_t total_iterations = input_total_iterations / 2;
uint32_t *a = (uint32_t *) calloc( 1, sizeof(uint32_t) );
uint32_t *b = (uint32_t *) calloc( 1, sizeof(uint32_t) );
uint64_t counter = 1;

if ( key_level == 1 )
  memset(((char *) b) + 2, UCHAR_MAX, sizeof(uint32_t) / 2 );
else if ( key_level == 2)
  memset(b, UCHAR_MAX, sizeof(uint32_t) );

struct timeval start_time;
struct timeval end_time;

gettimeofday( &start_time, NULL );

for ( uint64_t k = 0; k < total_iterations; k++ ) {
  counter += a[0] > b[0] || ( a[0] == b[0]++ );
  counter += a[0] > --b[0] || ( a[0] == b[0] );
}

gettimeofday( &end_time, NULL );
free(a); free(b);
if (counter == 0 )
  printf("Test completed with an entire counter's cycle.\n");

return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeBitwise_4B ( uint64_t input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations / 2;
uint32_t *a = (uint32_t *) calloc( 1, sizeof(uint32_t) );
uint32_t *b = (uint32_t *) calloc( 1, sizeof(uint32_t) );
uint64_t counter = 1;

if ( key_level == 1 )
  memset(((char *) b) + 2, UCHAR_MAX, sizeof(uint32_t) / 2 );
else if ( key_level == 2)
  memset(b, UCHAR_MAX, sizeof(uint32_t) );

struct timeval start_time;
```

```
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for ( uint64_t k = 0; k < total_iterations; k++ ) {
    counter += a[0] > b[0] | ( a[0] == b[0]++ );
    counter += a[0] > --b[0] | ( a[0] == b[0] );
  }

  gettimeofday( &end_time, NULL );
  free(a); free(b);
  if (counter == 0 )
    printf("Test completed with an entire counter's cycle.\n");

  return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeLogical_8B ( uint64_t input_total_iterations, int key_level ) {
  const uint64_t total_iterations = input_total_iterations / 2;
  uint64_t *a = (uint64_t *) calloc( 1, sizeof(uint64_t) );
  uint64_t *b = (uint64_t *) calloc( 1, sizeof(uint64_t) );
  uint64_t counter = 1;

  if ( key_level == 1 )
    memset(((int *) b) + 1, UCHAR_MAX, sizeof(uint64_t) / 2 );
  else if ( key_level == 2)
    memset(b, UCHAR_MAX, sizeof(uint64_t) );

  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for ( uint64_t k = 0; k < total_iterations; k++ ) {
    counter += a[0] > b[0] || ( a[0] == b[0]++ );
    counter += a[0] > --b[0] || ( a[0] == b[0] );
  }

  gettimeofday( &end_time, NULL );
  free(a); free(b);
  if (counter == 0 )
    printf("Test completed with an entire counter's cycle.\n");

```

```
return ( double ) ( ( end_time.tv_sec − start_time.tv_sec ) * 1000000 + ( end_time.tv_usec − start_time.tv_usec ) ) / 1000000;
}

double runTestThreeBitwise_8B ( uint64_t input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations / 2;
uint64_t *a = (uint64_t *) calloc( 1, sizeof(uint64_t) );
uint64_t *b = (uint64_t *) calloc( 1, sizeof(uint64_t) );
uint64_t counter = 1;

if ( key_level == 1 )
  memset(((int *) b) + 1, UCHAR_MAX, sizeof(uint64_t) / 2 );
else if ( key_level == 2)
  memset(b, UCHAR_MAX, sizeof(uint64_t) );

struct timeval start_time;
struct timeval end_time;

gettimeofday( &start_time, NULL );

for ( uint64_t k = 0; k < total_iterations; k++ ) {
  counter += a[0] > b[0] | ( a[0] == b[0]++ );
  counter += a[0] > --b[0] | ( a[0] == b[0] );
}

gettimeofday( &end_time, NULL );
free(a); free(b);
if (counter == -1 )
  printf("Test completed with an entire counter's cycle.\n");

return ( double ) ( ( end_time.tv_sec − start_time.tv_sec ) * 1000000 + ( end_time.tv_usec − start_time.tv_usec ) ) / 1000000;
}

double runTestThreeLogical_16B ( uint64_t input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations / 2;
uint64_t *a = (uint64_t *) calloc( 2, sizeof(uint64_t) );
uint64_t *b = (uint64_t *) calloc( 2, sizeof(uint64_t) );
uint64_t counter = 1;

```

```
    if ( key_level == 1 )
      memset(b + 1, UCHAR_MAX, sizeof(uint64_t) );
    else if ( key_level == 2)
      memset(b, UCHAR_MAX, sizeof(uint64_t) * 2 );

    struct timeval start_time;
    struct timeval end_time;

    gettimeofday( &start_time, NULL );

    for ( uint64_t k = 0; k < total_iterations; k++ ) {
      counter += a[0] > b[0] || ( a[0] == b[0]++ && ( a[1] > b[1] || ( a[1] == b[1]++ ) ) );
      counter += a[0] > --b[0] || ( a[0] == b[0] && ( a[1] > --b[1] || ( a[1] == b[1] ) ) );
    }

    gettimeofday( &end_time, NULL );
    free(a); free(b);
    if (counter == 0 )
      printf("Test completed with an entire counter's cycle.\n");

    return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeBitwise_16B ( uint64_t input_total_iterations, int key_level ) {
    const uint64_t total_iterations = input_total_iterations / 2;
    uint64_t *a = (uint64_t *) calloc( 2, sizeof(uint64_t) );
    uint64_t *b = (uint64_t *) calloc( 2, sizeof(uint64_t) );
    uint64_t counter = 1;

    if ( key_level == 1 )
      memset(b + 1, UCHAR_MAX, sizeof(uint64_t) );
    else if ( key_level == 2)
      memset(b, UCHAR_MAX, sizeof(uint64_t) * 2 );

    struct timeval start_time;
    struct timeval end_time;

    gettimeofday( &start_time, NULL );

    for ( uint64_t k = 0; k < total_iterations; k++ ) {
      counter += a[0] > b[0] | ( a[0] == b[0]++ & ( a[1] > b[1] | ( a[1] == b[1]++ ) ) );
```

