ELECTRICAL ENGINEERING

Benchmarking and performance enhancement framework for multi-staging object-oriented languages

Ahmed H. Yousef a,*, Tamer A. El-lateef b, Mona F. Ismail c

a Department of Computers and Systems Engineering, Ain Shams University, Cairo, Egypt
b Hewlett-Packard, Cairo, Egypt
c Department of Computers and Systems, Faculty of Engineering, Ain Shams University, Cairo, Egypt

Received 2 May 2012; revised 23 July 2012; accepted 28 August 2012
Available online 8 January 2013

KEYWORDS
Object-oriented languages; Multi-staging; Benchmarking; Performance enhancement framework; Java; C#; Performance analysis

Abstract This paper focuses on verifying the readiness, feasibility, generality and usefulness of multi-staging programming in software applications. We present a benchmark designed to evaluate the performance gain of different multi-staging programming (MSP) languages implementations of object oriented languages. The benchmarks in this suite cover different tests that range from classic simple examples (like matrix algebra) to advanced examples (like encryption and image processing). The benchmark is applied to compare the performance gain of two different MSP implementations (Mint and Metaphor) that are built on object oriented languages (Java and C# respectively). The results concerning the application of this benchmark on these languages are presented and analysed. The measurement technique used in benchmarking leads to the development of a language independent performance enhancement framework that allows the programmer to select which code segments need staging. The framework also enables the programmer to verify the effectiveness of staging on the application performance. The framework is applied to a real case study. The case study results showed the effectiveness of the framework to achieve significant performance enhancement.

© 2012 Ain Shams University. Production and hosting by Elsevier B.V. All rights reserved.

1. Introduction

Implementing a program with a high level of abstraction is a common goal for almost all software programmers. Abstraction mechanisms like functions, classes and recursion increase the software reusability, readability and maintainability. However, these abstraction mechanisms are always associated with accumulative runtime overhead. There are a lot of proposed approaches which try to ensure that abstraction can be used without the need to pay such overhead each time the code is executed. These approaches include macros or compile-time...
Meta-programming languages, partial evaluation and multi-stage languages.

Meta-programming systems are Meta-programs that manipulate object program code in some way [1]. This code manipulation may occur at compile time or run time. Compile-time Meta programming is concerned with code generation at compile-time. This includes C pre-processor that expands macros, C++ templates and Template Meta Haskell [2]. Macros are a synonym of this code generation process at compile-time. They expanded the Meta language code into code of the base language. Then, the resultant code is compiled as usual. The Meta-programming system is an extension to a base language. For example, C++ templates (Meta language) express parametric classes that could not be expressed in C++ (the object language). Generally, Program generation increases software reuse [3].

Partial evaluation [4–7] is based on the execution of some parts of the program code as soon as some program inputs are known. Usually these program inputs that is known early are called static inputs and used to evaluate the program partially. The remaining code that is not evaluated is named the residual program. It is executed later when the dynamic inputs are known. Since the residual program is usually faster than the original program, Partial evaluation is considered as a technique for program optimization [8,9]. For example, Tempo [8] is used as a partial evaluator for C and JSpec [9] is used as a partial evaluator for Java.

Multi-stage languages [10–12] are a Meta-programming system where the Meta and object languages are the same. The Meta-program generates specialized code for an object program. This optimization is done at runtime and dynamically improves the performance of a single stage program by changing the level of execution. Multi-stage programming (MSP) is a paradigm for developing generic software that does not pay a runtime overhead penalty for this generality [13,14]. This is achieved through concise, carefully-designed language extensions that support runtime code generation and program execution [1,10]. It depends on the reorganization of program’s execution and evaluation order. MSP gives the programmer a high degree of control compared to previously mentioned approaches.

MSP was first introduced in the context of functional language. This includes Meta ML that extends a functional subset of ML and supports the static type safety for the generated programs [12,15,16]. The Meta OCaml language is as an MSP extension of OCaml language [17,18]. There are other available MSP implementations [19–21] on several languages including Lisp. The Quasi-Quotation is one of the used implementation techniques [22].

Several Meta-programming language extensions exist for Java. This include Jumbo [23] for runtime code generation, Meta-AspectJ [24] for source code generation, JTS [25] for domain-specific language implementation and OpenJava [26] that is designed for offline generation of source code. However, all of these extensions were designed to support two level of staging and lack the full static type checking of the generated code. Although DynJava [27] has static type checking for dynamically generated code and SafeGen [28] is claimed to guarantee well-typedness of generated code, there is no rigorous proof or formalization of type safety. Formalization of staging languages can be found in [29–31]. 'C [32] is a fast run-time code generation extension to C but it is not a complete multi-stage language. Cyclone [33] is another C extension that supports type safety.

The first true multistage programming extension to an object oriented language is “Metaphor” that extended C# [34–37]. Metaphor is based on the introspection property of C# and Java. The most recently proposed MSP Java extension is ‘Java Mint’ [31,38,39]. It was based on extending the java compiler source code to support staging. Metaphor and Mint are considered the most complete implementations of multi-staging. Both of them support type safety and have strong formalization and proofs.

