TASK QUARTERLY 14 No 4, 397–404

# LOOP OPTIMIZATION IN MANAGED CODE ENVIRONMENTS WITH EXPRESSIONS EVALUATED ONLY ONCE

ADAM PIÓRKOWSKI AND MAREK ŻUPNIK

*Department of Geoinfomatics and Applied Computer Science,*
*AGH University of Science and Technology,*
*Al. Mickiewicza 30, 30-059 Cracow, Poland*
*pioro@agh.edu.pl*

(Received 3 December 2010; revised manuscript received 16 December 2010)

**Abstract:** This paper is concerned with optimizing code execution in virtual machine environments. Code motion is one of the many optimization techniques. We considered a special case of optimization – a loop containing expressions that can be effectively evaluated once. A solution for this case is discussed and two algorithms are proposed. Experimental results for Java VM, MS .NET and Mono are shown here in order to assess the performance of the proposed algorithms.

**Keywords:** loop optimization, component platforms, compile-time optimization, compilers

## 1. Introduction

Component technologies have become a modern method of software development. These technologies have gained popularity due to their numerous advantages. One of these is code reuse, *i.e.* the ability to use the same code on various platforms. Such portability enables, among other things, the construction of heterogeneous distributed environments for domains that require substantial computing power.

Certain component software solutions require high computing power to simulate for example important geoseismic phenomena [1, 2]. The simulation process can take up to several weeks. Other solutions require high computing power to calculate the response as quickly as possible [3]. In all such applications, it is vital to optimize the execution time.

## 2. Optimization techniques

Leading component environments, such as Java Virtual Machine, the .NET Framework and Mono, use dynamic translation called Just-in-Time [4, 5]. The

code is compiled into *bytecode* (in the case of Java, and into the Common Intermediate language (CIL) in the case of .NET and Mono) and subsequently translated on the fly into processor instructions by the virtual machine. Unlike compilers, which employ Ahead-of-Time optimization, virtual machines optimize the code during runtime. There are numerous optimization algorithms [6, 7]. Commercial environments do not reveal the principles of their operation, the Mono environment, however, is an open-source project, where individual algorithms can be switched on or off [8]. Users can select a set of algorithms to be used – peephole postpass, branch optimizations, inline method calls, constant folding, constant propagation, copy propagation, dead code elimination, linear scan global register allocation, conditional moves, emitting per-domain code, instruction scheduling, intrinsic method implementations, tail recursion and tail calls, loop-related optimizations, leaf-procedure optimizations, SSA-based partial redundancy elimination and the removal of array-bound checks [9]. There are certain hardware-specific optimizations that use, for instance, SSE2 instructions and other processor features.

One of the loop related optimizations is code motion [10]. This technique consists in moving the loop-invariant computations in front of the loop [11–13]. The moved code yields the same result regardless of how many times the loop is executed. An example of this case is presented in Table 1.

**Table 1.** An example of code motion

| CM1 – Original code | CM2 – Optimized code |
|---|---|
| `t = 10;`<br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  a = t * 10;`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`}` | `t = 10;`<br>`a = t * 10;`<br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`}` |

The original code (CM1) contains three lines in the loop. The first line calculates the value of one of the variables. This expression can change the value of the variable '$a$' only once, at the start of loop execution, and therefore it can be evaluated once, before the loop is executed (*cf.* CM2).

## 3. The idea of moving expressions evaluated only once out of the loop

A specific case of a loop code is presented in Table 2, example A1. One of the variables ('$a$') is changed only once during the first iteration. This variable cannot be moved by a code motion algorithm, because it is used by the previous expression in the loop, when its value has not been changed yet. For all subsequent

iterations, this variable is independent of all other variables and its value remains the same. The equivalent code (Table 2, example B) contains a similar expression, which is related to the loop counter, and hence cannot be optimized.