```
    counter += a[0] > --b[0] | ( a[0] == b[0] & ( a[1] > --b[1] | ( a[1] == b[1] ) ) );
  }

  gettimeofday( &end_time, NULL );
  free(a); free(b);
  if (counter == 0 )
    printf("Test completed with an entire counter's cycle.\n");

  return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}
double runTestThreeLogical_32B ( uint64_t input_total_iterations, int key_level ) {
  const uint64_t total_iterations = input_total_iterations / 2;
  uint64_t *a = (uint64_t *) calloc( 4, sizeof(uint64_t) );
  uint64_t *b = (uint64_t *) calloc( 4, sizeof(uint64_t) );
  uint64_t counter = 1;

  if ( key_level == 1 )
    memset(b + 2, UCHAR_MAX, sizeof(uint64_t) * 2 );
  else if ( key_level == 2)
    memset(b, UCHAR_MAX, sizeof(uint64_t) * 4 );

  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time, NULL );

  for ( uint64_t k = 0; k < total_iterations; k++ ) {
    counter += a[0] > b[0] || ( a[0] == b[0]++ && ( a[1] > b[1] || ( a[1] == b[1]++ && ( a[2] > b[2] || ( a[2] == b[2]++ && ( a[3] > b[3] || ( a[3] == b[3]++ ) ) ) ) ) ) );
    counter += a[0] > --b[0] || ( a[0] == b[0] && ( a[1] > --b[1] || ( a[1] == b[1] && ( a[2] > --b[2] || ( a[2] == b[2] && ( a[3] > --b[3] || ( a[3] == b[3] ) ) ) ) ) ) );
  }

  gettimeofday( &end_time, NULL );
  free(a); free(b);
  if (counter == 0 )
    printf("Test completed with an entire counter's cycle.\n");

```

```
return ( double ) ( ( end_time.tv_sec - start_time.
    tv_sec ) * 1000000 + ( end_time.tv_usec - start_time
    .tv_usec ) ) / 1000000;
}

double runTestThreeBitwise_32B ( uint64_t
    input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations
     / 2;
uint64_t *a = (uint64_t *) calloc( 4, sizeof(uint64_t) )
    ;
uint64_t *b = (uint64_t *) calloc( 4, sizeof(uint64_t) )
    ;
uint64_t counter = 1;

if ( key_level == 1 )
  memset(b + 2, UCHAR_MAX, sizeof(uint64_t) * 2 );
else if ( key_level == 2)
  memset(b, UCHAR_MAX, sizeof(uint64_t) * 4 );

struct timeval start_time;
struct timeval end_time;

gettimeofday( &start_time, NULL );

for ( uint64_t k = 0; k < total_iterations; k++ ) {
  counter += a[0] > b[0] | ( a[0] == b[0]++ & ( a[1] > b
      [1] | ( a[1] == b[1]++ & ( a[2] > b[2] | ( a[2] ==
       b[2]++ & ( a[3] > b[3] | ( a[3] == b[3]++ ) ) ) )
       ) ) );
  counter += a[0] > --b[0] | ( a[0] == b[0] & ( a[1] >
      --b[1] | ( a[1] == b[1] & ( a[2] > --b[2] | ( a[2]
       == b[2] & ( a[3] > --b[3] | ( a[3] == b[3] ) ) )
      ) ) ) );
}

gettimeofday( &end_time, NULL );
free(a); free(b);
if (counter == 0 )
  printf("Test completed with an entire counter's cycle
      .\n");

return ( double ) ( ( end_time.tv_sec - start_time.
    tv_sec ) * 1000000 + ( end_time.tv_usec - start_time
    .tv_usec ) ) / 1000000;
}

double runTestThreeLogical_64B ( uint64_t
    input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations
     / 2;
```

```
  uint64_t *a = (uint64_t *) calloc( 8, sizeof(uint64_t) );
  uint64_t *b = (uint64_t *) calloc( 8, sizeof(uint64_t) );
  uint64_t counter = 1;

  if ( key_level == 1 )
    memset(b + 4, UCHAR_MAX, sizeof(uint64_t) * 4 );
  else if ( key_level == 2)
    memset(b, UCHAR_MAX, sizeof(uint64_t) * 8 );

  struct timeval start_time;
  struct timeval end_time;

  gettimeofday( &start_time , NULL );

  for ( uint64_t k = 0; k < total_iterations; k++ ) {
    counter += a[0] > b[0] || ( a[0] == b[0]-- && ( a[1] > b[1] || ( a[1] == b[1]-- && ( a[2] > b[2] || ( a[2] == b[2]-- && ( a[3] > b[3] || ( a[3] == b[3]-- && ( a[4] > b[4] || ( a[4] == b[4]-- && ( a[5] > b[5] || ( a[5] == b[5]-- && ( a[6] > b[6] || ( a[6] == b[6]-- && ( a[7] > b[7] || ( a[7] == b[7]-- ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
    counter += a[0] > ++b[0] || ( a[0] == b[0] && ( a[1] > ++b[1] || ( a[1] == b[1] && ( a[2] > ++b[2] || ( a[2] == b[2] && ( a[3] > ++b[3] || ( a[3] == b[3] && ( a[4] > ++b[4] || ( a[4] == b[4] && ( a[5] > ++b[5] || ( a[5] == b[5] && ( a[6] > ++b[6] || ( a[6] == b[6] && ( a[7] > ++b[7] || ( a[7] == b[7] ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
  }

  gettimeofday( &end_time, NULL );
  free(a); free(b);
  if (counter == 0 )
    printf("Test completed with an entire counter's cycle.\n");

  return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeBitwise_64B ( uint64_t input_total_iterations, int key_level ) {
  const uint64_t total_iterations = input_total_iterations / 2;
  uint64_t *a = (uint64_t *) calloc( 8, sizeof(uint64_t) );
  uint64_t *b = (uint64_t *) calloc( 8, sizeof(uint64_t) );
```

