J. Software Engineering & Applications, 2009, 2: 144-149
doi:10.4236/jsea.2009.23021 Published Online October 2009 (http://www.SciRP.org/journal/jsea)

Applying Heuristic Search for Distributed Software Performance Enhancement

Omid BUSHEHRIAN

Computer and IT Department, Shiraz University of Technology, Shiraz, Iran.
Email: bushehrian@sutech.ac.ir

Received April 2nd, 2009; revised April 28th, 2009; accepted May 4th, 2009.

ABSTRACT

Software reverse engineering and reengineering techniques are most often applied to reconstruct the software architecture with respect to quality constraints, or non-functional requirements such as maintainability or reusability. In this paper, the performance improvement of distributed software is modeled as a search problem that is solved by heuristic search algorithms such as genetic search methods. To achieve this, firstly, all aspects of the distributed execution of a software is specified by an analytical performance evaluation function that not only evaluates the current deployment of the software from the performance perspective but also can be applied to propose the near-optimal object deployment for that software. This analytical function is applied as the Heuristic search objective function. In this paper a novel statement reordering method is also presented which is used to generate the search objective function such that the best solution in the search space can be found.

Keywords: *Performance Engineering, Heuristic Search Methods, Software Reverse Engineering*

1. Introduction

Automatic software reverse engineering and reengineering techniques are most often applied to reconstruct the software architecture with respect to quality constraints, or non-functional requirements such as maintainability or reusability [1–5]. However, there has been no effort to assess the architecture of existing distributed software from the performance viewpoint. All the architectural level performance engineering techniques are focused on the performance assessment at the early stages of the software development life cycle. However, the implemented software still may not meet its performance provisions and needs to be modified to improve the performance.

In this paper, a novel software reengineering approach is presented that proposes some alterations to the deployment of the distributed software to improve its overall performance. The performance improvement is achieved by providing the chance for concurrent execution among method calls including the remote calls or local ones. The concurrency among the method calls is obtained when some of the blocking invocations are transformed into non-blocking or asynchronous form. However, this transformation entails execution overheads that should be considered very carefully. Therefore, the question is how to automatically find a set of invocations to be transformed to non-blocking such that the highest amount of concurrency in the execution of distributed software over a cluster is obtained.

To address this problem, in this paper the program source code is analyzed to extract a performance evaluation function considering the characteristics of the current deployment of the distributed software. These characteristics include the number of available workstations, the number of processors of each workstation and the deployment constraints. According to these constraints, some of the objects must reside on specific workstations as they need to access hardware or software resources (such as Database, printer, file,…) on that workstation.

The current researches in the Software Performance Engineering (S.P.E) are dedicated to estimate the performance of software in the early stage of development process due to needs for *QOS*. In order to achieve this goal, several works have been done to transform the software architectural models to the analyzable formal models. Some examples are deriving Queuing Network models from UML diagrams [6] or translating some of the UML diagrams to the Perti Nets [7,8]. In [9] constructing and analyzing two kinds of performance models are proposed: *software execution model* and *system execution model*. The former represents the software execution behavior and is modeled by *execution graphs* and the latter is based on the queuing network models, which represent the computer system platform, including hardware and software components. The software and system execution models are applied to assess the performance of the intended software architecture.

There are also some related researches in the area of performance optimization of existing distributed programs. In a mixed dynamic and programmer-driven approach to optimize the performance of large-scale object-oriented distributed systems [10], an object-partitioning method is dynamically invoked at runtime to collocate objects that communicate often. Here, the partitioning criterion is to gather objects that often communicate in a same partition. In a distribution strategy for program objects [11], an object graph construction algorithm and a partitioning policy for the program objects based on object types is presented. The distribution strategy is to place the most communicating objects in the same machine to minimize the network traffic. However, when partitioning a program, in addition to minimizing the communication cost, the amount of concurrency in the execution of the distributed partitions has to be maximized. To achieve this, in this paper, a new performance-driven partitioning criterion is proposed.

The main contribution of this paper is to extend the Software Performance Engineering techniques to the reengineering area in order to optimize the performance of existing distributed software. To achieve this, a new parametric performance evaluation function to assess the performance of the current object deployment of the software over the computational nodes is presented. This function not only evaluates the current software deployment but also proposes the best object deployment for the software. This function is automatically constructed while traversing the program call flow graph and considers both blocking and non-blocking types for each invocation.

