Calderón, A., García, A., García-Carballeira, F., Carretero, J. & Fernández, J. (2016). Improving performance using computational compression through memoization: A case study using a railway power consumption simulator. *The International Journal of High Performance Computing Applications, 30*(4), 469–485.

DOI: 10.1177/1094342016637813

Improving performance using computational compression through memoization: A case study using a Railway Power Consumption Simulator

Alejandro Calderón, Alberto García, Félix García-Carballeira, Jesús Carretero, Javier Fernández

Avda. Universidad 30, 28911 Leganés, Madrid, Spain

Computer Science and Engineering Department

Carlos III University of Madrid

Abstract

The objective of data compression is to avoid redundancy in order to reduce the size of the data to be stored or transmitted. In some scenarios, data compression may help to increase global performance by reducing the amount of data at a competitive cost in terms of global time and energy consumption. We have introduced computational compression as a technique for reducing redundant computation, in other words, to avoid carrying out the same computation with the same input to obtain the same output. In some scenarios, such as simulations, graphic processing, and so on, part of the computation is repeated using the same input in order to obtain the same output, and this computation could have an important cost in terms of global time and energy consumption. We propose applying computational compression by using memoization in order to store the results for future reuse and, in this way, minimize the use of the same costly computation. Although memoization was proposed for sequential applications a long time ago, in the 1980s, and there are some projects that have applied it in very specific domains, we propose a novel, domain independent way of using it in high-performance applications, as a means of avoiding redundant computation.

Keywords: High Performance Computing, Memoization, Railway Simulation

1. Introduction

Scientists typically develop their workflow in a trial-and-error manner [1]. We can find examples on Scientific Applications from areas such as engineering [2], biomedical [3], or financial [4]. For those scientific applications scientists often need to reproduce/analyze/share previous results [5] [6]. On these cases, the used workload is the same. By inspecting the results likely the model or the experimental protocol is modified and the simulation is repeated [7]. On these studies the used workloads are very similar among them. Even within the execution of the scientific application with a single dataset, we can find that the internal computation follows an iterative process where two consecutive iterations have some inputs in common. In CFD (Computational fluid Dynamics) applications [8] for example, there

are incremental steps with little variation from one step to the next. On all these cases, there are opportunities for repeating the same internal computation within some functions because their internal computation works with the same input. In these cases, avoiding redundant computation might lead to a decrease in cost in terms of execution time.

Computation compression removes redundant computation by using memoization. The term memoization [9] refers to computing a slow function only once and storing the results in a table. Subsequent requests for the function are then handled by table lookup rather than by computing the function. Memoization is an optimization technique used to avoid redundant computing, so that some function calls can be replaced by the results obtained in a previous execution of the same function. Memoization lets us reduce the execution time provided that the time to search a previously computed value is going to be less than the time to compute it.

The computed values are stored temporally in memory in order to minimize the search time for a previously computed result. There is a trade-off between space and time. The goal of memoization is to reduce the execution time in exchange for memory space. However in order to achieve this goal with today's applications from different domains, we need to find a very fast memoization method, and a generic way to apply memoization to any function as easily as possible. For MPI [10] applications, in a previous paper we proposed [11] a new MPI_Info object implementation, used to store key-value pairs so quickly that it could be used as a shared storage solution among MPI processes. In that paper, we also explored the utilization of the memoization technique on MPI applications based on the proposed MPI_Info object for improving performance.

After successfully demonstrating that memoization can be used on an MPI parallel application, we have extended our work to non-MPI based applications. We have selected as case study a Railway Power Consumption Simulator [12] (hereinafter RPCS). Our goal is to reduce the simulation time for testing simulations of power consumption of a given railway, infrastructure and traffic. In this simulator we have found that the simulations are often executed for testing the viability of some configurations, and that the performed computation is redundant in some cases; there is, therefore, a good scenario for computation compression through memoization. We have applied memoization successfully to the RPCS, and in this project we want to share not only the experience, but the way we have found to deal with several problems that, as far as we know, have not all been covered before by a simulation system like this.

Further applications can be candidates for using our proposed computational compression too. Examples of those kind of applications for future evaluations are: video analysis, and multivariate analysis. In video analysis each frame from the studied video is analyzed, where close frames use to be similar. There are opportunities for redundant computation. Multivariate analysis usually implies performing not one but many simulations, using the same simulator but varying the input data. Examples of such kind of simulators can be found in [13]. With similar input data, there is opportunities for redundant computation too.

The rest of the paper is organized as follows: Section 2 discusses the related work; Section 3 and Section 4 describe our proposal; Section 5 describes the case study based on the RPCS; Section 6 shows the experimental evaluation results; and finally Section 7 summarizes our conclusions and presents future work.

2. Related work

Memoization [9] can be described as a way to reduce the cost of recomputing expensive functions by computing the functions only once and storing the results. Then, subsequent calls to the same function are handled by table lookup rather than by computing the function, thus saving execution time. Memoization has been used to improve performance in different (but specific) areas such as thread scheduling [14], out-of-order processor simulation [15], and as file caching of results for geo-scientists in satellite data [16]. Recently, it has also been applied in order to improve energy efficiency; for example on financial applications [17] and in general, it complies by avoiding the execution of groups of instructions. For example, in [18], the authors show Dynamic Trace Memoization (DTM), a reuse technique that employs memoization tables to skip the execution of sequences of redundant instructions. Memoization can be applied at different levels; in hardware, it has been used as a technique for reusing partial results on instruction blocks [19] or at CPU instruction level for arithmetic operations [20]. In the software area, it has been mainly used in logic programming in order to 'learn' intermediate results as predicates in Prolog [21], in functional programming, even with concurrency and communication [22], or to optimize functions with stochastic inputs [23].

In all cases we have been able to find the description of some work where memoization has been applied. But we have not found both a memoization framework and a systematic process intended for general usage. The closest work that can be found is [24] where despite the title of the paper, the associated work only shows the steps followed for an early prototype in a C++ specific application for DARPA project. It does not deal with a large amount of computed values to be stored and retrieved or with how to reuse this memoization; and it does not provide the steps to detect where this technique can be used (in which functions it could be applied). To the best of our knowledge, found memoization related works do not address all of these key factors.

One key factor of memoization's performance is the storage system supporting the previously computed values. A possible solution is to use a storage method based on a fast hash table [1]. While the application is running, this hash table has to provide a fast way to insert, search, and retrieve general input-output pairs. Moreover, to improve the performance, further opportunities like sharing the hash

[1] In this paper, the term "hash table" is used to designate a dictionary abstract type, composed of a collection of key value pairs. Neither transformations on the key, nor collisions which may happen between different keys are performed by the authors, unless otherwise indicated.

table among threads or processes (in order to increase computational compression) or storing the hash table at the end of the application (so that computed values will be available to future executions of the same application), have to be considered.

An important contribution to the related work with hash tables can be found in [25] where some of the main hash table implementations available are evaluated. `Khash` is the fastest implementation evaluated in this work. Moreover, this work has shown that the hash table mechanism behaves better than the search trees if ordering is not required. However, there is another interesting implementation, named `UThash` [26], that it is in the top 5 faster hash implementations (out of 27). It presents a very easy interface and deployment process, and it can also use almost any type of object as key or value (khash is limited to using strings as keys). We have selected `UThash` as the default option for computational compression within a process because it is fast, it does not limit the key type to strings, and it is easy to use. `UThash` is a non-persistent hash-table. At the end of the application's execution the computed values in memory should be stored in a persistent storage solution like a SSD o hard disk device. At the beginning of the next execution these computed values can be restored back to memory to avoid repeating the same computation over different executions. Even if this means some overhead now, it will almost disappear when new NVRAM memory devices can be used as persistent devices.

We have found NoSQL storage solutions that are fast and scalable for keeping results in memory, and they could be also useful for sharing these results among processes. In this cases, the hash table is a distributed service, not a data abstraction implemented by a single process. There are several alternatives, but memcached in-memory key-value store distributed system [27] seems to be the one with the best performance [28]. Memcached is a persistent storage solution in the sense that a stored computed value survives different executions of the distributed application. Nevertheless, a low latency network might be needed as a viable option in some border-cases [11].