Although the previous mentioned literature was concerned with the formalization and implementation of Multi-staging programming, this paper focuses on benchmarking MSP and providing a framework that guides the programmer to use MSP efficiently to improve the performance of his applications.

As a result of the increasing number of MSP implementations in OOP languages, the need for an effective reference benchmark to compare between these implementations has become a must for several reasons. Firstly it will show how ready the multi-staging implementations for real code are. It shows how multi-staging improves the performance of code. In this paper, a proposed benchmark suite is introduced and applied to measure the performance of the multi-staging to both Mint and Metaphor.

Because MSP is still new outside research communities, it is not practically possible to leave the programmers who are new comers to MSP deciding which code segments is the best candidate to staging. The staging decisions cannot be entirely left to the programmer sense. To solve this issue, this work extended a performance enhancement framework that guides the programmer to improve his application’s performance using MSP. The performance enhancement framework was published initially in [40,41]. It is enhanced and extended in this paper to verify its applicability to multiple languages (Java and C#) and to overcome the problems found in the original framework.

There are two main contributions of this paper. The first one is a benchmark suite that can compare the effectiveness of current and future multi-staging implementations. The second one is a performance enhancement framework that enables the programmer to enhance the performance of software application without being an expert in multi-staging.

The contribution of this paper is significant for a number of reasons. It proves the feasibility of using multi-staging for performance enhancement and shows empirically that multi-staging can be used for larger and more realistic introductory programs that have multiple functions with deeper call tree. It shows that programmers with fair knowledge of the application domain can apply the performance enhancement framework to stage their programs easily without being experts in multi-staging. The framework will help him know where performance bottlenecks will occur. This will be helpful in the transition of multi-staging from being research incubated to a widely accepted add-on to commercial software development environment. Results showed that image processing and encryption application domains are good candidates for multi-staging. Results highlighted that Mint is more efficient and faster than Metaphor. The RSA encryption case study shows substantial performance gain when applying multi-staging on multiple evolutions of the RSA encryption code.
The remainder of this paper is organized as follows: Section 2 presents basic multi-staging background. In Section 3, the Multi-staging performance evaluation and the benchmarks are discussed. This includes the followed comparison methodology and the results of applying the benchmarks. A thorough analysis is presented. Performance enhancement framework is described in Section 4. Section 5 discusses the case study which is performed using the proposed framework. Finally, Section 6 concludes the paper and suggests ideas of future work.

2. Multi-stage programming background

The goal of staging is to improve a program based on a priori information about how it will be used. Staging is a program transformation that involves reorganizing the program’s execution into stages [42]. The concept of a stage arises naturally in a wide variety of situations. Compilation-based program execution involves two distinct stages: compile-time, and run-time. Generated program execution involves three stages: generation-time, compile-time, and run-time. For calculating the benefit cost analysis of the staging, the most commonly referenced performance cost-evaluation model assume that the overall cost is the cost of the last stage. Other models assign weights to each of the program stages. This weight is based on the reusability of the output of each stage as an input to the other stages [11].

For example, the construction of sin method table at code specialization time would save the cost of computing the method at runtime stage when executed many times. Here, a tradeoff for staging effectiveness can be defined based on the usage of the defined sin table values. If we wish to obtain the sin value of many different angles, then the construction of a table containing sin values at code specialization time would be valuable to be used at runtime. On the other hand, there is no need to construct such table when calculating the sin values of one or two angles. In other words, the value of multi-staging emerges when applied to a large amount of data or repetitive scenarios.

In multi-staging, the programmer has the facility to decide and mark the program portions which will be reused, and therefore, need to be staged. The evaluation of these program portions will be postponed to later points of program lifecycle. In multi-staging, the marks used by the programmer are called “staging constructs”. Multi-staging languages are often implemented as extensions of existing languages like C# and Java. For this reason, the multi-staging languages syntax is based on the quotation representation for implementing these constructs.

Quotation representation is a mechanism that aims to make the syntax of the new multi-staging language much like the original language syntax, which helps make the original language programmers more comfortable with using and understanding the new language syntax. This is achieved by using simple quotation annotations around the original syntax [1,22].

Multi-staging introduces three constructs: brackets, escape and run. Brackets represent the quotation annotations [10,12]. They are written as <e> in both Mint and Metaphor, where e is the expression quoted by the brackets. The result of the bracket-quoted expression is of type Code<T> where T is the type of expression e. Code is a special class that stores any staged code segment. Brackets are used by the programmer to postpone the evaluation of quoted code to a later point of the program lifecycle. A small example of using these constructs is shown in Table 1. The example is referenced in [40,41] and is repeated here for the sake of reader’s convenience.

As shown in Table 1A, the unstaged expression int x = 2 + 3 will be translated at compilation time to the corresponding abstract syntax tree (AST) nodes. At runtime, this statement will be evaluated immediately, and the value 5 will be assigned to x.

