UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN

INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA

PSORT: AUTOMATED CODE TUNING

Advisor: Prof. Enoch Peserico Stecchini Negri De Salvi

Co-Advisor: Dott. Marco Bressan

Candidate: Giovanni Di Liberto

ANNO ACCADEMICO 2010 - 2011

ABSTRACT

This thesis describes the design and implementation of an automated code tuner for *psort*, a fast sorting library for large datasets. Our work, motivated by the necessity of guaranteeing a high performance while keeping a low cost on the end user, provides a reusable and portable framework that can be easily extended to automatically tune virtually every portion of the source code, including code that has not yet been written. Experiments show that our system produces code which is significantly faster than original code, suggesting that *psort* should include it among its tools.

SOMMARIO

Questa tesi descrive la progettazione e la realizzazione di un ottimizzatore di codice automatico per *psort*, una libreria di ordinamento veloce per grandi moli di dati. Il nostro lavoro, motivato dalla necessità di garantire alte prestazioni mantenendo un basso costo sull'utente finale, fornisce una infrastruttura riusabile e portabile che può essere facilmente estesa per ottimizzare in maniera automatica virtualmente ogni porzione di codice sorgente, incluso codice che ancora non è stato scritto. Gli esperimenti mostrano che il nostro sistema produce codice che è significativamente più veloce del codice originale, suggerendo che *psort* dovrebbe includerlo tra i suoi strumenti.

Contents

Chapter 1

Introduction

This thesis describes the design and implementation of an automatic code generator for *psort*, a fast, stable *external sorting* software. This section provides a gentle introduction to the problem of *external sorting* (Section 1.1) before describing the *psort* library (Section 1.2) and its limiting factors (Section 1.3) which are partially addressed by this work.

1.1 External sorting

Sorting is one of the best known and most important problems in computer science, arising in virtually every application. Indeed, some estimates suggest that few decades ago the global amount of time spent in sorting amounted to approximately 25% of the global CPU time (see [4]); and, while this amount has probably decreased today, sorting still plays a crucial role. Due to its primary importance, it has also been adopted as the core task by benchmarks such as the Sort Benchmark (see [8]) to measure the evolution of computing power over time.

Perhaps the best known scenario for the sorting problem is the RAM model – where accessing a datum has always the same cost, independently of the address where it is stored. However, this is not generally true, and in some cases is very inaccurate. This is the case of *external sorting*, where the size of the *main memory* (which is assumed to be unlimited in the above mentioned RAM model) is not sufficient to hold the data at every step of the algorithm, which must then resort to a larger *external memory* of higher access time cost. This happens, for example, when sorting large amounts of data (which is becoming more the rule than the exception) from disk to disk, and in fact this is a main requirement of the Sort Benchmark competition [8]. In all these cases, the *hierarchical memory model* [5] captures the basic essence of the typical structure of modern machines, which consists of several layers of progressively faster (but more expensive and thus smaller) memory. This *memory hierarchy* is exploited by the typical spatial and temporal locality exhibited by real programs, which ideally perform as many accesses as possible at the highest levels and as few as possible at the lowest ones.

In many practical cases, and especially in the case of desktop-class machines, the memory hierarchy often boils down to three main levels: the CPU (L2) cache, the main memory, and the external storage. Given the typical large gap between the main memory and the external storage (usually represented by mechanical hard disks) in terms of data transfer rate, it is crucial to provide an efficient access to the latter. However, depending on the structure of the input (more precisely, on the ratio between the length of record payloads to the length of record keys) efficient access to the main memory may become at least as crucial as efficient access to external storage. It is thus important to optimize access at every memory level in order to squeeze out a consistent fraction of the computing power of the machine.

1.2 *psort*

psort is a sorting library designed for large amount of data. According to the PennySort benchmark, *psort* was the fastest desktop-class external sorting library from 2008 to 2011[7, 8]. The input files are viewed as sequences of records of arbitrary size; these are sorted, according to an arbitrary infix (the key), by first obtaining individual "*runs*" of sorted data approximately sized as the main memory and then merging those runs into a single sorted file.

Towards the end of 2010 *psort* had dozens of parameters that needed to be chosen carefully in order to achieve high performance, and this amount is still increasing with the newer optimization for *multi-core* and *multi-disks*. So a manual configuration takes too long, and a user may not be inclined to afford this cost. Even a good programmer might not have in-depth knowledge of the hardware architecture underlying his programs, and knowing how a software will be executed is probably increasingly difficult in modern systems, especially those which are highly parallel [6]. Perhaps an *automated code tuning* of the source code could resolve this problem.

psort uses the methodology described in Section 1.1, in fact it divides the work into two stages. The first creates many sorted runs and the second merges those in a single output. There are many important aspects for each stage. In particular this thesis is focused on the low-level sorting routines in the file `cache_sorters.cpp`, that are the core of the first phase. The reasons are that *stage one* is the most complex and that it represents a serious bottleneck at the highest levels of the memory hierarchy.

1.2.1 Stage one

The stage one of *psort*, like shown in Figure 1.1, essentially involves reading from an input file sequences of data (runs) whose size can be chosen by the user; a data run can't be larger than the main memory. Then *psort* execute the sorting of the runs, saving these in the secondary memory. Finally stage two merges the sorted sequences into the output file, in some cases executing more additional merge steps. For I/O efficiency, *psort* allows the use of direct asynchronous I/O to minimise the cost of transferring data between the disks and a set of userspace buffers. A simple double-buffering technique ensures that the disks never fall idle: while the CPU reads/writes data from/to a buffer, the I/O subsystem reads/writes data from/to another buffer, thus completely overlapping I/O with computation. Unsurprisingly then, the number and size of the I/O buffers are crucial parameters that need to be set carefully.

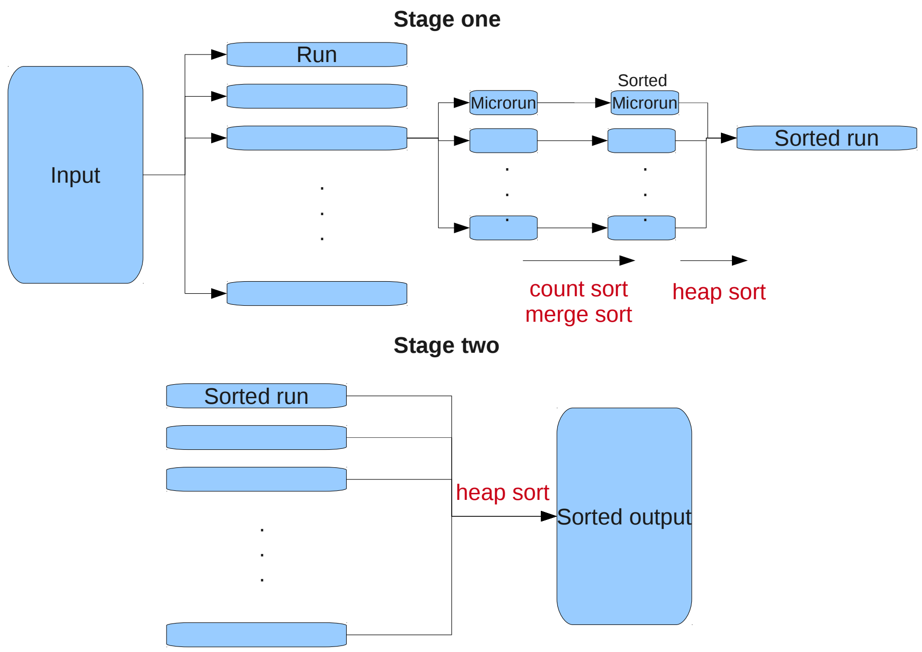


Figure 1.1: Data-flow representing the high-level execution of psort (data on cyan background, routines in red)

As described in Section 1.1, *psort* exploits data locality (and thus the hierarchical structure of the underlying machine) by splitting each run in smaller data blocks, called "*microruns*", of size roughly equivalent to the L2 cache; so the first operation is to sort the microruns and, after that, these will be processed by an heap merge implementation called `kMerger` (*stage two* uses a similar solution to merge the sorted runs). This appears to be a method that could reduce the running time using the higher speed of the cache. Unfortunately there are hardware-related problems, for example caused by the associativity of the CPU cache [3], that can be overcome by a careful *hand-tuning*. At the base case, *psort* employs a small, efficient counting sort to sort short sequences of records whose length is determined by the parameter `chunk_size`. The size of these input sequences is determined by the compilation parameter `chunk_size`. These sorted chunks are then sorted by a carefully designed merge sort routine which consumes only 5/4 of the total input size, rather than the "common" factor 2. There is also a variant of this algorithm, called `sorter_quasi_in_place_wave` (see Section 3.2), with the same space complexity but in some cases faster (although it is not clear in which cases). With a proper hand-tuning in compilation it is possible to choose the best implementation.

When *keys* are sufficiently small compared to the records' remaining *pay-*

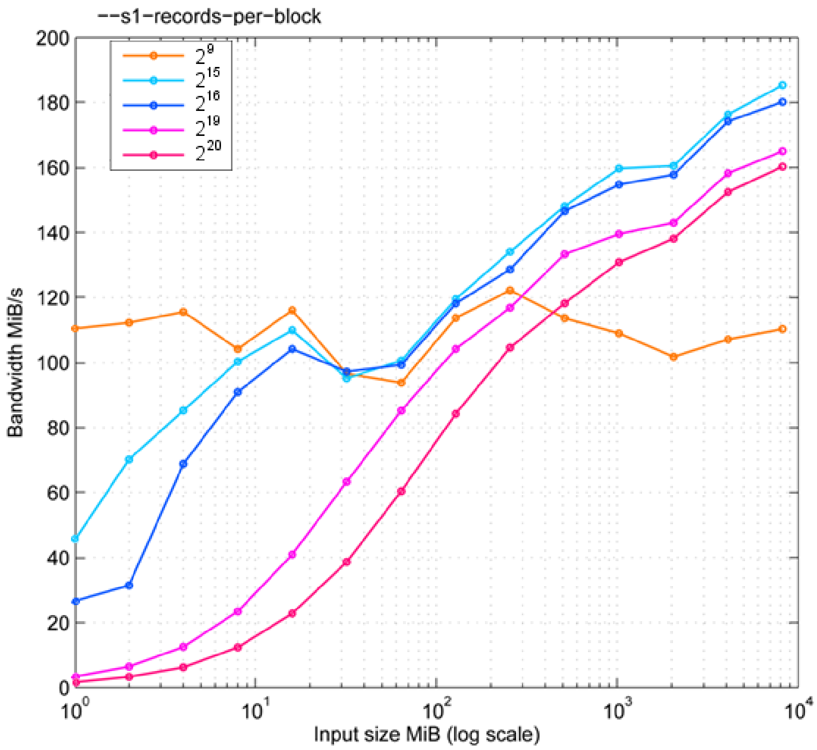


Figure 1.2: psort stage one: bandwidth as a function of the input size, for some values of –s1-records-per-block. Input consisted of 128-byte records with 8-byte keys, sorted using 3GB of allocated memory, direct I/O access and an I/O total buffer size of 300MB

loads, *psort* detaches keys from payloads, attaching to each key a pointer to the corresponding payload. The performance improvement is evident when payloads are longer than keys, that is a scenario arising in many applications (and in the Sort Benchmark competition). Thus parameter tuning must take into account that *psort* usually works with blocks of *extended keys*. For example, consider the parameter *–s1-records-per-block*, which indicates how many extended keys (or records if the payloads have not been separated by the keys) compose a microrun in the stage one. In order to improve the performance, the size of a microrun should be less than the L2 cache capacity, so the value of this parameter must satisfy the following inequality (up to constants):

$$size(microrun) \leq \frac{size(cache\, L2)}{size(extended\, key)}$$

Figure 1.2 shows that, for input sizes greater than about 50MiB, the optimal choice of *–s1-records-per-block* is the value that makes the microruns size around the size of L2 cache. In fact in this case the latter is 1MB and the optimal values appears to be 2^{15} and 2^{16} records; with an extended keys of $8 + 8 = 16$ bytes, this means respectively 0.5MB and 1MB for each microrun (other results shows that for larger inputs the trend remains the same).

Starting from the considerations above, it can be understood why the allocation of the memory for I/O buffers is not so simple. About this problem, the following parameters give a high flexibility:

- *–s1-read-buffer-size*: size of each read buffer,
- *–s1-read-buffers*: number of read buffers,
- *–s1-write-buffer-size*: size of each write buffer,
- *–s1-write-buffers*: number of write buffers,
- *–s1-io-space*: fraction of memory allocated for stage one I/O buffers.

In order to tune well these variables it has to be considered that more buffers don't mean necessarily increase the performance; furthermore using more memory for the buffers the runs will be more and of smaller size, thus the second phase could be slower.

The output of the final pass is directly written to the I/O buffers, potentially recombining keys and payloads. Note that disk writes can be almost entirely overlapped with the merge pass(es) described above.

1.2.2 Stage two

The second phase (which only takes place if the data-set does not fit in main memory) is much simpler: w sorted runs at a time are streamed from disk and merged (with the same code that merges microruns in the first phase), and the output is streamed back to the disk. It is reasonable to use a single merge pass if the number of data runs (approximately the ratio between data and memory size) does not exceed the ratio between the size of the memory and that of one "efficient" read from disk. This means that a single merge pass is possible only if the memory is large enough to contain, for each run, an amount of data that is large enough to ensure an efficient transfer from/to the I/O subsystem – which depends on the hardware.

Data are read with direct asynchronous I/O into (userspace) dynamically sized buffers, one per run. When the amount of data in a buffer falls below a threshold (*–s2-read-threshold*, a parameter to be tuned) the buffer is "*refilled*" from the appropriate run. This can be highly ineffective, however, if data are not consumed uniformly, and in particular if they are consumed more rapidly from recently refilled buffers. For this reason, *psort* allows the user to specify, via the *–s2-geometry-factor* option, a "geometry factor" that determines the dimension of a run's buffer in comparison to the previous run's one, choosing a trade-off between safety and optimization. Figure 1.3 illustrates how this parameter influences the performance of stage two.

1.2.3 Performance limitations

The goal of this work is to design and implement an automatic system to optimize the source code of *psort*. This is actually part of a more general process, consisting of three separate tasks: *automated hardware detection*, *tuning at run-time*, and *automated code tuning*. The first task has the goal to detect some

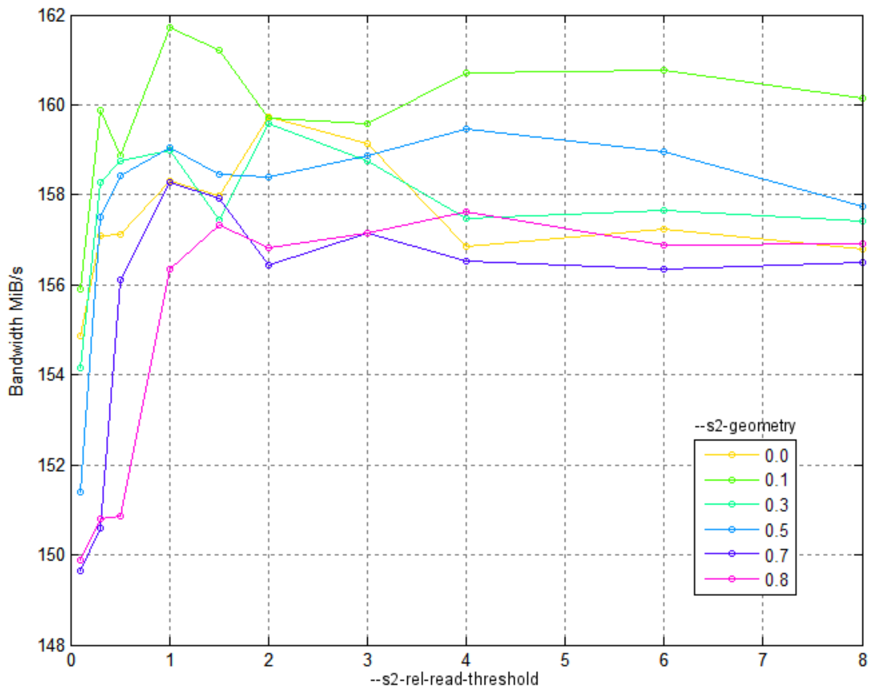


Figure 1.3: psort stage two: bandwidth as a function of the parameter –s2-rel-read-threshold (run queue size threshold for read request, relative to a minimum default value of queue size), for various values of the parameter –s2-geometry (the geometry factor used in the stage two). The input, consisting in 128-byte records with an 8-byte keys, was sorted into 128 runs, each of 256MiB (overall input size: 32GiB).

hardware parameters; these values have an impact on (and are taken into account by) the other tasks. Tuning at runtime pertains the choice of optimal execution parameters, like the size of each run, the number of records in a single block, or the buffers size. Automated code tuning concerns instead lower level optimizations like *loop-unrolling* and the choice between *bitwise* and *logical compares*.

The first task faced was tuning at runtime. The initial step was to find optimal combinations of the parameters on different machines, with the goal of understanding the relation between the optimal value of a parameter and the hardware architecture. To this end, we have written a Bash script that allows to automatically execute many unsupervised tests and a *Matlab* script that plots automatically the output data. This allowed us to rapidly produce large amounts of custom plots, which were of substantial help to our research. Thus the most important parameters were identified; which means that the running time of *psort* is very sensible to the variation of these variables.

In the stage one, some parameters are assigned at the execution but some others needs the re-compilation of the source code; the reason is that the routines in which they are used are repeated many times, so adding code to handle conditions at runtime could mean to reduce the performance visibly, and compilers produce significantly faster code if the values of these parameters are known at compile time. Thus the tuning of these parameters cannot be done at runtime. The performance optimization done using variables of this type is called *compiler-based auto-tuning* (see Section 2.1).

Important potential hurdles at this stage are associativity misses (in early experiments these reduced performance by as much as 20% [7]). However *psort* employs a careful data layout to minimize this problem, and the tuning at runtime tries to assign a reasonable value to each parameter using the results of automated hardware detection.

For all these reasons, it is crucial to provide *psort* with an automatic code generation systems, allowing for an efficient and effective generation of code which substitutes the costly (and boring!) manual tuning. The next section is devoted to this.

Chapter 2

Automated code tuning

Automated code tuning is the process with the purpose of finding an efficient solution (which typically means reducing the running time of the resulting code), evaluating different implementations of the same algorithms or data structures; each optimization is generally related to one or more parameters. This process can be (and often is) carried out by a developer, but with the significant disadvantages of take a very long time, of being error-prone and is clearly (except at an extremely high cost) neither portable nor reusable. The values of the working parameters can be calculated using an appropriate model or with an empirical search. Therefore, automated code tuning is the natural choice for high-performance software like *psort*. This chapter presents an overview of some existing solutions (Section 2.1) and a more detailed description of three of them (Sections 2.2-2.4).

2.1 Overview

Modern microprocessors can achieve high performance but this sometimes requires extensive machine-specific hand-tuning (e.g. [6]). There are many factors to be considered in the complex computer architectures: cache (multiple levels, capacity, associativity), memory latency, parallelism. Another point is that the compiler optimizations could be improved with some particular features. In fact, more sophisticated compiler optimizations, including loop unrolling and *software pipelining*, are either not performed or not very effective at producing the highest quality code [6]. Many of these improvements can often be converted in some coding guidelines that permits to write fast codes [12]. There are other low-level choice hard to be predicted, like the faster comparison between logical and bitwise. For these it is sometimes easier try and look which implementation is better. The optimizations at issue cannot be tuned with some execution parameters, like does tuning at runtime. In these cases for each value of such a parameter the code has to be modified and compiled. The automated code tuning has the goal to produce parametrized code generators rather than code by hand, to execute some tests and to obtain an optimal version of the software for the particular machine.

Some automated code tuning systems have already been developed, mainly in high-performance and scientific computing libraries. They usually tune once when installed to determine the best settings and optimizations for the hardware they are running on. This approach is appropriate when the library will be used in different architectures and should provide good performance for all of them. These systems are clearly very application specific.

One approach is to use *analytical models* to determine an optimal combination of values. Unfortunately if there are too many parameters it could be difficult to build a suitable model that covers the main hardware behaviours [11]. Moreover there is the risk to obtain a non optimal solution, if the model is too simple; nevertheless the probability to obtain a bad model increases with more complex approximations. In a nutshell, when the optimization space is too large a model driven methodology could be intractable.

Another approach is to perform a *global empirical search* over the space of parameters values. For example, the linear algebra library *ATLAS* [10] provides portable performance across a range of CPUs architectures by using a database of precomputed optimizations for known architectures, and by execute a *global search auto-tuning* for new architectures. *PHiPAC* [12] and *OSKI* [16] use similar techniques.