**Table 2.** Source code for optimization and equivalent unoptimizable code

| A1 – original code | B – equivalent code |
|---|---|
| `t = 10;`<br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`  a = t * 10;`<br>`}` | `t = 10;`<br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`  a = i * 10;`<br>`}` |

The presented code (A1) can be optimized using two different methods (*cf.* Table 3). The first method (A2) consists in adding a flag that allows to execute the considered line only once (this requires a conditional instruction). The second method (A3) consists in rearranging the loop into two blocks – the first one, containing the line in question, is executed only once, whereas the second block is a loop that takes care of the subsequent iterations, where the line in question is absent. In addition, owing to its structure, the second method (A3) enables multithreading (if possible), because it does not contain any thread-critical sections.

The proposed algorithms can be manually applied to the source code.

**Table 3.** Proposed optimizations

| A1 – original code | A2 – optimization with a flag | A3 – optimization by the rearrangement of the loop |
|---|---|---|
| `t = 10;`<br><br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`  a = t * 10;`<br>`}` | `t = 10;`<br>`int flag = 0;`<br><br>`for(i=0;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`  if (flag == 0)`<br>`  {`<br>`  a = t * 10;`<br>`  flag++;`<br>`  }`<br>`}` | `t = 10;`<br><br>`i=0;`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`  a = t * 10;`<br>`}`<br><br>`for(i=1;i<N;i++)`<br>`{`<br>`  b = 20 * i;`<br>`  y+= a * b;`<br>`}` |

## 4. Performance results of the proposed algorithms

To assess the performance of the proposed algorithms, several tests were carried out in two testing environments – E1 and E2.

The first environment (E1) was a personal computer:

- CPU: Intel Pentium M 1.5 GHz,
- RAM: 1024 MB DDR PC2100 (133 MHz).

The second environment (E2) was a workstation:

- CPU: Pentium(R) Dual-Core CPU E6500 @ 2.93 GHz,
- RAM: 2 GB DDR2-667 (333 MHz).

The performance of the proposed algorithms was measured for the following operating systems:

- Linux Ubuntu 10.04,
- MS Windows XP (SP3).

The following component environments were installed:

- .NET 2.0 & 4.0,
- Sun Java JRE 1.6.20, tested with and without the *-server* option,
- Mono 2.6.4 (WinXP) and 2.4.4 (Linux), gmcs compiler.

Each loop was executed 100 000 times. The priorities of the tested threads were increased to avoid preemptions during the run. Each test was repeated several times, and the minimum value was taken. We used time measurement techniques that provide sufficient measurement accuracy of $\pm 1$ ms.

The default optimization level for the .NET and Mono environments is geared towards generating the fastest code, similarly to the O2 optimization level for C compilers. The Java environment offers the *-server* option, which performs optimization method testing, and therefore it is substantially more time-consuming, but only at the beginning.

The results observed in the .NET 2.0 and .NET 4.0 environments were identical. It was observed that the default settings in Mono produce very good results. The results obtained with the default settings were very similar to those obtained using the single-loop optimization method, therefore we present only the results for the default case.

In the first step, the performance of a standard code motion algorithm was measured. The results of this test are shown in Table 4. Figure 1 presents these results in the form of a chart.

Loop execution times measured during the tests for the proposed algorithms are collected in Table 5. Figure 2 shows the loop execution times for the three versions of the code (A1, A2 and A3). Figure 3 presents time savings achieved by the proposed algorithms (A2 and A3).

We observe that although the original code (CM1) and the equivalent code (CM2) are very similar, their loop execution times differ significantly. Therefore, despite the fact that virtual machines effectively optimize the original code, further improvement can be obtained by employing manual optimization (CM2).