On the other hand, the staged version of the same expression is as shown in Table 1 (staged version). The code is as follows:

```
Code<Integer> x = <| 2+3 |>; (Mint)
Code<int>  x = <| 2+3 |>; (Metaphor)
```

This expression will be translated to the corresponding AST nodes, including the brackets node. At runtime, the evaluation of this expression will not be done immediately, but it will be postponed to a later point as shown in Table 1B (section c). At that later point, the result of the expression evaluation could be used multiple times.

Escape represents the reverse operation of brackets quotation annotation. It is written as ‘e in Mint and ~e in Metaphor. The programmer can use escape to un-bracket a Code object. This partial evaluation will allow the programmer to splice more than one Code object (generated as a result of several previous brackets) inside another Code object to construct a bigger Code object. In Table 1B, the AST of the unstaged expression int y = 1 + x will be constructed at compilation time, while at runtime, this expression will be evaluated, and the value 6 will be assigned to y. In the staged version, the situation is different:

```
Code<Integer> y = <| 1 + 'x |>; (Mint)
Code<int>  y = <| 1 + ~x |>; (Metaphor)
```

At compilation time, the corresponding AST tree will be constructed as shown in Table 1B. While at runtime, the escaped expression x, which was previously bracket-quoted, will be evaluated. The resulting expression <| 1 + (2+3) |> will then be assigned to y. Here the escape annotation was used to splice the brackets expression x inside another brackets expression. After the evaluation of the escaped expression, the overall evaluation of the brackets expression is postponed.

The last construct is the run construct. Run is used to perform the computation of the staged code at runtime. It is implemented in both Mint and Metaphor as a member method of Code class. The Run method is written as run () in Mint, and Run () in Metaphor. In the same example, after constructing y object, the run method is invoked to compute the value of y at runtime. After computation, the result value 6 will be assigned to the integer variable z.

3. Applying multi-staging programming

Here, we describe how MSP can be applied through the staging of the classic Power () method. The Power method takes two numerical inputs x and n. Then, the result of multiplying x by itself n times is calculated. Table 2 shows the complete unstaged implementation is shown both in C# and Java.
In Table 2, the example proceeds by calculating $x$ to the power 30. Most of the execution time is spent on the recursive call in line 7. The recursive call accesses the stack in repetitive pop and push operations for each of the 30 calls, which causes a lot of runtime overhead. This runtime overhead can be eliminated using MSP.

The complete staged SPower method implementation is shown in Table 3. The first step performed to stage the power example is to determine the dynamic and static inputs. For this example, the power of 30 will always be calculated for different $x$ values. Therefore, the dynamic input is $x$ and the static input is $n$ ($n = 30$). According to this fact, the dynamic input $x$ should be staged. The staging of $x$ variable is achieved by defining it as Code object ($\text{Code<int>}$ in Metaphor and $\text{Code<Integer>}$ in Mint) as shown in line 9.

The next step is to stage the costly recursive statement of line 13. It can be staged using brackets constructs in order to postpone the evaluation of this statement to a future point.
of the program life cycle. Both $x$ and the recursive call of SPower method are marked using the escape construct to splice them inside the brackets clause.

In OOP, any method must be defined inside a class object. For this reason, the final step to stage the power method is to wrap SPower method call inside a staged anonymous class.Anonymous class is an OOP feature which allows defining inline classes and methods inside method's body. The implementation of the staged anonymous class is shown through the lines 16–21.

The anonymous class is created on-the-fly with a new instance. The method evaluation would be postponed until the run method is called in line 22. Note that this class contains a method which takes the static variable $n = 30$ for reusability. In line 22, when the run method is invoked, the anonymous class code would be evaluated and the result would be as shown in Table 4.

The recursive statement is replaced with a direct multiplication statement which calculates the power of 30. The evaluation of the anonymous class code is performed as shown, and then the generated code can be reused to calculate the power of 30 for different $x$ values without any recursion overhead. The staged version eliminates overhead, which therefore leads to performance improvement.

### 4. Multi-staging performance evaluation and benchmarks

To assess the usefulness of MSP on object oriented languages, several benchmarks are designed, developed and applied to Mint and Metaphor. Mint (The MSP extension to Java) and Metaphor (The MSP extension of C#) are selected for the benchmark comparison for several reasons. These reasons include the popularity of their base language (Java and C#) in current modern and commercial development and their consideration as mainstream languages. They are also the only available MSP implementations on object oriented languages that have a mathematical proof to perform full static and dynamic type checking of generated code. They are also based on objected oriented paradigm which proved its usefulness in large applications. The comparison of the performance of these languages also highlights the consequences of the implementation and runtime environment.

