LAZY EVALUATION METHOD IN THE COMPONENT ENVIRONMENTS

Michał Kniotek

AGH University of Science and Technology, Kraków, Poland

Abstract: This paper describes the manually use of the lazy evaluation code optimization method in the component environments such as Java VM, MS .NET, Mono. Despite the implemented solutions in optimizers, there are occurrences when manual code optimization can accelerate execution of programs. In component environments, due to the optimization performed during JIT (Just In Time) compilation, the code cannot be fully optimized because of the short time available. JIT optimization takes place during execution of the currently used part of the code. That is the reason why the time spent on searching the best optimization methods must be balanced between the program response time and the choice of optimal optimization. This article presents optimization method ending with conclusion to answer in which component environment is recommended to use a given method manually. The presented method is called lazy evaluation.

Keywords: code optimization, component environments, lazy evaluation

1. Introduction

This article was written to prove that the use of manual code optimization allows to achieve additional speed as opposed to using only optimizers incorporated in the component environments. Similar research can be found in the article about loop optimization [16].

At present, the component environments are widely used for coding various programs, because of their main advantages: code portability between different operating systems and computer architectures; the standardized notation of the code, which allows for using universal optimization methods [1].

Currently used digital machines, reached its limit of computing due to the achievement of maximum technological potential capabilities, hence multiple programs may have long execution time. That is why, complicated calculations are performed in the schema of cloud computing. Even on a single machine, as well as on

Advances in Computer Science Research, vol. 10, pp. 53-65, 2013.

multiple machines, it is natural to search for the appropriate code optimizations that reduce the computation time.

There are two cases where optimization should be applied: programs whose running time is long, such as used in the geophysics (magnetotellurics) [13], or in programs where data must be processed quickly, such as in multimedia [17].

Optimization methods are currently looked for in many areas of science, such as: video reconstruction [7], the problem of resource block allocation for downlink Long Term Evolution (LTE) [10], acceleration of generalized minimum aberration designs of hadamard matrices [4], acceleration of element subroutines in finite element method [5], stereo images processing [8].

Optimization method presented in this article was examined. Method was tested on a different computer architecture and in different operating systems. In each case, program execution was timed, intermediate code examined (reverse engineering), and results summarized. The results allowed to answer when, why and in which component environments it is recommended to use a given method.

2. Compilation and optimizatfion methods

Compilation in all the environments mentioned in this article is performed in the same way. In the beginning, the source code is compiled through compilers provided with the package for developers into the intermediate code. The obtained files are then compiled once again by a virtual machine at the program's startup. Virtual machines compile the intermediate code to machine code which is executed by the processor. Compilation of the intermediate code to the machine code is performed by the JIT compiler; it takes place in stages and is subject to ad hoc optimizations at intervals set by the machine (thus the name: Just-in-Time).

JIT compilation occurs at the start of the application. It is performed at every launch of the program, because it could happen that the intermediate code has been moved to a different hardware platform or operating system. The purpose of the aforementioned compilation is to translate the intermediate code into the machine code of the currently used platform. Each code method or function is compiled only when there is need for it. Thus, the program can run without being fully compiled, because some methods may be unused. During the execution of the program, once the compiled parts are not lost and can be reused, they are loaded into the cache as ready to use the machine code [6,12,15].

JIT compilation is limited only by one factor: time. This is due to the fact of its execution during the launch of the application. Therefore, the analysis and code

optimization cannot last as long as they could at the AOT (Ahead-of-Time) compilation. [14] However, its advantages are code portability and application optimization suited to the currently used hardware and operating system. To compare, applications compiled using the JIT techniques run faster than scripts which are executed by interpreters [2].

The program can be implemented in many different ways and in all cases the result will be correct. However, certain approaches can be [3]:

- easier (the solution of the problem itself thanks to which the implementation can be easier and more understandable);
- cleaner (they use less memory);
- easier to maintain (to adapt the code to frequent changes and improvements);
- faster (time to obtain the result is shorter).

