FOLIA 204

Annales Universitatis Paedagogicae Cracoviensis

Studia Technica IX (2016)

ISSN 2081-5468

Karol Grondzak

Fast Algorithms for Reliability Importance Index Evaluation

Introduction

Reliability of any system is of great importance for its owner, as well as for its producer. More reliable systems have a higher value on the market and are more demanded by the customers. Naturally no system is 100 percent reliable and the reliability of the system varies in time. Information about the reliability of the components of the system can be useful for the system producer and for the user. From the point of the producer it can help to improve the overall reliability by improving the reliability of the critical component. From the point of the user it can indicate which component should be checked regularly to ensure that the system functions properly.

One of the goals of the reliability analysis is to allow for identifying the components which are critical from the point of view of proper functionality of some system (Grishkevich 2014: 249), (Grishkevich 2010: 97). The term "reliability importance" is used for this purpose.

Several reliability importance indices, such as Structure Importance, Birnbaum Component Importance, Reliability Criticality Importance, Upgrading Function and Operational Criticality Importance, were defined (Leemis 1995:110) and are widely used in engineering practice. Different indices have different purpose, from aiming at the greatest gain in system reliability improvement efforts, to providing the best assistance in system design and optimization or suggesting the most efficient way to operate a system or to prevent system failure.

One of the basic information about the system we can obtain is the calculation in which of the states, described by the states of its components it is in reliable state, e.g. it is functioning properly. In the rest of the paper we will concentrate on this task.

Formulation of the problem

Let us consider a Binary State System (BSS), which consists of n elements. From the point of view of the reliability of such system, it can be described by some function, defining

$$f(x) = F, F \in \{0,1\}, \quad [1]$$

where *i*-th component of vector *x*, denoted as x_i represents the state of *i*-th component of the system and its values are zero and one. Value zero represents the failure of the component, while value one represents the functional state of the component. The function f(x), known as structure function describes the overall state of the system, where value zero means the system failure, while value one represents the functional state of the system.

It is quite obvious that for a system with *n* components, of which both can be in two different states, the input vectors of function f(x) will have 2^n different values. It means that even for a relatively small amount of components the amount of input states is quite high. Determining the number of states of the system components for which the system is reliable can be quite time consuming. If we need to perform the evaluation many times e.g. in case that some optimization procedure is applied to improve the system reliability, the speed of the evaluation becomes important.

We can formulate our problem as follows: we have a vector which consists of values zero and one. The size of vector is 2^n. Our goal is to calculate how many of the values in a given vector are equal to one

In the following paragraphs we present several approaches to how to perform the evaluation and we demonstrate the possibilities of evaluation performance optimization.

The simplest possible algorithm to perform the just-formulated problem is shown in Fig. 1.

```
SUM ← 0
for i = 0 to sizeof(ARRAY) - 1 do
  SUM ← SUM + ARRAY[i]
    end for
```

Fig.1. Simple algorithm to perform formulated task
source: author

We consider this algorithm as the base and demonstrate how to improve it, considering the different properties of modern processors and operating systems.

Theoretical analysis

There are several possibilities of how to improve the performance of the basic algorithm presented in Fig. 1. We will start with considerations of how to access the input data. It is obvious that to process any data it must reside in the main memory of the computer. There are several ways how to get data from external storage into main memory in modern operating systems.

The traditional approach is to read data from file into a designated area in the main memory in chunks of fixed size. This approach means that all data are loaded into main memory, before they are further processed. The process of loading is fully controlled by the programmer which means that if the programmer is not aware of the computer system properties, the procedure may not be optimal.

Another approach to provide data for the program is to use the memory mapping technique. This technique is available in all modern operating systems which manage the main memory by virtual memory technique. Virtual memory technique allows the programs to allocate large pieces of memory without the limitations of how much physical memory is installed in the computer. It is obvious that such a approach can lead to significant performance issues. On the other hand it leaves the memory management on the operating system, which means that the techniques optimized for the particular computer system can be applied.

To use memory mapping technique there is a system call which allows asking the operating system to map content of a given file into address space of an application. The result of this memory mapping is a pointer to the virtual address space where the file is mapped. When the process accesses this region, the operating system automatically loads data from the file into the main memory.