Many of today's applications are multi-threaded applications, using a shared-memory model. In the paper [29], the authors proposed the usage of a hash table as a shared resource where the results are stored by one thread and reused by others. But this work is tied to object-oriented programming, and only evaluated with a benchmark (STMBench7) for software transactional memory. This hash table solution might have two major problems: it might become a potential performance bottleneck, and as such, forces major modifications to the original source code in order to implement the solution. If the hash table is a bottleneck with transactional memory support, the problem can be dealt with [30] where the authors proposed a technique, entitled lock-free parallel dynamic programming; this technique improves the performance of many threads accessing the hash table in a large number of situations. However it forces major (and not so simple) code modification of the original source code.

3. A systematic process for computational compression

In this section, we propose a user-driven workflow to facilitate adding the computational compression, through memoization, on new and existing applications. The usual minimal process for creating an application can be seen as, at least, a three-step workflow: 1) writing the source code, 2) compiling this source code, and 3) executing the application in its environment (e.g. multi-thread, or multi-processes environment). Our proposal introduces three additional steps (as shown in Figure 1): A) an application profiling study, B) the function/method selection process, and C) the addition of computational compression (using memoization).

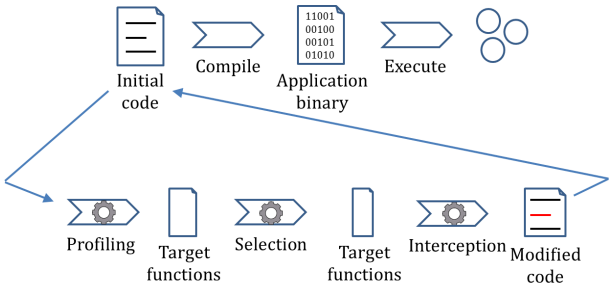

Figure 1: Steps of the proposed workflow for computational compression.

3.1. Application profiling phase

The application profiling step is focused on finding the ordered list of functions that need most of the execution time. To obtain this ordered list we need to perform the following steps:

1. Use the standard `-pg` flag at compilation phase. This flag is valid for GCC and many other compilers.

2. Execute the application with a representative workload. After the execution of the application, a `gmon.out` file is generated.

3. Execute `gprof 'application name' gmon.out > gmon.txt`, so that we can find the list we are looking for in the "Flat profile" section of the `gmon.txt` file.

3.2. Function/Method selection phase

Once we have the function list where the application spends most of the time, the next step is to choose which functions computational compression could be applied to. We are going to select from the function list those whose execution time is large enough to expect a performance improvement if we reduce the computational redundancy through memoization. For a function f of an application A to be considered a valid candidate for applying computational compression must fulfill the following properties:

(a) The function f is a *pure function*. Pure functions are those that return the same results (ρ) over the time (j,k) providing the input parameters are the same $(i, i = 1..n)$. Moreover, the execution of this function must not cause any side-effects on the system (S) (e.g., modification of a global variable).

$$f \text{ is pure} \iff \forall j,k \;\; \rho_j(f(i, i = 1..n)) \;=\; \rho_k(f(i, i = 1..n)) \;\wedge\; S_j = S_k = \emptyset$$

(b) The function makes use of resources f_r that are not altered outside the application scope A_s (e.g. I/O routines can read or write data that can be externally modified).

$$\forall r \in f_r \Rightarrow r \text{ is only modified in } A_s$$

(c) The function is a significant number of times called during the execution.

$$\eta(f) \;=\; n \text{ times}, n \neq 0$$

The first time the function call $f(i, i = 1..n)$ occurs, the result is computed. In subsequent times, this result is looked up. So if this memoized result is never discarded, the success ratio of the memoization technique (h) for $f(i, i = 1..n)$ is:

$$h = \frac{\eta(f,i) \;\; 1}{\eta(f,i)} \quad h \in [0..1]$$

(d) The function spends a minimum amount of time t_c in order to compute the output values from the input values. Then, a $t_i(i)$ time is needed to memorize the function call results (the input values act as the key, whereas output values are packed as the value-part of the pair). The computing time t_c must be longer than the time t_s to search in the hash table for the key-value pair where the output values are stored.

$$\tau_0(f(i, i = 1..n)) = t_c(i) + t_i(i)$$
$$\tau_1(f(i, i = 1..n)) = t_s(i)$$
$$\ldots$$
$$\tau_n(f(i, i = 1..n)) = t_s(i)$$

$$t_s(i) < t_c(i) \quad \forall \tau_k, k = 0..n$$

These four properties provide the conditions for reducing the execution time of this function. If searching for previously computed results ($t_s(i)$) requires less time than recomputing them ($t_c(i)$), and this happens several times ($\eta(f,i)$), this technique may reduce the total execution time of an application, providing there is enough computational redundancy.

The total execution time for a memoized function call ψ_m is:

$$\psi_m = \eta(f,i) * (h * t_s(i) + (1-h) * (t_c(i) + t_i(i))) \tag{1}$$

$h \in [0..1]$ being the success ratio.

The total execution time without memoization ψ_{nm} is:

$$\psi_{nm} = \eta(f,i) * t_c(i)$$

And the accumulated saved time by using memoization is:

$$\psi_{nm} - \psi_m = \eta(f,i) * t_c(i) - \eta(f,i) * (h * t_s(i) + (1-h) * (t_c(i) + t_i(i)))$$

If results memoized are never forgotten then $\psi_{nm} - \psi_m$ is:

$$\begin{aligned}
\psi_{nm} - \psi_m &= \eta(f,i) * t_c(i) - \eta(f,i) * h * t_s(i) - \eta(f,i) * (1-h) * (t_c(i) + t_i(i)) \\
&= \eta(f,i) * t_c(i) - \eta(f,i) * \frac{\eta(f,i)-1}{\eta(f,i)} * t_s(i) - \eta(f,i) * (1 - \frac{\eta(f,i)-1}{\eta(f,i)}) * (t_c(i) + t_i(i)) \\
&= \eta(f,i) * t_c(i) - (\eta(f,i)-1) * t_s(i) + t_c(i) + t_i(i) \\
&= (\eta(f,i)-1) * t_c(i) - (\eta(f,i)-1) * t_s(i) - t_i(i)
\end{aligned}$$

And in order to save time, $\psi_{nm} - \psi_m > 0$ so:

$$\begin{aligned}
(\eta(f,i)-1) * t_c(i) - (\eta(f,i)-1) * t_s(i) - t_i(i) > 0 \\
(\eta(f,i)-1) * t_c(i) > (\eta(f,i)-1) * t_s(i) + t_i(i)
\end{aligned}$$

The last expression tells us that $\eta(f,i) - 1$ computations of the function call $f(i, i = 1..n)$ needs more time than $\eta(f,i) - 1$ times searching this result, plus the time to memoize it ($t_i(i)$). If this condition is reached, then the memoization can be applied to the $f(i, i = 1..n)$ pure function call.

From the profiling results we obtain the number of times the function is called $\eta(f)$ (i.e. an approximation for $\eta(f,i)$). We also obtain the average computing time t_c of the function (i.e. an approximation for $t_c(i)$). The average search time t_s, and the average memoization time t_i are obtained from a simple benchmark on the hash table chosen. The best candidates are those functions with the longest saving times:

$$\psi_{nm} - \psi_m = (\eta(f) - 1) * (t_c - t_s) - t_i$$

Both elements t_c and $\eta(f)$ are important: the longer the better. The functions with longer saving time provide better results on lowering execution times of several executions of the applications, thanks to computational redundancy.

These profiling results are an approximation because what happens with each individual function call is unknown. The application source code can be modified in order to print a message each time the

candidate function is called; this message will contain the input parameters. After the execution, we can obtain $\eta(f, i)$ by counting the number of messages in which input $i, i = 0..n$ occurs. From the previously obtained candidate functions, the ones with a large number of calls on the same parameters ($\eta(f, i)$) provide the best results on lowering the execution time of one execution of the application, thanks to computational redundancy.

3.3. Adding computational compression

Once we have selected the functions suitable for computational compression from the list of functions which have large impact on the application execution time, the final step is to add the interception. This interception process checks whether the function call has been previously executed with the same arguments, so the resulting values have already been computed and can be reused. If the values are not present, they are computed using the original function call, and stored for future use. There are some examples [31] of adding memoization but they impose some restrictions: firstly, all function parameters have to perform as input for the memoization; and secondly, the output must be the return value of the function. We have pursued a more general solution: the users can decide what input parameters are going to be memoized (all of them or only a subset), and output parameters acting as memoized output.