```
    uint64_t counter = 1;

    if ( key_level == 1 )
      memset(b + 4, UCHAR_MAX, sizeof(uint64_t) * 4 );
    else if ( key_level == 2)
      memset(b, UCHAR_MAX, sizeof(uint64_t) * 8 );

    struct timeval start_time;
    struct timeval end_time;

    gettimeofday( &start_time, NULL );

    for ( uint64_t k = 0; k < total_iterations; k++ ) {
      counter += a[0] > b[0] | ( a[0] == b[0]-- & ( a[1] > b[1] | ( a[1] == b[1]-- & ( a[2] > b[2] | ( a[2] == b[2]-- & ( a[3] > b[3] | ( a[3] == b[3]-- & ( a[4] > b[4] | ( a[4] == b[4]-- & ( a[5] > b[5] | ( a[5] == b[5]-- & ( a[6] > b[6] | ( a[6] == b[6]-- & ( a[7] > b[7] | ( a[7] == b[7]--  ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
      counter += a[0] > ++b[0] | ( a[0] == b[0] & ( a[1] > ++b[1] | ( a[1] == b[1] & ( a[2] > ++b[2] | ( a[2] == b[2] & ( a[3] > ++b[3] | ( a[3] == b[3] & ( a[4] > ++b[4] | ( a[4] == b[4] & ( a[5] > ++b[5] | ( a[5] == b[5] & ( a[6] > ++b[6] | ( a[6] == b[6] & ( a[7] > ++b[7] | ( a[7] == b[7]  ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
    }

    gettimeofday( &end_time, NULL );
    free(a); free(b);
    if (counter == 0 )
      printf("Test completed with an entire counter's cycle.\n");

    return ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
}

double runTestThreeLogical_128B ( uint64_t input_total_iterations, int key_level ) {
    const uint64_t total_iterations = input_total_iterations / 2;
    uint64_t *a = (uint64_t *) calloc( 16, sizeof(uint64_t) );
    uint64_t *b = (uint64_t *) calloc( 16, sizeof(uint64_t) );
    uint64_t counter = 1;

    if ( key_level == 1 )
      memset(b + 8, UCHAR_MAX, sizeof(uint64_t) * 8 );
```

```
    else if ( key_level == 2)
      memset(b, UCHAR_MAX, sizeof(uint64_t) * 16 );

    struct timeval start_time;
    struct timeval end_time;

    gettimeofday( &start_time , NULL );

    for ( uint64_t k = 0; k < total_iterations; k++ ) {
            counter += a[0] > b[0] || ( a[0] == b[0]-- && (
                a[1] > b[1] || ( a[1] == b[1]-- && ( a[2] >
                b[2] || ( a[2] == b[2]-- && ( a[3] > b[3] ||
                 ( a[3] == b[3]-- && ( a[4] > b[4] || ( a[4]
                 == b[4]-- && ( a[5] > b[5] || ( a[5] == b
                [5]-- && ( a[6] > b[6] || ( a[6] == b[6]--
                && ( a[7] > b[7] || ( a[7] == b[7]-- && ( a
                [8] > b[8] || ( a[8] == b[8]-- && ( a[9] > b
                [9] || ( a[9] == b[9]-- && ( a[10] > b[10]
                || ( a[10] == b[10]-- && ( a[11] > b[11] ||
                ( a[11] == b[11]-- && ( a[12] > b[12] || ( a
                [12] == b[12]-- && ( a[13] > b[13] || ( a
                [13] == b[13]-- && ( a[14] > b[14] || ( a
                [14] == b[14]-- && ( a[15] > b[15] || ( a
                [15] == b[15]--  ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
                 ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
      counter += a[0] > ++b[0] || ( a[0] == b[0] && ( a[1] >
           ++b[1] || ( a[1] == b[1] && ( a[2] > ++b[2] || (
          a[2] == b[2] && ( a[3] > ++b[3] || ( a[3] == b[3]
          && ( a[4] > ++b[4] || ( a[4] == b[4] && ( a[5] >
          ++b[5] || ( a[5] == b[5] && ( a[6] > ++b[6] || ( a
          [6] == b[6] && ( a[7] > ++b[7] || ( a[7] == b[7]
          && ( a[8] > ++b[8] || ( a[8] == b[8] && ( a[9] >
          ++b[9] || ( a[9] == b[9] && ( a[10] > ++b[10] || (
           a[10] == b[10] && ( a[11] > ++b[11] || ( a[11] ==
           b[11] && ( a[12] > ++b[12] || ( a[12] == b[12] &&
           ( a[13] > ++b[13] || ( a[13] == b[13] && ( a[14]
          > ++b[14] || ( a[14] == b[14] && ( a[15] > ++b[15]
           || ( a[15] == b[15]  ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
          ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
    }

    gettimeofday( &end_time, NULL );
    free(a); free(b);
    if (counter == 0 )
      printf("Test completed with an entire counter's cycle
          .\n");

    return ( double ) ( ( end_time.tv_sec - start_time.
        tv_sec ) * 1000000 + ( end_time.tv_usec - start_time
        .tv_usec ) ) / 1000000;
    }
```

```
double runTestThreeBitwise_128B ( uint64_t input_total_iterations, int key_level ) {
const uint64_t total_iterations = input_total_iterations / 2;
uint64_t *a = (uint64_t *) calloc( 16, sizeof(uint64_t) );
uint64_t *b = (uint64_t *) calloc( 16, sizeof(uint64_t) );
uint64_t counter = 1;

if ( key_level == 1 )
  memset(b + 8, UCHAR_MAX, sizeof(uint64_t) * 8 );
else if ( key_level == 2)
  memset(b, UCHAR_MAX, sizeof(uint64_t) * 16 );

struct timeval start_time;
struct timeval end_time;

gettimeofday( &start_time, NULL );

for ( uint64_t k = 0; k < total_iterations; k++ ) {
  counter += a[0] > b[0] | ( a[0] == b[0]-- & ( a[1] > b[1] | ( a[1] == b[1]-- & ( a[2] > b[2] | ( a[2] == b[2]-- & ( a[3] > b[3] | ( a[3] == b[3]-- & ( a[4] > b[4] | ( a[4] == b[4]-- & ( a[5] > b[5] | ( a[5] == b[5]-- & ( a[6] > b[6] | ( a[6] == b[6]-- & ( a[7] > b[7] | ( a[7] == b[7]-- & ( a[8] > b[8] | ( a[8] == b[8]-- & ( a[9] > b[9] | ( a[9] == b[9]-- & ( a[10] > b[10] | ( a[10] == b[10]-- & ( a[11] > b[11] | ( a[11] == b[11]-- & ( a[12] > b[12] | ( a[12] == b[12]-- & ( a[13] > b[13] | ( a[13] == b[13]-- & ( a[14] > b[14] | ( a[14] == b[14]-- & ( a[15] > b[15] | ( a[15] == b[15]--  ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
  counter += a[0] > ++b[0] | ( a[0] == b[0] & ( a[1] > ++b[1] | ( a[1] == b[1] & ( a[2] > ++b[2] | ( a[2] == b[2] & ( a[3] > ++b[3] | ( a[3] == b[3] & ( a[4] > ++b[4] | ( a[4] == b[4] & ( a[5] > ++b[5] | ( a[5] == b[5] & ( a[6] > ++b[6] | ( a[6] == b[6] & ( a[7] > ++b[7] | ( a[7] == b[7] & ( a[8] > ++b[8] | ( a[8] == b[8] & ( a[9] > ++b[9] | ( a[9] == b[9] & ( a[10] > ++b[10] | ( a[10] == b[10] & ( a[11] > ++b[11] | ( a[11] == b[11] & ( a[12] > ++b[12] | ( a[12] == b[12] & ( a[13] > ++b[13] | ( a[13] == b[13] & ( a[14] > ++b[14] | ( a[14] == b[14] & ( a[15] > ++b[15] | ( a[15] == b[15]  ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) );
}