Benchmarking the performance of MSP implementations consists of performing a set of tests in order to measure the execution time. To ensure that both implementations are compared on the same basis, benchmarks are implemented using the same algorithm in both languages (Mint and Metaphor). The tests are very generic and the tests code is compact and can be ported easily to any future MSP language. The benchmark suite consists of seven different benchmarks in three groups. Five of these benchmarks represent simple programs from the previous literatures. Two new benchmarks are presented to cover larger programs for image processing and encryption algorithms. The benchmarks source code can be accessed from http://sdrv.ms/KFwvdO.

To compare performance results equally for both Mint and Metaphor on all benchmarks, both unstaged and staged code versions are implemented in the same way for each of the two languages. Benchmarks are executed for both Mint and Metaphor, and performance results are recorded and analyzed.

The used computing environment is as follows: a personal computer with an Intel core 2 Duo CPU (2.53 GHz), 2 MB of L2 cache and 2 GB RAM. The used operating system was Windows XP Professional Service Pack 3. The Java environment was JDK (1.6.0_07) and JVM (1.6.0_22) and the used C# environment was .Net Framework 3.5 SP1. Each benchmark is executed for several points on the computing environment. Both the staged and unstaged versions of the benchmark are run for about 2 s each. The time is chosen long enough for calibration that allows the JIT compiler to finish optimizing the program both for the unstaged and the staged code. The number of repetitions during this time is calculated. The average time taken by each run is calculated and recorded for this benchmark. The speedup is then calculated. To ensure the statistical validity of the benchmark, standard deviations are calculated and found to be less than 0.7% and the difference between minimum and maximum speedup was less than 3%.

#### 4.1. Benchmarks

Each benchmark consists of two main elements: code of the test method and the workload that execute the specialized staged test method.
method. We used the same metrics for evaluating the benchmarks. The metrics include both the execution time and speedup for both Mint and Metaphor. Although previous literature [31,35] included measurements of the execution time of staged code, the tests were not unified between Mint and Metaphor and the workload usually consists of only one point for each test.

The workload is chosen here to consist of several points (usually five) for each test. This ensures more fairness of comparison.

The proposed benchmark suite is composed of three groups with seven tests. The first group includes two versions of the power classic MSP benchmarks. Each version of them calculates the power of n for a dynamic base integer x.
Power version 1 (Looping): this version of the Power method is written by using loops.

Power version 2 (Self Recursive): this version of the Power method depends on self-recursive calls of Power method (the same as the example shown in Section 2).

The first group tests are selected to show that staging improves performance for different algorithm implementation of the same problem. The empirical results prove generality that will lead to the development of using MSP as an enhancement framework to improve performance for any application regardless of the algorithm. The second group contains other examples that are used in previous literatures [17, 35]. They include:

- Generalized fibonacci: the classic Fibonacci method which computes nth element in the standard Fibonacci sequence. Multi-staging is applied to reduce the overhead resulting from the recursive calls done to compute the nth element. n is considered static, while the first two Fibonacci elements are dynamic and need to be staged.

- Gaussian elimination: it uses the popular Gaussian elimination algorithm [18] to get the solution for n simultaneous equations with dynamic n coefficient variables. The matrix operations that are performed on the coefficients repeatedly to solve the equations are staged to reduce looping overhead. The number of equations is considered static, while the coefficients are dynamic and need to be staged. The used equations are the same for Metaphor and Mint.

- Binary search: it uses the popular binary search algorithm to get the position of an array element. Multi-staging is applied to reduce the overhead caused by the recursive operation associated with each recursive search trial. It also reduces the recursive calls overhead. The array size n is static, while the element to be searched for is dynamic and needs staging. The array elements and the selected element are chosen randomly. The same generated random variables are used for Metaphor and Mint.

Each of the previous benchmarks is selected to cover a different class of application algorithms. Recursive versions of Power and Fibonacci represent the recursion algorithms. They are also selected for compatibility and comparison with results published in literature [31, 35]. Other applications are selected to represent MSP improvement in different application domains. Gaussian Elimination is a sample of metrics algebra algorithms. Binary search covers the search algorithms area.

The third benchmarks group consists of two newly proposed benchmarks. This group is injected to cover other application domains like image and video processing (GreyFilter) and more complex algorithms like encryption algorithms (RSA encryption):

- GreyFilter: GreyFilter method takes an image of size \( n \times n \) (in pixels form) as an input, and returns the image pixels transformed to the grey scale as an output. Multi-staging is applied to reduce the overhead caused by the recursive operation of transforming each pixel to grey scale. The size of the image is considered static, while the pixel value (RGB value) is dynamic and needs to be staged. The implementation is written by self-recursive calls for GreyFilter method for each of the image’s pixels. This implementation showed significant performance enhancement.

- RSA encryption: The RSA encrypt () method takes three parameters as inputs: message m, modulus n and public key exponent e. Then the method performs the RSA encryption algorithm to encrypt n messages and return the corresponding ciphers. The key is mostly generated once, and therefore is chosen to be static, while the message is dynamic and needs to be staged. RSA uses Power method; therefore it would be a great chance to benefit from the staged version of Power. Since Power is called repeatedly, multi-staging plays an effective role when it is used to stage the calls to Power in order to reduce looping overhead.