3. Methodology and test performance characteristics

In order to distinguish test environments, later in the paper they are named by shortcuts E64 and E32. Numbers designate operating systems (32 and 64 bits) installed in the respective test environments.

Hardware specification (E64): Intel® Core™ 2 E6400 @ 2.46GHz (CPU), 1.5GB DDR2-667 (640MHz) (RAM). Operating systems (E64): Microsoft Windows 7 Professional SP1 (64bit), Fedora 16 (64bit).

Hardware specification (E32): Intel® Celeron® M520 @ 1.60GHz (CPU), 512MB DDR2-667 (532MHz) (RAM). Operating systems (E32): Microsoft Windows XP Professional SP2 (32bit), Fedora 11 (32bit).

Virtual machines installed in an E32 and E64 environments: Microsoft .NET 4.0 (Windows), Microsoft .NET 2.0 (Windows), Java Development Kit 7u4 (Windows and Linux), Mono 2.10.8 (Windows), Mono 2.4.3 (Fedora 11 - E32), Mono 2.10.5 (Fedora 16 - E64).

In order to reach objective test results, all testing was performed in the same way, in accordance with the principles set out below.

- Tested instructions were carried out in a loop; the total time of their operation was measured. The loop had a predefined amount of iterations.
- If the test required sample data, they were randomized for each iteration of the loop. Randomizing took place before the measurement of time and the values were stored in arrays. Data were randomized in order to keep the conditions close to real application run. Thanks to randomization, the data were both pessimistic and optimistic (those that can make to return result faster or slower).The randomizer was initiated using current time.

- The measurements were performed for each test ten times. It is always the shortest time of all that was chosen. This approach is burdened with the smallest error, due to the applications that run in the background in a test environment [9].
- To measure time with nanosecond accuracy, methods provided together with executing environments were used. In Mono and .NET, the Stopwatch class from System.Diagnostics package was used, while in Java, it was the System.nanoTime() method.
- The use of the source code compilers to bytecode:
  - .NET, compiler csc with /o flag (optimization launch),
  - Mono, compiler mcs with -optimize+ flag (optimization launch),
  - Java, compiler javac (by default, optimization is turned on).
- The use of JIT compilers:
  - .NET, lack of interference in the applied optimization,
  - Mono, compiler mono with flag -O=all (turn on all possible optimization),
  - Java, compiler java with flag -XX:+AggressiveOpts (optimizations foreseen for the next release of JVM), also separately launched in mode -client and -server.

4. Performance of the lazy evaluation

The test is performed in order to compare the time spans after the use of optimization of the lazy evaluation. [11] The test cases provide for using optimization manually and the lack of the optimization. In addition, tests were carried out for different ranges of the variables that are permitted by conditional statements. This is performed to check whether there is a correlation between the range of the permitted variables in conditional statements and the time of the program execution. In addition, the test is designed to draw attention to the memory usage and the intermediate code after disassembling.

The test was performed as described by pseudo-code shown in Alg. 1. During the randomization of sample data vector (variable vector) it is worth noting that the numbers are randomized from the closed interval <-100;100>. Four variants have been provided for the test; the fundamental difference is shown by Tab. 1. In general, in each variant timeConsumingFunction() (Alg. 2) have been executed and random number from the first element of the vector variable have been checked. The content of the Tab. 1 with the appropriate variant should be inserted in the comment's place in the pseudo-code shown in Alg. 1. The timeConsumingFunction() is designed to perform several calculations (addition, multiplication, division with remainder) that perform longer than it takes to compare numbers. The condition upon which the

function printOnScreen(count) is executed is noteworthy. In testing, the condition was never fulfilled, but using it ensured that the optimizer does not recognize the count variable as unused in the code. Optimizers encountering unused variables in the code often use additional optimizations, which had an impact on the test. Typically, fragments of the code which handle such variables are treated as redundant, and this fragment of code is the main part of the test. Lack of that condition caused shortening of the times measured.