gettimeofday( &end_time, NULL );
```

```
free(a); free(b);
if (counter == 0 )
  printf("Test completed with an entire counter's cycle.\n");

return ( double ) ( ( end_time.tv_sec − start_time.tv_sec ) * 1000000 + ( end_time.tv_usec − start_time.tv_usec ) ) / 1000000;
}


void printBooleanExpression(int num_bytes, int current_index) {
int max_index = num_bytes − 1;
if ( current_index <= max_index )
{
  printf("a_%dB[%d] > b_%dB[%d] || ( a_%dB[%d] == b_%dB[%d] ", num_bytes, current_index, num_bytes, current_index, num_bytes, current_index, num_bytes, current_index);
  if ( current_index < max_index )
    printf("&& ( ");
  printBooleanExpression(num_bytes, current_index + 1);
  if ( current_index != max_index )
    printf(" ) )");
  else
    printf(" )");
  if ( current_index == 0 )
    printf(";");
}
}

void printUnsignedInt64Arrays(int total_elements, uint64_t *array1, uint64_t *array2 ) {
printf( "\nArray 1 = " );
for ( int i = 0; i < total_elements; i++ )
  printf( "%ju ", array1[i] );
printf( "\nArray 2 = " );
for ( int i = 0; i < total_elements; i++ )
  printf( "%ju ", array2[i] );
printf( "\n" );
}
```

A.2 MAIN MEMORY AND CACHE TESTS SOURCE CODE

Listing 3: Extract from memory and cache tests source code

```
#include "pre-tuner_mem.h"

int main( int argc, char** argv ) {

using namespace std;

printf("\n### Starting MEM TESTS ###\n");

createOutputFiles();

if ( RUN_MEM_READ_TEST ) {

  printf("\nStarting test 1 (memory reads). It will allocate and array of size %lu MiB and it will\nperform %lu reads of %lu B each for a total of %lu MiB.\n",
      (long unsigned) (SIZE) / (1024*1024), (long unsigned) NUM_ACCESSES, (long unsigned) sizeof(elem_t), (long unsigned) ((NUM_ACCESSES) * sizeof( elem_t ) / (1024*1024)) );

  runFoolOperations(FOOL_OPERATIONS_AMOUNT_MB);
  double bandwidth = runMemoryReadTest();
  printf("\nRead bandwith: %0.02f MiB/s\n", bandwidth );

  //cut: save results code here
}


if ( RUN_MEM_WRITE_TEST )  {

  printf("\nStarting test 2 (memory writes). It will allocate and array of size %lu MiB and it will\nperform %lu writes of %lu B each for a total of %lu MiB.\n",
      (long unsigned) (SIZE) / (1024*1024), (long unsigned) NUM_ACCESSES, (long unsigned) sizeof(elem_t), (long unsigned) ((NUM_ACCESSES) * sizeof( elem_t ) / (1024*1024)) );

  runFoolOperations(FOOL_OPERATIONS_AMOUNT_MB);
  double bandwidth = runMemoryWriteTest();
  printf("\nWrite bandwith: %0.02f MiB/s\n", bandwidth );

  //cut: save results code here
```

```

  }

  if ( RUN_CACHE_SIZE_TEST )  {

    printf("\nStarting test 3 (cache size).\n\n");

    #ifdef _SC_LEVEL2_CACHE_SIZE
    long sysconf_L2_size = sysconf(_SC_LEVEL2_CACHE_SIZE);
    if ( sysconf_L2_size != -1 ) {
      printf("L2 Cache size using sysconf(): %lu KiB\n", sysconf_L2_size / 1024);
      //cut: save results code here
    }
    #endif

    #ifdef _SC_LEVEL3_CACHE_SIZE
    long sysconf_L3_size = sysconf(_SC_LEVEL3_CACHE_SIZE);
    if ( sysconf_L3_size != -1 ) {
      printf("L3 Cache size using sysconf(): %lu KiB\n", sysconf_L3_size / 1024);
      //cut: save results code here
    }
    #endif

    int proc_cache_size = getCacheSizeFromProc_kb();
    if ( proc_cache_size != -1 ) {
      printf("Cache size using cpuinfo: %d KiB\n\n", proc_cache_size);
      //cut: save results code here
    }
    else
      printf("\n");

    /* This array contains bandwidth values with input from 12 KiB (index 0) to 49152 KiB (index 12).
      Element at index i has input value which is the double of element at index i-1.
      We'll find the two largest bandwith variations and estimate cache size as the
      two-power between two contiguous-element of this array. */

    double bandwidths[13];
    int counter = 0;

    for ( uint64_t size = 8 * (1<<10); size <=  32 * (1<<20); size *= 2) {

      runFoolOperations(FOOL_OPERATIONS_AMOUNT_MB);
      double bandwidth = runCacheSizeTest(size);
```

```
        printf("Read bandwith with input of %ju KiB: %0.00f
            MiB/s\n", size / (1<<10), bandwidth );
        //cut: save results code here

        runFoolOperations(FOOL_OPERATIONS_AMOUNT_MB);
        bandwidth = runCacheSizeTest(size + size / 2);
        //cut: save results code here
        bandwidths[counter++] = bandwidth;
      }

      int caches[2];
      estimateCacheSize(bandwidths, caches);

      printf("\nFirst estimated cache size: %d KiB\nSecond
          estimated cache size: %d KiB\n", caches[1], caches
          [0]);
      //cut: save results code here

      /* Actually psort needs the size of the cache nearest
          to 512 KiB */
      char string[128];
      if ( abs((512 - caches[0])) < abs((512 - caches[1])) )
        sprintf(string, "cache-size: %d", caches[0]);
      else
        sprintf(string, "cache-size: %d", caches[1]);
       writeStringToFile("psortvalues", string);

    }
    printf("\n");
    }

    double runMemoryReadTest() {

    uint64_t checksum = 0;
    uint64_t v_pos = 0;

    struct timeval start_time;
    struct timeval end_time;

    /* allocate the vector */
    elem_t *v;
    const uint64_t n_elem = ( (SIZE) / sizeof( elem_t ) );
    if ( posix_memalign( ( void** ) &v, sizeof( elem_t ) ,
        sizeof( elem_t ) * n_elem ) != 0 ) {
      perror("Cannot allocate array");
      exit(2);
    }