No database or files applications are used in our benchmarks. We expect that the performance improvement of those types of applications will be low because most of the execution time is spent in input output operations. Also computation operations in database applications rely heavily on server side computation. This computation cannot be affected by the programming language because it depends on the database engine.

4.2. Results analysis

The results of the execution time of the versions of the power methods are shown in Fig. 1. Fig. 1A and B shows the
Figure 1  Execution time for two versions of power method (both unstaged and staged on Mint and Metaphor). (A) Execution time comparison of two unstaged versions of power (Java implementation), (B) execution time comparison of two unstaged versions of power (C# implementation), (C) execution time comparison of two staged versions of power (Mint implementation) and (D) execution time comparison of two staged versions of power (Metaphor implementation).
Figure 2  Benchmarks performance results. (A) Power (version 1): looping, (B) power (version 2): self recursive call, (C) fibonacci, (D) gaussian elimination, (E) binary search, (F) grey filter and (G) RSA encryption.
execution time of the unstaged code of the two versions of the power method. It is clear that the looping version is slow due to the counter initialization, incrementing and evaluating the test condition and jump operation. The self-recursive method is even slower because of the overhead resulting from recursion, stack push and pop operation and indirect access to data due to parameter passing and more memory usage.

Fig. 1C and D shows the execution time of the staged code of the two versions of the power method. It shows much less execution time and higher performance gain for both methods compared to the unstaged versions of Fig. 1A and B. The execution time changes from the order of microseconds to the order of hundreds on nanoseconds. This proves the effectiveness of Multi-stage programming empirically. It shows that the performance gain is high enough independently on the algorithm implementation and that any staged implementation is faster than the best unstaged version. It should be taken into consideration that loop unrolling performance may be worse than the original loop due to code cache management saturation.

Moreover, the differences in the maximum execution time of the staged version are about 0.5 ns in Mint and 14 ns in Metaphor respectively. This is much less than the difference in the unstaged versions which was 311 and 331 ns respectively. This ensures that Multi-staging can compensate for the optimization efforts done by the programmer to select the most efficient mathematical algorithm. It shows also that Mint is much faster than Metaphor which indicates the efficiency of Java Virtual Machine (JVM) in comparison with the C# Common Language Runtime (CLR).
The speedup results of the seven experiments of the three groups are shown in Fig. 2 for both Metaphor and Mint compilers. All results are shown in log scale. The log scale base changes for each figure to suit the recorded speedup range.

The results show improvement in the speedup of all cases. The speedup is defined as the ratio between the run times of unstaged and staged versions of the program. It is expected that staged version will be faster than the unstaged version. This means that speedup should be usually greater than one. It is worth here to mention that all execution times shown in this paper include both the compilation and generation time. However, it is possible to neglect the computation time to generate the specialized program and the compilation time intentionally because the application runs several times while it is compiled and staged once only. Therefore, the weight of the compilation and code generation cost will be negligible to the weight of the execution time. This happens when the number of execution repetition is more than the breakeven point mentioned in[35]. The case study of the performance enhancement framework can be considered as an empirical validation of this assumption.

The speedup improvement depends on the nature of benchmark. The highest improvement is recorded for the Power versions that use recursion, GreyFilter and RSA encryption benchmarks. This is a result of staging the recursive calls in these benchmarks. On the other hand, the improvement is lower in the case of Gaussian Elimination because of the loops. In the case of binary search, improvement is reasonable because of the multiple binary search trials achieved to find the random element inside the given array.

The performance study shows that multi-staging implementation applied to the benchmarks boosts the performance enhancement. This is clearly observed in Power, RSA encryption and GreyFilter benchmarks which suggest staging for basic functions that are found in built in libraries. It also suggests using staging in image and video applications. The programmer should take care that very large images may lead to code explosion problem.

The results proved that there is a direct proportional relation between number of computing operations and speedup. In other words, when the problem needs more computation, the performance speedup improves significantly. This calls to more use of multi-staging when loops are long or recursion is deep.

It is worth mentioning that Power and Fibonacci were used in testing Metaphor [35] and Mint [31,38], and that the performance results came consistent with the results in this paper. On the other hand, Binary Search results in Metaphor came inconsistent with the results obtained by Metaphor implementers. We contacted them for discussing this issue and they justified it by the differences in C# CLR versions and used hardware but this reasoning needs more validation.

Metaphor speedup results are frustrating when compared to Mint results. Consequently, the reason behind the speedup differences between Metaphor and Mint results needs to be studied. In order to do this, we performed an additional test to see whether this difference is resulting from the difference between Metaphor and Mint implementations, or it is inherited from the nature of C# and Java languages. This test code aims to perform a comparison between the execution time of two methods to calculate the power of 30. The first method RecPower uses recursion to calculate the power of 30 as in the Power version 1 benchmark (see benchmarks section). The second method MultPower uses the multiplication operation to calculate the power of 30. This method is a simulation of the staged code construction of the recursion Power method at runtime (see the explanation of staged code generation in Section 2). The code of both methods is shown in Table 5.