Algorithm 1 pseudo-code for the test of the lazy evaluation

```
Ensure: time
 1: for i=0 to i < 1 000 000; i++ do
 2:     for j = 0 to j < 10; j++ do
 3:         vector[i][j] = randomNumber()% 201 - 100;
 4:     end for
 5: end for
 6: startTime();
 7: for i=0 to i < 1 000 000; i++ do
 8:     if /* Different cases from Tab. 2 */ then
 9:         count++;
10:     end if
11: end for
12: stopTime();
13: if count > 500 000 then
14:     printOnScreen(count);
15: end if
```

Algorithm 2 timeConsumingFunction

```
Require: vector
Ensure: boolean
 1: sum = 0;
 2: var = 1;
 3: for i = 0 to i < 10; i++ do
 4:     sum += vector[i];
 5:     var *= vector[i];
 6: end for
 7: return ((sum + var) % 2 == 0) ? true : false;
```

Table 1. Variants for the test of the lazy evaluation

Variant	Code to insert
B1 – before optimization, permitted range (0;100), half of the range of values which have been randomized into the vector variable	timeConsumingFunction(vector[i]) && vector[i][0]<100 && vector[i][0]>0
B2 – after optimization, permitted range (0;100) , half of the range of values which have been randomized into the vector variable	vector[i][0]<100 && vector[i][0]>0 && timeConsumingFunction(vector [i])
B3 - before optimization, permitted range (50;100) , one fourth of the range of values which have been randomized into the vector variable	timeConsumingFunction(vector[i]) && vector[i][0]<100 && vector[i][0]>50
B4 - after optimization, permitted range (50;100), one fourth of the range of values which have been randomized into the vector variable	vector[i][0]<100 && vector[i][0]>50 && timeConsumingFunction(vector[i])

In Tab. 2 all the times measured during the test have been gathered. In addition, the acceleration of using various optimization variants with different order of conditions and ranges of permitted variables has been calculated.

The measured times in the test environment E64 and E32.

Execution time in variants B1 and B3 in every case is almost equal. Using optimization of the lazy evaluation (B2, B4) always reduced the duration of the program execution. More restrictive limit range of permitted values (variant B4) additionally reduces the program's execution time.

The measured times in the test environment E64.

The fastest is the program executed in .NET Framework and in the JVM, client version in Windows 7 in variant B4. The slowest is the program executed in variants B1 and B3 in the JVM, client version in Windows 7 and in Mono in Fedora 16.

The measured times in the test environment E32.

The fastest is the program executed in .NET Framework and in the JVM, client version on both operating systems in variant B4. The slowest is the program executed in variants B1 and B3 in the JVM, client version in both operating systems. Note the variant B4 in JVM client version on Fedora 11, where the execution time is almost equal to the same variant in E64 test environment on Fedora 16.

Table 2. Test results of the lazy evaluation

Environment			Time [ms]				Acceleration [%]	
			B1	B2	B3	B4	(B1-B2)/B1	(B3-B4)/B3
E64	Windows 7	.NET 2.0	30.918	21.645	30.398	16.446	29.99	45.90
		.NET 4.0	33.233	22.120	32.427	16.701	33.44	48.50
		Mono	41.313	28.161	41.305	20.334	31.84	50.77
		Java Client	50.554	20.104	49.915	16.061	60.23	67.82
		Java Server	32.990	22.159	32.781	19.872	32.83	39.38
	Fedora 16	Mono	50.927	33.221	50.068	25.772	34.77	48.53
		Java Client	37.411	28.111	37.797	23.177	24.86	38.68
		Java Server	39.251	26.705	35.841	24.076	31.96	32.83
E32	Windows XP	.NET 2.0	50.233	34.898	49.396	25.617	30.53	48.14
		.NET 4.0	53.881	35.941	52.697	26.166	33.30	50.35
		Mono	69.747	45.637	69.621	32.336	34.57	53.55
		Java Client	80.212	32.340	78.951	25.004	59.68	68.33
		Java Server	58.671	37.979	58.322	35.147	35.27	39.74
	Fedora 11	Mono	66.111	43.688	66.202	30.745	33.92	53.56
		Java Client	79.162	31.932	77.929	24.543	59.66	68.51
		Java Server	60.916	40.046	60.212	37.219	34.26	38.19

The acceleration in test environment E64 and E32.