    /* fill the vector */
    for ( uint64_t i = 0; i < n_elem; i++ ) {
      for ( uint64_t j = 0; j < STRUCT_SIZE; j++ )
        v[ i ].content[ j ] = i + j + 1;
```

```
}

gettimeofday( &start_time, NULL );

for ( uint64_t i = 0; i < NUM_ACCESSES; i += 1 ) {
  for ( uint64_t k = 0; k < STRUCT_SIZE; k++ )
    checksum += v[ v_pos ].content[ k ];
  v_pos = ( v_pos + STEP ) % n_elem;
}

gettimeofday( &end_time, NULL );

double time = ( double ) ( ( end_time.tv_sec -
    start_time.tv_sec ) * 1000000 + ( end_time.tv_usec -
     start_time.tv_usec ) ) / 1000000;
double band = ( ( ( ( NUM_ACCESSES ) * sizeof( elem_t )
    ) / time ) ) / (1024*1024); // in MiB/s

free(v);

if ( checksum == 0 ) // just try to avoid compiler's
    trick forcing it to use counter
  printf(" Checksum: %d", (int) checksum );

return band;
}

double runMemoryWriteTest() {

uint64_t checksum = 0;
uint64_t v_pos = 0;

struct timeval start_time;
struct timeval end_time;

/* allocate the vector */
elem_t *v;
const uint64_t n_elem = ( (SIZE) / sizeof( elem_t ) );
if ( posix_memalign( ( void** ) &v, sizeof( elem_t ) ,
    sizeof( elem_t ) * n_elem ) != 0 ) {
  perror("Cannot allocate array");
  exit(2);
}

/* fill the vector */
for ( uint64_t i = 0; i < n_elem; i++ ) {
  for ( uint64_t j = 0; j < STRUCT_SIZE; j++ )
    v[ i ].content[ j ] = i + j + 1;
}

gettimeofday( &start_time, NULL );

```

```
for ( uint64_t i = 0; i < NUM_ACCESSES; i++ ) {
  for ( uint64_t k = 0; k < STRUCT_SIZE; k+= 1 )
    v[ v_pos ].content[ k ] = i;
  v_pos = ( v_pos + STEP ) % n_elem;
}

gettimeofday( &end_time, NULL );

double time = ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000;
double band = ( ( ( ( NUM_ACCESSES ) * sizeof( elem_t ) ) / time ) ) / (1024*1024); // in MiB/s

checksum = v[ v_pos ].content[ v_pos & ( STRUCT_SIZE - 1 ) ];

free(v);

if ( checksum == 0 ) // just try to avoid compiler's trick forcing it to use counter
  printf(" Checksum: %d", (int) checksum );

return band;
}

double runCacheSizeTest(uint64_t size) {

uint64_t checksum = 0;
uint64_t v_pos = 0;

struct timeval start_time;
struct timeval end_time;

// allocate the vector
elem_t *v;
const uint64_t n_elem = ( size / sizeof( elem_t ) );
if ( posix_memalign( ( void** ) &v, sizeof( elem_t ) , sizeof( elem_t ) * n_elem ) != 0 ) {
  perror("Cannot allocate array");
  exit(2);
}

// fill the vector
for ( uint64_t i = 0; i < n_elem; i++ ) {
  for ( uint64_t j = 0; j < STRUCT_SIZE; j++ )
    v[ i ].content[ j ] = i + j + 1;
}

gettimeofday( &start_time, NULL );

for ( uint64_t i = 0; i < NUM_ACCESSES; i += 1 ) {
```

```
    for ( uint64_t k = 0; k < STRUCT_SIZE; k++ )
      checksum += v[ v_pos ].content[ k ];
    v_pos = ( v_pos + STEP ) % n_elem;
  }

  gettimeofday( &end_time, NULL );

  double time = ( double ) ( ( end_time.tv_sec -
      start_time.tv_sec ) * 1000000 + ( end_time.tv_usec -
       start_time.tv_usec ) ) / 1000000;
  double band = ( ( ( ( NUM_ACCESSES ) * sizeof( elem_t )
      ) / time ) ) / (1024*1024); // in MiB/s

  free(v);

  if ( checksum == 0 ) // just try to avoid compiler's
      trick forcing it to use counter
    printf(" Checksum: %d", (int) checksum );

  return band;
  }

  int getCacheSizeFromProc_kb()
  {
   char line[512], buffer[32];
   size_t column;
   FILE *cpuinfo;

   if (!(cpuinfo = fopen("/proc/cpuinfo", "r"))) {
      perror("/proc/cpuinfo: fopen");
      return -1;
   }

   while (fgets(line, sizeof(line), cpuinfo)) {
      if (strstr(line, "cache size")) {
        column = strcspn(line, ":");
        strncpy(buffer, line + column + 1, sizeof(buffer))
            ;
        fclose(cpuinfo);
        return (int)strtol(buffer, NULL, 10);
      }
   }
   fclose(cpuinfo);
   return -1;
  }

  void estimateCacheSize(const double *bandwidths, int*
      caches) {

  /* Element at index i contains variation between
      bandwidths at index i and i+1. */
  double variation[12] = {0};
```

```

  for ( int i = 0; i < 13; i++ ) {
    variation[i] = ( bandwidths[i] - bandwidths[i+1] ) / bandwidths[i];
    if ( variation[i] < 0 ) variation[i] *= (-1);
  }

  int maxIndex = 0;
  int secondHigherIndex = 0;
  double currentMax = 0;
  double currentSecondHigher = 0;

  /* Debug
  printf("\n");
  for ( int i = 0; i < 13; i++ ) {
    printf("bandwidths[%d]: %f\n", i, bandwidths[i]);
  }
  printf("\n");

  printf("\n");
  for ( int i = 0; i < 12; i++ ) {
    printf("variation[%d]: %f\n", i, variation[i]);
  }
  printf("\n");*/

  for ( int i = 0; i < 12; i++ ) {
    if ( variation[i] > currentMax ) {
      maxIndex = i;
      currentMax = variation[i];
    }
  }
  for ( int i = 0; i < 12; i++ ) {
    if ( variation[i] < currentMax && variation[i] > currentSecondHigher && abs( maxIndex - i ) > 2 ) {
      secondHigherIndex = i;
      currentSecondHigher = variation[i];
    }
  }

  caches[0] = 1 << (maxIndex + 4); // 4 as offset because first element is 12 KiB.
  caches[1] = 1 << (secondHigherIndex + 4);
  }

```

A.3 DISK TESTS SOURCE CODE

Listing 4: Extract from disks tests source code