An alternative manner is to use both a simple model and an empirical search. This methodology is called *local search [11]* because the parameters values are searched around the model answer. In this case is also needed a *search engine*, that permits to reduce the work executing only the tests that could give useful results.

Compiler-based auto-tuning is another important part of the problem. It concerns the improvement of some code optimizations, such as loop unrolling, to find the best for a particular program. For example, the compiler uses software pipelining more probably if the code is written following some particular rules. These indication could be dependent from the architecture or the compiler, so having an automated translation of the source code to one more suitable to the hardware used could be help the compiler in the optimization. This sort of operations are often join in a bigger search engine, that is responsible for identifying optimal values also for other parameters

Other examples of auto-tuned software are *FFTW* [13, 14] and *NukadaFFT* provide FFT (Fast Fourier Transform) implementations which use *automated code tuning* to improve performance. *NukadaFFT* runs on GPUs using *CUDA* and uses auto-tuning to choose various GPU optimizations. *FFTW* is a CPU implementation which uses auto-tuning to construct good execution plans. This library follows the idea of problem-by-problem optimization. An execution plan is created based on the hardware being used and the memory layout of the problem being solved. This plan can then be used when solving any problem of the same ‘shape’. *SPIRAL* is another auto-tuning based digital signal processing library.

2.2 *ATLAS*

The *ATLAS* (Automatically Tuned Linear Algebra Software) project is an ongoing research effort focusing on applying empirical techniques in order to provide portable performance [10, 11]. The original version uses a global search approach; instead the complete *ATLAS* system includes a search engine, a code generator and a test code that measure the execution time of some critical parts of the source code. It also includes hand-written code for various routines, which may be used to produce the library on some machine if it is found to outperform the code produced by the code generator. This situation could happen using well-known architecture for which precomputed optimizations has been developed very well.

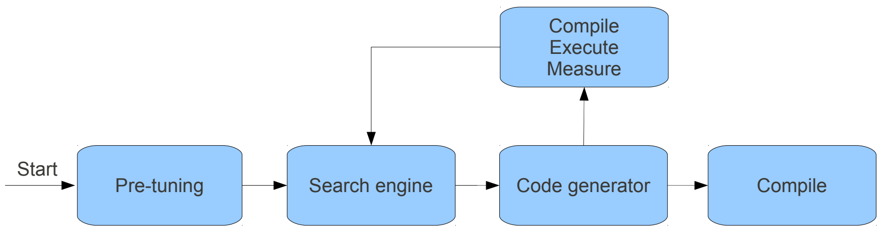


Figure 2.1: Empirical Optimization Architecture

For large parameter ranges, a full global search auto-tuning is very slow, so it is necessary to bound the search space. At the installation *ATLAS* executes a set of micro-benchmarks to measure certain hardware parameters such as the L1 data cache capacity, the number of registers, etc. The search engine bounds the search space using theses values. In Figure 2.1 it is schematized the sequence of operations performed by a software like *ATLAS*. The first block, automated hardware detection, indicates the measure of hardware parameters. After that there is a loop in which the search engine decide if the final source code could be produced and compiled, or if more tests are needed. The *Compile-Execute-Measure* block summarize the compilation of the little piece of code, the execution, and the performance measures. Thanks to these data the search engine can decide when the optimal values have been found.

An example of search is about the matrix-multiplication routine: the *naive code* is divided into sub-problems which size is assigned by a parameter. A simple function like that could depends on 4 parameters, for each the search engine have to determine the search space. Thus in a big code dozens of optimization parameters could be found, so it is very important to define small search spaces because the complexity increases quickly. Another issue is if try each combination or if study the trend of each parameter with the others fixed; a good solution could be to test separately some group of related parameters. All these choices depends on the number of parameters, on the search space size, and on how much time the user can spend in this operation. The *brute-force search* of all possible valuations is tractable for little application, otherwise is very slow. The *ATLAS* solution is a global search strategy called *orthogonal line search*. It tries to find the optimal values of a function $y = f(x_1, x_2, ..., x_n)$ by reducing the *n-dimensional* optimization problem into a sequence of *n 1-dimensional* optimization problems by sorting the parameters x_i, in some order, and optimizing them one at a time using reference values for parameters that have not been optimized yet [15].

2.3 *FFTW*

FFTW (Fastest Fourier Transform in the West) is a *C* subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data [13, 14]. This package was developed at MIT by Matteo Frigo and Steven G. Johnson. *FFTW*'s performance is portable: the same program will perform well on most architectures without modification. Thanks to this feature the performance is typically superior to that of other publicly available FFT software, and is even competitive with vendor-tuned codes (hand-tuned for a particular hardware)[14]. Like described above, *FFTW* follow the idea of problem-by-problem optimization. This means that an execution plan is based on the hardware being used and the memory layout of the problem being solved. This plan can then be used when solving any problem of the same 'shape'.

In *FFTW*, the computation of the transform is accomplished by an executor that consists of highly optimized, composable blocks of *C* code called *codelets*. A codelet is a specialized piece of code that computes part of the transform. The combination of codelets applied by the *executor* is specified by a special data structure called a plan. The plan is determined at runtime, before the computation begins, by a *planner* which uses a dynamic programming algorithm to find a fast composition of codelets. The planner tries to minimize the actual execution time, and not the number of floating point operations. Consequently, the planner measures the running time of many plans and selects the fastest.

The speed of the executor depends crucially on the efficiency of the codelets, but writing and optimizing them is a tedious and error-prone process. For this reason, *FFTW* generates the codelets automatically with a codelet generator. The main advantages of generating code are that it is simple to experiment with different algorithms or coding strategies, and it is easy to produce many long blocks of unrolled, optimized code.

The user interacts with *FFTW* only through the planner and the executor, as in the following example:

```
fftw_plan plan;
COMPLEX A[n], B[n];
/* plan the computation */
  plan = fftw_create_plan(n);
/* execute the plan */
  fftw(plan, A);
/* the plan can be reused for other inputs of size N */
  fftw(plan, B);
```

Figure 2.2: Simplified example of FFTW's use. The user must first create a plan, which can be then used at will.

2.4 *PHiPAC*

PHiPAC is a methodology for developing Portable High-Performance linear algebra libraries in *ANSI C*. Its goal is to produce, with minimal effort, libraries for a wide range of systems [12].

The *PHiPAC* methodology has three components. First a generic model of current *C* compilers and microprocessors that provides guidelines for writing high-performance code. Second, rather than hand-code particular routines, it uses parametrized generators that produce optimized code. Third, it automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

For example, using the *PHiPAC* methodology in the production of a portable, *BLAS*-compatible matrix multiply generator, the resulting code can achieve over 90% of peak performance on a variety of current workstations, and is sometimes faster than the vendor-optimized libraries [12].

The *PHiPAC code generator* follows some guidelines that can be used directly to improve performance in critical routines. For example, loop unroll explicitly to expose optimization opportunities, or increase locality to improve cache performance. Follows some others coding guidelines (see [12]):

- using local variables, reorder operations to explicitly remove false dependencies;
- exploit multiple integer and floating-point registers;
- minimize pointer updates by striding with constant offsets;
- hide multiple instruction FPU latency with independent operations;
- balance the instruction mix;
- convert integer multiplies to adds;
- minimize branches, avoid magnitude compares.

Chapter 3

Implementation

This thesis has the goal to create a code that generates a machine-optimized version of *psort*. The first choice was to realize a full working source code; so the quantity of tuned code faded into the background, and the focus went on a single critical part of *psort*. In this way the code has achieved a stable version and it uses some well-defined rules. Thus it is possible to expand the optimization to other parts of the software.

The automated code tuning in *psort* uses the local search approach (Section 3.3): the concept is to use a very simple model of the system to determine the values of the parameters. The latter are called start values and the tests should be done around them. The solution determined by this method is optimal, or in any case a good one. This approach generates high-performance code without increasing search time dramatically like in a brute-force code tuning.

First of all it is important to define the main steps of the execution.

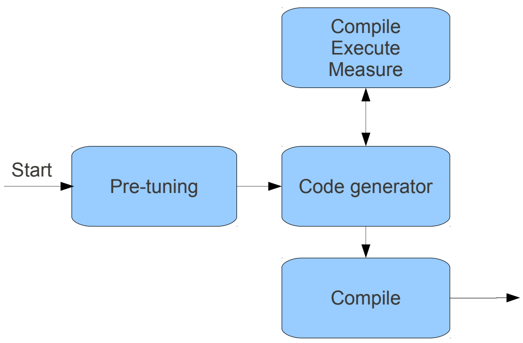


Figure 3.1: Empirical optimization architecture without a search engine

The chosen methodology is similar to the *ATLAS* empirical optimization architecture. In fact there are the automated hardware detection part, a code generator, a test and measure script, and the predisposition for a search engine. The latter isn't necessary because the optimization parameters have a small search space; however the code is thought to be easily adapted if required. The automated hardware detection code provides some hardware information about cache, hard disk, CPU and RAM. It also makes specific tests that permits a further optimization of the code; for example it tells if bitwise comparisons are faster or slower than logical. All these information are used by the other blocks in Figure 3.1, and also during tuning at runtime. The following block is the code generator: given a file, its parameters, and their search spaces, it generates a big code with the different versions of the interested routines (candidate code). The last block compile this file, executes many tests, and eventually chooses the best parameters. After this phase the code generator produces the final code.

A deep execution of the automated code tuning could take many hours, so there is a parameter that permits to choose the level of detail of the tuning. However this parameter should be set at the maximum value because probably it's better to loose some hours and have a fast software instead of to have a fast tuning but a slow software; in fact the tuning is made only once.

The operations shown in Figure 3.1 are executed by the file `tuner.sh`, a *Bash script* with as input the *tuning-level* (an integer specifying the level of detail of the tests) and some optional parameters. Thus this script has the tuned file as output and, when all the tuning operations have been finished, it compiles the whole *psort.*

3.1 Code Generator

The code generator, like described above, generates from each file that has to be tuned another source code with all the possible versions of the interested routines in the search space. The choice was to use a pre-existent software as support. The most suitable appeared to be *CodeWorker* [9]. It is a versatile Open Source parsing tool and a source code generator. This software permits to expand a code with a transformation source-to-source, following some specific rules described in a separated file in *extended-BNF* syntax, for recognizing the format of the specification to parse - a procedural language for manipulating easily parse trees, string, files, and directories [17].

CodeWorker provides two methods for performing a parsing:

- the reading of tokens is *procedural*;
- the BNF description is *declarative*, and conforms to a kind of BNF (the Backus-Naur Form represents a grammar in a particular syntax) extended with regular expressions.

The first file of *psort* chosen to be optimized is `cache_sorters.cpp`. Like described in Section 1.2 this is a critical file because it contains low level routines used many times in a single execution, so a little optimization could speed up significantly the software. The extension of a file to be tuned has to be change from *.c*, or *.cpp*, to *.tun*; this indicates that the syntax of the file is different from that of a *C* or *C++* source code. In fact, in order to obtain a tunable code it is necessary the uses of particular tags, defined and recognized by the grammar created in the extended-BNF file. For example, these tags permit to the parser to understand when implement a loop unrolling. In this case to the cycle is assigned a parameter with a specific search space and the code generator writes a separate function for each value of the parameter.

3.1.1 Grammars and generation rules

Extended Backus-Naur Form (extended-BNF) is a family of metasyntax notations used for expressing context-free grammars. It is used wherever exact descriptions of languages are needed, so it is suitable for the role of recognizer of programming languages or source-to-source translator. *CodeWorker* uses a variant of extended-BNF.

In order to make a code generator there must be a file that defines the generation rules. This file, with extension *.gen*, describes a grammar. First of all it permits to identify in the code the interested tags; after that it specify how to substitute this identifier with a valid source code. The extended-BNF, that is used in this kind of files, has the same basic rules of a simple grammar. Thus the programmer writes the production like in Figure 3.2 (where capital letters are used for indicate variables, instead the other characters are used for terminal symbols).

```
S ::= AB | C
A ::= a | aA
B ::= b | bB
C ::= aCb | => { writeText("Hello World");
                  if $exclamation==1$
                    writeText("!");
                }
```

Figure 3.2: A simple set of productions recognized by CodeWorker

The syntax recognized by *CodeWorker* permits to the programmer to write in the grammar definition a *C* like code (see the last production in Figure 3.2). The modified extended-BNF has also many specific keywords that permits to write a more simple code for even complex operations. Some of these are very useful, for example:

1. `implicitCopy` - it puts all scanned characters in the output;

2. `explicitCopy` - the scanned symbols doesn't go automatically in the output;

3. `readCString` - the *C* strings are ignored;

4. `readIdentifier`: "STR" - if it has be scanned the string “STR” this condition is fulfilled.

The file `cache_sorters_tuner.gen` contains an about 500 lines grammar. This has to read a tunable file, to identify the tuning tags (with the last of the keywords described above), to transform properly the code contained in these tags, and to copy the other lines without changes in the output (with the first keyword).

3.1.2 Tags

A tunable code recognize the parameters and the tags defined in the grammar. Before going deeper it is necessary a short description of the main ones.

Function:

```
< FUN functionName_PARAM1_PARAM2_PARAMN >< _FUN >
```

The code inside these tags will be repeated for each value of the parameters. These must be specified in the function name.

If it is used the parameter `< BYTES >` (it indicates the size of the extended key) inside `FUN` tag this code will be repeated for each value of `BYTES` and the tag is replaced by its value on the specific routine. Furthermore if there are N parameters the code will be repeated for each combination of values.

Assign-Value:

```
< ASSIGN PARAM value1 value2 valuen >
```

This one allows to assign an array of values to the interested parameter until the tag restore, that copies the old values to the variable. These tags were added because it could be useful to give the possibility of make some differences in particular cases. For example, it could be used for the parameter `BYTES` if we are interested in modify only the 16 bytes versions of a function.

Restore-Value :

```
< RESTORE PARAM >
```

It has been described above.

Only-for-part-1 :

```
< P1 > <_ P1 >
```

The code inside these two tags will be parsed only in the phase one, which generates the code to be tested.

Only-for-part-2 :

```
< P2 > <_ P2 >
```

The code inside these two tags will be parsed only in the phase two, which generates the final code.

Loop:

```
< LOOP _LOOP >
```

This tag is necessary for the loop unrolling. The code inside will be written as many times as the current value of the loop unroll factor specified in the function name. This tag is related to the variable `LOOP_UNROLL_FACTOR_CHUNK`. There are other elements like this related to other variables.

Select:

```
< SELECT FUNCTION_NAME value > < _SELECT >
```

These tags give a name to the code that they contain. It is useful when `TUNING_PART` is 2 and only the best sorter function must be written in the final source file. So it has the aim to permit a choice between different implementation of the same function. These tags could also be used when `TUNING_PART=1` if the choice has been already given. An example is the selection between bitwise and logical comparison.

The next subsection shows how to use the tags above with some practical examples.

3.1.3 Creation of tunable code

In order to transform a source in a tunable code the programmer has to simply modify the starting file, adding and modifying some lines. Let see this operation with an example.

```
9.  inline void chunk_sorter_16b( ureg_t *buffer, const ureg_t size ) {
10.   ureg_t b12, b13, b14, b15, b16, b17, b18, b23, b24, b25, b26, b27;
11.   ureg_t b28, b34, b35, b36, b37, b38, b45, b46, b47, b48, b56, b57;
12.   ureg_t b58, b67, b68, b78;
13.   ureg_t data1[ 2 ], data2[ 2 ], data3[ 2 ], data4[ 2 ], data5[ 2 ];
14.   ureg_t data6[ 2 ], data7[ 2 ], data8[ 2 ];
15.   for ( ureg_t i = 0; i < size * 2; i += 32 ) {
16.     MICRO_SORT_8B_8( (i) );
17.     MICRO_SORT_8B_8( (i + 16) );
18.   }
19. }
```

Figure 3.3: A non-tunable version of `chunk_sorter_16b`

```
20. < ASSIGN BYTES 8 16 24 32 >
21. < P1 >
22.   < FUN inline void chunk_sorter_BYTESb_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK >
          ( ureg_t *buffer, const ureg_t size ) {
23. < _P1 >
24. < P2 >
25.   < FUN inline void chunk_sorter_BYTESb >( ureg_t *buffer, const ureg_t size ) {
26. < _P2 >
27.   ureg_t < COMPARE_VARIABLES >
28.   ureg_t < DATA_VARIABLES >
29.   for ( ureg_t i = 0; i < size < SIZE_SCALING >; i += < STEP > ) {
30.     < LOOP MICRO_SORT_8B_CHUNK_SIZE( (i + INCREMENT) ); _LOOP >
31.   }
32. }
33. < _FUN >
```

Figure 3.4: A tunable routine `chunk_sorter` *(8b, 16b, 24b, 32b)*

Looking at Figure 3.4, the code might seem very difficult to understand. Surely after the transformation the it become more complex, but the transition from Figure 3.3 to Figure 3.4 isn't so intricate.

First of all it is added at line 20 an `ASSIGN` tag, indicating that the code has to been repeated for each value of `BYTES` (instead in Figure 3.3 is reported only the 16 *key-bytes* version). After that there is the difficult part: the *phase one* has to produce all the function to be tested, so they must have different names; instead the *phase two* returns only the optimal code with the final function name. At lines 27 and 28 there are particular tags created because for each value of `BYTES` this code needs a different number of variables. Finally there is the core of the optimization: a loop unrolling tag. `SIZE_SCALING` multiplies size by a number from 1 to 4 dependent by the value of `BYTES`, because larger is the *extended key* larger become the microrun size (the variable size is normalized to the 8 *key-bytes* version, so with an *extended key* of 16 bytes the value of size has to be doubled). The tag `STEP` depends on the value of the `loop_unroll_factor`. At line 30 there is a tag `LOOP` that repeats that macro like in Figure 3.3.

Figures 3.5-3.6 show a similar transformation applied on the function `merge`. The parameters and the considerations are similar to those explain above.

```
inline void merge_16b
( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
  ureg_t i = 0, j = 0, k = 0;
  ureg_t a[ 2 * 2 ];
  ureg_t b;
  while ( ( i < size * 2 ) & ( j < size * 2 ) ) {
    KEY_COPY_16B( &a[ 0 ], &s1[ i ] );
    KEY_COPY_16B( &a[ 2 ], &s2[ j ] );
    b = KEY_COMPARE_16B( &a[ 0 ], &a[ 2 ] );
    KEY_COPY_16B( &dest[ k ], &a[ b * 2 ] );
    k += 2;
    i += ( 1 - b ) * 2; j += b * 2;
  }
  for ( ; i < size * 2; i += 2 ) {
    KEY_COPY_16B( &dest[ k ], &s1[ i ] ); k += 2;
  }
  for ( ; j < size * 2; j += 2 ) {
    KEY_COPY_16B( &dest[ k ], &s2[ j ] ); k += 2;
  }
}
```

Figure 3.5: A non-tunable version of `merge_16b`

```
< ASSIGN BYTES 8 16 24 32 >
< P1 >
  < FUN inline void merge_BYTESb_LOOP_UNROLL_FACTOR_MERGE >
  ( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
    < INFO >
< _P1 >
< P2 >
  < FUN inline void merge_BYTESb >
  ( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
< _P2 >
    ureg_t i = 0, j = 0, k = 0;
    ureg_t a[ 2 < SIZE_SCALING >];
    ureg_t b;
    const ureg_t iMax =
(size/< LOOP_UNROLL_FACTOR_MERGE >)*< LOOP_UNROLL_FACTOR_MERGE > < SIZE_SCALING >;
    while ( ( i < iMax ) & ( j < iMax ) ) {
      < MERGELOOP KEY_COPY_< BYTES >B( &a[ 0 ], &s1[ i ] );
        KEY_COPY_< BYTES >B( &a[ BYTES_DIVIDED_BY_8 ], &s2[ j ] );
        b = KEY_COMPARE_< BYTES >B( &a[ 0 ], &a[ BYTES_DIVIDED_BY_8 ] );
        KEY_COPY_< BYTES >B( &dest[ k ], &a[ b * BYTES_DIVIDED_BY_8 ] );
        k += BYTES_DIVIDED_BY_8; i += ( 1 - b ) * BYTES_DIVIDED_BY_8;
        j += b * BYTES_DIVIDED_BY_8;
      _LOOP >
    }
    while ( ( i < size < SIZE_SCALING > ) & ( j < size < SIZE_SCALING > ) ) {
      KEY_COPY_< BYTES >B( &a[ 0 ], &s1[ i ] );
      KEY_COPY_< BYTES >B( &a[ < BYTES_DIVIDED_BY_8 > ], &s2[ j ] );
      b = KEY_COMPARE_< BYTES>B( &a[ 0 ], &a[ < BYTES_DIVIDED_BY_8 > ] );
      KEY_COPY_< BYTES >B( &dest[ k ], &a[ b * < BYTES_DIVIDED_BY_8 > ] );
      k += < BYTES_DIVIDED_BY_8 >;
      i += ( 1 - b ) * < BYTES_DIVIDED_BY_8 >;
      j += b * < BYTES_DIVIDED_BY_8 >;
    }
    for ( ; i < size < SIZE_SCALING >; i += < BYTES_DIVIDED_BY_8 > ) {
      KEY_COPY_< BYTES >B( &dest[ k ], &s1[ i ] );
      k += < BYTES_DIVIDED_BY_8 >;
    }
    for ( ; j < size < SIZE_SCALING >; j += < BYTES_DIVIDED_BY_8 > ) {
      KEY_COPY_< BYTES >B( &dest[ k ], &s2[ j ] );
      k += < BYTES_DIVIDED_BY_8 >;
    }
  }
< _FUN >
```

Figure 3.6: A tunable routine `merge` *(8b, 16b, 24b, 32b)*

```
< SELECT COMPARE16 1 >
  #define OR16 |
  #define AND16 &
< _SELECT >
< SELECT COMPARE16 2 >
  #define OR16 ||
  #define AND16 &&
< _SELECT >
```

Figure 3.7: A choice with the tag `SELECT` *- only one of the two blocks can be written in the final source code. Which one depends on the value of the parameter* `COMPARE16`.