The first acceleration was achieved after using the optimization of the lazy evaluation on the code from the B1 variant, while the second was obtained after using the optimization of the lazy evaluation on the code from the B3 variant. The main difference is the range limit of the first number from the vector variable in the conditional statement. In the mentioned cases, the acceleration of the program execution has been noted. There was a higher acceleration when the limiting range was from the interval (50;100). However, there is no linear relationship between the various ranges and acceleration.

The acceleration in test environment E64.

The smallest impact between the permitted range and acceleration is observed in the JVM, server version in Fedora 16. The highest acceleration occurs in the JVM, client version in the Windows 7 operating system. By optimizing the program in the JVM, client version in Windows 7, it has grown from the slowest to the fastest. The smallest acceleration was recorded in the JVM, client version in Fedora 16 with range (0;100).

The acceleration in test environment E32.

The smallest impact between the permitted range and acceleration is observed in the JVM, server version in both operating systems. The highest acceleration occurs in the JVM, client version in both operating systems. By optimizing the program in the JVM, client version in both operating systems, it has grown from the slowest to the fastest. The smallest acceleration was recorded in the JVM server version (both operating systems), .NET Framework and Mono (both operating systems) with range (0;100).

Below are presented the Java bytecodes of conditional statement after disassembling, which is the critical part of the test. Conditional statement in the B3 variant is presented on Tab. 3, while in the B4 variant on Tab. 4. It should be noted that during generation of the intermediate code by the compiler, it did not changed the sequence of conditions. In the B3 variant, the sequence of execution is: function timeConsumingFunction() (number 83:), comparison with the number 100 (number 92: and 94:), and finally a comparison with the number 50 (number 100: and 102:). While in the B4 variant the sequence of execution is (by changing the order obtained the acceleration in test): comparison with the number 100 (number 84: and 86:), comparison with the number 50 (number 92: and 94:), and finally execution of the function timeConsumingFunction() (number 99:), if the previous conditions were met.

Table 3. Disassembly of Java bytecode – lazy evaluation before optimization in the variant B3

```
81: aload_1
82: aload_2
83: invokevirtual #12; //Method timeConsumingFunction:([I)Z
86: ifeq   108
89: aload_2
90: iconst_0  91: iaload
92: bipush 100
94: if_icmpge     108
97: aload_2
98: iconst_0
99: iaload
100: bipush 50
102: if_icmple    108
```

Table 4. Disassembly of the Java bytecode – lazy evaluation after optimization in the variant B4

```
81: aload_2
82: iconst_0
83: iaload
84: bipush 100
86: if_icmpge     108
89: aload_2
90: iconst_0
91: iaload
92: bipush 50
94: if_icmple     108
97: aload_1
98: aload_2
99: invokevirtual #12; //Method timeConsumingFunction:([I)Z
102: ifeq  108
```

Below are presented managed code compiled using a csc compiler provided with .NET Framework disassembled by the Ildasm tool. It shows the managed code of conditional statement in the variants B3 (Tab. 5) and B4 (Tab. 6). It is worth noting that the instructions after compiling with mcs compiler provided with Mono environment are identical after disassemblation by the Ildasm tool. In the B3 variant the sequence of execution is: function timeConsumingFunction() (lines IL_0060 and IL_0065), comparison with the number 100 (line IL_006c), finally a comparison with the number 50 (line IL_0073). While in the B4 variant, the sequence of execution is (by changing the order obtained the acceleration in test): comparison with the number 100 (line IL_0063), comparison with the number 50 (line IL_006a), and finally execution of the function timeConsumingFunction() (lines IL_006e and IL_0073), if the previous conditions were met.