```
#include "pre-tuner_disks.h"

int main( int argc, const char *argv[] ) {

printf("\n### Starting DISKS TESTS ###\n");

createOutputFiles();

if ( RUN_SEQUENTIAL_RW_TEST ) {
  char *readFromFileName = new char[2048];
  strcpy(readFromFileName, quotes(SEQ_INPUT_FILE) );
  char *writeToFileName = (char *) malloc( strlen( readFromFileName ) + 4);
  strcpy(writeToFileName, readFromFileName);
  strcat(writeToFileName, "_w");

  printf("\nFirst Test: sequential write to %s and read from %s. Max filesize is %d MiB.\n\n", writeToFileName, readFromFileName, SEQ_MAX_BLOCKSIZE_MB);

  seqWriteToDisk( readFromFileName, SEQ_MAX_BLOCKSIZE_MB * 1024 * 1024 ); // To be read

  double prevSeqWriteBandwidth = 0;
  double prevSeqReadBandwidth = 0;

  for (size_t k = SEQ_MIN_BLOCKSIZE_B; k <= (size_t) SEQ_MAX_BLOCKSIZE_MB * 1024 * 1024; k *= SEQ_STEP_INCREMENT ) {

    double seqWriteBandwidth = 0;
    double seqReadBandwidth = 0;

    for (int i = 0; i < SEQ_NUM_TEST; i++ ) {
      seqWriteBandwidth += seqWriteToDisk( writeToFileName, k );
      seqReadFromDisk( writeToFileName, k );
      seqReadBandwidth += seqReadFromDisk( readFromFileName, k );
    }

    seqWriteBandwidth = seqWriteBandwidth / SEQ_NUM_TEST;
    seqReadBandwidth = seqReadBandwidth / SEQ_NUM_TEST;

```

```
    printf("Output size: %.03f MiB. Sequential write
        bandwidth: %.02f MiB/s\n", ((double) k ) /
        (1024*1024) , seqWriteBandwidth);
    printf("Input size:  %.03f MiB. Sequential read
        bandwidth:  %.02f MiB/s\n", ((double) k ) /
        (1024*1024) , seqReadBandwidth);

    // We need to know when the threshold of bandwidth
        is small enough to stop the test
    if ( prevSeqWriteBandwidth > 0 &&
        prevSeqReadBandwidth > 0 ) {
      if ( ABS(1 - seqWriteBandwidth /
          prevSeqWriteBandwidth) <=
          SEQ_PERCENT_THRESHOLD ) {
        if ( ABS(1 - seqReadBandwidth /
            prevSeqReadBandwidth) <=
            SEQ_PERCENT_THRESHOLD ) {
          printf("\nWrite delta is: %f\n", ABS(1 -
              seqWriteBandwidth / prevSeqWriteBandwidth)
               );
          printf("Read delta is:  %f\n", ABS(1 -
              seqReadBandwidth / prevSeqReadBandwidth) )
              ;
          //cut: save results code here
          break;
        }
      }
    }
    prevSeqWriteBandwidth = seqWriteBandwidth;
    prevSeqReadBandwidth = seqReadBandwidth;

  }

  if( remove( writeToFileName ) != 0 )
    perror( "Error deleting tmp writing file" );
  if( remove( readFromFileName ) != 0 )
    perror( "Error deleting tmp reading file" );

}

if ( RUN_RANDOM_READ_TEST  ) {

  printf("\nSecond Test: random read from device %s of %
      d KiB (in blocks of %d B).\n\n", RANDOM_DEVICE,
      RANDOM_SEQ_BLOCKSIZE_KB,
      RANDOM_MINISEQ_BLOCKSIZE_B);

  double randomReadAccessTime = randomReadFromDisk(
      RANDOM_DEVICE, RANDOM_SEQ_BLOCKSIZE_KB);
  printf("Random read access time: %.02f ms\n",
      randomReadAccessTime );

```

```
  writeRecordToFile("human", "disks", 3, "Random read bandwidth", randomReadAccessTime, -1, -1);
  writeRecordToFile("machine", "2", 3, "1", randomReadAccessTime, -1, -1);
}

printf("\n");

return 0;
}

double seqWriteToDisk(const char *pathname, size_t blocksize) {
struct timeval start_time;
struct timeval end_time;

void *buffer;
if ( posix_memalign(&buffer, blocksize, blocksize) != 0 ) {
  perror("Cannot allocate buffer");
  exit(2);
}
int file = open(pathname, O_CREAT|O_TRUNC|O_WRONLY|O_DIRECT, S_IRWXU);

checkFileForErrors( file );

gettimeofday( &start_time, NULL );
int check = write(file, buffer, blocksize);
gettimeofday( &end_time, NULL );

if ( check == -1 ) {
  perror("Error writing data");
  exit(3);
}

checkFileForErrors ( close(file) );
free(buffer);

double total_bandwidth = ( (double) blocksize / (1<<20) ) / ( ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000 );
// printf("Partial seq write: %.02f MiB/s\n", total_bandwidth);
return total_bandwidth;
}

double seqReadFromDisk(const char *pathname, size_t blocksize) {

struct timeval start_time;
```

```
struct timeval end_time;

void *buffer;
if ( posix_memalign(&buffer, blocksize, blocksize) != 0 ) {
  perror("Cannot allocate buffer");
  exit(2);
}
int file = open(pathname, O_RDONLY|O_DIRECT, S_IRWXU);

checkFileForErrors( file );

gettimeofday( &start_time, NULL );
int check = read(file, buffer, blocksize);
gettimeofday( &end_time, NULL );

if ( check == -1 ) {
  perror("Error reading data");
  exit(3);
}

checkFileForErrors ( close(file) );
free(buffer);

double total_bandwidth = ( (double) blocksize / (1<<20) ) / ( ( double ) ( ( end_time.tv_sec - start_time.tv_sec ) * 1000000 + ( end_time.tv_usec - start_time.tv_usec ) ) / 1000000 );
// printf("Partial seq read: %.02f MiB/s\n", total_bandwidth);
return total_bandwidth;
}


double randomReadFromDisk(const char *disk, size_t blocksize_kb) {

struct timeval start_time;
struct timeval end_time;
const size_t miniblocksize = RANDOM_MINISEQ_BLOCKSIZE_B;
unsigned long iterations = ( ((unsigned long) blocksize_kb) * 1024 ) / miniblocksize;

char *buffer = new char[miniblocksize];
unsigned long numberOfBlocks;
off64_t offset;

int file = open( disk, O_RDONLY );
checkFileForErrors( file );

if ( ioctl(file, BLKGETSIZE, &numberOfBlocks) == -1 ) {
  perror("Cannot get total block number from the disk");
```