The remaining parts of this paper are organized as follows: in section 2 the main steps of the proposed method are described. Section 3 presents an optimization technique called *statement reordering* which is applied to improve the amount of concurrency in the distributed program code. In section 4 the implementation of the proposed method is described. Finally the conclusions and future works are presented in section 5.

2. Software Performance Modeling

To improve the performance of distributed software, some of the blocking invocations among objects should be transformed to non-blocking ones. The non-blocking invocations are implemented either by means of remote asynchronous calls (supported in some middlewares such as JavaSymphoney [12]) or Java Threads. However, an invocation is converted to non-blocking only when this conversion results in concurrent execution between the caller and callee considering the resultant overheads. Each non-blocking invocation incurs communication and initiation overheads. The former is the amount of required time for sending the invocation parameters and receiving the return values between caller and calee. The latter is the amount of time required for starting the non-blocking invocation (such as asynchronous RMI or Java Threads).

The object deployment for a given program is defined as pair *(r,d)*. *r* denotes the set of all invocations in the program along with each invocation status (blocking or non blocking) an *d* denotes the deployment of caller and called objects, for each invocation I_i in *r,* over the available computational nodes. The performance evaluation function Θ(r,d) estimates the amount of execution time for object deployment (r,d). The optimal object deployment (r_o,d_o) is the one for which the amount of Θ is minimum. To find the optimal object deployment all possible object deployment for the software should be evaluated using function Θ. However, this is a NP-Complete problem and should be solved using heuristic search algorithms such as Genetic algorithms. The deployment constraints must be considered during the search for the optimal object distribution. According to these constraints, some of the objects must reside on specific workstations as they need to access hardware or software resources (such as Database, printer, file,…) on that workstation. We have used a Constrained Genetic Clustering algorithm to find the optimal object deployment. In this algorithm function Θ is used as the search objective function.

The main steps can be summarized as follows:

- Extracting *Call Graph* from the program source code.
- Unfolding the all graph to obtain the *Call Tree.*
- Extracting function Θ by analyzing the program source code.
- Search for the optimal object deployment (r_o,d_o). This is achieved by using a genetic clustering algorithm that uses Θ as the search objective function.

We have omitted the cycles in the extracted call graph to obtain the program call tree. This is necessary as in the call graph each node represents a class with multiple incoming invocations. However at runtime each invocation may be performed using a separate instance of a class. Each object deployment (r,d) corresponds to a *labeled partitioning* of the program call tree. Each label of a node in the call tree specifies the workstation on which the object represented by that node resides. The status of invocations is determined by the partitioning. The invocations among partitions are assumed non-blocking while the invocations inside a partition are blocking. Therefore each labeled partitioning of the call tree specifies an object deployment (r,d) of the software uniquely and vice versa. To search for the optimal object deployment (r_o,d_o) all possible labeled partitioning of the call tree are evaluated by the constrained genetic clustering algorithm.

The performance evaluation function Θ is built automatically while traversing a program call flow graph. A call flow graph represents the flow of method calls among program classes. Θ is built considering the estimated execution times of all instructions within the program code including the invocations. Since, the type of invocations affects the overall execution time of the program and is not determined until the program is partitioned; Θ is built as a general function including all the possible invocation types for each method call. There are four types of invocations: local blocking, remote blocking (implemented using RMI), local non-blocking (implemented using Java Thread) and remote non-blocking (implemented using asynchronous RMI). For each type of invocation an overhead is defined as shown in Table 1.

As described in the previous section, Θ maps each object distribution (*r,d*) to a value representing the estimated execution time of the distributed software with object distribution (*r,d*). Object distribution (*r,d*) is defined using a set of functions presented in Table 2 .

For instance, consider a method invocation I_1 that performs another invocation I_2 during its execution. The estimated execution time of I_1, denoted by T_{I1}, when I_2 type is (1) local blocking, (2) remote blocking,(3) local non-blocking and (4) remote non-blocking , is shown by relations (1) to (4) below respectively.