Another example of transformation is from `inlines.h` to `inlines.tun`. The source code in Figure 3.7 shows a simple tag `SELECT` that permits to write on the final code only the best choice between bitwise and logical compares. Without this methodology, in order to make the optimal choice, the programmers should probably compile two times the code and measure the performance. With many parameters like this the hand tuning could become very tedious and error-prone.

The file resulting by the first phase of the tuning, that is the source code with all the variants that has to be tested, is very big. In fact, from the about 3000 lines of the tunable file the obtained source code could have more than 9000 lines; this value depends on what optimization has been enabled and on the size of the search space.

3.2 Critical optimizations

The file `cache_sorters.cpp` has been chosen to be automatically tuned. This source code contains low level routines divided by the key size, many macros used for the hand-tuning, and many versions of the same functions with different values of the parameters `chunk_size` and `loop_unroll_factor`. Thus the source code appeared to be often repeated with minor changes from a function to another. These little variations, together, could speed up the execution over the 10%.

There were 3 main functions that are now tuned: `chunk_sorters`, `merge`, and `sorter`. The first sorts a chunk of data (microruns) of 2, 4, or 8 elements with a count-sort; its size depends on the parameter `chunk_size` . The second merges two sorted *chunks* of data. The third uses `chunk_sorters` and the merge functions to sort a block of data whose size is defined by the execution parameters *–s1-records-per-block* and *–s2-records-per-block*, in one or more steps (it is a *merge-sort* that starts from an array of elements sorted in groups which sizes are the value of `chunk_size`).

The example in Figure 3.8 shows the sorting of a simple input sequence with the function `sorter_8b`. First of all it calls the routine `chunk_sorters` having `chunk_size = 2`. Usually the optimal value of the latter is higher, in fact the advantage is when it permits to avoid more than one merge executions. In this simple case after the first operation the array will be composed by sorted chunks, each of two elements. Thus the execution of the merge needs only two merge passes. These are the steps of *psort* in the sorting of a microrun.

There are also many versions of the functions `merge` and `sorter`. For example, the routine *sorter* has three possible implementations:

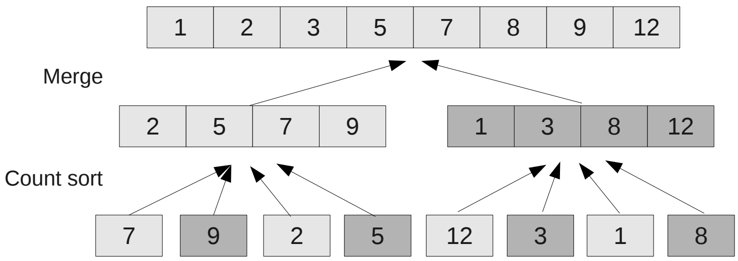


Figure 3.8: The two iterations performed by the `sorter()` *function on an input containing 8 keys. In this case, chunk_size equals 4. With a larger input the merge is repeated many times.*

- `sorter`,
- `sorter_quasi_in_place`,
- `sorter_quasi_in_place_wave`.

In each of these functions, where needed, is introduced a distinct parameter for the loop unrolling. In particular the function `chunk_sorters` has parameters like `loop_unroll_factor` and `loop_unroll_factor_chunk`, that are related respectively to an explicit loop unrolling and to the value of `chunk_size`, that comports the uses of different macros. These parameters are strongly machine-dependent. So the automated code tuning has to determine the fastest versions of these functions and the optimal values of their variables.

Another tunable file is `inlines.h`. This file defines the compare functions, so for these has to be chosen logical or bitwise compares; that is a little but sometimes an important tuning operation.

3.3 Local search

The automated code tuning in *psort* follows a local search approach (see Section 2.1). Actually the source code has been written to support this methodology, although the search space of the parameters is quite small. Some parameters has been grouped and, for them, the code generator produces routines in all possible combinations in the search space (for example in `chunk_sorters` the parameters `loop_unroll_factor` and `loop_unroll_chunk` are grouped). Our approach is similar to the *ATLAS* global search strategy: we use an orthogonal line search (see Section 2.2), with the additional possibility of grouping some parameters and execute a full global search between them (it tries all the combinations). The fixed values chosen for the parameters are a good compromise that appears to go adequately with many architectures.

Some parameters have a very small search space, for example `BYTES` and `loop_unroll_chunk`. In particular the first is a particular parameter, in the sense that *psort* needs the implementations for all the values in its search space.

In this case, the tuning has to find the best combination of `loop_unroll_chunk` and `loop_unroll_factor` distinctly for each value of `BYTES`.

Chapter 4

Results

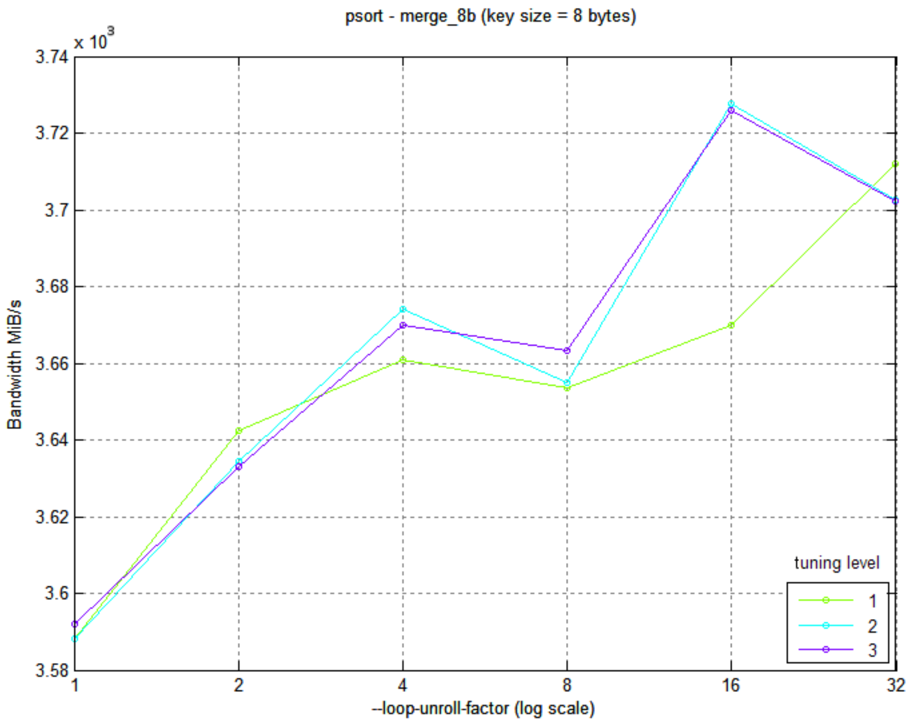
This chapter shows some tests on the performance of *psort* and the automated code tuning and argues some consideration starting from them. First of all, the bandwidth improvement is evident by some tests executed on architectures where hand-tuning hasn't been done yet. An important issue is how the trend of the performance depends on the tuning level and if a fast tuning is useful or not.

Figures 4.1 and 4.2 show the performance of the function `merge` for many values of `loop_unroll_factor` and for different sizes of the records (or extended keys). Each test has also been repeated for each available *tuning level*, that is the level of detail of a test: in this case, considering `merge_8b`, with the first level of tuning the test has been executed 20 times with an input of 1MB; with the third level the result is an arithmetic average of 200 executions of the same function with the same input size. This function is described in Subsection 1.2.1: it implements a simple merge sort between two arrays.

Increasing the value of tuning level to 2 has the effect of expanding the search space: the search is not anymore orthogonal but it takes into account more combinations. Furthermore, with tuning level 3 the number of executions of each test increase from 20 to 200, obtaining more reliable values.

Looking at Figures 4.1 and 4.2 it is important to focus on the relative variation of bandwidth. Certainly with another hardware composition the results will be different, but the situation found in this case is quite common. First of all it is important to specify that a larger value of the *tuning level* should mean a greater reliability of the resulting temporal data. Starting from this consideration different trend with tuning level 1 in the plot of `merge_8b` is not probably caused by noise. Unfortunately this situation is not detected with *tuning-level*=1 and this could lead to a non optimal choice. About the case of `merge_24b` the search space should be enlarged, because it is not clear if the bandwidth will fall or not with the increasing `loop_unroll_factor` (this argument follows the local search approach, starting from the value 1). A very different situation is shown in the plot of `merge_16b`, where it is difficult to understand what happens. In this cases, where the values are very near but the trend is systematic the automated code tuning can't do anything but a global search in the whole search space.

The search of the optimal values of `chunk_size` is simpler, in fact the default search space is {2, 4, 8} and the tests show that there is no need to expand it.



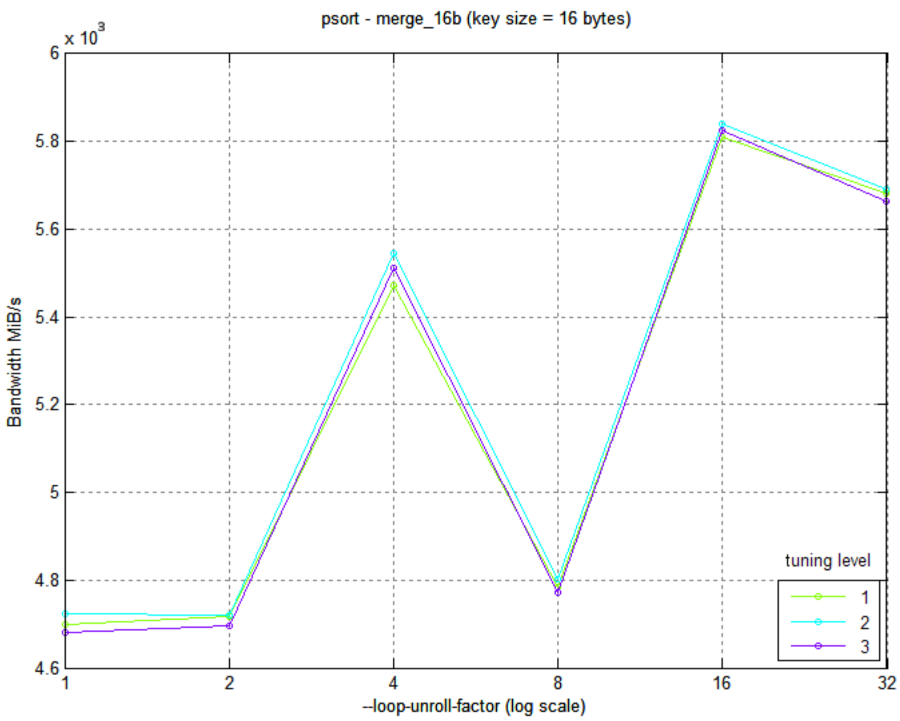
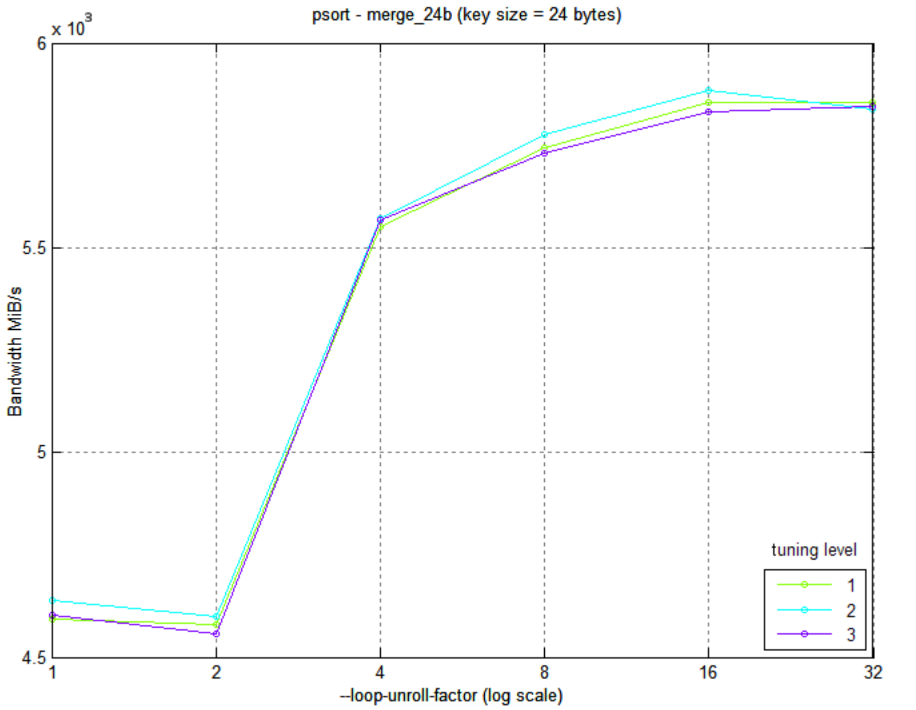


Figure 4.1: Performance of the functions `merge_8b` *and* `merge_16b` *(key size 8 and 16) in* `cache_sorters.cpp` *for different values of* `loop_unroll_factor` *and of the tuning level. Hardware used: 3.07 GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz*



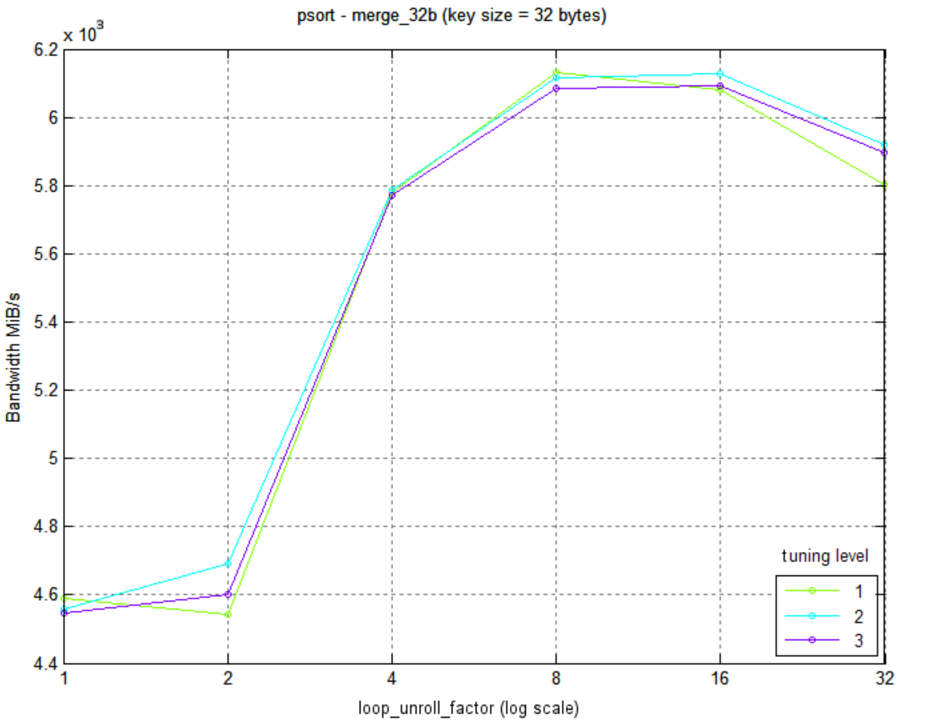


Figure 4.2: Performance of the functions `merge_24b` *and* `merge_32b` *(key size 24 and 32) in* `cache_sorters.cpp` *for different values of* `loop_unroll_factor` *and of the tuning level. Hardware used: 3.07 GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz*

tuning level	*tuning time* [sec]	`chunk_size`	*loop unroll factor*	
			`chunk_sorter`	`merge`
1	110	8, 4, 8, 8	*1, 32, 1, 1*	*32, 16, 32, 16*
2	210	8, 4, 8, 8	1, 2, 1, 8	16, 16, 16, 16
3	695	4, 4, 8, 8	32, 1, 1, 1	16, 16, 32, 16

Table 4.1: Running time and output values of the tuning phase (in these tests the sorter routine used was `sorter_quasi_in_place`*), for three different tuning levels. For each tuning level, the cell relative to a routine contains the estimated optimal values for its 8, 16, 24, and 32-byte versions. System setup: 3.07* GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz, RAID 6x1TB

tuning level	*input size [GiB]*	*execution time* (stage 1) [sec]	*average bandwidth [MB/s]*
0	32	113.7	288.2
0	128	450.7	290.8
1	32	112.2	292.0
1	128	446.1	293.8
2	32	110.8	295.7
2	128	444.5	294.9
3	32	109.6	299.0
3	128	440.2	297.8

Table 4.2: Performance of the automatically generated `stage_one()` *code for different tuning levels (0 indicates the original version), for 32GiB 128GiB inputs. Hardware used:* 3.07 GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz, RAID 6x1TB

Perhaps in some cases it could be useful to add the value 16, but it usually has worse performance.

Table 4.2 shows the performance of *psort*'s stage one using different level of tuning. It is important to remember that these measures of time are only about the tuning of `cache_sorters.cpp` (and `inlines.h`). With more tunable files the following relative considerations become even more justified. From Table 4.1 and 4.2 is simple to verify that with `tuning-level=3` a user saves from 1.3% to 2.3% of the time spent using `tuning-level=1`, in the sorting. This means that the additionally 585 seconds, spent in the tuning phase, will be made up sorting about 7TB of data. Using the `tuning-level=2` a user saves 1.2% of the time respect to the `tuning-level=1`(in the 32GiB version), and the user will be made up the additional 100 seconds after 2TB of data. The machine used for the Sort Benchmark had similar performance, although the hardware used for these tests are probably better for the issue[1]. This because the automated code tuning can come closer but hardly exceed the performance of a very well hand-tuned source code. Thus with a more accurate tuning at runtime and an automated code tuning extended to other critical aspects, the performance

[1] *Sort Benchmark machine: 2.7 Ghz AMD Sempron, 4 GB RAM DDR3 1333, 5x320 GB 7200 RPM Samsung SpinPoint F4 HD332GJ, Linux*

Automated Code Tuning test machine: 3.07 GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz, *RAID 6x1TB*

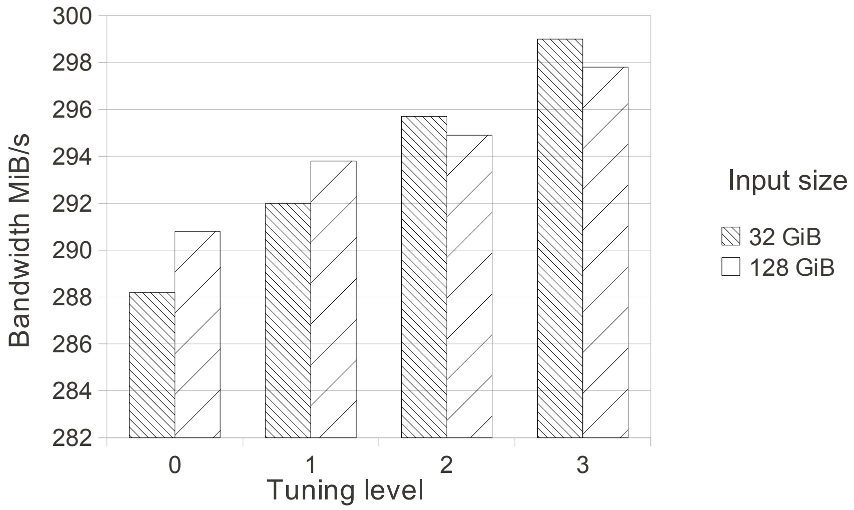


Figure 4.3: Performance of the automatically generated `stage_one()` code for different tuning levels (0 indicates the original version), for 32GiB 128GiB inputs. See Table 4.2. Hardware used: *3.07 GHz Intel(R) Core(TM) i7-950, 3 banks of 4 GB tri-channel DDR3 running at 1.6 GHz, RAID 6x1TB*

could overcome that obtained with the Sort Benchmark machine.