```
    return -1;
  }

  unsigned int seed = (unsigned int) time(NULL);
  srand(seed);

  gettimeofday( &start_time, NULL );

  for (int i = 0; i < iterations; i++) {
    offset = (off64_t) numberOfBlocks * random() /
        RAND_MAX;
    if ( (int) lseek64(file, miniblocksize * offset,
        SEEK_SET) == -1 ) {
      perror("Cannot locate next block");
      return -1;
    }
    if ( read(file, buffer, miniblocksize) < 0 ) {
      perror("Cannot read data from disk");
      return -1;
    }
  }

  gettimeofday( &end_time, NULL );
  free(buffer);
  double total_time = ( double ) ( ( end_time.tv_sec -
      start_time.tv_sec ) * 1000000 + ( end_time.tv_usec -
       start_time.tv_usec ) ) / 1000000;
  return (double) total_time / iterations * 1000;

  }


  void checkFileForErrors(int file) {
  if ( file == -1 ) {
    perror("Error with test file. Please check dir
        permissions");
    exit(1);
  }
  }
```

B EXECUTION LOG

This appendix contains the execution log of the entire *psort* tuning package installer starting from the hardware detection and ending with a test execution of *psort*. The log has been recorded on an *Intel Core i7 920* (cache: L1 *32 KiB*, L2 *256 KiB*, L3 *8192 KiB* shared) with *6 GiB* of RAM and a single *7200* RPM low-end disk. The package performs, in order:

1. Estimation of hardware parameters.
2. Code tuning.
3. *psort* compiling.
4. Installation test.

Listing 5: Execution log on Intel Core i7 920

```
----------- PSORT INSTALLER -----------

Using extreme pre-tuner level.
All TESTS will be performed (cpu, disks, mem).
Code-tuning level is 2 (medium).
Using different keys optimization.

--- STARTING HARDWARE DETECTION PROCESS

Compiling pre-tuner files...

g++ -O3 -funroll-loops -funsafe-loop-optimizations -
    march=native -mtune=native -c -DOUTPUT_PATH=../ -
    DPSORT_PATH=../psort/ misc/pre-tuner_functions.cpp -
    o misc/pre-tuner_functions.o
g++ -O3 -funroll-loops -funsafe-loop-optimizations -
    march=native -mtune=native -DCPU_PRETUNING_LEVEL=2
    cpu-test/pre-tuner_cpu.cpp misc/pre-tuner_functions.
    cpp -o cpu-test/pre-tuner_cpu
g++ -O3 -funroll-loops -funsafe-loop-optimizations -
    march=native -mtune=native -DDISKS_PRETUNING_LEVEL=2
     -DSEQ_INPUT_FILE=/tmp/tmp.data disks-test/pre-
    tuner_disks.cpp misc/pre-tuner_functions.cpp -o
    disks-test/pre-tuner_disks
g++ -O3 -funroll-loops -funsafe-loop-optimizations -
    march=native -mtune=native -DMEM_PRETUNING_LEVEL=2
    mem-test/pre-tuner_mem.cpp misc/pre-tuner_functions.
    cpp -o mem-test/pre-tuner_mem
```

```

### Starting CPU TESTS ###

Starting test 1 (pointers). It will be repeated 8 time(s
    ). Every test works with an array of size 4096 MiB.

Subscript notation total time: 12.889888
Offset notation total time: 12.897998

Starting test 2 (branch-merge). It will be repeated 8
    time(s). Every test performs 3000000000 comparisons.

If-else approach total time: 18.111178
Boolean approach total time: 15.395987

Starting test 3 (logical-bitwise). Every test performs
    300000000 comparisons.

4 Byte (equal key) Logical time: 0.283646
4 Byte (equal key) Bitwise time: 0.014774
8 Byte (equal key) Logical time: 0.283638
8 Byte (equal key) Bitwise time: 0.014776
16 Byte (equal key) Logical time: 0.532310
16 Byte (equal key) Bitwise time: 0.787882
32 Byte (equal key) Logical time: 0.925768
32 Byte (equal key) Bitwise time: 1.812099
64 Byte (equal key) Logical time: 1.536332
64 Byte (equal key) Bitwise time: 2.314368
128 Byte (equal key) Logical time: 3.112589
128 Byte (equal key) Bitwise time: 4.293876

4 Byte (half equal key) Logical time: 0.323030
4 Byte (half equal key) Bitwise time: 0.014774
8 Byte (half equal key) Logical time: 0.323070
8 Byte (half equal key) Bitwise time: 0.014774
16 Byte (half equal key) Logical time: 0.531815
16 Byte (half equal key) Bitwise time: 0.787880
32 Byte (half equal key) Logical time: 0.679554
32 Byte (half equal key) Bitwise time: 1.812540
64 Byte (half equal key) Logical time: 0.984845
64 Byte (half equal key) Bitwise time: 2.314361
128 Byte (half equal key) Logical time: 1.890885
128 Byte (half equal key) Bitwise time: 4.294293

4 Byte (total different key) Logical time: 0.323025
4 Byte (total different key) Bitwise time: 0.014774
8 Byte (total different key) Logical time: 0.323038
8 Byte (total different key) Bitwise time: 0.014810
16 Byte (total different key) Logical time: 0.226513
16 Byte (total different key) Bitwise time: 0.787860
32 Byte (total different key) Logical time: 0.216670
32 Byte (total different key) Bitwise time: 1.812091
```

```
64 Byte (total different key) Logical time: 0.236359
64 Byte (total different key) Bitwise time: 2.314354
128 Byte (total different key) Logical time: 0.196980
128 Byte (total different key) Bitwise time: 4.294296


### Starting DISKS TESTS ###

First Test: sequential write to /tmp/tmp.data_w and read
     from /tmp/tmp.data. Max filesize is 1024 MiB.

Output size: 8.000 MiB. Sequential write bandwidth:
    72.31 MiB/s
Input size:  8.000 MiB. Sequential read bandwidth:
    69.15 MiB/s
Output size: 16.000 MiB. Sequential write bandwidth:
    84.91 MiB/s
Input size:  16.000 MiB. Sequential read bandwidth:
    66.35 MiB/s
Output size: 32.000 MiB. Sequential write bandwidth:
    75.54 MiB/s
Input size:  32.000 MiB. Sequential read bandwidth:
    77.60 MiB/s
Output size: 64.000 MiB. Sequential write bandwidth:
    77.89 MiB/s
Input size:  64.000 MiB. Sequential read bandwidth:
    75.30 MiB/s

Write delta is: 0.031083
Read delta is:  0.029666

Second Test: random read from device /dev/sdc of 1024
    KiB (in blocks of 512 B).