Both Power implementations were used to calculate the power of 30 for the base 2 in both C# and Java, and the speed-up ratio between the execution time of MultPower and RecPower is calculated. The speedup ratio was 4.527 in C# and 172.651 in Java. This result shows that the speedup difference in the benchmarks is influenced by the C# language implementation. It emphasizes that the issue is related to C# IL (intermediate language) assembler which needs more optimization.

Performance results come with a conclusion that Mint and Metaphor compilers prove performance improvement when staging is applied. The speedup is very high compared to the runtime overhead of staging implementation. This overhead comes from the reconstruction of the code and calling the compiler at runtime to compile the code (javac in Mint and intermediate language ‘IL’ generator in Metaphor). The reflection library is then used to execute the code. Future enhancements

<table>
<thead>
<tr>
<th>C# implementation of Power30</th>
<th>Java implementation of Power30</th>
</tr>
</thead>
</table>
| static double RecPower(double x, int n)
  if(n == 1)
    return x;
  else
    return RecPower (x,n-1)*x ;
|
| static double MultPower(double x)
  return x*x*x ...*x ; //30 time |

| static double RecPower(double x, int n)
  if(n == 1)
    return x;
  else
    return RecPower (x,n-1)*x ;
|
| static double MultPower(double x)
  return x*x*x ...*x ; //30 time |
Figure 3  Performance enhancement framework design.
in the compilers can be achieved to overcome this overhead and show more improvement in the speedup.

5. Performance enhancement framework

Till now, most MSP programming is used in small applications to test MSP implementations in a research environment. In real-world applications, a large application is developed with multiple developers. It is not practically possible to let the programmer decide which code segments are suitable for multi-staging. The staging decisions cannot be entirely left to the programmer sense. For this purpose, a performance enhancement framework is proposed. This framework introduces guidelines for tailoring the program using multi-staging programming to enhance the results of runtime performance metrics and the time taken to perform each of the program methods. The framework represents a defined methodology which guides the programmer throughout the process of improving his application using multi-staging.

The framework guides the programmer to determine the performance improvement opportunities through staging. These opportunities are identified by code hot spots (code segments which consume most of the application execution time). These hotspots are characterized with long mathematical calculations that are called repetitively. The high level design of the framework is shown in Fig. 3.

Slow code that runs once or twice may not be good candidate for staging. For example, in RSA encryption, the key generation part does not need to be staged because the key is mostly generated only once at the communication startup, and then used to encrypt the exchanged messages between communicating parties, and therefore nothing would be gained from staging this part. After determining these hot spots, the programmer’s next step is to stage these hot spots.

Although the performance enhancement framework appears similar to any enhancement framework, many building blocks are added to support the specific situations of staging. As shown in Fig. 3, the input for the framework is the original unstaged software program. The next step is to monitor the program execution by the performance analyzer tool (profiler), and generate the performance report. Two available Java and C# performance analyzers are used to measure and analyze runtime performance metrics. In Java case the Mint’s Performance measure library is used, while in C# the Microsoft Visual Studio 2010 Performance Analyzer tool is used. It is worth to highlight that whenever a programmer intends to extend this framework for another similar language: a suitable performance analyzer should be selected and used.

The performance analysis report shows each of the program methods associated with its execution time. Our methodology is to analyze this report. Then, these methods are sorted by the execution time in descending order. The slowest hot spot code is then staged if it is suitable for multi-staging. Suitability requires a certain set of conditions: (1) the function should be called many times with a fixed subset of inputs, (2) partial evaluation of those fixed inputs leads to at least modest reduction in the complexity or code path length of that function. After staging these code portions, a new staged code evolution would be generated and returned back to the framework to examine its performance.

If the speedup of the current version is less than the speedup of the previous version, then an analysis is needed to define the reason of this degradation. In order to do this, all the unstaged methods (callers) must be checked if they call any staged method (callee). If this situation is found, then the unstaged ‘caller’ must be turned to call the unstaged version of this ‘callee’. Actually, this can be the reason for the speedup decrease, because each time the unstaged ‘caller’ calls the staged ‘callee’, an evaluation of the staged ‘callee’ code is needed and this increases the runtime overhead. An example of this situation will be shown in the second Evolution of the next Case Study section.

On the other hand, if the performance of the current version shows improvement against the previous version, then the next step is to check the performance report. If the performance report does not contain any further unstaged hot spots that are called repetitively, then the current evolution is considered as the final code version. Otherwise, the code evolution is returned back to the framework to start the next improvement cycle.

To verify that the performance enhancement framework is generic and language independent, the RSA encryption case study is applied to both staged Java (Mint) and staged C# (Metaphor).