Listing 1: Skeleton with the original C++ source code

```
1 void functionX (int in1, double * out2)
2 {
3    /* example of pure function */
4 }
```

In listing 1 we show an example of a function without the necessity of a canonical format. It has two arguments, one is the input, and the other one is the output parameter used to retrieve the result of the calling function, once the computational part is completed. In addition, the output could depend on only some, rather that all, of the input parameters. Our proposal allows programmers to add computational compression through memoization into their codes in a more general way.

In listing 2 we have shown how the function we selected is modified in order to add computational compression.

- In lines 1 to 4 we have included the function where we just change the name by adding an `original_` prefix.
- In line 6 we have defined a proxy object which is going to store the input-output pairs, and we have named it after the name of the function to be optimized (i.e. `functionX`). This object is going to act as interface to the hash table, obtaining and storing elements (see lines 10 and 15).

- From line 8 to 16 we have created a function that is going to intercept the original function calls, which, firstly, at line 10 is going to search for a previously computed value. If a pair is found, it is returned. Otherwise, it will be computed (for the first time) and stored for future use.

Listing 2: Skeleton with the modified C++ source code

```
1  void original_functionX (int in1, double * out2)
2  {
3      /* example of pure function */
4  }
5
6  MemoSupport proxy("functionX");
7
8  void functionX (int in1, double * out2)
9  {
10     found = proxy.find(&(in1), sizeof(int), out2, sizeof(double));
11     if (true == found)
12         return;
13
14     original_functionX(in1, out2);
15     proxy.add(&(in1), sizeof(int), out2, sizeof(double));
16 }
```

The proxy object stores key-values, making a copy of the data that constitutes the key (i.e. the inputs) and the value (i.e. the outputs, once computed). In order to make this copy, the proxy object receives the memory address where that value and the number of bytes it takes in memory are stored.

If several parameters constitute the input, then we have to gather all their values and flatten them into a single structure at the beginning of the interception function (`functionX` at Listing 2). The aim is to provide the proxy function with one single object to be stored. Similarly, if the intercepted function has several output parameters, we have to store them, once computed, in one single structure. And when retrieving one memorized value from the hash table, we have to deserialize the stored value, and map the byte stream to the output parameters before returning it to the function caller.

Figure 2 shows the structure of the skeleton of the information flow through the modified source code. The `slow()` function represents the `original_functionX` in listing 2, and the `intercept()` function represents the `functionX()` of the same listing. The `intercept()` function before performing the slow computational part, tries to avoid it by searching for a previously computed value.

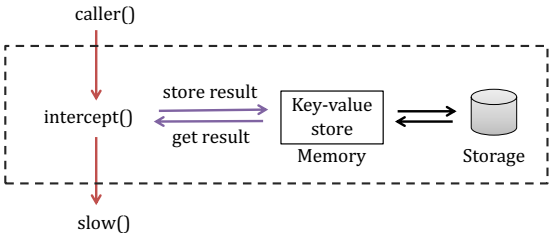

Figure 2: Computational compressed version of a slow function.

4. Proposed storage architecture for computational compression

In the previous section we have introduced the process to apply computational compression through memoization. In this section we will focus on solving the problem of how to conveniently store the input-output pairs. The appropriate storage support is very important to successfully add computational compression to current applications.

The functions we want to optimize could be executed on different threads within an application, and this application could be executed several times. Therefore the computational redundancy may appear in an application at different levels: 1) spatial locality in the same execution, 2) temporal locality in different executions, 3) a combination of both. If the application is executed several times, it could be possible to find similar computations over a period of time. For instance, for a process that is filtering a specific region of a video stream, many frames could be very similar. If several threads from the same application are executed in parallel, it could be possible to find similar computations over these threads in a particular execution. Another example could be n-body-like problems where two processes manage collocated regions of the space and a shadow zone is used.

For storing and retrieving the input-output pairs quickly, a fast in-memory hash-table solution is used. In order to avoid computational redundancy during one execution (over space), the hash-table solution has to be not only fast, but also shared among all the threads, so that any one of them could find whether another (or itself) has previously performed a computation. In order to avoid computational redundancy along several executions of the application, we have to consider a persistent storage for storing the computed values (The Storage element in Figure 2). In this way, several executions of the same application could obtain an improved speed-up, because the values computed during the current and previous executions can be reused in the subsequent ones. In the related work section, we have introduced several hash-table solutions that can be used. Depending on some properties of the computational redundancy in a particular function, the programmer could find a hash-table more suitable for the memoization (or even try some of them, and select the best one).

In this paper we propose the two-layer architecture represented in Figure 3. This architecture gives the programmers the flexibility to decide which underlying technology is going to be implemented. The upper

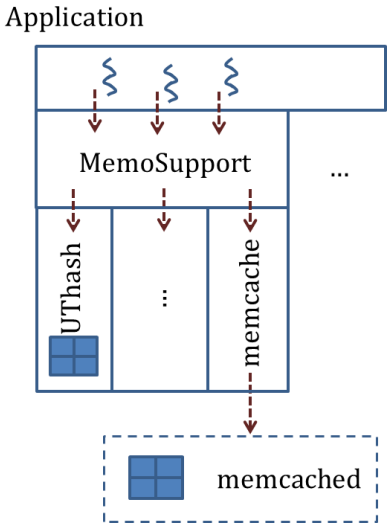

Figure 3: Proposed architecture for storing computational-compression data.

layer provides a simple interface to be used by the programmers, as shown in listing 2. The lower layer implements a plug-in like subsystem, where several hash-table solutions could be used for implementing the upper layer interface. In this way, programmers can also select a different hash-table solution for each function to be optimized, so that several solutions can be used and combined. UTHash is used by default, but an extra parameter in MemoSupport constructor lets programmers select the desired under-layered key-value store to be used, as for example: `MemoSupport proxy("functionX","memcached");`

In a near future scenario, all computing elements will use a Non-Volatile Dual In-line Memory Module or NVDIMM [32]. Since data is retained even without electrical power, and they provide a large capacity, it could be used to store the input-output pairs of the hash-table solution used in a persistence way. If the underlying key-value solution does not provide persistence, the MemSupport can provide it: the MemSupport destructor stores the input-output pairs, and the MemSupport constructor load the previous saved pairs. In this case, an additional time is needed to load the information before the sequence of simulations, and an extra time is also needed to save the information after the sequence of simulation ends. Depending on the storage technology used this time could vary very much. The difference observed between SSD and HDD is up to an order of magnitude better [33].

If MemSupport is going to use a hard disk to save/retrieve the input-output pairs from/to memory, we propose two optimizations in order to reduce the associated additional time:

- Using compression in order to reduce the size of the file where pairs are saved and to speed up the read/write process. For example the zlib library is free and portable across multiple platforms [34].

- A hits counter associated to each entry. The initial value is 1, and each time an entry is used again

the counter is incremented. On saving the input-output pairs, the programmer could skip the ones with a hits counter of less than a threshold value. For instance, by skipping input-output pairs with an associated hit counter of 1, we avoid storing the pairs that were never reused in this execution.

The MemSupport constructor loads previously stored input-output pairs if the file where they were saved is detected. In order to avoid reusing the former stored values, the user only has to remove the computational compression supporting files before the next execution.

5. Case study: railway power consumption simulator

In order to demonstrate the benefits of the memorization, we implemented this technique on a Railway Power Consumption Simulator (RPCS), developed by our research group.

We selected this tool for several reasons. First of all, it is a real tool currently used by ADIF, the Spanish railway company, to test and verify different scenarios (e.g. developing new routes, increasing train traffic across the tracks, or testing failure situations where services have to be operated on degraded mode), so it portraits the general sort of engineering simulators commonly used. Secondly, the complexity of the source code (more than 13,000 lines of code disregarding GUI classes) is sufficiently high to accurately approach the workload necessary to memorize complex applications. Finally, the tool requires a large amount of computing power, performing multiple matrix operations for each simulated instant (and a typical train traffic scenario has to be simulated during the whole day), so it will likely benefit from the memorization technique. Previous research project [12] have been conducted using this simulator as a test bed due to these characteristics.