In summary, the best solution is to use optimization of the lazy evaluation in all environments. Its effectiveness depends on the permitted values in conditional statement. More restrictive conditions must be applied at the beginning of a conditional statement, because they allow eliminating most cases at the beginning and probably further check will not be necessary. Due to this, greater acceleration could be achieved, but it also still depends on the specific runtime environment. But we should not forget that more restrictive conditions may be much more expensive computationally. In this case, times should by measured again, because less expensive computationally conditions maybe should be checked earlier. There should be found

Table 5. Disassembly of thr managed code compiled with csc compiler and used Ildasm tool – lazy evaluation before optimization in the variant B3

```
IL_005e: ldloc.0
IL_005f: ldloc.1
IL_0060: callvirt  instance bool SprawdzanieWarunkowPRZED.
    SprawdzanieWarunkowPRZED:: timeConsumingFunction (int32[])
IL_0065: brfalse.s IL_0079
IL_0067: ldloc.1
IL_0068: ldc.i4.0
IL_0069: ldelem.i4
IL_006a: ldc.i4.s  100
IL_006c: bge.s     IL_0079
IL_006e: ldloc.1
IL_006f: ldc.i4.0
IL_0070: ldelem.i4
IL_0071: ldc.i4.s  50
IL_0073: ble.s     IL_0079
```

Table 6. Disassembly of the managed code compiled with csc compiler and used Ildasm tool – lazy evaluation after optimization in the variant B4

```
IL_005e: ldloc.1
IL_005f: ldc.i4.0
IL_0060: ldelem.i4
IL_0061: ldc.i4.s  100
IL_0063: bge.s     IL_0079
IL_0065: ldloc.1
IL_0066: ldc.i4.0
IL_0067: ldelem.i4
IL_0068: ldc.i4.s  50
IL_006a: ble.s     IL_0079
IL_006c: ldloc.0
IL_006d: ldloc.1
IL_006e: callvirt  instance bool SprawdzanieWarunkowPO.
    SprawdzanieWarunkowPO:: timeConsumingFunction (int32[])
IL_0073: brfalse.s IL_0079
```

the middle ground solution between the time of checking and the range of permissible values. The most important thing in this case is the programmer's knowledge about the problem. Particular attention should be paid to the fact that when the optimization is not applied, the effect of limiting the range is negligible. The use of optimization does not affect the additional memory consumption, which is at a constant level. *Garbage Collector* and the optimizations implemented in all environments can easily handle memory management.

5. Conclusions

The authorial contribution is the analysis of acceleration in the component environments after using various optimization with conclusions about it. Benchmarks presented in this paper may help developers to write their own code. On this basis, they can determine that in their case, the optimization would be effective or not. The analysis specifies opportunities to optimize the code in the various stages of the work with it. The entire testing process, allow to explain and understand what the optimizations do. Testing is also a time-consuming task, so they will not have to undergo the same testing process as shown in article, thanks to the results and conclusions to every test presented here. In the test, also the factors on which effectiveness of optimization method may depend have been pointed, such as the limiting range in the optimization of the lazy evaluation. In this article, the intermediate code was analyzed by using reverse engineering methods. The analysis of the intermediate code which provides instructions similar to those that would be executed by the processor, allows to understand the essence of the optimizations.

References

[1] Aho A. V., Lam M. S., Sethi R., Ullman. J. D. (2006), Compilers. Principles, Techniques, and Tools (second edition), Prentice Hall, Upper Saddle River.

[2] Aycock J. (2003), A brief history of just-in-time., ACM Computing Surveys, 35(2), 97–113.

[3] Gray J. (2003), Writing Faster Managed Code: Know What Things Cost. MSDN Library, 06.2003.

[4] Calhoun, J., Graham, J., Hong Zhou, Hai Jiang (2012), Acceleration of Generalized Minimum Aberration Designs of Hadamard Matrices on Graphics Processing Units., 2012 IEEE 9th International Conference on High Performance Computing and Communication, Liverpool, 25-27 June 2012.