6. RSA encryption case study

In order to prove the effectiveness of the proposed framework, it is applied to a case study application. This case study is the implementation of RSA encryption algorithm in both Java (Mint) and C# (Metaphor) languages. It covers the encryption and decryption of 1000 messages. Fig. 4 shows the invocation tree of RSA program.

6.1. Unstaged version

First, the unstaged code version is executed and the performance is monitored by the analysis tools to generate the performance report. The report shows the execution time for each method in the program invocation tree. This is illustrated in Table 6.

6.2. First evolution

As seen in Table 6, after sorting the methods based on execution time in a descending order, it is observed that Decrypt() method takes the longest execution time among all methods. Accordingly, we need to start by staging Decrypt() method. When we applied multi-staging to Decrypt() method, it showed a speedup of 5.3 (Metaphor) and 1.11 (Mint), which resulted in an overall program speedup of 1.85 (Metaphor) and 1.04 (Mint). The staging results report, including speedup, is shown in Table 7.
6.3. Second evolution

According to the results report of Table 7, now the \texttt{Power()} method called by \texttt{Decrypt()} is the most time consuming method, and therefore it is the next candidate to be staged. After staging the \texttt{Power()} method, we have two \texttt{Power()} versions for \texttt{Encrypt()} to pick from; the staged and the unstaged.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of calls</th>
<th>Metaphor call time (nsec)</th>
<th>Mint call time (nsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt (exclusive the power call)</td>
<td>1000</td>
<td>700.000</td>
<td>74.000</td>
</tr>
<tr>
<td>Power (called by encrypt)</td>
<td>2000</td>
<td>430.000</td>
<td>15.000</td>
</tr>
<tr>
<td>Decrypt (exclusive the power call)</td>
<td>1000</td>
<td>1060.000</td>
<td>140.000</td>
</tr>
<tr>
<td>Power (called by decrypt)</td>
<td>4000</td>
<td>970.000</td>
<td>82.000</td>
</tr>
<tr>
<td>Overall program</td>
<td>–</td>
<td>3160.000</td>
<td>311.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of calls</th>
<th>Metaphor call time (nsec)</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt (exclusive the power call)</td>
<td>1000</td>
<td>800.000</td>
<td>–</td>
<td>74.000</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Power (called by encrypt)</td>
<td>2000</td>
<td>430.000</td>
<td>–</td>
<td>15.000</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Decrypt (exclusive the power call)</td>
<td>1000</td>
<td>200.000</td>
<td>5.300</td>
<td>126.000</td>
<td>1.110</td>
<td>82.000</td>
</tr>
<tr>
<td>Power (staged version called by decrypt)</td>
<td>4000</td>
<td>970.000</td>
<td>0.010</td>
<td>8,200.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall program</td>
<td>–</td>
<td>2400.000</td>
<td>1.850</td>
<td>297.000</td>
<td>1.040</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of calls</th>
<th>Metaphor call time (nsec)</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt (exclusive the power call)</td>
<td>1000</td>
<td>819,000.000</td>
<td>–</td>
<td>1,281,000,000</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Power (called by encrypt)</td>
<td>2000</td>
<td>430.000</td>
<td>–</td>
<td>15.000</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Decrypt (exclusive the power call)</td>
<td>1000</td>
<td>200.000</td>
<td>5.300</td>
<td>126.000</td>
<td>1.110</td>
<td></td>
</tr>
<tr>
<td>Power (staged version called by decrypt)</td>
<td>4000</td>
<td>970.000</td>
<td>0.010</td>
<td>8,200.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall program</td>
<td>–</td>
<td>819,631.000</td>
<td>0.003</td>
<td>1,281,141,000</td>
<td>0.0002</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of calls</th>
<th>Metaphor call time (nsec)</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
<th>Mint call time (nsec)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypt</td>
<td>1000</td>
<td>200.000</td>
<td>3.500</td>
<td>26.000</td>
<td>2.890</td>
<td></td>
</tr>
<tr>
<td>Power (called by encrypt)</td>
<td>2000</td>
<td>2.000</td>
<td>215.000</td>
<td>0.100</td>
<td>150.000</td>
<td></td>
</tr>
<tr>
<td>Decrypt</td>
<td>1000</td>
<td>10.000</td>
<td>106.000</td>
<td>43.000</td>
<td>3.255</td>
<td></td>
</tr>
<tr>
<td>Power (called by decrypt)</td>
<td>4000</td>
<td>1.000</td>
<td>970.000</td>
<td>0.010</td>
<td>8,200.000</td>
<td></td>
</tr>
<tr>
<td>Overall Program</td>
<td>–</td>
<td>213.000</td>
<td>14.900</td>
<td>69.000</td>
<td>4.507</td>
<td></td>
</tr>
</tbody>
</table>
This can be interpreted into two evolutions 2a and 2b respectively. The results in Table 8 show an undesirable increase in the execution time of Encrypt() in evolution 2a. This is due to re-evaluating the staged value returned by Power() each time it is called by Encrypt(). This result can be generally concluded that any staged caller method that calls unstaged called method will degrade the performance.