**Table 4.** Loop execution times for the code motion algorithm

| environment | | | loop time [s] | |
|---|---|---|---|---|
| | | | CM1 | CM2 |
| E1 | WinXP | .NET 2.0 | 0.3354 | 0.2942 |
| | | Mono 2.6.4 | 0.3414 | 0.2844 |
| | | Java 1.6.0_20 | 0.6232 | 0.4582 |
| | | Java 1.6.0_20 S | 0.1958 | 0.0993 |
| | Linux | Mono 2.4.4 | 0.3684 | 0.3139 |
| | | Java 1.6.0_20 | 0.6721 | 0.4947 |
| | | Java 1.6.0_20 S | 0.2158 | 0.1074 |
| E2 | WinXP | .NET 2.0 | 0.1067 | 0.1029 |
| | | Mono 2.6.4 | 0.1215 | 0.0916 |
| | | Java 1.6.0_20 | 0.2049 | 0.1474 |
| | | Java 1.6.0_20 S | 0.0695 | 0.0438 |
| | Linux | Mono 2.4.4 | 0.1152 | 0.0910 |
| | | Java 1.6.0_20 | 0.1928 | 0.1474 |
| | | Java 1.6.0_20 S | 0.0697 | 0.0437 |

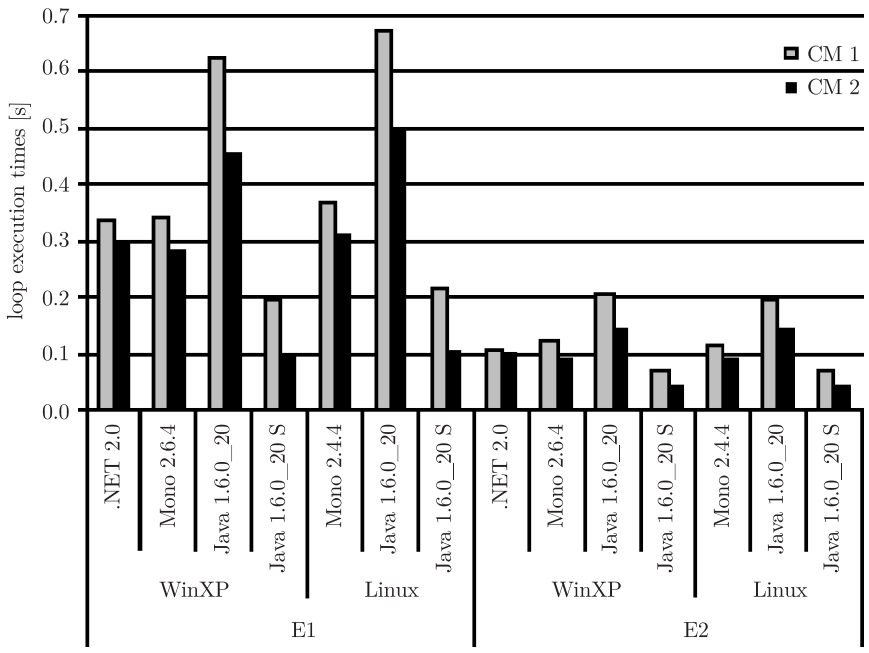


**Figure 1.** Loop execution times for the code motion algorithm

The proposed algorithm A3 was effective in all the tested cases. In several cases, it resulted in time savings exceeding 50% for the presented synthetic code. The algorithm A2 gave less satisfactory results, particularly in the case of the .NET environment where loop execution times were increased by 20%.