Both of the above-mentioned approaches have their advantages and disadvantages, but the latter one is generally recommended for modern operating systems (Silberschatz 2002: 329). As will be demonstrated later, the proper choice of the data loading procedure can significantly influence the performance of the calculation.

Once we have data available in the main memory, we can concentrate our optimization effort on the calculation process itself. In the following paragraphs we will consider computers equipped with modern central processing units (CPU), compatible with Intel IA–32 architecture. We also consider that data are stored in bytes, e.g. each value of the structure function defined by equation (1) is stored in one byte.

Let us consider IA–32 Intel CPU architecture. It has several general-purpose registers which are 32 bit wide. Arithmetic and logic operations are performed with the use of these registers which allow us to perform one operation on 32-bit values in one instruction. In 1997 Intel Company introduced a single instruction multiple data (SIMD) instruction set into their processors (*Intel 64 and IA–32 Architectures Optimization*...). They allow performing operations on 64-bits of data in one instruction. The idea is to allow process packed data. Instead of performing the calculation on one 64 bit value, it is possible to perform the calculation on two 32-bit numbers, or four 16-bit numbers or eight 8-bit numbers. The successors of this instructions set are SSE, SSE2, SSE3 and SSE4 instructions sets. They utilize 128-bit wide registers and improved set of instructions. When utilizing SSE technology we process the array of bytes by 128 bits or 16 bytes at once in one operation.

To explain this idea, let us consider a simple system with four components. Such system states can be described by structure function having 16 states values for its four components (from state {0, 0, 0, 0} representing all components in the state zero to state {1, 1, 1, 1} when all components are in state one). These sixteen states will be stored in sixteen bytes, in each of which only the least significant bit (LSB) is occupied, all other bits are equal to zero. To determine the overall number of states in which the system is functional, we can sum up all the sixteen values of the system structure function into one accumulator (denoted as SUM in Fig. 1.). This would require fifteen addition operations. But when we consider 32-bit CPU architecture, we can load four bytes into a CPU register by one load operation and sum them up by four-tuples. It will require three addition operations to obtain partial result of summing up four four-tuples followed by another three addition operations to sum up the partial results (Fig. 2).

We can formally describe the process of summation reorganization as follows. Instead of a direct summation:

$$SUM = \sum_{i=0}^{15} b_1 \qquad [2]$$

which requires fifteen addition operations on byte values we will perform the following calculation:

$$SUM_j = \sum_{i=0}^{3} b_{4i+j}\,, j = 0,1,\cdots,3$$
$$SUM = \sum_{j=0}^{3} SUM_j \qquad [3]$$

which requires six addition operation, but in this case on 32-bit values instead of bytes. Considering the SIMD instructions, we can perform the summation of up to sixteen values simultaneously when using 128-bit registers. We can expect the performance improvement comparing to base algorithm shown in Fig. 1.

$$0000000b_0 \,|\, 0000000b_1 \,|\, 0000000b_2 \,|\, 0000000b_3$$
$$+$$
$$0000000b_4 \,|\, 0000000b_5 \,|\, 0000000b_6 \,|\, 0000000b_7$$
$$+$$
$$0000000b_8 \,|\, 0000000b_9 \,|\, 0000000b_{10} \,|\, 0000000b_{11}$$
$$+$$
$$0000000b_{12} \,|\, 0000000b_{13} \,|\, 0000000b_{14} \,|\, 0000000b_{15}$$
$$=$$
$$sum_0 \,|\, sum_1 \,|\, sum_2 \,|\, sum_3$$

Fig.2. Demonstration of the concept of parallel calculation of the sum
source: author

When speaking of modern multicore CPUs architectures, we can take into consideration the fact that they are equipped with several processing units. It is possible to employ all the processing units to perform the calculation faster. Nowadays compilers support the use of several processing units by the concept of threads. Thread of execution is the smallest unit, which is considered by scheduler. A group of threads within one process shares data, which makes the communication among them easier and reduces the communication overhead.

While it would be possible to process data in parallel by creating a set of threads and assign to them parts of input data to process, we will apply different approach.