Recently the *psort* team has added the support to multidisks and multicore. These new features increase the software performance, but they also adds many parameters to be tuned. Probably in the future *psort* will become more complex, and thus the hand-tuned approach harder to realize.

Chapter 5

Conclusions

This works describes the design and implementation of an automated code tuning system for the external sorting library *psort*. Our system, which employs a variant of the Backus-Naur Form notation, allows programmers to efficiently and effectively write "tunable code" which is automatically expanded into several different candidate codes. The system then automatically performs code compilation and testing, selects the more efficient candidate code and integrates it into the *psort* library.

Our solution takes into account not only the performance of the resulting code, but also the reusability and modularity of the system itself. Thanks to these guidelines, the system can be extended to support those parts of code that were beyond the (experimental) scope of the present work, including code that has not been written yet. We evaluate the performance of several low-level sorting routines, including *psort*'s merge sort and heap sort, and show that automatically tuned versions perform up to 4% better than their (already highly efficient) original counterparts. Moreover, users can specify the tuning level, exploiting the trade-off between time spent tuning *psort* and time spent running *psort*. Our results show that automated code tuning provides substantial performance improvements at a modest cost in terms of end-user effort (indeed, the whole process aims to be transparent to the end user), and should thus be included in the *psort* library.

We suggest three main future directions of work. First, we propose to further reduce the programmers' effort by organizing and simplifying the grammar in a strict, well-defined manner. This would result in a simpler, better structured, and more readable code – desirable properties not only for future programmers but also for expert end users who want to hack into the source code. Second, we propose to investigate better strategies to find the (local) optimal point in the space of the automatically generated candidate codes. The current enumerative strategy, far from optimal, could be replaced by smarter search strategies – for example, the well-known taboo search – that would drastically reduce the tuning time, resulting in higher performance. An other issue is the reduction of the tuning time. Now it is acceptable for the user (for example see Table 4.1) but the tuning operations take longer than expected, so it is another direction of work.

Appendix A

Source code

This appendix contains the source code of *psort* used by the automated code tuning. Thus what is reported below are not the entire files.