Table1. Different overhead types

α	RMI initiation overhead
β	Thread creation overhead
γ	Asynch RMI initiation overhead

Table 2. The maps that specify a *labeled portioning* and its equivalent object deployment (*r,d*)

Φ	Maps invocation I_i to a value 0 or 1, if I_i is inside a partition it is 1 otherwise it is 0.
μ	Maps invocation I_i to a value 0 or 1, if the caller and callee objects of I_i reside on the same workstation it is 0, otherwise it is 1.
ω	Maps invocation I_i to a workstation name on which Ii is executed
Δ	Maps invocation I_i to a value indicating the communication cost of the network link over which I_i is sending parameters or receiving return values as a RMI or asynchronous RMI
Γ	Maps each workstation w to the maximum number of threads created on it
Π	Maps each workstation w to the number of processing units installed on it

$$T_{I1} = T_0 + T_{I2}, \quad T_{I2}=T_1 \qquad (1)$$

$$T_{I1} = T_0 + \alpha + T_{I2} + \delta(I_2) \ , \ T_{I2}=T_1 \qquad (2)$$

$$T_{I1} = T_0 + \beta + S_2 \quad , S_2=\max(T_1-d_2,0) \qquad (3)$$

$$T_{I1} = T_0 + \gamma + S_2 \quad , S_2=\max(T_1+ \delta(I_2)-d_2,0) \qquad (4)$$

Assuming that the target of the invocation I_1 is method *m*, T_0 is the total execution time of all the instructions excluding I_2 within *m*. When an invocation such as I_2 is non-blocking, the caller should wait for the results of I_2 at some synchronization point during its execution. In relation (3) and (4), S_2 denotes the amount of required time at the synchronization point of I_2, to receive the results of I_2. The above relations can be combined as a single relation as follows:

$$T_{I1}= T_0+\Phi(I_2)*(T_{I2} +\mu(I_2)*(\alpha+\delta(I_2))) + (1-\Phi(I_2))*(S_2+\mu(I_2)*\gamma+(1-\mu(I_2))*\beta) \qquad (5)$$
$$S_2=\max(T_1+\mu(I_2)* \delta (I_2) -d_2,0)$$

There may be several invocations, I_i, within method *m* and each invocation itself may include other invocations. Therefore, relation (5) for estimating the execution time of I_1 can be generalized as follows:

$$T_{I1} = T_0 + \sum \Phi (I_i)*(T_{Ii} + \mu(I_i)*(\alpha+ \delta(I_i))) +\sum (1-\Phi (I_i))*(S_i + \mu(I_i)*\gamma+(1-\mu(I_i))*\beta) \qquad (6)$$
$$S_i= \max(T_{Ii}+ \mu(I_i)*\delta(I_i) - d_i,0)$$

In the above relation, S_i denotes the time elapsed to wait for the results of the invocation I_i, d_i denotes the estimated execution time of the program statements located between each call statement, I_i, and the first locations where the results of the call are required (the synchronization point of I_i).

We can generalize relation (6) to obtain function Θ. this function that returns the estimated execution time for object deployment (r,d) is built by applying relations (6) recursively starting from the *main*() method of the program. Assuming that the program call flow graph is cycle-free, Θ (*r,d*) can always be computed recursively. However, there may be cycles in the call flow graph, resulting from direct or indirect recursive calls. Assuming that I_i is an invocation to a method in the cycle (and itself is not in the cycle) and the estimated number of recursions is n_i then the estimated execution time of invocation I_i is multiplied by n_i. An invocation I_i or a synchronization point S_i may be located within a loop statement. Therefore to consider the impact of loop iterations on the time estimation, coefficients m_i and k_i have been added to relation (7):

$$\Theta_{main}(r,d) = T_0 +\sum\Phi(I_i)*n_i*m_i*(\Theta_{Ii}(r,d) +\mu(I_i)*(\alpha+\delta(I_i)))+\sum (1-\Phi(I_i))* (S_i + \mu(I_i)* \gamma+(1- \mu(I_i))* \beta) \qquad (7)$$
$$S_i= k_i*\max(H(I_i)*\Theta_{Ii}(r,d)+ \mu(I_i)*\delta(I_i) - d_i,0)$$

In the above relation, $\Theta_{I_i}(r,d)$ denotes the estimated execution time of the program starting from invocation I_i. $H(I_i)$ is the adjustment factor that adjusts the execution time of invocation I_i according to the hardware capacity of the workstation on which I_i will be executed. $H(I_i)$ is defined as follows:

$$H(I_i)= \Gamma\ (\omega(I_i))\ /\ \Pi(\omega(I_i)) \tag{8}$$

As described in Table 2, Γ maps each workstation w to the number of possible threads created on it due to remote or local non-blocking invocations executed on w considering an object distribution (*r,d*). Γ(w) for each workstation *w* in object distribution (*r,d*) is calculated as follows:

$$\Gamma(w)=\sum\ \Phi(I_i)*\mu(I_i)+(1-\ \Phi(I_i))\ , \tag{9}$$

for all I_i such that: $\omega(I_i)=w$

3. Statement Reordering

In the preceding sections, the idea of partitioning the program call tree directed by Θ function was described. The main idea was to search for a partitioning of the program classes which results in the highest amount of concurrency among invocations. Obviously the concurrency is achieved by performing some of the method invocations among actors asynchronously. Generally, converting an ordinary method call to an asynchronous one, poses two kinds of overheads on the execution: initiation and communication (described earlier). The former is denoted by *s* and the latter is denoted by *O*. Therefore, from the performance perspective, converting an ordinary call I_k to an asynchronous call is only beneficial when the sum of execution times of program instructions located between I_i and its synchronization point S_i, denoted by d_i, is greater than $s+O$. Obviously, the larger the amount of d_i, the more concurrency between the caller and the callee is obtained.

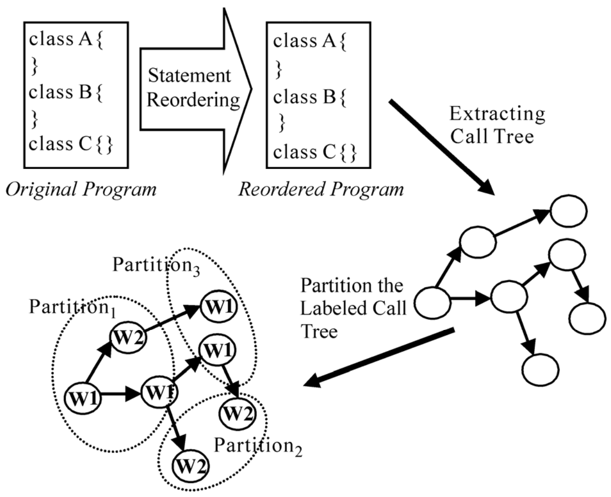

Figure 1. The statement reordering

However, a major difficulty is that programmers usually use the results of any method call I_i immediately after (d_i=0). Therefore there will be no chance for concurrency when transforming method calls to asynchronous calls. To resolve the difficulty, we have applied the ideas of *statement reordering* to enhance the potential parallelism degree of a program by increasing the amount of d_i for each invocation. The algorithm attempts to insert as many statements as possible between an invocation and its first data-dependent statement, considering the data dependencies between the statements. Obviously the reordering should be performed such that the original semantic of the program is preserved. To do this during the statement reordering the data and control dependency among statements must not be violated. Data and control dependency among program statements are represented by a acyclic graph called *Task Graph* [13]. The statement reordering is performed such that the dependencies represented by the program task graph are not violated. The overall steps including statement reordering is illustrated in Figure 1.

In the statement reordering algorithm, the program statements are classified as follows:

- *Call*: a method invocation
- *Use*, statement which is data dependent on a *Call*
- *Common*, an ordinary statement which is neither a *Call* nor a *Use*

The algorithm comprises two main steps. In the first step the program statements are moved from the *Original Code* to the *Reordered Code* gradually. In this step, presented in Figure 2, *Call* statements are moved to the reordered cod first and *Use* statements are moved as late as possible. In the second step, the reordered code resulted by applying the first step, is further optimized by reducing the time elapsed to wait for the results of *Call* statements. This is achieved by inserting as many statements as possible between each *Call* and its corresponding *Use*.

In the first step, *Call* statements of longer execution time are moved to the reordered code first because the longer the execution time of a *Call*, the more statements should be inserted between that *Call* and the corresponding *Use*. Obviously, before a statement is moved into the reordered code, all of its parent statements in the program task graph should be moved into the reordered code.

In the second step, the reordered code resulted in the first step is further optimized. To achieve this, the time required to wait for the results of *Call* statements is minimized. This is achieved by pushing down *Use* statements with positive wait time as far as their wait time reaches zero. Here Use statements with longer wait time are selected and pushed down first.