Modern compilers implement OpenMP API, which is a high–level API for automatic parallel execution of program parts, especially loops. OpenMP API It is a shared memory parallel paradigm, designed and maintained by the consortium OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors. It supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms (Quinn 2003: 404). It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Experimental Results

The algorithm was implemented using five different strategies in C language and executed on computer equipped with processor Intel Xeon CPU E3–1220 and 8GiB of main memory. This processor is equipped with 4x32KiB of L1 cache, 4x256KiB of L2 cache and 8MiB of L3 cache. It was compiled using gcc compiler on 64-bit CentOS GNU/Linux distribution.

Tab. 1. Results obtained for different algorithms (column alg), problem size (column Size) when data are loaded into memory without using memory mapping (column Time1 and Relative speed-up1) and when applying memory mapping technique (column Time2 and Relative speed-up2).

Alg	Size	Time1 [s]	Relative speed-up1 (Grishkevich, Grishkevich 2014)	Time2 [s]	Relative speed-up2 (Grishkevich, Grishkevich 2014)
1	10	0,00000295	1	0,00000955	1
1	16	0,0000212	1	0,0000723	1
1	18	0,0000772	1	0,000242	1
1	20	0,000291	1	0,000833	1
1	24	0,00497	1	0,00969	1
1	28	0,0795	1	0,0993	1
1	30	0,319	1	0,393	1
2	10	0,00000285	1,034	0,00000286	3,346
2	16	0,0000274	0,773	0,0000265	2,730
2	18	0,000101	0,764	0,0000981	2,463
2	20	0,000365	0,797	0,000368	2,264
2	24	0,00678	0,733	0,00858	1,130
2	28	0,111	0,714	0,132	0,753
2	30	0,446	0,714	0,527	0,745
3	10	0,0000488	0,060	0,0000385	0,248
3	16	0,00243	0,009	0,00161	0,045
3	18	0,00966	0,008	0,00898	0,027
3	20	0,0398	0,007	0,0358	0,023
3	24	0,645	0,008	0,548	0,018
3	28	9,935	0,008	9,927	0,010

3	30	39,8	0,008	39,7	0,010
4	10	0,00000266	1,108	0,00000308	3,106
4	16	0,0000122	1,736	0,0000114	6,326
4	18	0,0000449	1,718	0,0000495	4,878
4	20	0,000181	1,605	0,000207	4,023
4	24	0,00359	1,385	0,00540	1,795
4	28	0,0588	1,352	0,0768	1,293
4	30	0,236	1,349	0,306	1,281
5	10	0,00000319	0,925	0,00000298	3,210
5	16	0,0000121	1,757	0,0000110	6,592
5	18	0,0000435	1,775	0,0000472	5,112
5	20	0,000162	1,799	0,000164	5,077
5	24	0,00357	1,391	0,00534	1,814
5	28	0,0593	1,340	0,0777	1,278
5	30	0,236	1,348	0,312	1,257

The main results are presented in Table 1. They summarize the results obtained for different implementations of the algorithm using different strategies to load data into the main memory. Column Alg indicates the version of algorithm used, column Size represents the exponent of the power of two of the data size (it means when Size=10, then we processed 2^{10} bytes). Column Time is the time of the execution in seconds. It was obtained as the average of ten executions of the algorithm. Column Relative speed-up gives the speed-up achieved with respect to the base algorithm number one.

First strategy denoted by number one in column Alg is the naïve, straightforward approach. It is a direct implementation of the algorithm presented in Fig. 1. This is a referential implementation used to compare and evaluate the other implementations.

Algorithm two represents modification of the first algorithm, where the loop was partially unrolled, as was described in theoretical analysis section (equation 2 and 3 and Fig. 2). It tries to benefit from the fact that data are stored in only least significant bite of each byte and employs 32-bit addition to process four bytes at once.

The third algorithm is implemented using OpenMP directives of the compiler. It is parallel implementation using threads to calculate the results. Fourth and fifth algorithms use special SIMD instructions, to process eight or sixteen bytes at once. These implementations are similar to second implementation, but use special 64 or 128 bit registers.

When comparing the data in the table it becomes clear that memory mapping approach to data access is in general faster than programmatic loading data into memory.