A.1 cache_sorters.tun

```
< P1 >
#include <iostream>
using namespace std;
< _P1 >

/***************************************************
 **********  CODE FOR SORTING
 **********  LEXICOGRAPHICALLY
 ***************************************************/

/***************************************************
 *      Specific code for 8-byte extended keys
 ***************************************************/

#define CHUNK_SORT_8B chunk_sorter_8b
#define MERGE_8B merge_8b


/* Used for first-level cache sorters; assumes that variables   b0 ... bN,
    data1, ..., dataM   exist  */
#define MICRO_SORT_8B_4( i )             \
  KEY_COPY_8B( data1, &buffer[ i ] );          \
  KEY_COPY_8B( data2, &buffer[ i + 1 ] );        \
  KEY_COPY_8B( data3, &buffer[ i + 2 ] );        \
  KEY_COPY_8B( data4, &buffer[ i + 3 ] );        \
  b12 = KEY_COMPARE_8B( data1, data2 );          \
  b13 = KEY_COMPARE_8B( data1, data3 );          \
  b14 = KEY_COMPARE_8B( data1, data4 );          \
  b23 = KEY_COMPARE_8B( data2, data3 );          \
  b24 = KEY_COMPARE_8B( data2, data4 );          \
  b34 = KEY_COMPARE_8B( data3, data4 );          \
  KEY_COPY_8B( &buffer[ i + ( b12 + b13 + b14 ) ], data1 );   \
  KEY_COPY_8B( &buffer[ i + ( 1 - b12 + b23 + b24 ) ], data2 );   \
  KEY_COPY_8B( &buffer[ i + ( 2 - b13 - b23 + b34 ) ], data3 );   \
  KEY_COPY_8B( &buffer[ i + ( 3 - b14 - b24 - b34 ) ], data4 );


/* Used for first-level cache sorters; assumes that variables   b0 ... bN,
    data1, ..., dataM   exist  */
#define MICRO_SORT_8B_8( i )             \
  KEY_COPY_8B( data1, &buffer[ i ] );          \
  KEY_COPY_8B( data2, &buffer[ i + 1 ] );        \
  KEY_COPY_8B( data3, &buffer[ i + 2 ] );        \
```

```
  KEY_COPY_8B( data4, &buffer[ i + 3 ] );        \
  KEY_COPY_8B( data5, &buffer[ i + 4 ] );        \
  KEY_COPY_8B( data6, &buffer[ i + 5 ] );        \
  KEY_COPY_8B( data7, &buffer[ i + 6 ] );        \
  KEY_COPY_8B( data8, &buffer[ i + 7 ] );        \
  b12 = KEY_COMPARE_8B( data1, data2 );          \
  b13 = KEY_COMPARE_8B( data1, data3 );          \
  b14 = KEY_COMPARE_8B( data1, data4 );          \
  b15 = KEY_COMPARE_8B( data1, data5 );          \
  b16 = KEY_COMPARE_8B( data1, data6 );          \
  b17 = KEY_COMPARE_8B( data1, data7 );          \
  b18 = KEY_COMPARE_8B( data1, data8 );          \
  b23 = KEY_COMPARE_8B( data2, data3 );          \
  b24 = KEY_COMPARE_8B( data2, data4 );          \
  b25 = KEY_COMPARE_8B( data2, data5 );          \
  b26 = KEY_COMPARE_8B( data2, data6 );          \
  b27 = KEY_COMPARE_8B( data2, data7 );          \
  b28 = KEY_COMPARE_8B( data2, data8 );          \
  b34 = KEY_COMPARE_8B( data3, data4 );          \
  b35 = KEY_COMPARE_8B( data3, data5 );          \
  b36 = KEY_COMPARE_8B( data3, data6 );          \
  b37 = KEY_COMPARE_8B( data3, data7 );          \
  b38 = KEY_COMPARE_8B( data3, data8 );          \
  b45 = KEY_COMPARE_8B( data4, data5 );          \
  b46 = KEY_COMPARE_8B( data4, data6 );          \
  b47 = KEY_COMPARE_8B( data4, data7 );          \
  b48 = KEY_COMPARE_8B( data4, data8 );          \
  b56 = KEY_COMPARE_8B( data5, data6 );          \
  b57 = KEY_COMPARE_8B( data5, data7 );          \
  b58 = KEY_COMPARE_8B( data5, data8 );          \
  b67 = KEY_COMPARE_8B( data6, data7 );          \
  b68 = KEY_COMPARE_8B( data6, data8 );          \
  b78 = KEY_COMPARE_8B( data7, data8 );          \
  KEY_COPY_8B( &buffer[ i + ( b12 + b13 + b14 + b15 + b16 + b17 + b18 ) ], data1 ); \
  KEY_COPY_8B( &buffer[ i + ( 1 - b12 + b23 + b24 + b25 + b26 + b27 + b28 ) ], data2 ); \
  KEY_COPY_8B( &buffer[ i + ( 2 - b13 - b23 + b34 + b35 + b36 + b37 + b38 ) ], data3 ); \
  KEY_COPY_8B( &buffer[ i + ( 3 - b14 - b24 - b34 + b45 + b46 + b47 + b48 ) ], data4 ); \
  KEY_COPY_8B( &buffer[ i + ( 4 - b15 - b25 - b35 - b45 + b56 + b57 + b58 ) ], data5 ); \
  KEY_COPY_8B( &buffer[ i + ( 5 - b16 - b26 - b36 - b46 - b56 + b67 + b68 ) ], data6 ); \
  KEY_COPY_8B( &buffer[ i + ( 6 - b17 - b27 - b37 - b47 - b57 - b67 + b78 ) ], data7 ); \
  KEY_COPY_8B( &buffer[ i + ( 7 - b18 - b28 - b38 - b48 - b58 - b68 - b78 ) ], data8 );


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_8B_2( i )            \
  KEY_COPY_8B( data1, &buffer[ i ] );         \
  KEY_COPY_8B( data2, &buffer[ i + 1 ] );        \
  b12 = KEY_COMPARE_8B( data1, data2 );          \
  KEY_COPY_8B( &buffer[ i + b12 ], data1 );       \
  KEY_COPY_8B( &buffer[ i + 1 - b12 ], data2 );


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_8B_1( i )

/* TUNED CODE */
/** Sorts quadruples of elements in an array. In particular, independently
 *  sorts every subarray indexed from  k * subsize  to  k * subsize + 1
 *  \param buffer an array of shortrec_t records
 *  \param size the size of the array
 */

< P1 >
< ASSIGN BYTES 8 16 24 32 >
```

```
< FUN inline void chunk_sorter_BYTESb( ureg_t *buffer, const ureg_t size ) { } >
< _FUN >
< FUN inline void merge_BYTESb( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) { } >
< _FUN >
< FUN inline void merge_BYTESb_right( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) { } >
< _FUN >

ureg_t log_ceiling( ureg_t x ) {
    ureg_t depth = 0;
    ureg_t compare = 1;
    while( compare < x ) {
        compare <<= 1;
        depth++;
    }
    return depth;
}
< _P1 >

< ASSIGN BYTES 8 16 24 32 >
< P1 >
< FUN inline void chunk_sorter_BYTESb_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK >( ureg_t *buffer, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN inline void chunk_sorter_BYTESb >( ureg_t *buffer, const ureg_t size ) {
< _P2 >
  ureg_t < COMPARE_VARIABLES >
  ureg_t < DATA_VARIABLES >
  for ( ureg_t i = 0; i < size < SIZE_SCALING >; i += < STEP > ) { // STEP = CHUNK_SIZE * LOOP_UNROOL_FACTOR
    < LOOP MICRO_SORT_8B_CHUNK_SIZE( (i + INCREMENT) ); _LOOP >
  }
}< _FUN >

/** Merges two arrays
 *  Bit-based version
 */
< ASSIGN BYTES 8 16 24 32 >
< P1 >
< FUN inline void merge_BYTESb_LOOP_UNROLL_FACTOR_MERGE >( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN inline void merge_BYTESb >( const ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
< _P2 >
    ureg_t i = 0, j = 0, k = 0;
    ureg_t a[ 2 < SIZE_SCALING >];
    ureg_t b;
    const ureg_t iMax = (size/< LOOP_UNROLL_FACTOR_MERGE >)*< LOOP_UNROLL_FACTOR_MERGE > < SIZE_SCALING >;

    while ( ( i < iMax ) & ( j < iMax ) ) {
       < MERGELOOP KEY_COPY_< BYTES >B( &a[ 0 ], &s1[ i ] );
    KEY_COPY_< BYTES >B( &a[ BYTES_DIVIDED_BY_8 ], &s2[ j ] );
    b = KEY_COMPARE_< BYTES >B( &a[ 0 ], &a[ BYTES_DIVIDED_BY_8 ] );
    KEY_COPY_< BYTES >B( &dest[ k ], &a[ b * BYTES_DIVIDED_BY_8 ] );
    k += BYTES_DIVIDED_BY_8;
    i += ( 1 - b ) * BYTES_DIVIDED_BY_8;
    j += b * BYTES_DIVIDED_BY_8; _LOOP >
    }
    while ( ( i < size < SIZE_SCALING > ) & ( j < size < SIZE_SCALING > ) ) {
       KEY_COPY_< BYTES >B( &a[ 0 ], &s1[ i ] );
    KEY_COPY_< BYTES >B( &a[ < BYTES_DIVIDED_BY_8 > ], &s2[ j ] );
    b = KEY_COMPARE_< BYTES>B( &a[ 0 ], &a[ < BYTES_DIVIDED_BY_8 > ] );
    KEY_COPY_< BYTES >B( &dest[ k ], &a[ b * < BYTES_DIVIDED_BY_8 > ] );
    k += < BYTES_DIVIDED_BY_8 >;
    i += ( 1 - b ) * < BYTES_DIVIDED_BY_8 >;
```

```
      j += b * < BYTES_DIVIDED_BY_8 >;
      }
      for ( ; i < size < SIZE_SCALING >; i += < BYTES_DIVIDED_BY_8 > ) {
       KEY_COPY_< BYTES >B( &dest[ k ], &s1[ i ] );
      k += < BYTES_DIVIDED_BY_8 >;
      }
      for ( ; j < size < SIZE_SCALING >; j += < BYTES_DIVIDED_BY_8 > ) {
       KEY_COPY_< BYTES >B( &dest[ k ], &s2[ j ] );
      k += < BYTES_DIVIDED_BY_8 >;
      }
}< _FUN >


/** Sorts an array using a classical mergesort
 *  \param input  the source array
 *  \param output the destination array; must be allocated
 *  \param size   number of elements
 *  \return 0 if the sorted data is in input, 1 if in output
 */
< SELECT SORTER 1 > // Ignore this block if the value of sorter is neither 0 nor 1
< ASSIGN BYTES 8 16 24 32 >
< P1 >
  // < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
< FUN ureg_t sorter_BYTESb_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK >( ureg_t *input, ureg_t *output, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN ureg_t sorter_BYTESb >( ureg_t *input, ureg_t *output, const ureg_t size ) {
< _P2 >
    const ureg_t start = log_ceiling( < CHUNK_SIZE > );
    const ureg_t stop = log_ceiling( size );

< P1 >    chunk_sorter_< BYTES >b_< CHUNK_SIZE >_< LOOP_UNROLL_FACTOR_CHUNK >( input, size );< _P1 >
< P2 >    chunk_sorter_< BYTES >b( input, size );< _P2 >

      for ( ureg_t l = start; l < stop; l++ ) {
       /* merge pairs of subarrays of size 2^l */
       for ( ureg_t s = 0; s < size; s += 1 << (l + 1) ) {
< P1 >      merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( &input[ s < SIZE_SCALING >], &input[ ( s + (1 << l) ) < SIZE_SCALING >], &output[ s < SIZE_SCALING >], (1 << l) );< _P1 >
< P2 >      merge_< BYTES >b( &input[ s < SIZE_SCALING >], &input[ ( s + (1 << l) ) < SIZE_SCALING >], &output[ s < SIZE_SCALING >], (1 << l) );< _P2 >
         }
       TYPE_SWAP( ureg_t*, input, output );
      }

      return ( ( stop - start ) & 1 );

}< _FUN >
< P1 >
    < RESTORE LOOP_UNROLL_FACTOR_MERGE >
    < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >
//   < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
< _P1 >
< _SELECT >

/** Sorts an array using a classical mergesort, quasi-in-place (uses 1.5 times the input size)
 *  \param input  the source array
 *  \param output the destination array; must be allocated
 *  \param size   number of elements
 *  \return 0 if the sorted data is in input, 1 if in output
 */
< SELECT SORTER 2 > // Ignore this block if the value of sorter is neither 0 nor 2
< ASSIGN BYTES 8 16 24 32 >
```

```
< P1 >
//   < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
< FUN ureg_t sorter_BYTESb_quasi_in_place_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK >( ureg_t *input, ureg_t *output, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN ureg_t sorter_BYTESb_quasi_in_place >( ureg_t *input, ureg_t *output, const ureg_t size ) {
< _P2 >

    const ureg_t start = log_ceiling( < CHUNK_SIZE > );
    const ureg_t stop = log_ceiling( size );

< P1 >
chunk_sorter_< BYTES >b_< CHUNK_SIZE >_< LOOP_UNROLL_FACTOR_CHUNK >( input, size );< _P1 >
< P2 >     chunk_sorter_< BYTES >b( input, size );< _P2 >

    for ( ureg_t l = start; l < stop - 1; l++ ) {
    /* merge pairs of subarrays of size 2^l */
    ureg_t s = 0;
    for ( s = 0; s < size; s += 1 << (l + 1) ) {
< P1 >       merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( &input[ s < SIZE_SCALING >], &input[ ( s + (1 << l) ) < SIZE_SCALING >], &input[ ( s - (1 << l) ) < SIZE_SCALING >], (1 << l) );< _P1 >
< P2 >       merge_< BYTES >b( &input[ s < SIZE_SCALING >], &input[ ( s + (1 << l) ) < SIZE_SCALING >], &input[ ( s - (1 << l) ) < SIZE_SCALING >], (1 << l) );< _P2 >
     }
     input -= (1 << l) < SIZE_SCALING >;
    }
< P1 >     merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( input, &input[ (1 << (stop-1) ) < SIZE_SCALING >], output, (1 << (stop-1)) );< _P1 >
< P2 >     merge_< BYTES >b( input, &input[ (1 << (stop-1) ) < SIZE_SCALING >], output, (1 << (stop-1)) );< _P2 >

    return ( 1 );

}< _FUN >
< P1 >
   < RESTORE LOOP_UNROLL_FACTOR_MERGE >
   < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >
 //  < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
< _P1 >
< _SELECT >

#endif


/***************************************************
 *     Specific code for 16-byte extended keys
 ***************************************************/

#if ( ENABLE_16B == 0 ) /* If not enabled, use empty functions */

ureg_t sorter_16b( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}
ureg_t sorter_16b_quasi_in_place( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}

#else

#define CHUNK_SORT_16B chunk_sorter_16b
#define MERGE_16B merge_16b
#define MERGE_16B_RIGHT merge_16b_right
```

```
/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_16B_2( i )          \
  KEY_COPY_16B( data1, &buffer[ i ] );          \
  KEY_COPY_16B( data2, &buffer[ i + 2 ] );        \
  b12 = KEY_COMPARE_16B( data1, data2 );         \
  KEY_COPY_16B( &buffer[ i + b12 * 2 ], data1 );       \
  KEY_COPY_16B( &buffer[ i + ( 1 - b12 ) * 2 ], data2 );


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_16B_4( i )          \
  KEY_COPY_16B( data1, &buffer[ i ] );          \
  KEY_COPY_16B( data2, &buffer[ i + 2 ] );        \
  KEY_COPY_16B( data3, &buffer[ i + 2 * 2 ] );        \
  KEY_COPY_16B( data4, &buffer[ i + 3 * 2 ] );        \
  b12 = KEY_COMPARE_16B( data1, data2 );         \
  b13 = KEY_COMPARE_16B( data1, data3 );         \
  b14 = KEY_COMPARE_16B( data1, data4 );         \
  b23 = KEY_COMPARE_16B( data2, data3 );         \
  b24 = KEY_COMPARE_16B( data2, data4 );         \
  b34 = KEY_COMPARE_16B( data3, data4 );         \
  KEY_COPY_16B( &buffer[ i + ( b12 + b13 + b14 ) * 2 ], data1 );    \
  KEY_COPY_16B( &buffer[ i + ( 1 - b12 + b23 + b24 ) * 2 ], data2 );  \
  KEY_COPY_16B( &buffer[ i + ( 2 - b13 - b23 + b34 ) * 2 ], data3 );  \
  KEY_COPY_16B( &buffer[ i + ( 3 - b14 - b24 - b34 ) * 2 ], data4 );

/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
/* #define MICRO_INSORT_16B( i )          \
  for ( ureg_t _ii_ = 1; _ii_ < 4; _ii_++ ) {        \
    KEY_COPY_16B( data1, &buffer[ i + _ii_ * 2 ] );      \
    ureg_t _jj_ = _ii_;          \
    while ( KEY_COMPARE_16B( &buffer[ i + ( _jj_ - 1 ) * 2 ], data1 ) & ( _jj_ > 0 ) ) { _jj_--; } \
    memmove( &buffer[ i + ( _jj_ + 1 ) * 2 ], &buffer[ i + _jj_ * 2 ], ( _ii_ - _jj_ ) * 16 ); \
    KEY_COPY_16B( &buffer[ i + _jj_ * 2 ], data1 );      \
  }          \
*/

/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_INSORT_16B( i )          \
  for ( ureg_t _ii_ = 1; _ii_ < 4; _ii_++ ) {        \
    unsigned int _jj_ = _ii_;          \
    while ( ( _jj_ > 0 ) && ( KEY_COMPARE_16B( &buffer[ i + ( _jj_ - 1 ) * 2 ], &buffer[ i + _jj_ * 2 ] ) ) ) ) { \
      KEY_COPY_16B( data1, &buffer[ i + _jj_ * 2 ] );      \
      KEY_COPY_16B( &buffer[ i + _jj_ * 2 ], &buffer[ i + ( _jj_ - 1 ) * 2 ] ); \
      KEY_COPY_16B( &buffer[ i + ( _jj_ - 1 ) * 2 ], data1 );    \
      _jj_--;          \
    }          \
  }


/* Used for first-level cache sorters; assumes that variable data1 exists */
#define MICRO_INSORT_16B_bis( i )          \
  for ( ureg_t _ii_ = 1; _ii_ < 4; _ii_++ ) {        \
    unsigned int _jj_ = _ii_ * 2;          \
    while ( ( _jj_ > 0 ) && KEY_COMPARE_16B( &buffer[ i + _jj_ - 2 ], &buffer[ i + _jj_ ] ) ) { \
      KEY_COPY_16B( data1, &buffer[ i + _jj_ ] );      \
      KEY_COPY_16B( &buffer[ i + _jj_ ], &buffer[ i + _jj_ - 2 ] ); \
      KEY_COPY_16B( &buffer[ i + _jj_ - 2 ], data1 );      \
      _jj_ -= 2;          \
    }          \
  }


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
```

```
#define MICRO_INSORT_16B_4( i )             \
  for ( ureg_t _ii_ = 1; _ii_ < 8; _ii_++ ) {         \
    int _jj_ = _ii_;               \
    while ( KEY_COMPARE_16B( &buffer[ i + ( _jj_ - 1 ) * 2 ], &buffer[ i + _jj_ * 2 ] ) & ( _jj_ > 0 ) ) { \
      KEY_COPY_16B( data1, &buffer[ i + _jj_ * 2 ] );      \
      KEY_COPY_16B( &buffer[ i + _jj_ * 2 ], &buffer[ i + ( _jj_ - 1 ) * 2 ] ); \
      KEY_COPY_16B( &buffer[ i + ( _jj_ - 1 ) * 2 ], data1 );    \
      _jj_--;              \
    }              \
  }


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist  */
#define MICRO_INSORT_16B_4_bis( i )             \
  for ( ureg_t _ii_ = 1; _ii_ < 8; _ii_ += 2 ) {          \
    unsigned int _jj_ = _ii_;              \
    while ( KEY_COMPARE_16B( &buffer[ i + ( _jj_ - 2 ) ], &buffer[ i + _jj_ ] ) & ( _jj_ > 0 ) ) { \
      KEY_COPY_16B( data1, &buffer[ i + _jj_ ] );      \
      KEY_COPY_16B( &buffer[ i + _jj_ ], &buffer[ i + ( _jj_ - 2 ) ] ); \
      KEY_COPY_16B( &buffer[ i + ( _jj_ - 2 ) ], data1 );    \
      _jj_ -= 2;                 \
    }              \
  }




/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist  */
#define MICRO_SORT_16B_8( i )            \
  KEY_COPY_16B( data1, &buffer[ i ] );           \
  KEY_COPY_16B( data2, &buffer[ i + 2 ] );         \
  KEY_COPY_16B( data3, &buffer[ i + 4 ] );         \
  KEY_COPY_16B( data4, &buffer[ i + 6 ] );         \
  KEY_COPY_16B( data5, &buffer[ i + 8 ] );         \
  KEY_COPY_16B( data6, &buffer[ i + 10 ] );        \
  KEY_COPY_16B( data7, &buffer[ i + 12 ] );        \
  KEY_COPY_16B( data8, &buffer[ i + 14 ] );        \
  b12 = KEY_COMPARE_16B( data1, data2 );         \
  b13 = KEY_COMPARE_16B( data1, data3 );         \
  b14 = KEY_COMPARE_16B( data1, data4 );         \
  b15 = KEY_COMPARE_16B( data1, data5 );         \
  b16 = KEY_COMPARE_16B( data1, data6 );         \
  b17 = KEY_COMPARE_16B( data1, data7 );         \
  b18 = KEY_COMPARE_16B( data1, data8 );         \
  b23 = KEY_COMPARE_16B( data2, data3 );         \
  b24 = KEY_COMPARE_16B( data2, data4 );         \
  b25 = KEY_COMPARE_16B( data2, data5 );         \
  b26 = KEY_COMPARE_16B( data2, data6 );         \
  b27 = KEY_COMPARE_16B( data2, data7 );         \
  b28 = KEY_COMPARE_16B( data2, data8 );         \
  b34 = KEY_COMPARE_16B( data3, data4 );         \
  b35 = KEY_COMPARE_16B( data3, data5 );         \
  b36 = KEY_COMPARE_16B( data3, data6 );         \
  b37 = KEY_COMPARE_16B( data3, data7 );         \
  b38 = KEY_COMPARE_16B( data3, data8 );         \
  b45 = KEY_COMPARE_16B( data4, data5 );         \
  b46 = KEY_COMPARE_16B( data4, data6 );         \
  b47 = KEY_COMPARE_16B( data4, data7 );         \
  b48 = KEY_COMPARE_16B( data4, data8 );         \
  b56 = KEY_COMPARE_16B( data5, data6 );         \
  b57 = KEY_COMPARE_16B( data5, data7 );         \
  b58 = KEY_COMPARE_16B( data5, data8 );         \
  b67 = KEY_COMPARE_16B( data6, data7 );         \
  b68 = KEY_COMPARE_16B( data6, data8 );         \
  b78 = KEY_COMPARE_16B( data7, data8 );         \
  KEY_COPY_16B( &buffer[ i + ( b12 + b13 + b14 + b15 + b16 + b17 + b18 ) * 2 ], data1 );      \
  KEY_COPY_16B( &buffer[ i + ( 1 - b12 + b23 + b24 + b25 + b26 + b27 + b28 ) * 2 ], data2 ); \
```

```
  KEY_COPY_16B( &buffer[ i + ( 2 - b13 - b23 + b34 + b35 + b36 + b37 + b38 )
       * 2 ], data3 ); \
  KEY_COPY_16B( &buffer[ i + ( 3 - b14 - b24 - b34 + b45 + b46 + b47 + b48 )
       * 2 ], data4 ); \
  KEY_COPY_16B( &buffer[ i + ( 4 - b15 - b25 - b35 - b45 + b56 + b57 + b58 )
       * 2 ], data5 ); \
  KEY_COPY_16B( &buffer[ i + ( 5 - b16 - b26 - b36 - b46 - b56 + b67 + b68 )
       * 2 ], data6 ); \
  KEY_COPY_16B( &buffer[ i + ( 6 - b17 - b27 - b37 - b47 - b57 - b67 + b78 )
       * 2 ], data7 ); \
  KEY_COPY_16B( &buffer[ i + ( 7 - b18 - b28 - b38 - b48 - b58 - b68 - b78 )
       * 2 ], data8 );



/** Merges two arrays to the right (starting from the end)
 *  Bit-based version
 */
< ASSIGN BYTES 8 16 24 32 >
< P1 >
< FUN inline void merge_BYTESb_LOOP_UNROLL_FACTOR_MERGE_RIGHT_right >( const
     ureg_t *s1, const ureg_t *s2, ureg_t *dest, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN inline void merge_BYTESb_right >( const ureg_t *s1, const ureg_t *s2,
     ureg_t *dest, const ureg_t size ) {
< _P2 >
    ureg_t i = size, j = size;
    ureg_t a[ 2 < SIZE_SCALING > ];
    ureg_t b;

    s1 -= < BYTES_DIVIDED_BY_8 >;
    s2 -= < BYTES_DIVIDED_BY_8 >;
    dest -= < BYTES_DIVIDED_BY_8 >;

    #if ( IF_BASED_MERGER == 1 )

    KEY_COPY_16B( a, s1 ); /* use local vars to avoid load-and-store */
    KEY_COPY_16B( a + < BYTES_DIVIDED_BY_8 >, s2 ); /* use local vars to
         avoid load-and-store */

    for ( ureg_t k = size/< LOOP_UNROLL_FACTOR_MERGE_RIGHT >; k > 0; k-- ) {
      < MERGERIGHTLOOP if ( KEY_COMPARE_< BYTES >B( a, a + BYTES_DIVIDED_BY_8
           ) ) {
        KEY_COPY_< BYTES >B( dest, s1 );
        i--;
        s1 -= BYTES_DIVIDED_BY_8;
        KEY_COPY_< BYTES >B( a, s1 );
      }
      else {
         KEY_COPY_< BYTES >B( dest, s2 );
         j--;
         s2 -= BYTES_DIVIDED_BY_8;
         KEY_COPY_< BYTES >B( a + BYTES_DIVIDED_BY_8, s2 );
      }
      dest -= BYTES_DIVIDED_BY_8; _LOOP >
    }

    while ( ( i > 0 ) & ( j > 0 ) ) {
      if ( KEY_COMPARE_< BYTES >B( a, a + < BYTES_DIVIDED_BY_8 > ) ) {
        KEY_COPY_< BYTES >B( dest, s1 );
        i--;
        s1 -= < BYTES_DIVIDED_BY_8 >;
        KEY_COPY_< BYTES >B( a, s1 );
      }
      else {
        KEY_COPY_< BYTES >B( dest, s2 );
        j--;
        s2 -= < BYTES_DIVIDED_BY_8 >;
        KEY_COPY_< BYTES >B( a + < BYTES_DIVIDED_BY_8 >, s2 );
      }
      dest -= < BYTES_DIVIDED_BY_8 >;
    }
```

```
#else
     while ( ( i > < LOOP_UNROLL_FACTOR_MERGE_RIGHT >-1 ) & ( j > < LOOP_UNROLL_FACTOR_MERGE_RIGHT >-1 ) ) {
       < MERGERIGHTLOOP KEY_COPY_< BYTES >B( a, s1 );
       KEY_COPY_< BYTES >B( a + < BYTES_DIVIDED_BY_8 >, s2 );
       b = 1 - KEY_COMPARE_< BYTES >B( a, a + < BYTES_DIVIDED_BY_8 > );
       KEY_COPY_< BYTES >B( dest, a + b * < BYTES_DIVIDED_BY_8 > );
       dest -= < BYTES_DIVIDED_BY_8 >;
       i -= 1 - b;
       s1 -= (1 - b) * < BYTES_DIVIDED_BY_8 >;
       j -= b;
       s2 -= b * < BYTES_DIVIDED_BY_8 >; _LOOP >
    }
#endif

    while ( i-- > 0 ) {
      KEY_COPY_< BYTES >B( dest, s1 );
      dest -= < BYTES_DIVIDED_BY_8 >;
      s1 -= < BYTES_DIVIDED_BY_8 >;
    }
    while ( j-- > 0 ) {
      KEY_COPY_< BYTES >B( dest, s2 );
      dest -= < BYTES_DIVIDED_BY_8 >;
      s2 -= < BYTES_DIVIDED_BY_8 >;
    }
}< _FUN >

/** Sorts an array using a classical mergesort, quasi-in-place (uses 1.5 times the input size)
 *  \param input  the source array
 *  \param output the destination array; must be allocated
 *  \param size   number of elements
 *  \return 0 if the sorted data is in input, 1 if in output
 */
< SELECT SORTER 3 > // Ignore this block if the value of sorter is neither 0 nor 3
< ASSIGN BYTES 8 16 24 32 >
< P1 >
//   < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
   < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
< FUN ureg_t sorter_BYTESb_quasi_in_place_CHUNK_SIZE_wave_LOOP_UNROLL_FACTOR_CHUNK >( ureg_t *input, ureg_t *output, const ureg_t size ) {
   < INFO >
< _P1 >
< P2 >
< FUN ureg_t sorter_BYTESb_quasi_in_place_wave >( ureg_t *input, ureg_t *output, const ureg_t size ) {
< _P2 >
    const ureg_t start = log_ceiling( < CHUNK_SIZE > );
    const ureg_t stop = log_ceiling( size );

    ureg_t count = 0;

< P1 >    chunk_sorter_< BYTES >b_< CHUNK_SIZE >_< LOOP_UNROLL_FACTOR_CHUNK >( input, size );< _P1 >
< P2 >    chunk_sorter_< BYTES >b( input, size );< _P2 >

    ureg_t l = start;
    ureg_t s;

    while ( l < < BYTES_DIVIDED_BY_8 > * ( ( stop - 1 ) / < BYTES_DIVIDED_BY_8 > ) ) {

      /* merge pairs of subarrays of size 2^l */
      for ( s = 0; s < size; ) {
< P1 >      merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( &input[ s * < BYTES_DIVIDED_BY_8 > ], &input[ ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P1 >
< P2 >      merge_< BYTES >b( &input[ s * < BYTES_DIVIDED_BY_8 > ], &input[ ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P2 >
```

```

      s += 1 << (l + 1);
    }
    input = &input[ - (1 << l) * < BYTES_DIVIDED_BY_8 > + size * < BYTES_DIVIDED_BY_8 > ];
    l++;

    for ( s = 0; s < size; ) {
< P1 >      merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >_right( &input[ -s * < BYTES_DIVIDED_BY_8 >  ], &input[ - ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ - ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P1 >
< P2 >      merge_< BYTES >b_right( &input[ -s * < BYTES_DIVIDED_BY_8 >  ], &input[ - ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ - ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P2 >

      s += 1 << (l + 1);
    }
    input = &input[ (1 << l) * < BYTES_DIVIDED_BY_8 > - size * < BYTES_DIVIDED_BY_8 > ];
    l++;

  }

  if ( ( stop - 1 - start ) % 2 == 1 ) {
    for ( s = 0; s < size; ) {
< P1 >      merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( &input[ s * < BYTES_DIVIDED_BY_8 > ], &input[ ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P1 >
< P2 >      merge_< BYTES >b( &input[ s * < BYTES_DIVIDED_BY_8 > ], &input[ ( s + (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], &input[ ( s - (1 << l) ) * < BYTES_DIVIDED_BY_8 > ], (1 << l) );< _P2 >