4. Implementation

We have developed a tool support that implements the steps described in section 2 to find the optimal object distribution for distributed software. Within this environment, a Java source code analyzer implemented using COMPOST [14] library, analyzes the program source to extract the call graph, call flow graph, call tree and build the Θ function. To build the Θ function for a program, first the number of clock cycles for each statement in the program is determined, according to the JOP microinstruction definition [15], and saved in an XML document. JOP is an implementation of Java Virtual Machine in

```
Algorithm Reorder (Task-Graph: DAG) OUT: Reordered-Code : List
Begin
  1. If there is no Call in Task-Graph which is not moved
  2. While there is a Common-statement in Task-Graph which is not moved
  3. Select a Common-statement whose predecessors in Task-Graph are already moved
  4. Append this statement to Reordered-Code
  5. Label this instruction as moved
  6. End While
  7. While there is a Use in Task-Graph which is not moved
  8. Select a Use whose predecessors in Task-Graph are already moved
  9. Append this instruction to Reordered-Code
  10. Label this instruction as moved
  11. End While
  12. Else
  13. Find a Call C with the longest execution time in Task-Graph which is not moved
  14. Let New-Task-Graph be a Sub-graph of Task-Graph, including Predecessors of C
  15. Reorder(New-Task-Graph)
  16. Append C to Reordered-Code
  17. Label C as moved
  18. Reorder(Task-Graph)
  19. End If
End

Algorithm Optimize (Task-Graph: DAG, Reordered-Code : List ) OUT: New-Reordered-Code: List
Begin
    1.  While there is a Use in Task-Graph which is not selected
    2.  Select a Use U with the longest wait time W in Task-Graph
    3.  Find all unselected nodes in Task-Graph which are not connected to U through a path in the task graph and
    4.  Add them to Moving-List
    5.  Remove all those nodes from Moving-List which do not share an immediate control dependent parent with U
    6.  While W>0
    7.  Select an instruction I, from Moving-List, whose predecessors in Task-Graph are not in Moving-List
    8.  Let P be the set of immediate predecessors of I
    9.  Let C be a Call whose results are used by U
    10. Let P=P ∪ {C}
    11. Move I to a position after instructions in P and before U in Reordered-Code
    12. Remove I from Moving-List
    13. Subtract the estimated time of I form W
    End While
End While
End
```

Figure 2. The statement reordering algorithms

hardware. Our tool also inputs the loop bounds in the program as they are needed during the Θ generation. The generated XML document is applied by a statement reordering [13] engine to maximize the distances between each call statement and its very first data-dependent statement. The reordered program and the XML document are input to a separate module to produce the Θ function. The resultant Θ function is applied as an objective function of a constrained genetic clustering algorithm [16] to find a near-optimal object deployment for the distributed software.

A practical evaluation of the proposed method to optimize the performance of distributed object-oriented programs is presented in this section. We have used two Java case-studies: *TSP* and *Consolidated Clustering (CC)*. The first case study evaluates the impact of applying the proposed approach on a TSP program containing 18 classes and 129 method calls. This program finds near optimal Hamiltonian Circuit in a graph, using minimum spanning trees. The second case study measures the amount of speedup achieved by optimizing the performance of a program called Consolidated Clustering [17]. Consolidated Clustering is a graph clustering application written in Java. This program comprises 16 classes and 23 method calls. In this program, a graph is clustered several times using heuristic clustering algorithms. The results of each clustering are stored in a database for further uses. This program consolidates the clustering results to obtain a clustering with a specific confidence threshold. The program is relatively slow, because it applies the heuristic algorithms for clustering.

The case studies were analyzed first to extract function Θ for them. Then a genetic clustering algorithm was applied to partition the call tree of each program to find the optimal object deployment using Θ as the objective function. The chosen test bed was a cluster with 3 single processor Pentium computers running *JavaSymphoney* [12] as the cluster middleware. The parameter passing mechanism in remote non-blocking invocations in this test bed is implemented using copy-restore technique.

Before applying the genetic clustering algorithm on the case-studies, the values for parameters α, β, γ and $\delta(I_i)$ were measured in the test bed. The amount of communication cost over the Pentium cluster, regarding our underlying communication middleware, *JavaSymphony*, was measured less than 100 ms. To estimate the communication cost in terms of JOP clock cycles, the number of clock cycles of a sample program was divided by the measured execution time of that program. According to this estimation, the communication cost $\delta(I_i)$ was nearly 10^7 clock cycles for all I_i. The measured amount of parameters α, β and γ in the test bed were almost the same and was estimated 10^4 clock cycles. We applied a constraint for the CC program as 3 of its classes in the call tree should necessarily reside on the workstation on

Table 3. Measured speedups

	20	40	100	130	400	500
TSP:Θ	0.8	0.9	1	1.2	1.35	1.4
CC: Θ	0.6	0.7	0.8	1	1.3	1.4
TSP:MCC	0.3	0.5	0.6	0.7	0.78	0.8
TSP:TM	0.6	0.8	0.2	0.8	1	0.8

which the Data Base server was running.