Random read access time: 12.71 ms (nominal seek time 8.5
     ms)


### Starting MEM TESTS ###

Starting test 1 (memory reads). It will allocate and
    array of size 256 MiB and it will
perform 268435456 reads of 256 B each for a total of
    65536 MiB.

Read bandwith: 6430.53 MiB/s

Starting test 2 (memory writes). It will allocate and
    array of size 256 MiB and it will
perform 268435456 writes of 256 B each for a total of
    65536 MiB.

```

```
Write bandwith: 7803.61 MiB/s

Starting test 3 (cache size).

L2 Cache size using sysconf(): 256 KiB
L3 Cache size using sysconf(): 8192 KiB
Cache size using cpuinfo: 8192 KiB

Read bandwith with input of 8 KiB: 20087 MiB/s
Read bandwith with input of 12 KiB: 20095 MiB/s
Read bandwith with input of 16 KiB: 20155 MiB/s
Read bandwith with input of 24 KiB: 20119 MiB/s
Read bandwith with input of 32 KiB: 20244 MiB/s
Read bandwith with input of 48 KiB: 19630 MiB/s
Read bandwith with input of 64 KiB: 19622 MiB/s
Read bandwith with input of 96 KiB: 19595 MiB/s
Read bandwith with input of 128 KiB: 18169 MiB/s
Read bandwith with input of 192 KiB: 19587 MiB/s
Read bandwith with input of 256 KiB: 17972 MiB/s
Read bandwith with input of 384 KiB: 16415 MiB/s
Read bandwith with input of 512 KiB: 15968 MiB/s
Read bandwith with input of 768 KiB: 15920 MiB/s
Read bandwith with input of 1024 KiB: 15865 MiB/s
Read bandwith with input of 1536 KiB: 15851 MiB/s
Read bandwith with input of 2048 KiB: 15846 MiB/s
Read bandwith with input of 3072 KiB: 14045 MiB/s
Read bandwith with input of 4096 KiB: 12138 MiB/s
Read bandwith with input of 6144 KiB: 10188 MiB/s
Read bandwith with input of 8192 KiB: 7590 MiB/s
Read bandwith with input of 12288 KiB: 6025 MiB/s
Read bandwith with input of 16384 KiB: 5973 MiB/s
Read bandwith with input of 24576 KiB: 5941 MiB/s
Read bandwith with input of 32768 KiB: 5914 MiB/s
Read bandwith with input of 49152 KiB: 5851 MiB/s

First estimated cache size: 256 KiB
Second estimated cache size: 8192 KiB


--- STARTING CODE-TUNING PROCESS

Pre-tuning exectued.
*** inlines.tun ***
*** Translating the tuning file to a C++ source code ***

*** cache_sorters.tun ***
*** Translating the tuning file to a C++ source code ***
*** Compiling the extended source code ***
*** Executing test and evaluating the best options ***
*** Generating the optimal source code ***
The details has been saved in tuningLog_cache_sorters.
    txt
```

```


--- STARTING PSORT INSTALLATION

-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/
    Documents/PSORT-TUNED-PACKAGE/psort
Scanning dependencies of target libpsort
[18%] Building CXX object CMakeFiles/libpsort.dir/
    functions.cpp.o
[27%] Building CXX object CMakeFiles/libpsort.dir/
    kmerger.cpp.o
[36%] Building CXX object CMakeFiles/libpsort.dir/
    stage_one.cpp.o
[45%] Building CXX object CMakeFiles/libpsort.dir/
    stage_two.cpp.o
Linking CXX static library libpsort.a
[63%] Built target libpsort
Linking CXX executable psort
[72%] Built target psort
Scanning dependencies of target checksort
[81%] Building CXX object tools/CMakeFiles/checksort.dir
    /checksort.cpp.o
Linking CXX executable checksort
[81%] Built target checksort
[90%] Built target generator
[100%] Built target psortInfoParser


--- TESTING PSORT INSTALLATION

Generating 1048576 sort test data records to file ../../
    test-files/test-input.txt
Completed writing 1048576 Records to file ../../test-
    files/test-input.txt

--- RUNNING PSORT

psort - yet another fast external sorter
-- Stage 1 --
input:            1048576 x (8,128,0) = 134217728 bytes
    in 1 runs
block size:       16384 records ( 262144 bytes )
allocated blocks: 64
heap merger:      64 ways
I/O buffer size:  read: 540672 recs (69206016 bytes);
    write: 540672 recs (69206016 bytes)
memory used:      419430656
writing run 0 (1048576 records)
done.

```

```
--- CHECKING SORTED FILE

verbose level:   1
record length:   128
key length:      8
key offset:      0
tot records:     1048576
sort order:      1
buffer size:     85852160
E037B94C

--- END OF INSTALLATION PROCESS

```

BIBLIOGRAPHY

[1] CMake home page. `http://www.cmake.org/`, Available: September 2011.

[2] Codeworker home page. `http://www.codeworker.org/`, Available: September 2011.

[3] Matlab® home page. `http://www.mathworks.com/products/matlab/`, Available: September 2011.

[4] Gnu Octave home page. `http://www.gnu.org/software/octave/`, Available: September 2011.

[5] Wikipedia home page. `http://www.wikipedia.org`, Available: September 2011.

[6] Sort Benchmark home page. `http://sortbenchmark.org/`, Available: September 2011.

[7] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 305–314, 1987.

[8] P. Bertasi, M. Bressan, and E. Peserico. psort, yet another fast stable external sorting software. *Proceedings of the 8th Symposium on Experimental Algorithms*, 2009.

[9] D. Burger, Goodman J. R., and G. S. Sohi. *Memory systems in The Computer Science and Engineering Handbook*. CRC Press, 1997.

[10] G. Di Liberto. psort: automated code tuning. *Thesis paper - University of Padua*, July 2011.

[11] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Addison-Wesley Professional, 2th edition, 2006.

[12] D. E. Knuth. *Art of Computer Programming. Vol. 3: Sorting and searching*. Addison-Wesley Professional, 2th edition, 1998.

[13] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A Cache-Sensitive Parallel External Sort. *VLDB Journal 4*, pages 603–627, 1995.

[14] L. Torvald. Re: O_direct question. *Email*, January 2007.

[15] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Potomac (MD), 1983.