      s += 1 << (l + 1);
    }
    input = &input[ - (1 << l) * < BYTES_DIVIDED_BY_8 > ];
    l++;
  }

< P1 >    merge_< BYTES >b_< LOOP_UNROLL_FACTOR_MERGE >( input, &input[ ( 1 << l ) * < BYTES_DIVIDED_BY_8 > ], output, ( 1 << l ) );< _P1 >
< P2 >    merge_< BYTES >b( input, &input[ ( 1 << l ) * < BYTES_DIVIDED_BY_8 > ], output, ( 1 << l ) );< _P2 >

  return ( 1 );

}< _FUN >
< P1 >
   < RESTORE LOOP_UNROLL_FACTOR_MERGE >
   < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >
//   < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
< _P1 >
< _SELECT >


/***************************************************
 *      Specific code for 24-byte extended keys
 **************************************************/

#if ( ENABLE_24B == 0 ) /* If not enabled, use empty functions */

ureg_t sorter_24b( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}
ureg_t sorter_24b_quasi_in_place( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}

#else

#define CHUNK_SORT_24B chunk_sorter_24b
#define MERGE_24B merge_24b
```

```
/* Used for first-level cache sorters; assumes that variables  b0 ... bN,
    data1, ..., dataM  exist */
#define MICRO_SORT_24B_2( i )          \
  KEY_COPY_24B( data1, &buffer[ i ] );          \
  KEY_COPY_24B( data2, &buffer[ i + 3 ] );         \
  b12 = KEY_COMPARE_24B( data1, data2 );          \
  KEY_COPY_24B( &buffer[ i + b12 * 3 ], data1 );         \
  KEY_COPY_24B( &buffer[ i + ( 1 - b12 ) * 3 ], data2 );

/* Used for first-level cache sorters; assumes that variables  b0 ... bN,
    data1, ..., dataM  exist */
#define MICRO_SORT_24B( i )          \
  KEY_COPY_24B( data1, &buffer[ i + 0 * 3 ] );         \
  KEY_COPY_24B( data2, &buffer[ i + 1 * 3 ] );         \
  KEY_COPY_24B( data3, &buffer[ i + 2 * 3 ] );         \
  KEY_COPY_24B( data4, &buffer[ i + 3 * 3 ] );         \
  b12 = KEY_COMPARE_24B( data1, data2 );          \
  b13 = KEY_COMPARE_24B( data1, data3 );          \
  b14 = KEY_COMPARE_24B( data1, data4 );          \
  b23 = KEY_COMPARE_24B( data2, data3 );          \
  b24 = KEY_COMPARE_24B( data2, data4 );          \
  b34 = KEY_COMPARE_24B( data3, data4 );          \
  KEY_COPY_24B( &buffer[ i + ( b12 + b13 + b14 ) * 3 ], data1 );    \
  KEY_COPY_24B( &buffer[ i + ( 1 - b12 + b23 + b24 ) * 3 ], data2 );  \
  KEY_COPY_24B( &buffer[ i + ( 2 - b13 - b23 + b34 ) * 3 ], data3 );  \
  KEY_COPY_24B( &buffer[ i + ( 3 - b14 - b24 - b34 ) * 3 ], data4 );


/* Used for first-level cache sorters; assumes that variables  b0 ... bN,
    data1, ..., dataM  exist */
#define MICRO_SORT_24B_4( i )          \
  KEY_COPY_24B( data1, &buffer[ i ] );          \
  KEY_COPY_24B( data2, &buffer[ i + 3 ] );         \
  KEY_COPY_24B( data3, &buffer[ i + 6 ] );         \
  KEY_COPY_24B( data4, &buffer[ i + 9 ] );         \
  KEY_COPY_24B( data5, &buffer[ i + 12 ] );         \
  KEY_COPY_24B( data6, &buffer[ i + 15 ] );         \
  KEY_COPY_24B( data7, &buffer[ i + 18 ] );         \
  KEY_COPY_24B( data8, &buffer[ i + 21 ] );         \
  b12 = KEY_COMPARE_24B( data1, data2 );         \
  b13 = KEY_COMPARE_24B( data1, data3 );         \
  b14 = KEY_COMPARE_24B( data1, data4 );         \
  b15 = KEY_COMPARE_24B( data1, data5 );         \
  b16 = KEY_COMPARE_24B( data1, data6 );         \
  b17 = KEY_COMPARE_24B( data1, data7 );         \
  b18 = KEY_COMPARE_24B( data1, data8 );         \
  b23 = KEY_COMPARE_24B( data2, data3 );         \
  b24 = KEY_COMPARE_24B( data2, data4 );         \
  b25 = KEY_COMPARE_24B( data2, data5 );         \
  b26 = KEY_COMPARE_24B( data2, data6 );         \
  b27 = KEY_COMPARE_24B( data2, data7 );         \
  b28 = KEY_COMPARE_24B( data2, data8 );         \
  b34 = KEY_COMPARE_24B( data3, data4 );         \
  b35 = KEY_COMPARE_24B( data3, data5 );         \
  b36 = KEY_COMPARE_24B( data3, data6 );         \
  b37 = KEY_COMPARE_24B( data3, data7 );         \
  b38 = KEY_COMPARE_24B( data3, data8 );         \
  b45 = KEY_COMPARE_24B( data4, data5 );         \
  b46 = KEY_COMPARE_24B( data4, data6 );         \
  b47 = KEY_COMPARE_24B( data4, data7 );         \
  b48 = KEY_COMPARE_24B( data4, data8 );         \
  b56 = KEY_COMPARE_24B( data5, data6 );         \
  b57 = KEY_COMPARE_24B( data5, data7 );         \
  b58 = KEY_COMPARE_24B( data5, data8 );         \
  b67 = KEY_COMPARE_24B( data6, data7 );         \
  b68 = KEY_COMPARE_24B( data6, data8 );         \
  b78 = KEY_COMPARE_24B( data7, data8 );         \
  KEY_COPY_24B( &buffer[ i + ( b12 + b13 + b14 + b15 + b16 + b17 + b18 ) * 3
      ], data1 );     \
  KEY_COPY_24B( &buffer[ i + ( 1 - b12 + b23 + b24 + b25 + b26 + b27 + b28 )
      * 3 ], data2 ); \
  KEY_COPY_24B( &buffer[ i + ( 2 - b13 - b23 + b34 + b35 + b36 + b37 + b38 )
      * 3 ], data3 ); \
```

```
  KEY_COPY_24B( &buffer[ i + ( 3 - b14 - b24 - b34 + b45 + b46 + b47 + b48 ) * 3 ], data4 ); \
  KEY_COPY_24B( &buffer[ i + ( 4 - b15 - b25 - b35 - b45 + b56 + b57 + b58 ) * 3 ], data5 ); \
  KEY_COPY_24B( &buffer[ i + ( 5 - b16 - b26 - b36 - b46 - b56 + b67 + b68 ) * 3 ], data6 ); \
  KEY_COPY_24B( &buffer[ i + ( 6 - b17 - b27 - b37 - b47 - b57 - b67 + b78 ) * 3 ], data7 ); \
  KEY_COPY_24B( &buffer[ i + ( 7 - b18 - b28 - b38 - b48 - b58 - b68 - b78 ) * 3 ], data8 );




#endif


/****************************************************
 *      Specific code for 32-byte extended keys
 ****************************************************/

#if ( ENABLE_32B == 0 ) /* If not enabled, use empty functions */

ureg_t sorter_32b( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}
ureg_t sorter_32b_quasi_in_place( ureg_t *input, ureg_t *output, const ureg_t size ) {
  return 0;
}

#else

#define CHUNK_SORT_32B chunk_sorter_32b
#define MERGE_32B merge_32b

/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_32B_2( i )            \
  KEY_COPY_32B( data1, &buffer[ i ] );          \
  KEY_COPY_32B( data2, &buffer[ i + 4 ] );        \
  b12 = KEY_COMPARE_32B( data1, data2 );          \
  KEY_COPY_32B( &buffer[ i + b12 * 4 ], data1 );        \
  KEY_COPY_32B( &buffer[ i + ( 1 - b12 ) * 4 ], data2 );

/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_32B( i )            \
  KEY_COPY_32B( data1, &buffer[ i + 0 * 4 ] );        \
  KEY_COPY_32B( data2, &buffer[ i + 1 * 4 ] );        \
  KEY_COPY_32B( data3, &buffer[ i + 2 * 4 ] );        \
  KEY_COPY_32B( data4, &buffer[ i + 3 * 4 ] );        \
  b12 = KEY_COMPARE_32B( data1, data2 );          \
  b13 = KEY_COMPARE_32B( data1, data3 );          \
  b14 = KEY_COMPARE_32B( data1, data4 );          \
  b23 = KEY_COMPARE_32B( data2, data3 );          \
  b24 = KEY_COMPARE_32B( data2, data4 );          \
  b34 = KEY_COMPARE_32B( data3, data4 );          \
  KEY_COPY_32B( &buffer[ i + ( b12 + b13 + b14 ) * 4 ], data1 );    \
  KEY_COPY_32B( &buffer[ i + ( 1 - b12 + b23 + b24 ) * 4 ], data2 );  \
  KEY_COPY_32B( &buffer[ i + ( 2 - b13 - b23 + b34 ) * 4 ], data3 );  \
  KEY_COPY_32B( &buffer[ i + ( 3 - b14 - b24 - b34 ) * 4 ], data4 );


/* Used for first-level cache sorters; assumes that variables  b0 ... bN, data1, ..., dataM  exist */
#define MICRO_SORT_32B_4( i )            \
  KEY_COPY_32B( data1, &buffer[ i + 0 * 4 ] );        \
  KEY_COPY_32B( data2, &buffer[ i + 1 * 4 ] );        \
  KEY_COPY_32B( data3, &buffer[ i + 2 * 4 ] );        \
  KEY_COPY_32B( data4, &buffer[ i + 3 * 4 ] );        \
  KEY_COPY_32B( data5, &buffer[ i + 4 * 4 ] );        \
  KEY_COPY_32B( data6, &buffer[ i + 5 * 4 ] );        \
  KEY_COPY_32B( data7, &buffer[ i + 6 * 4 ] );        \
```

```
  KEY_COPY_32B( data8, &buffer[ i + 7 * 4 ] );        \
  b12 = KEY_COMPARE_32B( data1, data2 );        \
  b13 = KEY_COMPARE_32B( data1, data3 );        \
  b14 = KEY_COMPARE_32B( data1, data4 );        \
  b15 = KEY_COMPARE_32B( data1, data5 );        \
  b16 = KEY_COMPARE_32B( data1, data6 );        \
  b17 = KEY_COMPARE_32B( data1, data7 );        \
  b18 = KEY_COMPARE_32B( data1, data8 );        \
  b23 = KEY_COMPARE_32B( data2, data3 );        \
  b24 = KEY_COMPARE_32B( data2, data4 );        \
  b25 = KEY_COMPARE_32B( data2, data5 );        \
  b26 = KEY_COMPARE_32B( data2, data6 );        \
  b27 = KEY_COMPARE_32B( data2, data7 );        \
  b28 = KEY_COMPARE_32B( data2, data8 );        \
  b34 = KEY_COMPARE_32B( data3, data4 );        \
  b35 = KEY_COMPARE_32B( data3, data5 );        \
  b36 = KEY_COMPARE_32B( data3, data6 );        \
  b37 = KEY_COMPARE_32B( data3, data7 );        \
  b38 = KEY_COMPARE_32B( data3, data8 );        \
  b45 = KEY_COMPARE_32B( data4, data5 );        \
  b46 = KEY_COMPARE_32B( data4, data6 );        \
  b47 = KEY_COMPARE_32B( data4, data7 );        \
  b48 = KEY_COMPARE_32B( data4, data8 );        \
  b56 = KEY_COMPARE_32B( data5, data6 );        \
  b57 = KEY_COMPARE_32B( data5, data7 );        \
  b58 = KEY_COMPARE_32B( data5, data8 );        \
  b67 = KEY_COMPARE_32B( data6, data7 );        \
  b68 = KEY_COMPARE_32B( data6, data8 );        \
  b78 = KEY_COMPARE_32B( data7, data8 );        \
  KEY_COPY_32B( &buffer[ i + ( b12 + b13 + b14 + b15 + b16 + b17 + b18 ) * 4 ], data1 );     \
  KEY_COPY_32B( &buffer[ i + ( 1 - b12 + b23 + b24 + b25 + b26 + b27 + b28 ) * 4 ], data2 ); \
  KEY_COPY_32B( &buffer[ i + ( 2 - b13 - b23 + b34 + b35 + b36 + b37 + b38 ) * 4 ], data3 ); \
  KEY_COPY_32B( &buffer[ i + ( 3 - b14 - b24 - b34 + b45 + b46 + b47 + b48 ) * 4 ], data4 ); \
  KEY_COPY_32B( &buffer[ i + ( 4 - b15 - b25 - b35 - b45 + b56 + b57 + b58 ) * 4 ], data5 ); \
  KEY_COPY_32B( &buffer[ i + ( 5 - b16 - b26 - b36 - b46 - b56 + b67 + b68 ) * 4 ], data6 ); \
  KEY_COPY_32B( &buffer[ i + ( 6 - b17 - b27 - b37 - b47 - b57 - b67 + b78 ) * 4 ], data7 ); \
  KEY_COPY_32B( &buffer[ i + ( 7 - b18 - b28 - b38 - b48 - b58 - b68 - b78 ) * 4 ], data8 );



#endif

< P1 >
void randomOperations(ureg_t *buffer, ureg_t num_recs){
   for ( ureg_t i = 0; i < num_recs; i++ ) {
      buffer[ i ] = (i % 16 + 1)*rand();
      buffer[ i ] *= buffer[ num_recs ] + buffer[ num_recs/2 ];
   }
}

void generateRecords(ureg_t *buffer, ureg_t num_recs){
   for ( ureg_t i = 0; i < num_recs; i++ ) {
      buffer[ i ] = rand()%(1024);
   }
}

int checkSort(ureg_t *buffer, ureg_t num_recs, int bytes) {
   bytes /= 8;
   for (ureg_t i=0; i<(num_recs-2)*bytes; i+=bytes){
     double j = buffer[i+bytes]-buffer[i];
     if (j<0){      cout << endl << "Indice␣=␣"<<i<<"␣Valore␣=␣"<<buffer[i] << endl<< "Indice␣=␣"<<(i+bytes)<<"␣Valore␣=␣" << endl; return 0;}
   }
   return 1;
}
```

```
#define RANDOM_OPERATIONS randomOperations(buffer, (input_size)) // randomOperations(buffer, (num_recs*BYTES/8)/10)

int main( int argc, char **argv ) {
   double elapsed;                // variable used to store the execution time
   ureg_t tuning_level_factor = < TUNING_LEVEL >;
   if (tuning_level_factor==3) tuning_level_factor=4;
   ureg_t num_recs, input_size = 1024*1024; //tuning_level_factor*4*1024*1024;   // 16Mb * BYTES (it must be a power of 2)
   ureg_t *buffer = new ureg_t[input_size*32]; // 4*1.5
   ureg_t *bufferOut = new ureg_t[input_size*2];
   ureg_t num_executions;
   ureg_t chunk_tuned[4], chunk_current;
   ureg_t sorter_tuned[4]; // 1 == sorter, 2 == sorter_quasi_in_place
   ureg_t loopChunk_tuned[4], loopChunk_current;
   ureg_t loopMerge_tuned[4], loopMerge_current;
   ureg_t loopMergeRight_tuned[4], loopMergeRight_current;
   ureg_t bytes_current; // 8, 16, 24, 32
   double bandBest[4], bandTmp;
   for (int i=0; i<4; i++){ bandBest[i]=0; chunk_tuned[i]=0; loopChunk_tuned[i]=0; loopMerge_tuned[i]=0; loopMergeRight_tuned[i]=0; }

   FILE *fResults;
   if((fResults=fopen("tunerFinal.sh", "w")) == NULL) {
    printf("Error:␣Cannot␣open␣file␣for␣writing.\n");
    exit(1);
   }

   // This first execution could be slower because of the cpu energy saving
      generateRecords(buffer, input_size*32);
      sorter_8b_4_1( (ureg_t *)(buffer), (ureg_t *)(bufferOut), (ureg_t)(input_size/16) );


   // Valuating the best chunk_size value (LOOP_UNROLL_FACTOR are arbitrarily fixed) - first sorter function
     if (< TUNING_LEVEL > == 1) num_executions = 20;
     else if (< TUNING_LEVEL > == 2) num_executions = 20;
     else if (< TUNING_LEVEL > == 3) num_executions = 200;
     cout << "Tuning␣Chunk_size␣using␣sorter_8b,␣sorter_16b,␣...␣" << endl;
     cout << "Bytes\tChunk_size\ttime[s]\t\tband(obj/sec)" << endl;
     < ASSIGN BYTES 8 16 24 32 >
//     < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 >  // arbitrary - these values have to be the same setted before the relative sorter function
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
     < FUN prinit( 1, "sorter_eval" ); num_recs=input_size/(BYTES/8); for (int i=0; i<num_executions; i++){ generateRecords(buffer, num_recs*BYTES/8); RANDOM_OPERATIONS; prstart(1); sorter_BYTESb_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK ( (ureg_t *)(buffer), (ureg_t *)(bufferOut), (ureg_t)(num_recs/2) ); pracc(1); /*cout << checkSort((ureg_t *)(bufferOut), (ureg_t)(num_recs), BYTES);*/} >
       prteval( 1, &elapsed );
       prterm();
       bandTmp = num_recs/2 * num_executions / elapsed;
       chunk_current = < CHUNK_SIZE >;
        bytes_current = < BYTES_DIVIDED_BY_8 > -1;
       cout << < BYTES > << "\t" << chunk_current;
       cout << "\t\t" << elapsed << "\t\t" << bandTmp << endl;
       if (bandTmp > bandBest[bytes_current]) {
         bandBest[bytes_current] = bandTmp;
         sorter_tuned[bytes_current] = 1;
         chunk_tuned[bytes_current] = chunk_current;
         loopChunk_tuned[bytes_current] = < LOOP_UNROLL_FACTOR_CHUNK >;
       }
     < _FUN >
//     < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >
        /*  for (int i=0; i<4; i++)
             cout << endl << "The best value with " << ((i+1)*8) << "b is Chunk_size = " << chunk_tuned[i] << endl; */
          cout << endl;
```

```

    // Valuating the best chunk_size value (LOOP_UNROLL_FACTOR are arbitrarily fixed) - second sorter function
     cout << "Tuning␣Chunk_size␣using␣sorter_8b_quasi_in_place,␣sorter_16b_quasi_in_place,␣...␣" << endl;
     cout << "Bytes\tChunk_size\ttime[s]\t\tband(obj/sec)" << endl;
     < ASSIGN BYTES 8 16 24 32 >
//      < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 > // arbitrary - these values have to be the same setted before the relative sorter function
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
     < FUN prinit( 1, "sorter_eval" ); num_recs=input_size/(BYTES/8); for ( int i=0; i<num_executions; i++){ generateRecords((ureg_t *)(buffer), num_recs*BYTES/8); RANDOM_OPERATIONS; prstart(1); sorter_BYTESb_quasi_in_place_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK ((ureg_t *)(buffer), (ureg_t *)(bufferOut), (ureg_t)(num_recs/2) ); pracc(1); /*cout << checkSort((ureg_t *)(bufferOut), (ureg_t)(num_recs), BYTES); */} >
       prteval( 1, &elapsed );
       prterm();
       bandTmp = num_recs/2 * num_executions / elapsed;
       chunk_current = < CHUNK_SIZE >;
        bytes_current = < BYTES_DIVIDED_BY_8 > -1;
       cout << < BYTES > << "\t" << chunk_current;
       cout << "\t\t" << elapsed << "\t\t" << bandTmp << endl;
       if (bandTmp > bandBest[bytes_current]) {
         bandBest[bytes_current] = bandTmp;
         sorter_tuned[bytes_current] = 2;
         chunk_tuned[bytes_current] = chunk_current;
         loopChunk_tuned[bytes_current] = < LOOP_UNROLL_FACTOR_CHUNK >;
       }
     < _FUN >
//      < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >

      cout << endl;

       // Valuating the best chunk_size value (LOOP_UNROLL_FACTOR are arbitrarily fixed) - third sorter function
     cout << "Tuning␣Chunk_size␣using␣sorter_8b_quasi_in_place_wave,␣sorter_16b_quasi_in_place_wave,␣...␣" << endl;
     cout << "Bytes\tChunk_size\ttime[s]\t\tband(obj/sec)" << endl;
     < ASSIGN BYTES 8 16 24 32 >
//      < ASSIGN LOOP_UNROLL_FACTOR_CHUNK 4 >  // arbitrary - these values have to be the same setted before the relative sorter function
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE 4 >
     < ASSIGN LOOP_UNROLL_FACTOR_MERGE_RIGHT 4 >
     < FUN prinit( 1, "sorter_eval" ); num_recs=input_size/(BYTES/8); for ( int i=0; i<num_executions; i++){ generateRecords((ureg_t *)(buffer), num_recs*BYTES/8); RANDOM_OPERATIONS; prstart(1); sorter_BYTESb_quasi_in_place_CHUNK_SIZE_wave_LOOP_UNROLL_FACTOR_CHUNK ( (ureg_t *)(buffer), (ureg_t *)(bufferOut), (ureg_t)(num_recs/2) ); pracc(1); /*cout << checkSort((ureg_t *)(bufferOut), (ureg_t)(num_recs), BYTES);*/} >
       prteval( 1, &elapsed );
       prterm();
       bandTmp = num_recs/2 * num_executions / elapsed;
       chunk_current = < CHUNK_SIZE >;
        bytes_current = < BYTES_DIVIDED_BY_8 > -1;
       cout << < BYTES > << "\t" << chunk_current;
       cout << "\t\t" << elapsed << "\t\t" << bandTmp << endl;
       if (bandTmp > bandBest[bytes_current]) {
         bandBest[bytes_current] = bandTmp;
         sorter_tuned[bytes_current] = 3;
         chunk_tuned[bytes_current] = chunk_current;
         loopChunk_tuned[bytes_current] = < LOOP_UNROLL_FACTOR_CHUNK >;
       }
     < _FUN >
//      < RESTORE LOOP_UNROLL_FACTOR_CHUNK >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE >
     < RESTORE LOOP_UNROLL_FACTOR_MERGE_RIGHT >
           for (int i=0; i<4; i++){
```

```
            cout << endl << "The best value with " << ((i+1)*8) << "b is Chunk_size = " << chunk_tuned[i];
            if (sorter_tuned[i]==1) cout << " using sorter" << endl;
            else if (sorter_tuned[i]==2) cout << " using sorter_quasi_in_place" << endl;
            else if (sorter_tuned[i]==3) cout << " using sorter_quasi_in_place_wave" << endl;
          }
          cout << endl;

  // Valuating loop_unroll_factorChunk. The best value of chunk_size has already been found
   if (< TUNING_LEVEL > == 1){
     for (int i=0; i<4; i++) bandBest[i]=0;
    if (< TUNING_LEVEL > == 1) num_executions = 20;
    else if (< TUNING_LEVEL > == 2) num_executions = 20;
    else if (< TUNING_LEVEL > == 3) num_executions = 200;
    cout << "Tuning Loop_unroll_factor for the functions chunk_sorter_8b, chunk_sorter_16b, ... " << endl;
    cout << "Bytes\tChunk_size\tLoop_unroll_factorChunk\t\ttime[s]\t\tband(obj/sec)" << endl;
    < ASSIGN BYTES 8 16 24 32 >
       < FUN bytes_current = BYTES /8 -1; num_recs=input_size/(BYTES/8); chunk_current = CHUNK_SIZE; if (chunk_current==chunk_tuned[bytes_current]){ prinit( 1, "sorter_eval" ); for (int i=0; i<num_executions; i++){ generateRecords(buffer, num_recs*BYTES/8); RANDOM_OPERATIONS; prstart(1); chunk_sorter_BYTESb_CHUNK_SIZE_LOOP_UNROLL_FACTOR_CHUNK ((ureg_t *)(buffer), (ureg_t)(num_recs)); pracc(1); } >
          prteval( 1, &elapsed );
          prterm();
          bandTmp = num_recs * num_executions / elapsed;
          loopChunk_current = < LOOP_UNROLL_FACTOR_CHUNK >;
          cout << < BYTES > << "\t" << chunk_current << "\t\t" << loopChunk_current;
          cout << "\t\t\t\t" << elapsed << "\t" << bandTmp << endl;
          if (bandTmp > bandBest[bytes_current]) {
             bandBest[bytes_current] = bandTmp;
             loopChunk_tuned[bytes_current] = loopChunk_current;
             chunk_tuned[bytes_current] = chunk_current;
          }
       }
       < _FUN >
       for (int i=0; i<4; i++)
          cout << endl << "The best value with " << ((i+1)*8) << "b is Loop_unroll_factor_chunk = " << loopChunk_tuned[i] << endl;
       cout << endl;
   }
   // Valuating loop_unroll_factorMerge and loop_unroll_factorMergeRight. The best values of chunk_size and loop_unroll_factor_chunk have already been found
     for (int i=0; i<4; i++) bandBest[i]=0;
    if (< TUNING_LEVEL > == 1) num_executions = 20;
    else if (< TUNING_LEVEL > == 2) num_executions = 20;
    else if (< TUNING_LEVEL > == 3) num_executions = 200;
    cout << "Tuning Loop_unroll_factor for the functions merge_8b, merge_16b, .." << endl;
    cout << "Bytes\tChunk_size\tLoop_unroll_factorMerge\t\ttime[s]\t\tband(obj/sec)" << endl;
    < ASSIGN BYTES 8 16 24 32 >
       < FUN bytes_current = BYTES /8 -1; num_recs=input_size/(BYTES/8); chunk_current = CHUNK_SIZE; if (chunk_current==chunk_tuned[bytes_current]){ prinit( 1, "sorter_eval" ); for (int i=0; i<num_executions; i++){ generateRecords(buffer, num_recs*BYTES/8); RANDOM_OPERATIONS; prstart(1); merge_BYTESb_LOOP_UNROLL_FACTOR_MERGE ((ureg_t *)(buffer), (ureg_t *)(buffer+(num_recs/2)*BYTES_DIVIDED_BY_8), (ureg_t *)(bufferOut), (ureg_t)(num_recs/2)); pracc(1); } >
          prteval( 1, &elapsed );
          prterm();
          bandTmp = num_recs * num_executions / elapsed;
          loopMerge_current = < LOOP_UNROLL_FACTOR_MERGE >;
          cout << < BYTES > << "\t" << chunk_current << "\t\t" << loopMerge_current;
```

```
            cout << "\t\t\t\t" << elapsed << "\t" << bandTmp << endl;
            if (bandTmp > bandBest[bytes_current]) {
               bandBest[bytes_current] = bandTmp;
               loopMerge_tuned[bytes_current] = loopMerge_current;
            }
          }
        < _FUN >
      for (int i=0; i<4; i++){
         cout << endl << "The␣best␣value␣with␣" << ((i+1)*8) << "b␣is␣
              Loop_unroll_factor_merge␣=␣" << loopMerge_tuned[i] << endl;
      }
      cout << endl;

      for (int i=0; i<4; i++) bandBest[i]=0;
      cout << "Tuning␣Loop_unroll_factor␣for␣the␣functions␣merge_8b_right,␣
           merge_16b_right,␣.." << endl;
     cout << "Bytes\tChunk_size\tLoop_unroll_factorMergeRight\t\ttime[s]\t\
          tband(obj/sec)" << endl;
     < ASSIGN BYTES 8 16 24 32 >
        < FUN bytes_current = BYTES /8 -1; num_recs=input_size/(BYTES/8);
             chunk_current = CHUNK_SIZE; if ((chunk_current==chunk_tuned[
             bytes_current])&&(sorter_tuned[bytes_current]==3)){ prinit( 1, "
             sorter_eval" ); for (int i=0; i<num_executions; i++){
             generateRecords(buffer, num_recs*BYTES/8); RANDOM_OPERATIONS;
             prstart(1); merge_BYTESb_LOOP_UNROLL_FACTOR_MERGE_RIGHT_right ((
             ureg_t *)(buffer+(num_recs/2)*BYTES_DIVIDED_BY_8), (ureg_t *)(
             buffer+(num_recs)*BYTES_DIVIDED_BY_8), (ureg_t *)(bufferOut+(
             num_recs)*BYTES_DIVIDED_BY_8), (ureg_t)(num_recs/2)); pracc(1);
             } >
            prteval( 1, &elapsed );
            prterm();
            bandTmp = num_recs * num_executions / elapsed;
            loopMergeRight_current = < LOOP_UNROLL_FACTOR_MERGE_RIGHT >;
            cout << < BYTES > << "\t" << chunk_current << "\t\t" <<
                 loopMergeRight_current;
            cout << "\t\t\t\t" << elapsed << "\t" << bandTmp << endl;
            if (bandTmp > bandBest[bytes_current]) {
               bandBest[bytes_current] = bandTmp;
               loopMergeRight_tuned[bytes_current] = loopMergeRight_current;
            }
          }
        < _FUN >
        for (int i=0; i<4; i++){
           if (sorter_tuned[i]==3)
              cout << endl << "The␣best␣value␣with␣" << ((i+1)*8) << "b␣is␣
                   Loop_unroll_factor_merge_right␣=␣" << loopMergeRight_tuned
                   [i] << endl;
        }
        cout << endl;

    fprintf(fResults, "./codeworker␣-translate␣cache_sorters_tuner.gen␣
         cache_sorters.tun␣cache_sorters.cpp␣-args␣TUNING_PART␣2␣");

    fprintf(fResults, "LOOP_UNROLL_FACTOR_CHUNK␣%u␣%u␣%u␣%u␣
         LOOP_UNROLL_FACTOR_MERGE␣%u␣%u␣%u␣%u␣LOOP_UNROLL_FACTOR_MERGE_RIGHT␣%
         u␣%u␣%u␣%u␣CHUNK_SIZE␣%u␣%u␣%u␣%u␣SORTER␣%u␣%u␣%u␣%u␣COMPARE␣%u␣
         TUNING_LEVEL␣%u",
                     loopChunk_tuned[0], loopChunk_tuned[1], loopChunk_tuned
                          [2], loopChunk_tuned[3],
                     loopMerge_tuned[0], loopMerge_tuned[1], loopMerge_tuned
                          [2], loopMerge_tuned[3],
                     loopMergeRight_tuned[0], loopMergeRight_tuned[1],
                          loopMergeRight_tuned[2], loopMergeRight_tuned[3],
                     chunk_tuned[0], chunk_tuned[1], chunk_tuned[2],
                          chunk_tuned[3],
                     sorter_tuned[0], sorter_tuned[1], sorter_tuned[2],
                          sorter_tuned[3],
                     < COMPARE_TYPE >, < TUNING_LEVEL >);
    fclose(fResults);
}
< _P1 >
```