To be able to compare different implementations we have calculated relative speed-up for corresponding problem sizes. Relative speed-up is calculated for each result with respect to the basic, naïve implementation. As it can be seen, OpenMP implementation (alg. 3) did not achieve any speed-up regardless of data size. Our explanation is that because of the relatively simple nature of the calculation the overhead of thread creation and management is too high for such a simple task.

Unrolled algorithms benefiting from the fact that we can process several data simultaneously using 32-bit, 64-bit or 128-bit registers are the fastest. Among them algorithm 2 exhibits speed-up for combination of smaller task sizes up to 2^{24} and for the case when data are read into main memory directly (table 1). Its performance is poor for memory mapping access to data which can be explained by the overhead needed to access data. SIMD versions of this algorithm (alg 4 and alg 5) perform more consistently regardless of data access method. They are also more efficient on data of size up to 2^{24}, but perform well also on larger data.

Conclusions and future work

The goal of this paper was to demonstrate different possibilities of modern processor architectures and operating systems to perform simple calculation efficiently. We have presented two different possibilities of data access (traditional access using files and modern memory mapping approach) and demonstrated that memory mapping is generally a faster approach.

Next, we have presented different possibilities of how to speed up simple calculation, benefiting from the knowledge of the Central Processing Units of modern computers. We have presented the concept of SIMD instructions and demonstrated that their use can improve the performance of the calculation.

The performance analysis of algorithm which uses OpenMP instructions has indicated that explicit thread parallelization approach is not suitable in cases when the overhead of thread creation and management is larger than the size of the job calculated. Another explanation of the poor performance of this algorithm could be the necessity to flush data cache when switching among threads. Because each thread is processing a different part of data, when switching from one thread to another, the data cache in cache are invalidated and cache must be populated by new set of data from different part of input vector.

As an outcome of the performed analysis the author would recommend using memory mapping technique to access data files and, if possible, employ modern SIMD instructions of processors when possible. This is possible by the explicit use of them, or by enabling performance optimization of modern C/C+ compilers.

In the future, the author would like to continue the study of possible speed-up of the algorithm using Graphical Processor Unit (GPU). The actual trend is to employ modern GPUs for calculations as their computational power outperforms the CPU power by the factor of then or even hundredths. Of course, it is possible only for a specific kind of problems, which can benefit from the massively parallel architecture of modern GPUs.

References

Grishkevich A., Burmutaew A., 2010. Modelling the organization of maintenance and emergency repairs for calculating the reliability of electric power systems, The issue of renewable energy sources, operating forecasting in electric power systems: Series Monographs. Sekcja Wydawnictwa Wydziału Zarządzenia Politechniki Częstochowskiej, Częstochowa.

Grishkevich A., Grishkevich M., 2014. Interwałowe szacowania wskaźników niezawodności strukturalnej systemów elektroenergetycznych. *Przegląd Elektrotechniczny (Electrical Review)*, 88 (6).

Intel 64 and IA–32 Architectures Optimization Reference Manual, [online:] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia–32-architectures-woptimization-manual.html [access: 14.03.2016].

Leemis L.M., 1995. Reliability – Probabilistic Models and Statistical Methods. Prentice Hall, Inc., Upper Saddle River.

Quinn M.J., 2003. Parallel Programming in C with MPI and OpenMP. Mc Graw Hill, Boston.

Silberschatz A., Galvin P.B., Gagne G., 2002. *Operating System Concepts,* 6th ed. John-Wiley and Sons, New York.

Fast algorithms for reliability importance index evaluation

Abstract

Nowadays the reliability of products is of interest of their designers and users. It concerns any area of products, from simple home electronic devices to complex and critical industrial systems like atomic power plants or space ships. Depending on the complexity and structure of the investigated system, two basic mathematical models are used to evaluate the reliability – Binary State System and Multivalued State System. To identify the importance of some component of the system to its overall reliability, the importance index of the components is to be calculated. In this paper we present a comparison of different approaches how to calculate importance index of particular component of the investigated system.

Key words: system reliability, binary state system, multiple value state system, reliability importance index

Karol Grondzak
Pedagogical University of Cracow
Institute of Technology
ul. Podchorążych 2
30–084 Kraków, Poland
e-mail: grondz@up.krakow.pl