The aim of this simulator is to calculate if the amount of power supplied by the electrical substations is sufficient or not; this depends on a number of trains circulating along the lines. Starting from a description of the railway infrastructures i.e. tracks, catenaries deployed over the tracks, electricity substations located along the tracks, as well as additional elements like feeders and switches, the simulator reads the position of the trains and its instantaneous power demand at each second. Then, the electric circuit formed by the trains and the infrastructure is composed and solved using modified nodal analysis (MNA). Useful mean voltages, voltage drops and the temperatures of the wires are some of the results provided by the tool.

The core functionality of the simulator is to perform a circuit analysis for every second, thus solving the system and obtaining the electric status in the infrastructure for each instant. The MNA general formulation is:

$$\begin{bmatrix} A_1 Y_1 A_1^T & A_2 \\ M_2 A_2^T & N_2 \end{bmatrix} \cdot \begin{bmatrix} u^n \\ i_2^r \end{bmatrix} = \begin{bmatrix} -As \cdot is \\ ws2 \end{bmatrix} \tag{2}$$

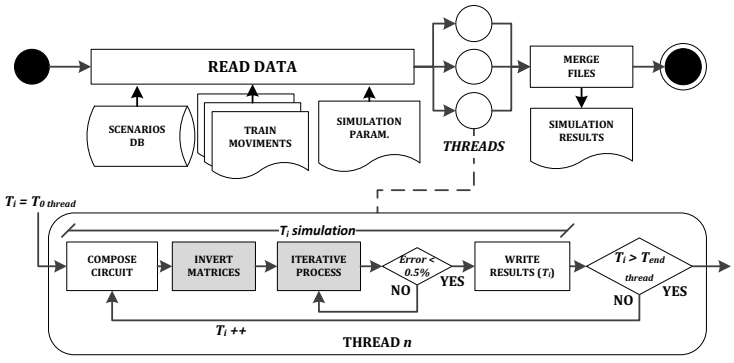

Figure 4: Railway power consumption algorithm.

In this problem, branches are considered resistors, and there are only independent voltage sources, so the previous equation can be simplified as:

$$\begin{bmatrix} G & B \\ C & 0 \end{bmatrix} \cdot \begin{bmatrix} u^n \\ i^r \end{bmatrix} = \begin{bmatrix} i \\ e \end{bmatrix} \tag{3}$$

where G, B, and C are matrices of known values obtained from the circuit elements (connection, conductances, etc.), u^n and i^r are the unknown voltages and and currents, and finally i and e contain the sum of the currents through the passive elements, and the values of the independent voltage sources respectively. More details about MNA can be found on [35].

Figure 4 describes the central algorithm of the simulator. The application is multi-threaded, thus the workload (instants to be simulated) is spread across all available cores. For each instant, the following operations are conducted:

1. Given the infrastructure data and train positions at current instant, the matrices representing the electric circuit are composed following the MNA technique.

2. The main matrix is inverted using LU decomposition.

3. Given train consumption at one instant, the aim is to obtain the corresponding values of current and voltage (both unknown). The user proposes a reference value and a percentage of error, and an iterative process is conducted based on that reference, until the results obtained are below the indicated error.

4. Finally, results are written on the disk.

Steps 2 and 3 involve dense matrix operations. In particular, step 2 performs a matrix inversion following the LU decomposition, and at step 3, a matrix multiplication is conducted. These operations are very suitable for being memoized.

RPCS core (without GUI and data management classes) is written in C++ along 142 files (plus one Makefile file). Using the cloc utility we have come up with the statistics described in Table 1.

Table 1: Source code statistics of RPCS core from cloc utility

Language	files	blank	comment	code
C++	69	2233	482	9445
C/C++ Header	73	1204	92	4241
make	1	9	1	15
SUM:	143	3446	575	13701

6. Evaluation

Once RPCS has been described, we are going to describe the platform where the RPCS was executed, and the workload that the RPCS worked with. Then, we are going to review how our proposed computational compression process has been applied to the RPCS, and the results that were obtained in the evaluation comparing both versions: with and without computational compression. This evaluation is focused on analyzing the most relevant effects of computational compression. Among all these effects, the most important is the execution time, since our proposal is intended to save computation time; but memory consumption is also important, because we save time by exchanging stored computed values in memory (computational compression through memoization). Therefore, the results will include time measurements as well as memory consumption measurements.

Moreover, we want to provide some insights into the internal behavior of memoization within the RPCS. We are particularly interested in the number of elements stored, the hit ratio (percentage of all accesses that avoid redundant computation), and the size of the file where the hash table is stored (using a compressed format). The hit/miss ratio will help us know what degree of computational redundancy is found for RPCS and the selected workload. File size becomes relevant because in this paper we are using a hard disk as a permanent storage subsystem for this evaluation. So if the user wants to reuse memoized values across different work sessions (several executions of simulations), all the entries must be stored in the file (permanent storage) and loaded again later on. Note that, due to the observer effect, these internal measurements may cause variations in both execution time and memory footprint, so they are gauged performing a separate study.

Finally, with regard to the possibility of reusing values across different executions of the same application, we want to analyze how the execution time and memory consumption evolve, as subsequent executions of the application are performed. The evaluation is going to compare the execution without computational compression, the first time the computational compression is used, and the second time the computational compression is used (with the same workload as in the first time). The "Without computational compression" represents the baseline result. The "second time the computational compression

is used" with the same workload represents the best scenario (reusing everything). If the workload is changed between two executions, depending on the computational redundancy between these executions, the results will be between the baseline and the best scenario.

6.1. Evaluation platform

The hardware used in the evaluation is a Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz (4 cores), with 32GiB RAM, Ubuntu 14.04.2 LTS, UThash 1.9.7, and GCC 4.8.2. The generated code has been optimized with the *-03* compilation flag.

6.2. Workload used with the RPCS

In order to perform the evaluation using a representative workload of real use cases of the RPCS simulator, we have followed the test case proposed in the European normative UNE-EN-50641[36] [2]. This draft intends to set standards for validating railway power supply simulators, providing a description of a typical workload and the results that are supposed to be obtained. Therefore, it is meaningful to use this scenario as the representative workload for this application.

This workload includes a simulation of trains circulating across a double track during a period of one hour and twenty minutes of simulated time with all the typical elements present (stations, tracks, catenaries, electric substations, feeders, switches and so on). There are five trains with different characteristics (high speed train, suburban train, freight train and so on), three power substations, and a number of connections between the two tracks composing a dense electric circuit.

The RPCS simulator has 27 parameters that define different aspects of the configuration. The selected workload also includes the information for the default values for these parameters. From these parameters we have selected two of them related to the electric behavior of the trains. Thus, they are very important for simulations related with electrical studies. We are going to use these two parameters in the evaluation, and they are:

- *uMin2*. This parameter establishes the threshold of the minimum permanent voltage for the trains. Under normal operating conditions (i.e. absence of system failures) the trains must maintain their voltages within the range minimum-maximum permanent voltage. If this is not possible, different simulation events can be dispatched, related to the train behavior (change of running mode) or the power sources (allocate extra groups in order to maintain a steady supply flow). Note that due to the fact that it is not possible to know in advance the evolution of the circuit as these events take place, changing this threshold requires repeating the whole simulation.

[2]This normative is still a proposal under vote by the CENELEC committee until July, 2015, when it will finally either be approved or discarded.

- *ErrorMaxPC.* This parameter establishes the threshold for the convergence in the iterative process performed after analyzing the circuit. Adjusting this parameter allows to obtain more precise results from the simulation, but a very tight value may provoke the algorithm not to converge. Again, experimenting with different error threshold requires the repetition of the simulation.

6.3. Proposed workflow applied to the RPCS

We have used the proposed three-step workflow in order to apply computational compression into the RPCS:

1. *Application profiling* This step identified the top execution time function list displayed in Table 2.

Table 2: Execution profile of RPCS from gprof utility: top execution time functions list

Each sample counts as 0.01 seconds						
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
66.84	1086.57	1086.57	16752	0.00	0.00	LibMath::ArrayOperator::Inv(...)
22.85	1458.02	371.45	16752	0.00	0.00	LibMath::LUMatrix::Calcular()
9.70	1615.74	157.72	47658	0.00	0.00	LibMath::ArrayOperator::MultiplyMatrix(...)
0.28	1620.36	4.62	25521	0.00	0.00	LibMath::ArrayOperator::MultiplyMatrix(...)
...						