This leads us to evolution 2b which shows remarkable progress in the overall execution time. This resulted in a speedup of 2.208 (Metaphor) and 1.446 (Mint).

6.4. Third Evolution

As Table 8 section b shows, The Encrypt() method becomes the most time consuming method, and therefore, it is time to stage it to get the most possible speedup of the overall program. The results report after staging Encrypt() (calling the staged version of Power() method) is shown in Table 9. This final report illustrates how the staging of Encrypt() has completed the picture by adding more improvement to the execution time and performance of the whole program.

When reviewing the results of the case study, it can be noticed that each time the program is executed, the time taken by a certain method may slightly differ. This is due to environmental factors (like CPU and memory usage) that are shared and affected by other running applications. We stress here that these slight variations do not affect statistical validity as explained in the benchmarks section.

The final performance evaluation report generated from both profilers shows that speedup results of Metaphor are greater than those of Mint, which contradicts with the benchmark results (see benchmarks section). In order to assess and validate the results, we applied the same timing technique used for the benchmarks to calculate execution time and speedup of the Metaphor version of the case study. Consequently, we obtained an overall unstaged rate of 1507,642 nsec, and an overall staged rate of 793.099 nsec for the third evolution, which led to an overall speedup of 1.9. These results are consistent with those of the benchmarks. After investigating how the profiler works, it is found that the profiler measured Metaphor execution time exclusive of the framework execution time. When the profiler measures the inclusive execution time, the speedup value is consistent with other results. Regardless of this, the profiler was useful in guiding the programmer to determine the program’s hot spots through the different code versions.

Figure 5 Overall evolutions execution time.

After applying the framework to RSA encryption algorithm case study, we can conclude the following:

- The results of the case study came consistent with those of the benchmarks section in that MSP has a significant positive effect on application performance improvement.
- The framework provides the programmer with a set of guidelines which enable him to stage an application with only fair knowledge about multi-staging and the application domain. This was illustrated in the case study when we based our selection of the method to be staged on comparing execution time of methods, regardless of how they were implemented.
- Staging a method does not mean that all callers should use the new staged method version as seen in the case of Power() in evolution 2. This needs to be cautiously considered by the programmer when applying MSP to an application.

Execution times of all evolutions are summarized in Fig. 5. It shows that MSP improves the performance of the application building blocks, which in turn is dramatically reflected in the overall improvement of the whole application.

7. Conclusion

In this paper, a multi-staging benchmark suite was proposed to evaluate different multi-staging implementations. A variety of both classic and newly proposed examples were included in the benchmark suite. Results of using the new proposed benchmark proved the substantial effectiveness of using multi-staging. Results showed that multi-staging can be used for both small and large applications and is ready now to increase the performance of commercial applications. In order to apply multi-staging easily and effectively on complex applications, a performance enhancement framework was proposed. A case study was implemented to illustrate performance improvement using the proposed framework. The results showed the effectiveness of the proposed framework, and emphasized the strength of multi-staging in real applications.

The benchmark can be extended by adding more tests in the future. Future research of the framework includes a thorough analysis of which application areas are more suitable for multi-staging and how a staging should be used for different types of hotspots. A pattern based approach could be used for this purpose.
Acknowledgments

We thank Mint team, especially Walid Taha, Edwin Westbrook for their important comments. We thank our colleagues who supported us in reviewing this paper.

References

A. H. Yousef, is an associate professor in the Computers and Systems Engineering Department, Ain Shams University since 2009. He is the executive Director of the Information and Communication Technology Project (ICTP), Ministry of Higher Education, Egypt. He got his Ph.D., M.Sc. and B.Sc. from Ain Shams University in 2004, 2000, 1995 respectively. He works also as the secretary of the IEEE, Egypt section since 2012. His research interests include Data Mining, Software Engineering, Programming Languages, Artificial Intelligence and Automatic Control.

T. M. Abd El Latif, has been a Project Manager at HP, Egypt since 2010. He got his M.Sc. and B.Sc. from Ain Shams University, Faculty of Engineering, Computer and Systems Department in 2011, 2004 respectively. He has been working in the software industry for about 10 years now. His research interests include Software Engineering, Agile Software Model, Artificial Intelligence and Decision Making.

M. F. Mona, is Associate Professor in Computer and Systems Eng. Dept., Faculty of Eng., Ain Shams University since 2005. Mona has graduated at Ain Shams University Faculty of Engineering, Cairo, Egypt in 1979. She got her Master of Science, and Doctorate of Philosophy degree at Ain Shams University Faculty of Engineering, Cairo, Egypt in 1985 and 1992. She has strong background in domain of Data Base management system and its application, Software Engineering, Artificial Intelligent, tutoring Systems and E-learning, and Logic Design. She is experienced in managing projects from conception to completion in Software Process Improvement, Quality Assurance, and CMMI. Currently, she works as a senior Engineering Manager in Mentor Graphics Egypt.