A.2 inlines.tun

```
#define OR |
#define AND &
< SELECT COMPARE16 1 >
#define OR16 |
#define AND16 &
< _SELECT >
< SELECT COMPARE16 2 >
#define OR16 ||
#define AND16 &&
< _SELECT >
< SELECT COMPARE24 1 >
#define OR24 |
#define AND24 &
< _SELECT >
< SELECT COMPARE24 2 >
#define OR24 ||
#define AND24 &&
< _SELECT >
< SELECT COMPARE32 1 >
#define OR32 |
#define AND32 &
< _SELECT >
< SELECT COMPARE32 2 >
#define OR32 ||
#define AND32 &&
< _SELECT >
[...]
```

A.3 tuner.sh

```
#!/bin/bash
#
#===========================================================================
# Name        : tuner.sh
# Author      : Giovanni Di Liberto
# Description : psort Code Tuning. This script executes the operations
#               necessary to the code tuning in pre-compilation.
#               It tests many critical functions  in order to make the
#               optimal choice and then it writes the final source code.
#               It needed of soma parameters that must be indicated as
#               ARGS.
#===========================================================================
#


# How to use ./codeworker:
# ./codeworker -translate cache_sorters_tuner.gen cache_sorters.cpp
    function_name.c -args loop_unrol_values chunk_size_values


#for i in $(seq $PARINITVALUE $PARINCRVALUE $PARENDVALUE)
#       do

# SETTING DEFAULT VALUES
COMPARE[0]=2;
COMPARE[1]=2;
COMPARE[2]=2;
COMPARE[3]=2;
TUNING_LEVEL=2;

# PARSING ARGS
while [ "$*" != "" ];
   do
      if [ "$1" == "--compare" ] ; then
         for i in $(seq 0 1 3)
          do
            shift
            test -n "$1" && COMPARE[$i]=$1
            if [ "${COMPARE[$i]}" != "1" ]&&[ "${COMPARE[$i]}" != "2" ] ;
                then
```

```
                echo "Value$i␣compare␣incorrect␣-␣it␣has␣been␣assigned␣the␣value␣2"
                COMPARE[$i]=2;
              fi
            done
        elif [ "$1" == "--tuning-level" ]; then
            shift
            test -n "$1" && TUNING_LEVEL=$1
            if [ "$TUNING_LEVEL" != "1" ]&&[ "$TUNING_LEVEL" != "2" ]&&[ "$TUNING_LEVEL" != "3" ] ; then
              echo "Value␣tuning␣incorrect␣-␣it␣has␣been␣assigned␣the␣value␣2"
              TUNING_LEVEL=2;
            fi
        elif [ "$1" == "--help" ]; then
            echo "Usage:␣bash␣tuner.sh␣[--compare␣value]"
            echo ""
            echo "␣␣--compare:␣␣␣␣␣␣it␣needed␣4␣parameters,␣one␣for␣each␣value␣of␣key_length␣(8,␣16,␣24,␣32);"
            echo "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣1␣==␣bitwise␣comparisons,␣2␣==␣logical␣comparisons."
            echo "␣␣--tuning-level:␣1␣==␣fast,␣2␣==␣medium,␣3␣==␣better␣(slow)."
            echo ""
            exit 0
        fi
        shift
    done


# IT SAVES THE OLD BINARY AND THE OLD LOG FILE
touch tuningTests_cache_sorters
mv tuningTests_cache_sorters tuningTests_cache_sorter_old
touch tuningLog_cache_sorters.txt
mv tuningLog_cache_sorters.txt tuningLog_cache_sorters.old
#touch
#tail -f tuningLog_cache_sorters.txt &

# MOVING THE OLD STABLE SOURCE FILES (we don't want to delete the last stable sources)
touch inlines.h
mv inlines.h inlines_old.h
touch cache_sorters.cpp
mv cache_sorters.cpp cache_sorters_old.cpp

# TUNING
echo "***␣inlines.tun␣***"
echo "***␣Translating␣the␣tuning␣file␣to␣a␣C++␣source␣code␣***"
./codeworker -translate cache_sorters_tuner.gen inlines.tun inlines.h -args TUNING_PART 2 COMPARE ${COMPARE[0]} ${COMPARE[1]} ${COMPARE[2]} ${COMPARE[3]} >> tuningLog_cache_sorters.txt 2>&1
echo ""
echo "***␣cache_sorters.tun␣***"
echo "***␣Translating␣the␣tuning␣file␣to␣a␣C++␣source␣code␣***"
./codeworker -translate cache_sorters_tuner.gen cache_sorters.tun cache_sorters_for_tuning.cpp -args TUNING_PART 1 LOOP_UNROLL_FACTOR_CHUNK 1 2 4 8 16 LOOP_UNROLL_FACTOR_MERGE 1 2 4 8 16 LOOP_UNROLL_FACTOR_MERGE_RIGHT 1 2 4 8 16 CHUNK_SIZE 2 4 8 COMPARE ${COMPARE[0]} ${COMPARE[1]} ${COMPARE[2]} ${COMPARE[3]} TUNING_LEVEL $TUNING_LEVEL >> tuningLog_cache_sorters.txt 2>&1
echo "***␣Compiling␣the␣extended␣source␣code␣***"
make tuning >> tuningLog_cache_sorters.txt 2>&1
echo "***␣Executing␣test␣and␣evaluating␣the␣best␣options␣***"
./tuningTests_cache_sorters >> tuningLog_cache_sorters.txt 2>&1
echo "***␣Generating␣the␣optimal␣source␣code␣***"
bash tunerFinal.sh >> tuningLog_cache_sorters.txt 2>&1 # this file is generated by tuningTests_cache_sorters

echo "The␣details␣has␣been␣saved␣in␣tuningLog_cache_sorters.txt"
```