2. *Function/Method selection* Time is mostly spent on two functions which correspond to the most expensive matrix operations performed by the simulator: inverting a matrix (`LibMath::ArrayOperator::Inv`) and multiplying two matrices (`LibMath::ArrayOperator::MultiplyMatrix`). Both are pure functions. Because 66.94% of the time is spent on the `LibMath::ArrayOperator::Inv` method, we will select this function as our priority target.

3. *Adding Computational Compression* We modify the function `Array2D<double>* ArrayOperator::Inv (LUMatrix* LU)` in order to add the interception at the beginning and at the end. In this way, we can skip the computational expensive task of performing the LU decomposition and calculating the inverse matrix, provided that the same calculation has been done before. This can be possible because repetitive patterns in train timetables (trains are scheduled periodically) can introduce redundant computations. We have used the UTHash locally, saving the input-output pairs for future executions in a gzip compressed file (using the zlib library). The `LU` matrix parameter is the input parameter, and corresponds to the matrix to be inverted (prior to the calculation). The `Array2D<double>` is the output, and corresponds to the already inverted matrix.

Listings 3 and 4 provide an overview of the `Array2D<double>` and the `LUMatrix` respectively. The `Array2D<double>` class represents a classic matrix of double elements, whereas the `LUMatrix` class extends the `Array2D<double>` class by adding the pivots to be used in the LU decomposition, and some functions

related to the inversion procedure. The underlying array where the data is stored is the variable `_a`, and `_rows` and `_cols` denote the number of rows and columns of the matrix (note that $rows * cols$ will be equal to the length of `_a`). Several constructors are provided, as well as functions to obtain or set an element of the matrix.

```
template <class T>
class Array2D
{
protected:
        T* _a;
        int _cols, _rows;
        unsigned int *_digest = NULL;

public:
        Array2D(int rows, int cols);
        Array2D(int rows, int cols, T initialValue);
        Array2D(T* a, int rows, int cols);
        ~Array2D(void);

        inline T Get(int row, int col);
        inline void Set(int row, int col, T value);

        inline int Rows();
        inline int Cols();
        ...
```

Listing 3: A quick glance at the class Array2D

```
class LUMatrix : public Array2D<double>
{
private:
        int pivsign;
        int* piv;

public:
        LUMatrix(int rows, int cols);
        LUMatrix(Array2D<double>* a);
        ~LUMatrix(void);

        int* Piv();
        int Pivsign();

        void Calculate();

        ...
```

Listing 4: A quick glance of the class LUMatrix

6.4. Evaluation of simulation time

The first analysis we performed was focused on the time that RPCS needs for simulating the scenario described in Section 6.2. We performed three measurements of the time required to conduct the simulation: without computational compression (the original application); with computational compression, but without re-using results from a previous execution; and with computational compression and re-using results from a previous execution.

In order to re-use results between two subsequent executions, those memoized values which have been hits (at least one hit on the hash table) are saved to permanent storage by one execution, and loaded again by the next. We executed the RPCS simulator ten times with the previously described workload, and we computed the average time of the experiments. We found that the standard deviation is less than 5% of the value measured.

Figure 5 shows the average simulation times for the RPCS simulation: without computational compression (the leftmost bar, labeled as 'Baseline'), with computational compression but without reusing

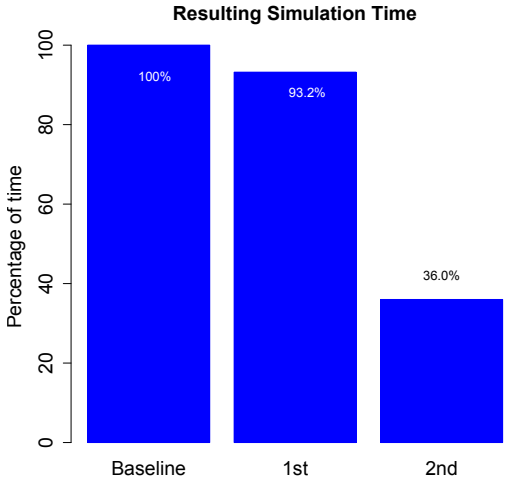

Figure 5: Average simulation times without using computational compression, the first time is used, and the second time is used.

previous values (the next bar, labeled as '1st'), and with computational compression but reusing previous values (the bar labeled '2nd').

Applying computational compression to the RPCS simulator leads to an 6.8% reduction in the average simulation time of the first execution where the application is monitorized. Total simulation time is further decreased in the second (2nd) execution. By re-using values from previous executions, we reduced the simulation time by up to 64.0% with regard to the original version (near to three time faster). We obtained the best result by re-using all values (computational compression avoiding computation that is redundant). The results with any other workload (with a different degree of computational redundancy) fall between the baseline and the second case. But these results prove that our proposed computational redundancy reduction technique will help to reduce simulation time, provided there is computational redundancy. With only one function/method optimized, more than twofold reduction of simulation time can be achieved (almost 2/3 of the simulation time was reduced by optimizing one single function in our case).

Finally, in order to know what the impact of the optimization on the intercepted function is, we have again analyzed the execution profile of RPCS, but this time with computational compression. Table 3 displays the execution profile when computational compression is used for the first time. Table 4 displays when is used for second time. Without computational compression `LibMath::ArrayOperator::Inv` function represents 62.83% of the total execution time. The first time computational compression is used it represents 66.30%, and the second time it represents 0.0%, so the impact of the function is clearly decreased. The difference without computational compression and the first time computational

compression is used is only 3.5%, due to the overhead introduced by the execution of the extra code needed for profiling (see Figure 5).

Table 3: Execution profile of RPCS with computational compression (1st): top execution time functions list

Each sample counts as 0.01 seconds						
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
62.83	938.10	938.10	16752	0.00	0.00	LibMath::ArrayOperator::Inv(...)
24.69	1306.78	368.68	16752	0.00	0.00	LibMath::LUMatrix::Calcular()
10.46	1462.94	156.16	47658	0.00	0.00	LibMath::ArrayOperator::MultiplyMatrix(...)
0.71	1473.48	10.54	16752	0.00	0.00	MemoSupport::find(...)
0.62	1482.74	9.26	2346	0.00	0.00	MemoSupport::add(...)
...						

Table 4: Execution profile of RPCS with computational compression (2nd): top execution time functions list

Each sample counts as 0.01 seconds						
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
66.30	395.68	395.68	16752	0.02	0.02	LibMath::LUMatrix::Calcular()
26.57	554.23	158.56	47658	0.00	0.00	LibMath::ArrayOperator::MultiplyMatrix(...)
3.04	572.35	18.12	2	9.06	9.06	MemoSupport::restore_bin()
1.86	583.43	11.08	16752	0.00	0.00	MemoSupport::find(...)
0.87	588.62	5.19	25521	0.00	0.00	LibMath::ArrayOperator::MultiplyMatrix(...)
...						

As said before, the best scenario is the second execution with the same workload, where all values are re-used. During an average simulation session the same input data is used but the configuration changes, thus the workload changes. In order to evaluate and to compare the behavior of the memoized application with different workloads (in a common simulation session), we have executed the application six times, each one with a different combination of values for the two parameters described in subsection 6.2.

The initial execution uses the values $errorMaxPC = 0.2$ and $uMin2 = 2000$, and the memoized results are reused for the next six executions. For the second execution, simulation is based on the same workload ($errorMaxPC = 0.2$ and $uMin2 = 2000$) so all values are reused (best case). In the following executions the pair $(errorMaxPC, uMin2)$ takes values $(0.1, 2000), (0.5, 2000), (0.2, 2500), (0.1, 2500), (0.5, 2500)$.