**Table 5.** Loop execution times for the proposed algorithms

| environment | | | loop time [s] | | | | ratio [%] | |
|---|---|---|---|---|---|---|---|---|
| | | | B | A1 | A2 | A3 | A2/A1 | A3/A1 |
| E1 | WinXP | .NET 2.0 | 0.3834 | 0.3101 | 0.3499 | 0.2869 | 112.83% | 92.52% |
| | | Mono 2.6.4 | 0.3576 | 0.3645 | 0.3529 | 0.2874 | 96.82% | 78.85% |
| | | Java 1.6.0_20 | 0.5794 | 0.5092 | 0.4715 | 0.4487 | 92.60% | 88.12% |
| | | Java 1.6.0_20 S | 0.3137 | 0.1986 | 0.0992 | 0.0964 | 49.95% | 48.54% |
| | Linux | Mono 2.4.4 | 0.3579 | 0.3609 | 0.3547 | 0.2860 | 98.28% | 79.25% |
| | | Java 1.6.0_20 | 0.5829 | 0.5101 | 0.4735 | 0.4500 | 92.82% | 88.22% |
| | | Java 1.6.0_20 S | 0.2908 | 0.1986 | 0.0992 | 0.0964 | 49.95% | 48.54% |
| E2 | WinXP | .NET 2.0 | 0.1297 | 0.1099 | 0.1211 | 0.1028 | 110.19% | 93.54% |
| | | Mono 2.6.4 | 0.1184 | 0.1419 | 0.1090 | 0.0923 | 76.81% | 65.05% |
| | | Java 1.6.0_20 | 0.1837 | 0.1534 | 0.1571 | 0.1474 | 102.41% | 96.09% |
| | | Java 1.6.0_20 S | 0.0931 | 0.0679 | 0.0430 | 0.0437 | 63.33% | 64.36% |
| | Linux | Mono 2.4.4 | 0.1416 | 0.1926 | 0.1095 | 0.0919 | 56.85% | 47.72% |
| | | Java 1.6.0_20 | 0.1133 | 0.0681 | 0.0431 | 0.0438 | 63.29% | 64.32% |
| | | Java 1.6.0_20 S | 0.1199 | 0.0682 | 0.0433 | 0.0439 | 63.49% | 64.37% |

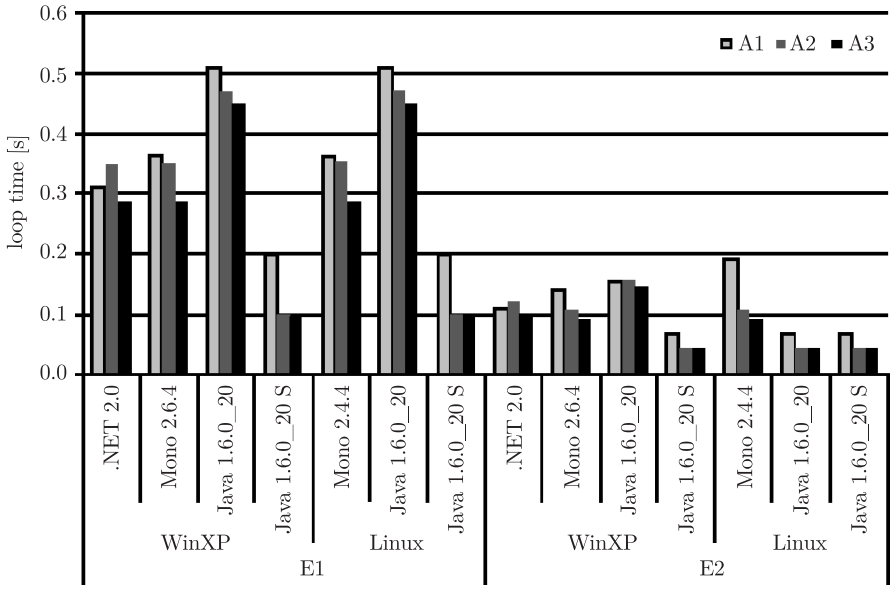


**Figure 2.** Loop execution times for the codes A1, A2 and A3

An interesting observation was made with regard to the performance of the code motion, *i.e.* that manually performed code motion is effective in the three aforementioned environments with Just-in-Time compilers.