[5] Filipovic, J., Fousek, J., Lakomy, B., Madzin, M. (2012), Automatically Optimized GPU Acceleration of Element Subroutines in Finite Element Method., 2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC), Chicago IL, 10-11 July 2012.

[6] Hind M. (2006), Dynamic Compilation and Adaptive Optimization in Virtual Machines, http://www.research.ibm.com/people/h/hind/ACACES06.pdf, available 07.07.2012.

[7] Jones, D.R., Schlick, R.O., Marcia, R.F. (2012), Compressive video recovery with upper and lower bound constraints., 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, 25-30 March 2012.

[8] Kaaniche, M., Pesquet-Popescu, B., Pesquet, J.-C. (2012), l1-adapted non separable vector lifting schemes for stereo image coding., Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European, Bucharest, 27-31 Aug. 2012.

[9] Kalibera T., Tuma P. (2006), Precise regression benchmarking with random effects: Improving Mono benchmark results., Formal Methods and Stochastic Models for Performance Evaluation, LNCS, 4054, 63–77.

[10] Lin S., Ping W., Fuqiang L. (2012), Particle swarm optimization based resource block allocation algorithm for downlink LTE systems., 2012 18th Asia-Pacific Conference on Communications (APCC), Jeju Island, 15-17 Oct. 2012.

[11] McCarthy J., Abrahams P. W., Edwards D. J., Hart T. P., Levin M. I. (1962), LISP 1.5 PROGRAMMER'S MANUAL, http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD0406138&Location=U2&doc=GetTRDoc.pdf, available 05.12.2013.

[12] Microsoft (2004), CLR (Common Language Runtime), http://msdn.microsoft.com/pl-pl/netframework/cc511286, available 05.12.2013.

[13] Mudge, J.C., Chandrasekhar, P., Heinson, G.S., Thiel, S. (2011), Evolving Inversion Methods in Geophysics with Cloud Computing - A Case Study of an eScience Collaboration. E-Science (e-Science), 2011 IEEE 7th International Conference on 5-8 Dec..

[14] Noriskin G. (2003), Writing High-Performance Managed Applications: A Primer., MSDN Library, 06.2003.

[15] Oracle Documentation (2009), Understanding JIT Compilation and Optimizations, http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html, available 05.12.2013.

[16] Piórkowski A., Żupnik M. (2010), Loop optimization in manager code environments with expressions evaluated only once., TASK QUARTERLY, 14(4), 397–404.

[17] Tsai, M.-H., Sung, J.-T., Huang, Y.-M. (2010), Resource management to increase connection capacity of real-time streaming in mobile WiMAX., Communications, IET 4(9), 1108 - 1115.

METODA „LAZY EVALUATION" W ŚRODOWISKACH KOMPONENTOWYCH

Streszczenie Artykuł opisuje użycie metody optymalizacji kodu "lazy evaluation" w środowiskach komponentowych (Java VM, MS .NET, Mono). Pomimo zaimplementowanych rozwiązań w optymalizatorach, występują przypadki, gdy doraźne zoptymalizowanie kodu skutkuje przyspieszeniem pracy programu. Optymalizacja kodu jest przeprowadzana podczas kompilacji JIT (Just In Time) w środowiskach komponentowych, dlatego kod nie może zostać w pełni zoptymalizowany. Optymalizacja i kompilacja następuje w momencie wywołania danej części kodu przez aplikację. Skutkuje to ograniczonym czasem, który jest dostępny na poszukiwanie najlepszej optymalizacji. Dostępny czas musi zostać zbalansowany pomiędzy czas odpowiedzi programu, a wybór optymalnej metody optymalizacji. Artykuł zakończono wnioskami, które pozwalają odpowiedzieć na pytanie, kiedy użycie metody "lazy evaluation" jest zalecane.

Słowa kluczowe: optymalizacja kodu, środowiska komponentowe, lazy evaluation