The figure 6 shows the normalized simulation time (taking the initial execution as baseline). The simulation time for the second execution (the best case) was a 36.0% of the baseline (simulation time for the first execution). In case of changing the workload (different combination of $errorMaxPC$ and $uMin2$ parameters) we have found that the simulation time is a 36.6% of the baseline case. Therefore there was redundant computation when the simulation was executed with a different workload, and

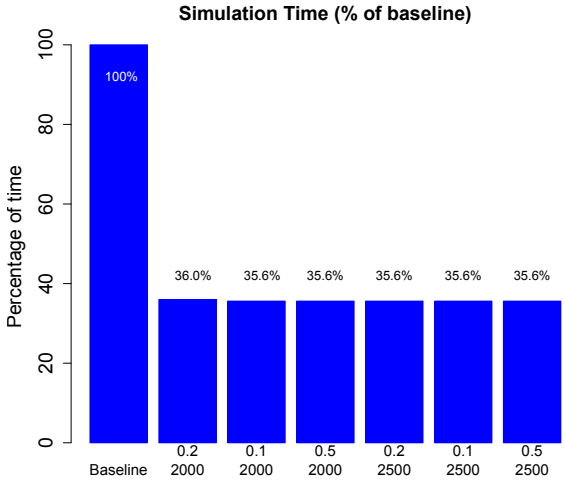

Figure 6: Average simulation time varying errorMaxPC (0.1, 0.2, 0.5) and uMin2 (2000, 2500) after the second time computational compression is used with errorMaxPC = 0.2 and uMin2 = 2000.

the proposed computational compression successfully was able to reduce this redundant computation. The computational compression captures the circuit topology information that is re-used on subsequent executions.

6.5. Impact on memory usage

As we have previously started, computational compression through memoization is a trade-off between memory consumption and performance enhancement opportunity. The more computed elements are stored in memory, the more opportunities there are for reusing them. But memory is limited, and it has to be used carefully. Figure 7 shows the impact on memory usage the first time the RPCS is executed with computational compression, without reusing former execution results, and the second time the RPCS is executed, with computational compression and reusing previously saved results. The quantity of main memory used for the hash table on the first and second executions is 16.3 GiB.

We have proposed the usage of two optimizations for improving the speed and reducing memory footprint usage while saving and retrieving from the permanent storage subsystem: packing and compression. The memory space for the previously computed pair and its associated metadata for being in its hash-table are allocated at once. As Figure 8 shows, by packing both into a single memory space we can reduce the number of times the allocation routine is invoked (by half), and the metadata needed for the memory allocator subsystem (at least for the size allocated and the pointer to the next element allocated).

Compression is used for storing the input-output pairs on a permanent storage subsystem between

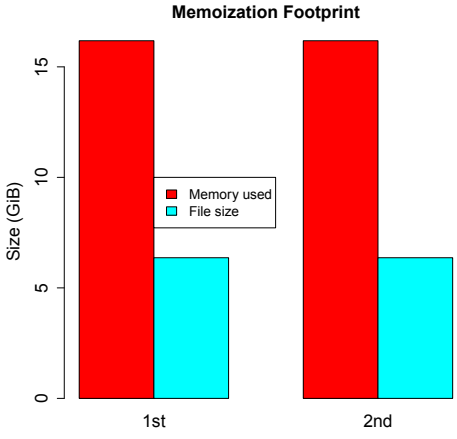

Figure 7: Memoization footprint on memory and disk.

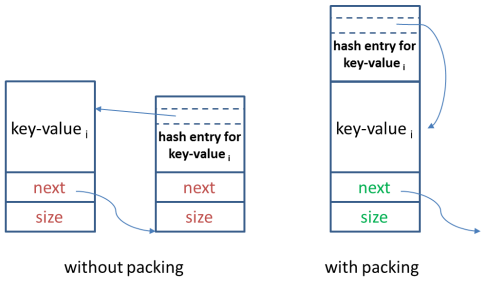

Figure 8: Proposed packing of key-value and its associated hash entry into a single allocation request.

executions (a hard disk in our evaluation). In our evaluation, the file size (by using the default compression level) is shrunk by around 6.5 GiB (less than half). In order to keep only the meaningful pairs, only the ones that have been reused at least once during the execution are saved. Nevertheless it would be possible to change this through a threshold that defines how many times a pair has to be reused to be saved (e.g. zero for storing all pairs). Figure 7 also shows the size of this file.

We think that it is also interesting to know how many elements are stored, and which ones are the most re-used. Figure 9 shows the number of pairs stored in the hash table, and the elements that have been used more than twice in the execution of the RPCS simulator with computational compression of the inv routine. The number of saved results (number of input-output pairs stored in the hash table) is 2346 in both executions, first and second execution with computational compression.

Figure 10 shows the percentage of accesses that are inputs that we had already computed (hits), and the percentage of accesses that are inputs whose associated output hadn't previously been computed (misses). The misses go from 14.0% in the first execution down to 0.0% in the second execution (because all the results of the optimized function are reused). Figure 10 also shows that there are a lot of hits but with small size matrix, thus the impact on time of these saved computation is not so high for the first execution. We have analyzed the dimensions of the matrices involved in the simulation done with the described workload. We have found two main type of dimensions used: 2x2 and 805x805. The relative frequency of the 2x2 matrices is 83%, and the relative frequency of 805x805 matrices is 17%.

6.6. Computational compression results: multithreading

The RPCS simulator can be executed using threads in order to divide the workload, and to perform the simulation in parallel. While searching for an input can be done in parallel without problems, the addition of a new pair of input-output values cannot be carried out in parallel with other additions or

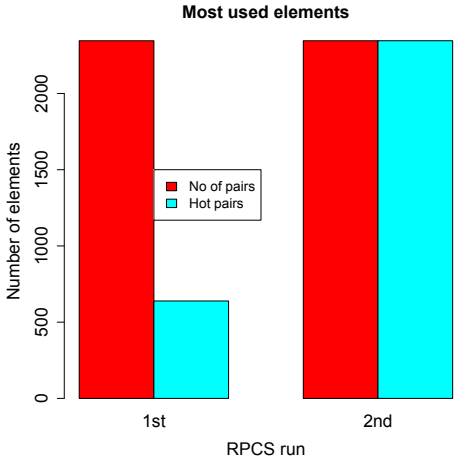

Figure 9: The most re-used elements for the two consecutive executions of the RPCS.

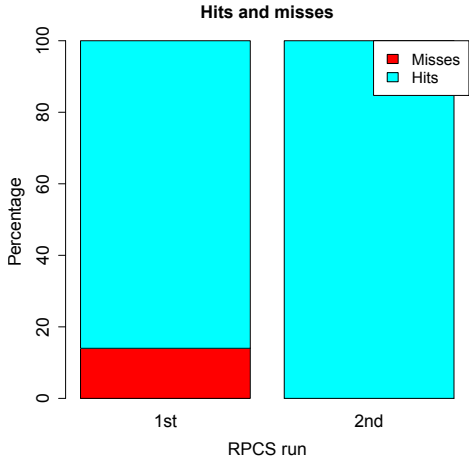

Figure 10: Memoization hits/misses for the two consecutive executions of the RPCS.

with other searches. For this reason, we use a read-write locking protection mechanism (rwlock) in order to protect concurrent writes, and concurrent read-write requests.

Figure 11 shows the simulation times of the scenario tested in previous sections using one, two, and four threads, as a percentage of the simulation time of the original RPCS simulator (without computational compression) with one thread. The first bar in each case is the simulation time without computational compression, the second bar is the simulation time the first time the computational compression is used, and the third bar is the simulation time the second time the computational compression is used (so former results are re-used).

The results are very interesting. Firstly, when the number of threads is increased, the simulation time drops. For example, with two threads the simulation time is 73.2% of the simulation time of executing with one thread and not using computational compression. With four threads the simulation time is 53.9% of the baseline. This behavior in general is not linear because two main factors: the CPU overhead of switching many threads rather than executing the simulation, and when the workload slice for its thread is so small that the overhead for creating threads and sharing data is higher than the corresponding thread simulation. Secondly, in some cases (with two and four threads) the first execution with computational compression needs a little more time. This small overhead is due to the strict locking protocol: the use of read-write locks, and when a thread adds a result to be memoized, there is a critical section where only one thread can be executed (no other thread can modify, and none of the other threads can search for a previous computation). For example with two threads the overhead is around 1%, and with four threads is around 1,7%. In future versions, this protocol could be replaced by an optimistic one to reduce this overhead. Thirdly, in all cases (one, two, and four threads), the second time the computational compression is used, by having all previously computed values, the simulation times greatly decrease.

For example, with four threads the second time that computational compression is used, the simulation is around six times faster than the original. There are many elements fetched (by a reader thread), and in this critical section, all readers can be executed in parallel.