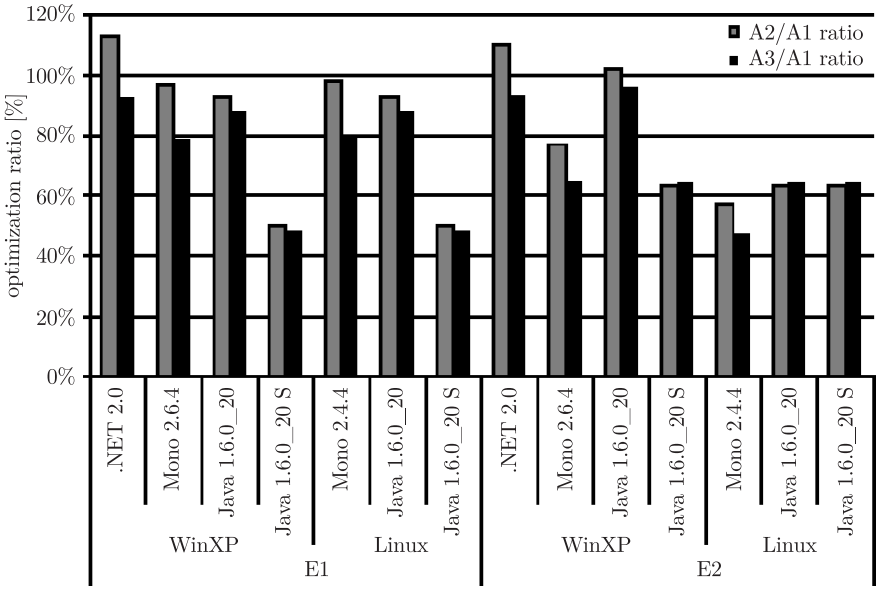


**Figure 3.** Time savings achieved by the algorithms A2 and A3

## 5. Conclusions

This article is concerned with the problem of numerical code optimization for leading virtual machine environments. We prove that manually performed code motion can be effective for these platforms. A new case of code motion, namely, that of code containing expressions evaluated only once in a loop is examined. This paper proposes two solutions – the first one is based on a flag-checking, the second one – on rearranging one loop into two blocks. Careful testing revealed that the second method was effective for all leading virtual machines. At present, the method must be applied manually and it allows to speed up the execution of the numerical code in specific cases, *i.e.* when the execution time is a crucial factor. Automatic optimization is a direction for future work.

### *References*

[1] Kowal A, Piórkowski A, Danek T and Pięta A 2010 *Innovations and Advances in Computer Sciences and Engineering*, Springer, pp. 359–362

[2] Kowal A, Piórkowski A, Pięta A and Danek T 2009 *Mineralia Slovaca*, supl. Geovestnik **41** (3) 361

[3] Piórkowski A and Plodzien D 2009 *16th Conf. Computer Networks, CN 2009* (Kwiecień A, Gaj P and Stera P, Eds), Wisła, Poland, Springer, Berlin-Heidelberg, CCIS, **39**, pp. 225–232

[4] Hoste K, Georges A and Eeckhout L 2010 *Proc. 8th Annual IEEE/ACM Int. Symposium on Code Generation and Optimization*, Toronto, pp. 62–72

[5] Vinodh Kumar R, Lakshmi Narayanan B and Govindarajan R 2002 *Proc. 9th Int. Conf. on High Performance Computing (HiPC-02)*, Bangalore, pp. 495–505

[6] Gampe A, Niedzielski D, von Ronne J and Psarris K 2010 *Safe, Multiphase Bounds Check Elimination in Java*, *Technical Report* Dept. of Computer Science, Univ. of Texas at San Antonio **CS-TR-2010-001**

[7] Gal A, Probst Ch W and Franz M 2006 *Proc. 2nd Int. Conf. Virtual Execution Environments (VEE '06)*, ACM, New York, USA, pp. 144–153
[8] Kalibera T and Tuma P 2006 *Formal Methods and Stochastic Models for Performance Evaluation*, LNCS, **4054** 63
[9] Mono Project homepage: http://www.mono-project.com/
[10] Aho A V, Sethi R and Ullman J D 1986 *Compilers: Principles, Techniques, and Tools*, Addison Wesley
[11] Song L, Futamura Y, Glück R and Hu Z 2000 *IEICE Trans. Information & System* **E83-D** (10) 1841
[12] Song L, Futamura Y, Glück R and Hu Z 2000 *Proc. IFIP Conf. Software: Theory and Practice (16th World Computer Congress 2000)*, Beijing, pp. 80–90
[13] Song L and Kavi K 2002 *Proc. 5th Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'02)*, pp. 0390