A.4 cache_sorters_tuner.gen

```
//=============================================================================
```

```
// Name        : cache_sorters_tuner.gen
// Author      : Giovanni Di Liberto
// Description : psort translation script for automated code tuning.
//               Translation script: an extended-BNF script that allows
//               generating code in the same time.
//               It extend the source code when it founds some particular
//               tags. Useful when the developer wants to obtain many
//               version of the same code with different values of
//               a param (loop unroll is a typical example)
//============================================================================


// We want to put all scanned characters in the output
#implicitCopy

// The head of the grammar is the first production rule encountered
inlineCodeExpander    ::=
    =>{ global tuningPart=1;                 // 1==tuning code, 2==final code (default 1)
        global loopName;    // It permits to use the same code (the production expandMERGELOOPMarkup) for differents loops
        /* param Loop unroll factor variables */
          global loop_unroll_factorChunk;        // array with the interested values for this param
          global loopChunkCount=0;               // the number of interested values for this param
          global loopChunkCurrent=0;             // the current value
          global loopChunkActive=0;              // !=0 if the current function is related to this param
          global loop_unroll_factorChunk_stored; // when there is an ASSIGN tag the old values must be copied here
        /* param Loop unroll factor merge variables */
          global loop_unroll_factorMerge;
          global loopMergeCount=0;
          global loopMergeCurrent=0;
          global loopMergeActive=0;
          global loop_unroll_factorMerge_stored;
        /* param Loop unroll factor merge right variables */
          global loop_unroll_factorMergeRight;
          global loopMergeRightCount=0;
          global loopMergeRightCurrent=0;
          global loopMergeRightActive=0;
          global loop_unroll_factorMergeRight_stored;
        /* param Chunk size variables */
          global chunk_size;
          global chunkCount=0;
          global chunkCurrent=0;
          global chunkActive=0;
          global chunk_size_stored;

        global bytes;             // psort extended key-length - select the values with, for example < ASSIGN BYTES 8 16 >
        bytes[0]=8;               // default: only 8 bytes
        global bytesCount=1;      // size of the array bytes
        global bytesCurrent=0;

        global functionName;      // the name of the current function (it will be rewrite many times)
        global functionLength=0;
        local  functionCount=0;
        global isFunctionStarted=0;  // if we are between two tags < FUN ... > and < _FUN >
        global strFunctionName="";

        global sorter;            // array[4] - the IDs of the best sorter functions for each value of bytes
        global sorterCount=0;
        global sorterCurrent=0;
        global tuningLevel=2;   // 1 = fast, 2 = medium, 3 = better (slow)
        global compareType;     // 1 = bitwise, 2 = logical
        compareType[0]=2; compareType[1]=2; compareType[2]=2; compareType[3]=2;
        global compareCurrent;
```

```
        compareCurrent[0]=0; compareCurrent[1]=0; compareCurrent[2]=0; compareCurrent[3]=0;
        global compareCount=0;
        // sorter has to be init to 0 0 0 0 (the SELECT tag with the SORTER variable will not ignore)
        sorter[0]=0; sorter[1]=0; sorter[2]=0; sorter[3]=0;

        global iStartInputPosition=0; // useful when we want to overwrite a tag in the generated file.
    }

    // Parsing _ARGS
    => { local current_variable=0;
         foreach i in _ARGS{
           //traceText(i+endl());
           switch(i){
             case "LOOP_UNROLL_FACTOR_CHUNK":       set current_variable=1; break; // it admits an array of values
             case "CHUNK_SIZE":                     set current_variable=2; break; // it admits an array of values
             case "TUNING_PART":                    set current_variable=3; break; // it admits only one value
             case "LOOP_UNROLL_FACTOR_MERGE":       set current_variable=4; break; // it admits an array of values
             case "SORTER":                         set current_variable=5; break; // it admits 4 values (one for each value of bytes)
             case "LOOP_UNROLL_FACTOR_MERGE_RIGHT": set current_variable=6; break; // it admits an array of values
             case "COMPARE":                        set current_variable=7; break; // it admits 4 values
             case "TUNING_LEVEL":                   set current_variable=8; break; // it admits one value
             default:
               switch(current_variable){
                 case "1": loop_unroll_factorChunk[loopChunkCount]=i;
                           loopChunkCount = $loopChunkCount+1$;
                           break;
                 case "2": chunk_size[chunkCount]=i;
                           chunkCount = $chunkCount+1$;
                           break;
                 case "3": tuningPart=i;
                           break;
                 case "4": loop_unroll_factorMerge[loopMergeCount]=i;
                           loopMergeCount = $loopMergeCount+1$;
                           break;
                 case "5": sorter[sorterCount]=i;
                           sorterCount = $sorterCount+1$;
                           break;
                 case "6": loop_unroll_factorMergeRight[loopMergeRightCount]=i;
                           loopMergeRightCount = $loopMergeRightCount+1$;
                           break;
                 case "7": compareType[compareCount]=i;
                           compareCount = $compareCount+1$;
                           break;
                 case "8": tuningLevel=i;
                           break;
               }
           }
         }
       }
  #ignore(C++)    // ignore C++ comments and whitespaces between terminals and non-terminals.
  [
     => local iOutputCurrentPosition = getOutputLocation();
        #readCString     // Jump over strings
     |
        // Handle a markup
        '<'
        // The script keeps the position just before '<'
        => local iStartPosition = $getOutputLocation() - 1$;
        // from now, scanned characters aren't put in the output
        #explicitCopy
        [
```

```
        #readIdentifier:"FUN"[ // function start
        #implicitCopy
        [
              => {
                 setOutputLocation($iStartPosition$);
                 /* If it is writing a sorter function and the best sorter function are specified then we skip
                    the non-optimal combination */
                   bytesCurrent=0;
                   if (sorterCurrent>0){
                     while ((bytesCurrent<4)&&(sorter[bytesCurrent]!=sorterCurrent)&&(sorter[bytesCurrent]!=0))
                       bytesCurrent=$bytesCurrent+1$;
                   }
                 if (bytesCurrent<4){
                    if (tuningPart==2){ // If it's generating the final code
                      chunkActive=1;
                      loopChunkActive=1;
                      loopMergeActive=1;
                      loopMergeRightActive=1;
                    }
                    isFunctionStarted=1;
                    chunkCurrent=0;
                    loopChunkCurrent=0;
                    loopMergeCurrent=0;
                    loopMergeRightCurrent=0;

                    while (lookAhead(" ")) readChar(); // ignore the ' '
                    strFunctionName="";
                    while (!lookAhead(" >")){
                      strFunctionName += readChar();
                    }
                    readChar();readChar();             // jump over the substring " >"
                    iStartInputPosition=$getInputLocation()$;
                 }
              }
              #check(isFunctionStarted==0)[ // jump to the end of the function
                 #explicitCopy
                 [^"_FUN >"]*
                 #explicitCopy
                 "_FUN >"
              ]|[
                 writeFunctionName
              ]
        ]
        ]
        |
        #readIdentifier:"_FUN" [ // function end
           =>{
             if(isFunctionStarted==1){
               setOutputLocation(iStartPosition);
               writeText(endl()+endl());
               if(tuningPart==1){ // tuning part
                 chunkCurrent=$chunkCurrent+1$;
                 if(($chunkCurrent<chunkCount$)&&(chunkActive!=0)){
                   setInputLocation(iStartInputPosition); // it turns back to the start of the block, if it hasn't finished
                 }else{
                   if(chunkActive!=0) chunkCurrent=0;
                   loopChunkCurrent=$loopChunkCurrent+1$;
                   if(($loopChunkCurrent<loopChunkCount$)&&(loopChunkActive!=0)){ // loopChunk and loopMerge won't be used at the same time (they are mutually exclusive)
                     setInputLocation(iStartInputPosition); // it turns back to the start of the block, if it hasn't
                   }else{
                      if(loopChunkActive!=0) loopChunkCurrent=0;
                      loopMergeCurrent=$loopMergeCurrent+1$;
```

```
                              if(($loopMergeCurrent<loopMergeCount$)&&(loopMergeActive!=0)){
                                setInputLocation(iStartInputPosition); // it turns back to the start of the block, if it hasn't
                              }else{
                                 if(loopMergeActive!=0) loopMergeCurrent=0;
                                 loopMergeRightCurrent=$loopMergeRightCurrent+1$;
                                 if(($loopMergeRightCurrent<loopMergeRightCount$)&&(loopMergeRightActive!=0)){
                                   setInputLocation(iStartInputPosition); // it turns back to the start of the block, if it hasn't
                                 }else{
                                    if(loopMergeRightActive!=0) loopMergeRightCurrent=0;
                                    bytesCurrent=$bytesCurrent+1$;
                                    if (sorterCurrent>0){
                                      while ((bytesCurrent<4)&&(sorter[bytesCurrent]!=sorterCurrent)&&(sorter[bytesCurrent]!=0))
                                        bytesCurrent=$bytesCurrent+1$;
                                    }
                                    if(bytesCurrent<$bytesCount$){
                                      setInputLocation(iStartInputPosition); // it turns back to the start of the block, if it hasn't
                                      loopChunkCurrent=0;
                                      loopMergeCurrent=0;
                                      loopMergeRightCurrent=0;
                                      chunkCurrent=0;
                                    }else { isFunctionStarted=0; readChar(); readChar();}
                           }}}}
                        }else{
                           bytesCurrent=$bytesCurrent+1$;
                           if (sorterCurrent>0){
                             while ((bytesCurrent<4)&&(sorter[bytesCurrent]!=sorterCurrent)&&(sorter[bytesCurrent]!=0))
                               bytesCurrent=$bytesCurrent+1$;
                           }
                           if(bytesCurrent<bytesCount){ // if (tuningPart != 1)
                             setInputLocation(iStartInputPosition); // torno all'inizio del blocco, se necessario
                             loopChunkCurrent=bytesCurrent;
                             loopMergeCurrent=bytesCurrent;
                             loopMergeRightCurrent=bytesCurrent;
                           }else { isFunctionStarted=0; readChar(); readChar();}
                        }
                      }
                   }
                   writeFunctionName
                 ]
                 |
                 #readIdentifier:"INFO"
                    => setOutputLocation(iStartPosition);
                    expandINFOMarkup
                 |
                 #readIdentifier:"DATA_VARIABLES"
                    => setOutputLocation(iStartPosition);
                    expandDATA_VARIABLESMarkup
                 |
                 #readIdentifier:"COMPARE_VARIABLES"
                    => setOutputLocation(iStartPosition);
                    expandCOMPARE_VARIABLESMarkup
                 |
                 #readIdentifier:"STEP"
                    => setOutputLocation(iStartPosition);
                    expandSTEPMarkup
                 |
                 #readIdentifier:"SIZE_SCALING"
                    => setOutputLocation(iStartPosition);
                    expandSIZE_SCALINGMarkup
```

```
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_CHUNK"
           => setOutputLocation(iStartPosition);
           expandLOOP_UNROLL_FACTOR_CHUNKMarkup
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE_RIGHT"
           => setOutputLocation(iStartPosition);
           expandLOOP_UNROLL_FACTOR_MERGE_RIGHTMarkup
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE"
           => setOutputLocation(iStartPosition);
           expandLOOP_UNROLL_FACTOR_MERGEMarkup
        |
        #readIdentifier:"CHUNK_SIZE"
           => setOutputLocation(iStartPosition);
           expandCHUNK_SIZEMarkup
        |
        #readIdentifier:"COMPARE_TYPE"
           => setOutputLocation(iStartPosition);
           expandCOMPARE_TYPEMarkup
        |
        #readIdentifier:"LOOP"
           => setOutputLocation(iStartPosition);
           expandLOOPMarkup
        |
        #readIdentifier:"MERGERIGHTLOOP"
           => setOutputLocation(iStartPosition);
           => loopName="MergeRight";
           expandMERGELOOPMarkup
        |
        #readIdentifier:"MERGELOOP"
           => setOutputLocation(iStartPosition);
           => loopName="Merge";
           expandMERGELOOPMarkup
        |
        #readIdentifier:"ASSIGN"
           => setOutputLocation(iStartPosition);
           expandASSIGNMarkup
        |
        #readIdentifier:"RESTORE"
           => setOutputLocation(iStartPosition);
           expandRESTOREMarkup
        |
        #readIdentifier:"BYTES_DIVIDED_BY_8"
           => setOutputLocation(iStartPosition);
           expandBYTES_DIVIDED_BY_8Markup
        |
        #readIdentifier:"BYTES"
           => setOutputLocation(iStartPosition);
           expandBYTESMarkup
        |
        #readIdentifier:"TUNING_LEVEL"
           => setOutputLocation(iStartPosition);
           expandTUNING_LEVELMarkup
        |
        #readIdentifier:"P1" // only for tuning
         [  => setOutputLocation(iStartPosition);
           #check(tuningPart!=1)[
             #explicitCopy // ignore all the block
             [^"_P1 >"]*
             #explicitCopy
             "_P1 >"
             => if (lookAhead(endl())) readChar();
           ]|[#explicitCopy // ignore only the characters until we find '>'
             [^'>']*
             #explicitCopy
             '>'
           ]
         ]
        |
        #readIdentifier:"_P1" // only for tuning // la '/' è un problema !! forse è un carattere speciale
           => setOutputLocation(iStartPosition);
```

```
                #explicitCopy
                [^'>']*
                #explicitCopy
                '>'
             |
             #readIdentifier:"P2" // only for final source code
              [  => setOutputLocation(iStartPosition);
                #check(tuningPart!=2)[
                  #explicitCopy
                  [^"_P2 >"]*
                  #explicitCopy
                  "_P2 >"
                  => if (lookAhead(endl())) readChar();
                ]|[#explicitCopy
                  [^'>']*
                  #explicitCopy
                  '>'
                ]
              ]
             |
             #readIdentifier:"_P2" // only for tuning // the '/' can't be
                 used
                => setOutputLocation(iStartPosition);
                #explicitCopy
                [^'>']*
                #explicitCopy
                '>'
             |
             #readIdentifier:"SELECT" // it tells which version of the
                 indicated function is parsed
              [ => setOutputLocation(iStartPosition);
                expandSELECTMarkup

                #check(($(compareCurrent[0]!=0)&&(compareCurrent[0]!=
                     compareType[0])$)
                       ||($(compareCurrent[1]!=0)&&(compareCurrent[1]!=
                            compareType[1])$)
                       ||($(compareCurrent[2]!=0)&&(compareCurrent[2]!=
                            compareType[2])$)
                       ||($(compareCurrent[3]!=0)&&(compareCurrent[3]!=
                            compareType[3])$))[ // we ignore this section if
                            it is relative to a compare value and if that isn'
                            t the value we have chosen
                  #explicitCopy
                  [^"_SELECT >"]*
                  #explicitCopy
                  "_SELECT >"
                  => if (lookAhead(endl())) readChar();
                ]|[
                  #explicitCopy
                  [^'>']*
                  // ignore the end of the markup - only one: '>'
                  #explicitCopy
                  '>'
                ]
             ]
             |
             #readIdentifier:"_SELECT"
                => setOutputLocation(iStartPosition);
                =>{ sorterCurrent=0; }
                #explicitCopy
                [^'>']*
                #explicitCopy
                '>'
          ]
      |
          // If not a string and not a markup, reading of any
          // character
          #readChar
  ]*;

// Replaces a LOOP markup with the C++ corresponding code
expandLOOPMarkup    ::=
  => local count=0;
```

```
  => local strRepeat;
  => local strIndex=0;
  => local strCount=0;

  // Copy of the query in the output
  #implicitCopy
  => {
     while (!lookAhead(" _LOOP >")){       // copying in strRepeat each input line until " >" is found
       strRepeat[strCount] += readChar();
       if (lookAhead(endl())){ readChar(); strCount=$strCount+1$; }  // ignore endl and increment the counter
     }

     while $count < loop_unroll_factorChunk[loopChunkCurrent]*chunk_size[chunkCurrent]*(bytes[bytesCurrent]/8)${
       strIndex=0;
       while (strIndex <= strCount){
         if ($count!=0$) || ($count==0$ && $strIndex!=0$) writeText(endl());
         if $strIndex==0$ && $count!=0$ @    @
         local s=strRepeat[strIndex];
         if $count==0$ { s=replaceString(" + INCREMENT", "", s); s=replaceString("+ INCREMENT", "", s); } // ugly but it works
         else          s=replaceString("INCREMENT", count, s);

         s=replaceString("CHUNK_SIZE", chunk_size[chunkCurrent], s);
         s=replaceString("< BYTES_DIVIDED_BY_8 >", $bytes[bytesCurrent]/8$, s);
         s=replaceString("BYTES_DIVIDED_BY_8", $bytes[bytesCurrent]/8$, s);
         s=replaceString("< BYTES >", bytes[bytesCurrent], s);
         s=replaceString("BYTES", bytes[bytesCurrent], s);
         writeText(s);
         strIndex = $strIndex+1$;
       }
       count = $count + chunk_size[chunkCurrent]*(bytes[bytesCurrent]/8)$;
     }}

  // ignore the end of the markup - ne ignora solo 1: '>'
  #explicitCopy
  "_LOOP >"
  ;

// Replaces a MERGELOOP markup with the C++ corresponding code
expandMERGELOOPMarkup    ::=
  => local count=0;
  => local strRepeat;
  => local strIndex=0;
  => local strCount=0;

  // Copy of the query in the output
  #implicitCopy
  => {
     local maxValue;
     if (loopName=="Merge") maxValue=loop_unroll_factorMerge[loopMergeCurrent];
     else if (loopName=="MergeRight") maxValue=loop_unroll_factorMergeRight[loopMergeRightCurrent];
     while (!lookAhead(" _LOOP >")){   // copying in strRepeat each input line until " >" is found
       strRepeat[strCount] += readChar();
       if (lookAhead(endl())){readChar(); strCount=$strCount+1$; }  // ignore endl and increment the counter
     }
     while $count < maxValue${
       strIndex=0;
       while ($strIndex <= strCount$){
         if ($count!=0$) || ($count==0$ && $strIndex!=0$) writeText(endl());
         if $strIndex==0$ && $count!=0$ @      @
         local s=strRepeat[strIndex];
         if $count==0$ { s=replaceString(" + INCREMENT", "", s); s=replaceString("+ INCREMENT", "", s); } // ugly but it works
         else          s=replaceString("INCREMENT", count, s);
         s=replaceString("CHUNK_SIZE", chunk_size[chunkCurrent], s);
```

```
          s=replaceString("< BYTES_DIVIDED_BY_8 >", $bytes[bytesCurrent]/8$, s);
          s=replaceString("BYTES_DIVIDED_BY_8", $bytes[bytesCurrent]/8$, s);
          s=replaceString("< BYTES >", bytes[bytesCurrent], s);
          s=replaceString("BYTES", bytes[bytesCurrent], s);
          writeText(s);
          strIndex = $strIndex+1$;
        }
        count = $count + 1$;
      }}

  // ignore the end of the markup - ne ignora solo 1: '>'
  #explicitCopy
  "_LOOP >"
  ;


// Replaces a STEP markup with the C++ corresponding code
expandSTEPMarkup    ::=
        => {
            if (loopChunkActive!=0)
              writeText($loop_unroll_factorChunk[loopChunkCurrent]*chunk_size[chunkCurrent]*(bytes[bytesCurrent]/8)$);
            else if (loopMergeActive)
              writeText($loop_unroll_factorMerge[loopMergeCurrent]*chunk_size[chunkCurrent]*(bytes[bytesCurrent]/8)$);
            else if (loopMergeRightActive)
              writeText($loop_unroll_factorMergeRight[loopMergeRightCurrent]*chunk_size[chunkCurrent]*(bytes[bytesCurrent]/8)$);
        }
        #implicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a SIZE_SCALING markup with the C++ corresponding code
expandSIZE_SCALINGMarkup    ::=
        => {
            if $(bytes[bytesCurrent]/8)>1$
              writeText("* "+$bytes[bytesCurrent]/8$);
        }
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a LOOP_UNROLL_FACTOR_CHUNK markup with the C++ corresponding code
expandLOOP_UNROLL_FACTOR_CHUNKMarkup    ::=
        => writeText(loop_unroll_factorChunk[loopChunkCurrent]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a LOOP_UNROLL_FACTOR_MERGE markup with the C++ corresponding code
expandLOOP_UNROLL_FACTOR_MERGEMarkup    ::=
        => writeText(loop_unroll_factorMerge[loopMergeCurrent]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a LOOP_UNROLL_FACTOR_MERGE_RIGHT markup with the C++ corresponding code
expandLOOP_UNROLL_FACTOR_MERGE_RIGHTMarkup    ::=
        => writeText(loop_unroll_factorMergeRight[loopMergeRightCurrent]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
```

```
        ;

// Replaces a CHUNK_SIZE markup with the C++ corresponding code
expandCHUNK_SIZEMarkup    ::=
        => writeText(chunk_size[chunkCurrent]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a COMPARE_TYPE markup with the C++ corresponding code
expandCOMPARE_TYPEMarkup    ::=
        => writeText(compareType[0]+","+compareType[1]+","+compareType
            [2]+","+compareType[3]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a BYTES_DIVIDED_BY_8 markup with the C++ corresponding code
expandBYTES_DIVIDED_BY_8Markup     ::=
        => writeText($bytes[bytesCurrent]/8$);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a BYTES markup with the C++ corresponding code
expandBYTESMarkup     ::=
        => writeText(bytes[bytesCurrent]);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a TUNING_LEVEL markup with the C++ corresponding code
expandTUNING_LEVELMarkup     ::=
        => writeText(tuningLevel);
        #explicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a INFO markup with the C++ corresponding code
expandINFOMarkup    ::=
        => {
            writeText("// ");
            if (chunkActive!=0) writeText("CHUNK_SIZE == "+chunk_size[
                chunkCurrent]+" - ");
            if (loopChunkActive!=0) writeText("LOOP_UNROLL_FACTOR_CHUNK = "+
                loop_unroll_factorChunk[loopChunkCurrent]+" - ");
            if (loopMergeActive!=0) writeText("LOOP_UNROLL_FACTOR_MERGE = "+
                loop_unroll_factorMerge[loopMergeCurrent]);
            if (loopMergeRightActive!=0) writeText("
                LOOP_UNROLL_FACTOR_MERGE_RIGHT = "+
                loop_unroll_factorMergeRight[loopMergeRightCurrent]);
        }
        #implicitCopy
        [^'>']*
        #explicitCopy
        '>'
        ;

// Replaces a DATA_VARIABLES markup with the C++ corresponding code
expandDATA_VARIABLESMarkup    ::=
        => {
            local i=0;
            while $i<chunk_size[chunkCurrent]$ {
              if $i>0$ writeText(", ");
```

```
                writeText("data"+$i+1$+"[ "+$bytes[bytesCurrent]/8$+" ]");
                i=$i+1$;
              }
              writeText(";");
          }
          #implicitCopy
          [^'>']*
          #explicitCopy
          '>'
          ;

// Replaces a COMPARE_VARIABLES markup with the C++ corresponding code
expandCOMPARE_VARIABLESMarkup    ::=
          => {
              local i=1;
              while $i<chunk_size[chunkCurrent]$ {
                local j=$i+1$;
                while $j<=chunk_size[chunkCurrent]$ {
                  if $i>1$ || $j>2$ writeText(", ");
                  writeText("b"+i+j);
                  j=$j+1$;
                }
                i=$i+1$;
              }
              writeText(";");
          }
          #implicitCopy
          [^'>']*
          #explicitCopy
          '>'
          ;

// Replaces a ASSIGN markup with the C++ corresponding code
expandASSIGNMarkup    ::=
          #readIdentifier:"BYTES"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               bytesCount=0;
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
                 if (c==' ')||(c=='>'){ // now we'll store the next value
                   bytes[bytesCount]=s;
                   s="";
                   bytesCount=$bytesCount+1$;
                   do{
                     c=readChar(); // moves to the next symbol
                   }while (c==' ');
                 }
               }
            }
          |
          #readIdentifier:"CHUNK_SIZE"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               // Storing the old data
                 chunkCount=0;
                 foreach i in chunk_size {
                   chunk_size_stored[chunkCount] = i;
                   chunkCount=$chunkCount+1$;
                 }
               chunkCount=0;
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
                 if (c==' ')||(c=='>'){ // now we'll store the next value
                   chunk_size[chunkCount]=s;
                   s="";
                   chunkCount=$chunkCount+1$;
                   do{
```

```
                  c=readChar(); // moves to the next symbol
                }while (c==' ');
              }
            }
        }
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_CHUNK"
          => {
            readChar(); // the ' '
            local s="";
            local c=readChar();
            // Storing the old data
              loopChunkCount=0;
              foreach i in loop_unroll_factorChunk {
                loop_unroll_factorChunk_stored[loopChunkCount] = i;
                loopChunkCount=$loopChunkCount+1$;
              }
            loopChunkCount=0;
            while c>='0' && c<='9' {
              s=s+c;
              c=readChar();
              if (c==' ')||(c=='>'){ // now we'll store the next value
                loop_unroll_factorChunk[loopChunkCount]=s;
                s="";
                loopChunkCount=$loopChunkCount+1$;
                do{
                  c=readChar(); // moves to the next symbol
                }while (c==' ');
              }
            }
        }
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE_RIGHT"
          => {
            readChar(); // the ' '
            local s="";
            local c=readChar();
            // Storing the old data
              loopMergeRightCount=0;
              foreach i in loop_unroll_factorMergeRight {
                loop_unroll_factorMergeRight_stored[loopMergeRightCount] = i
                    ;
                loopMergeRightCount=$loopMergeRightCount+1$;
              }
            loopMergeRightCount=0;
            while c>='0' && c<='9' {
              s=s+c;
              c=readChar();
              if (c==' ')||(c=='>'){ // now we'll store the next value
                loop_unroll_factorMergeRight[loopMergeRightCount]=s;
                s="";
                loopMergeRightCount=$loopMergeRightCount+1$;
                do{
                  c=readChar(); // moves to the next symbol
                }while (c==' ');
              }
            }
        }
        |
        #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE"
          => {
            readChar(); // the ' '
            local s="";
            local c=readChar();
            // Storing the old data
              loopMergeCount=0;
              foreach i in loop_unroll_factorMerge {
                loop_unroll_factorMerge_stored[loopMergeCount] = i;
                loopMergeCount=$loopMergeCount+1$;
              }
            loopMergeCount=0;
            while c>='0' && c<='9' {
              s=s+c;
              c=readChar();
```

```
                if (c==' ')||(c=='>'){ // now we'll store the next value
                  loop_unroll_factorMerge[loopMergeCount]=s;
                  s="";
                  loopMergeCount=$loopMergeCount+1$;
                  do{
                    c=readChar(); // moves to the next symbol
                  }while (c==' ');
                }
              }
            }
          ;

// Replaces a SELECT markup with the C++ corresponding code
expandSELECTMarkup    ::=
          => { compareCurrent[0]=0; compareCurrent[1]=0; compareCurrent[2]=0;
              compareCurrent[3]=0; }
          #readIdentifier:"SORTER"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               bytesCount=0;
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
               }
               sorterCurrent=s;
            }
          |
          #readIdentifier:"COMPARE8"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
               }
               compareCurrent[0]=s;
            }
          |
          #readIdentifier:"COMPARE16"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
               }
               compareCurrent[1]=s;
            }
          |
          #readIdentifier:"COMPARE24"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
               }
               compareCurrent[2]=s;
            }
          |
          #readIdentifier:"COMPARE32"
            => {
               readChar(); // the ' '
               local s="";
               local c=readChar();
               while c>='0' && c<='9' {
                 s=s+c;
                 c=readChar();
               }
```

```
              compareCurrent[3]=s;
            }
          ;


// Replaces a RESTORE markup with the C++ corresponding code
expandRESTOREMarkup    ::=
          [#readIdentifier:"CHUNK_SIZE"
            => {
               // Restoring the old data
                 chunkCount=0;
                 foreach i in chunk_size_stored {
                   chunk_size[chunkCount] = i;
                   chunkCount=$chunkCount+1$;
                 }
            }
          |
          #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE_RIGHT"
            => {
               // Restoring the old data
                 loopMergeRightCount=0;
                 foreach i in loop_unroll_factorMergeRight_stored {
                   loop_unroll_factorMergeRight[loopMergeRightCount] = i;
                   loopMergeRightCount=$loopMergeRightCount+1$;
                 }
            }
          |
          #readIdentifier:"LOOP_UNROLL_FACTOR_MERGE"
            => {
               // Restoring the old data
                 loopMergeCount=0;
                 foreach i in loop_unroll_factorMerge_stored {
                   loop_unroll_factorMerge[loopMergeCount] = i;
                   loopMergeCount=$loopMergeCount+1$;
                 }
            }
          |
          #readIdentifier:"LOOP_UNROLL_FACTOR_CHUNK"
            => {
               // Restoring the old data
                 loopChunkCount=0;
                 foreach i in loop_unroll_factorChunk_stored {
                   loop_unroll_factorChunk[loopChunkCount] = i;
                   loopChunkCount=$loopChunkCount+1$;
                 }
            }]

          #implicitCopy
          [^'>']*
          #explicitCopy
          '>'
          ;

writeFunctionName    ::=
    => { if (isFunctionStarted==1){
            local s=strFunctionName;
            // This flag is necessary to iterate only the declared params
            s=replaceString("BYTES_DIVIDED_BY_8", $bytes[bytesCurrent]/8$, s)
                ;
            s=replaceString("BYTES", bytes[bytesCurrent], s);
            chunkActive=countStringOccurences(s, "CHUNK_SIZE");
            s=replaceString("CHUNK_SIZE", chunk_size[chunkCurrent], s);
            loopChunkActive=countStringOccurences(s, "
                LOOP_UNROLL_FACTOR_CHUNK");
            s=replaceString("LOOP_UNROLL_FACTOR_CHUNK",
                loop_unroll_factorChunk[loopChunkCurrent], s);
            loopMergeRightActive=countStringOccurences(s, "
                LOOP_UNROLL_FACTOR_MERGE_RIGHT");
            s=replaceString("LOOP_UNROLL_FACTOR_MERGE_RIGHT",
                loop_unroll_factorMergeRight[loopMergeRightCurrent], s);
            loopMergeActive=countStringOccurences(s, "
                LOOP_UNROLL_FACTOR_MERGE");
            s=replaceString("LOOP_UNROLL_FACTOR_MERGE",
                loop_unroll_factorMerge[loopMergeCurrent], s);
```

```
            writeText(s);
            if (tuningPart==2){
              chunkActive=1;
              loopChunkActive=1;
              loopMergeActive=1;
              loopMergeRightActive=1;
            }
            setInputLocation(iStartInputPosition);
         }
    }
    ;
```

Bibliography

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., Pearson Education, 2008.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, 3rd ed., The MIT Press, 2009.

[3] John L. Hennessy, and David A. Patterson. *Computer Architecture: A Quantitative Approach.*, Kaufmann u.a., Amsterdam u.a., 2007.

[4] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Addison-Wesley Professional, April 1998.

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

[6] E. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[7] Paolo Bertasi, Marco Bressan, and Enoch Peserico. *psort*, yet another fast stable external sorting software. *Proceedings of the 8th Symposium on Experimental Algorithms*, 2009.

[8] Sort Benchmark Home Page. `http://sortbenchmark.org/`.

[9] CodeWorker Home Page. `http://www.codeworker.org/`.

[10] ATLAS Home Page. `http://math-atlas.sourceforge.net/`.

[11] K. Yotov, K. Pingali, and P. Stodghill. *Think Globally, Search Locally,* Cornell University, `http://hdl.handle.net/1813/5669`.

[12] J. Bilmes and K. Asanovi'c and J. Demmel and D. Lam and C.W. Chin. *PHiPAC*: A Portable, High-Performance, *ANSI C* Coding Methodology and its application to Matrix Multiply, *LAPACK Working Note 111*, UTK, `http://www.netlib.org/lapack/lawn`, 1996.

[13] M. Frigo and S.G. Johnson. *FFTW*: an adaptive software architecture for the FFT, *Proceedings of the IEEE 93(2), 2005.* `http://www.fftw.org/fftw-paper-icassp.pdf`.

[14] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3, *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, `http://www.fftw.org/fftw-paper-ieee.pdf`.

[15] Karmen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padue, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE 93(2), 2005. Special issue on "Program Generation, Optimization, and Adaptation".*

[16] Vuduc, R., Demmel, J.W., Yelick, K.A.: *OSKI*: A library of automatically tuned sparse matrix kernels. *J. Phys. Conf. Ser. 16, 521–530,* 200, `http://iopscience.iop.org/`.

[17] ISO/IEC 14977:1996. *Information technology - Syntactic metalanguage - Extended BNF,* `http://www.iso.org/iso/`.

[18] Alberto Bedin. Personal communication.