To summarize, in the first execution there is a small overhead because of the locking protocol but the results prove that it is possible to combine computational compression and multi-threading for further improvement.

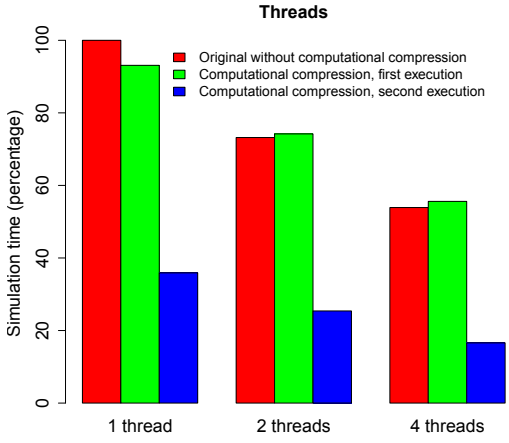

Figure 11: Simulation time executing the RPCS simulator with 1, 2, and 4 threads.

6.7. Computational compression results: data compression level

If the user wants to reuse memoized values across different work sessions (several executions of simulations), these memoized values must be stored in a permanent storage, and loaded again later on. In this paper we are using a file in a hard disk as a permanent storage subsystem, and we use data compression to reduce the size of this file.

Data compression is especially useful for systems with high CPU performance but low I/O bandwidth, where it is possible to reduce the amount of data to be transferred and execution time to be saved globally. But the selected compression level to be used depends on these two factors in each platform (CPU performance and I/O bandwidth). For example, low CPU performance or high I/O bandwidth requires a lower compression level. We have used the zlib library that is free and portable across multiple platforms [34]. This library supports the algorithm called DEFLATE, that is a variation of LZ77. We have evaluated the lowest compression level (no compression), the highest compression level, and the best speed compression level. We want to know the differences in our scenario (platform, simulator, and workload) among these three compression levels.

As Figure 10 shows, the simulation workload used for the first execution is one of the worst scenarios

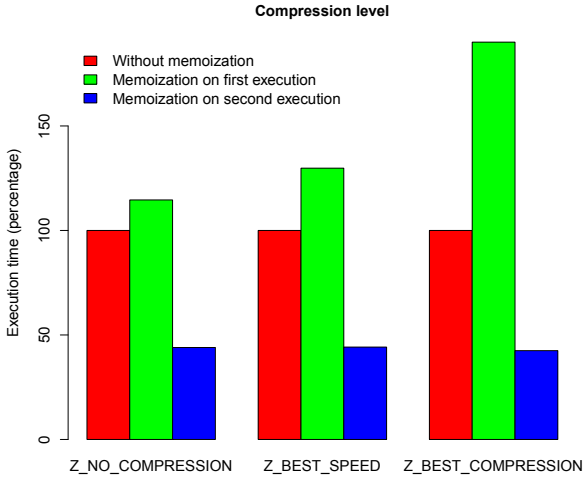

Figure 12: RPCS execution time with three data compression levels.

because the impact of the saved time with this workload is very little. The same simulation workload used for the second execution is one of the best scenarios because all previously computed values are reused. One of the worst cases let us to measure the overhead that our proposal introduces, while one of the best cases let us to measure the best improvement our proposal could reach for the studied conditions.

In Figure 12, we can see the execution time for three compression levels: without compression; the best speed compression level; and thirdly the best compression option. The first bar in each case is the simulation time without computational compression that is shown as a reference value. The second bar is the execution time relating to the first time the computational compression is used, and the third bar is the execution time the second time relating to the computational compression is used. All values are the percentages relative to the reference values (without computational compression), and computational compression values include both, compression and decompression time.

The results show that the simulation time with computational compression is increased when a higher level of compression is requested. With more compression, less data has to be stored in secondary memory but more time is needed to compress more information. If the compression time is increased, it will not be worth the time saved by transfer or storing less data. As said before, with few redundant computation for the first execution, this workload becomes the worst scenario. The overhead measured in the experiment with the described conditions is an extra 15% of execution time.

The second time the computational compression is used, by having all previously computed values with the same workload, no more elements are added, therefore, the time is the same because no new elements are compressed. But it is interesting to notice that even by reading and decompressing the input-output pairs is still room for an improvement of 50% in time. By using NVDIMM in the future

we expect further improvements: time for compressing/decompressing and saving/restoring is gone.

7. Conclusion and future works

Computational compression through memoization provides a way to obtain results from function calls that are likely to be executed several times with the same parameters. The first time the function call is performed, its results are computed and stored. But the next time a call to the same function with the same input happens, the previously computed result is searched and used. In general, an arbitrary application will benefit from computational compression whenever the time to compute the results is longer than the time to retrieve the previously cached results from memory. Therefore, both computing intensive and I/O intensive applications may benefit from this technique.

The work introduced here provides a modular architecture for storing the computed values that is both flexible and simple and also helps to achieve the best performance. It also introduces a systematic process to apply computational compression through memoization, and a way for re-using the memoization. All of it can be used in multi-threaded applications.

In order to demonstrate the feasibility of our proposal we have used a Railway Power Consumption Simulator (RPCS) as a case study for adding our proposed computational compression. It is written in C++, and has 13,701 lines of source code. The evaluation shows that it is possible to reduce the average total execution time by up to 64%. In other words, the simulation with the proposed computational compression lets us run the workload used up to 2.7 times faster, just by applying the optimization to the most CPU-demanding function.

In future projects, we will be working to extend our previous work for pre-processing C code in order to semi-automatically add computational compression. The idea is to let the user add some pragma/attribute before the function he/she wants to apply computational compression to; a pre-processor stage will automatically replace the pragma/attribute with the appropriate code. We will work at a function or task level such as [37]. We also are working in thresholds for programmers that define some limits. For example, where the computation is so small that he/she wants better compute rather than computational compress, or for example the soft limit of the number of elements to be memoized. We are working on big-size matrix support by using a hash to represent the matrix. The drawback of this technique is a hash has not warranty to be unique (although collisions are really minimized). The use of hashing let us going a step forwared: deal with approximate results (e.g. imprecise computation [38]). We are studying carefully the accumulated error, and future works go in the direction to balance the speed and the error in the approximate result obtained.

One major research line derived from this work includes the definition of automatic heuristics to detect which functions are candidates to be memoized. This heuristics will open up the possibility of creating automatic tools for memoizing modern applications, after working on breaking down the adoption barriers

commented in [39]. The programmers could also add an attribute (or pragma) to force one particular function to be memoized. We also want to provide several alternatives for storing memoization results (inputs-outputs pairs) for the same function to be optimized, and switch to the most appropriate one dynamically (similar to autotuning [39]). Additionally, we would like to extend our study to include different kinds of parallel applications (e.g. OpenMP/OpenACC/CUDA based), as we think that this technique could be also very effective in accelerators.

8. Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness under the project TIN2013-41350-P (Scalable Data Management Techniques for High-End Computing Systems).

[1] Görlach K, Sonntag M, Karastoyanova D et al. *Conventional Workflow Technology for Scientific Simulation*. Guide to e-Science, Springer-Verlag. ISBN 978-0-85729-438-8, 2011. pp. 323–352. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2011-01&engl=1.

[2] Hoffman J and Frankel S. *Numerical Methods for Engineers and Scientists, Second Edition,*. Taylor & Francis, 2001. ISBN 9780824704438. URL https://books.google.es/books?id=VKs7Afjkng4C.

[3] Aitken M, Broadhurst B and Hladky S. *Mathematics for Biological Scientists*. Garland Science, 2010. ISBN 9780815341369. URL https://books.google.es/books?id=eShFAQAAIAAJ.

[4] Sengupta C. *Financial Modeling Using Excel and VBA*. Wiley Finance, Wiley, 2004. ISBN 9780471651093. URL https://books.google.es/books?id=B3GPL__ehRsC.

[5] Chen W and Deelman E. WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments. https://www.ci.uchicago.edu/escience2012/pdf/WorkflowSim-A_Toolkit_for_Simulating_Scientific_Workflows_in_Distibuted_Environments.pdf, 2012. URL https://www.ci.uchicago.edu/escience2012/pdf/WorkflowSim-A_Toolkit_for_Simulating_Scientific_Workflows_in_Distibuted_Environments.pdf.

[6] Chen W and Deelman E. Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science)*. E-SCIENCE '12, Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4673-4467-8, pp. 1–8. DOI:10.1109/eScience.2012.6404430. URL http://dx.doi.org/10.1109/eScience.2012.6404430.

[7] Antolk J and Davison AP. Integrated workflows for spiking neuronal network simulations. *Frontiers in Neuroinformatics* 2013; 7(34). DOI:10.3389/fninf.2013.00034. URL `http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2013.00034/abstract`.

[8] Shen Q, Pang A and Uselton SP. Data level comparison of wind tunnel and computational fluid dynamics data. In *IEEE Visualization*. pp. 415–418. URL `http://dblp.uni-trier.de/db/conf/visualization/visualization1998.html#ShenPU98`.

[9] Bentley JL. *Writing efficient programs*, chapter Appendix C as "Space-For-Time Rule 2.". Upper Saddle River, NJ, USA: Prentice Hall, Inc. ISBN 0-13-970251-2, 1982. pp. 1–170.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, volume 1, chapter The Info Object. University of Tennessee. ISBN EAN-1114444410030, 2012. pp. 367–372. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[11] Calderón A, Jesús C, García-Carballeira F et al. Improving mpi applications with a new mpi_info and the use of the memoization. In *Proceedings of the 20th European MPI Users' Group Meeting*. EuroMPI '13, New York, NY, USA: ACM. ISBN 978-1-4503-1903-4, pp. 7–12. DOI:10.1145/2488551.2488554. URL `http://doi.acm.org/10.1145/2488551.2488554`.

[12] Caíno-Lores S, García A, García-Carballeira F et al. A cloudification methodology for numerical simulations. In Lopes L, Žilinskas J, Costan A et al. (eds.) *Euro-Par 2014: Parallel Processing Workshops*, *Lecture Notes in Computer Science*, volume 8806. Springer International Publishing. ISBN 978-3-319-14312-5, 2014. pp. 375–386. DOI:10.1007/978-3-319-14313-2_32. URL `http://dx.doi.org/10.1007/978-3-319-14313-2_32`.

[13] Saa R, Garcia A, Gomez C et al. An ontology-driven decision support system for high-performance and cost-optimized design of complex railway portal frames. *Expert Systems with Applications* 2012; 39(10): 8784 – 8792. DOI:http://dx.doi.org/10.1016/j.eswa.2012.02.002. URL `http://www.sciencedirect.com/science/article/pii/S0957417412002436`.

[14] Cui H, Wu J, Tsai CC et al. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX conference on Operating Systems design and implementation*. USENIX Association, pp. 1–13.

[15] Schnarr E and Larus JR. Fast out-of-order processor simulation using memoization. *SIGOPS Oper Syst Rev* 1998; 32(5): 283–294. DOI:10.1145/384265.291063. URL `http://doi.acm.org/10.1145/384265.291063`.

[16] Yamada D, Sonobe T, Tezuka H et al. Grid spider: a framework for data intensive research with data process memoization cache. In *INTENSIVE 2012, The Fourth International Conference on Resource Intensive Applications and Services*. pp. 5–8.

[17] Agosta G, Bessi M, Capra E et al. Automatic memoization for energy efficiency in financial applications. *Sustainable Computing: Informatics and Systems* 2012; 2(2): 105 – 115. DOI:10.1016/j.suscom.2012.02.002. URL http://www.sciencedirect.com/science/article/pii/S2210537912000066.

[18] Costa ATD, Franca FM and Filho EMC. The dynamic trace memoization reuse technique. In *In 9th PACT, p. 92-99, 2000, IEEE CS*. pp. 92–99.

[19] Kamimura K, Oda R, Yamada T et al. A speed-up technique for an auto-memoization processor by reusing partial results of instruction regions. In *Networking and Computing (ICNC), 2012 Third International Conference on*. IEEE, pp. 49–57.

[20] Richardson SE. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, Mountain View, CA, USA, 1992.

[21] Bratko I. *Prolog programming for artificial intelligence*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-14224-4.

[22] Ziarek L, Sivaramakrishnan K and Jagannathan S. Partial memoization of concurrency and communication. *SIGPLAN Not* 2009; 44(9): 161–172. DOI:10.1145/1631687.1596575. URL http://doi.acm.org/10.1145/1631687.1596575.

[23] Mulalic EH, Stankovic MS and Stankovic RS. Memoization technique for optimizing functions with stochastic input. *CoRR* 2012; abs/1211.5173. URL http://dblp.uni-trier.de/db/journals/corr/corr1211.html#abs-1211-5173.

[24] Hall M and McNamee JP. Improving software performance with automatic memoization. In *Johns Hopkins APL Technical Digest*, volume 18-2. Applied Physics Laboratory, pp. 254–260.

[25] Chaos A. Another Look at my old Benchmark. http://attractivechaos.wordpress.com/2008/10/07/another-look-at-my-old-benchmark/, 2008. URL http://attractivechaos.wordpress.com/2008/10/07/another-look-at-my-old-benchmark/.

[26] Troy HD. UThash. http://uthash.sourceforge.net/, 2012. URL http://uthash.sourceforge.net/.

[27] Kovacs K. Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris comparison. http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis, 2013. URL http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis.

[28] Hecht R and Jablonski S. Nosql evaluation: A use case oriented survey. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*. CSC '11, Washington, DC, USA: IEEE Computer Society. ISBN 978-1-4577-1635-5, pp. 336–341. DOI:10.1109/CSC.2011.6138544. URL http://dx.doi.org/10.1109/CSC.2011.6138544.

[29] Rito H and Cachopo Ja. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. PPPJ '10, New York, NY, USA: ACM. ISBN 978-1-4503-0269-2, pp. 89–98. DOI:10.1145/1852761.1852775. URL http://doi.acm.org/10.1145/1852761.1852775.

[30] Stivala A, Stuckey PJ, Garcia de la Banda M et al. Lock-free parallel dynamic programming. *J Parallel Distrib Comput* 2010; 70(8): 839–848. DOI:10.1016/j.jpdc.2010.01.004. URL http://dx.doi.org/10.1016/j.jpdc.2010.01.004.

[31] Jarod24. Memoization functor wrapper in C++. http://stackoverflow.com/questions/24637619/memoization-functor-wrapper-in-c, 2014. URL http://stackoverflow.com/questions/24637619/memoization-functor-wrapper-in-c.

[32] Intel and Micron. Intel and Micron Produce Breakthrough Memory Technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, 2015. URL http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[33] Panitkin SY, Benjamin D, Montoya GC et al. Distributed analysis with proof in atlas collaboration. *Journal of Physics: Conference Series* 2010; 219(7): 072014. URL http://stacks.iop.org/1742-6596/219/i=7/a=072014.

[34] Jean-loup and Adler M. zlib. http://www.zlib.net/, 2014. URL http://www.zlib.net/.

[35] Jahn S, Margraf M, Habchi V et al. Qucs technical papers. URL http://qucs.sourceforge.net/tech/node14.html. http://qucs.sourceforge.net/tech/node14.html. Last accessed November 2015.

[36] BS-EN-50641. Railway applications. fixed installations. requirements for the validation of simulation tools used for the design of traction power supply systems. Technical report, CENELEC, 2014.

[37] Bellens P, Perez JM, Badia RM et al. Making the best of temporal locality: Just-in-time renaming and lazy write-back on the cell/b.e. *International Journal of High Performance Computing Applications* 2011; 25(2): 137–147. DOI:10.1177/1094342010369115. URL http://hpc.sagepub.com/content/25/2/137.abstract. http://hpc.sagepub.com/content/25/2/137.full.pdf+html.

[38] Liu JWS, Lin KJ, Shih WK et al. Algorithms for scheduling imprecise computations. *IEEE Computer* 1991; 24(5): 58–68. URL http://dblp.uni-trier.de/db/journals/computer/computer24.html#LiuLSYCZ91.

[39] Basu P, Hall M, Khan M et al. Towards making autotuning mainstream. *International Journal of High Performance Computing Applications* 2013; 27(4): 379–393. DOI:10.1177/1094342013493644. URL http://hpc.sagepub.com/content/27/4/379.abstract. http://hpc.sagepub.com/content/27/4/379.full.pdf+html.