The measured speedups resulted from executing TSP and CC on the test bed are presented in Table 3. The results are compared to other methods: (1) applying *MCC* function and (2) and applying trivial method. MCC denotes *Minimum Communication Cost* described in section 1. In the trivial method, denoted by TM, the invocations are assigned to a workstation with the lowest load at runtime for execution.

5. Conclusions

In this paper the software reengineering research area has been extended to include performance improvement of existing distributed software. To achieve this first a performance assessment function is extracted from the program source code. Then this function is applied to find optimal object deployment of the software using a constrained genetic clustering algorithm. The result is a labeled partitioning of the program call tree. The invocations inside a partition are assumed blocking while the invocations among partitions are non-blocking. The labels in the labeled partitioning graph indicate the workstations on which objects reside. We have implemented this approach and applied that on two case studies. The result of our measurements shows this approach can be applied to improve the performance of legacy software.

This is an ongoing research in the field of Software Performance Engineering. As the future work we intend to extend the idea of architectural level performance assessment in forward engineering to validate the software models in the sense that whether they satisfy the performance provisions or not.

REFERENCES

[1] B. Bellay and Gallh, "Reverse engineering to recover and describe a systems architecture," Development and Evolution of Software Architectures for Product Families Lecture Notes in Computer Science, Vol. 1429, 1998.

[2] D. R. Harris, H. B. Reubenstein and A.S.Yeh, "Reverse engineering to the architectural level," Proc. 17th Int. Conf. Software Engineering, Seattle, Washington, US, 1995.

[3] S. Parsa and O. Bushehrian, "The design and implementation of a tool for automatic software modularization," J. Supercomput., Vol. 32, No. 1, pp. 71–94, 2005.

[4] B. S. Mitchell and M. Spiros, "Bunch: A clustering tool for the recovery and maintenance of software system structure," Proc. Int. Conf. Software Maintenance, 1999. (IEEE)

[5] L. Tahvildari, K. Kontoglannis and J. Mylopoulos, "Qualitydriven software re-engineering," J. Syst. Softw., Vol. 66, pp. 225–239, 2003.

[6] Hyunsang Youn, Suhyeon Jang and Eunseok Lee, "Deriving queuing network model for UML for software performance prediction," Fifth International Conference on Software Engineering Research, Management and Application, pp. 125–131, 2007. (IEEE)

[7] J. M. Fernandes, S. Tjell, J. B. Jorgensen and O. R. Ribeiro, "Designing tool support for translating use cases and UML 2.0 sequence diagrams into a colored Petri Net," Proc. 16th international Workshop on Scenarios and State Machines, 2007. (IEEE)

[8] R. G. Pettit and H. Gomma, "Analyzing behavior of concurrent software designs for embedded systems," Proc. 10th International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007. (IEEE)

[9] Andolfif., F. Aquilani, S. Balsamo, and P. Inverardi, "Deriving performance models of software architectures from message sequence charts," Proc. 2nd Int. Workshop on Software and Performance (WOSP2000), Canada, 2000.

[10] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic, and B. Tangney, "Dynamic clustering in an object-oriented distributed system," Proc. OLDA-II (Objects in Large Distributed Applications), Ottawa, Canada, October 1992.

[11] D. Deb, M. Fuad, and M. J. Oudshoom, "Towards autonomic distribution of existing object oriented programs," Int. Conf. Autonomic and Autonomous Systems (ICAS' 06), 2006. (IEEE)

[12] T. Fahringer and A. Jugravu, "JavaSymphony: New directives to control and synchronize locality, parallelism, and load balancing for cluster and GRID-computing," Proc. Joint ACM Java Grande–ISCOPE 2002 Conf., Seattle, Washington, November 2002.

[13] S. Parsa and O. Bushehrian, "Genetic clustering with constraints," Journal of research and practice in information technology, Vol. 39, No. 1, pp. 47–60, 2007.

[14] http://www.info.uni-karlsruhe.de/~compost, last visit: 12th September 2009.

[15] M. Schoeberl, "A time predictable Java processor," Proc. Conf. Design, Automation and Test in Europe, Germany, pp. 800–805, 2006.

[16] S. Parsa and O. Bushehrian, "Performance-driven object oriented program re-modularization," Journal of IET Software, Vol. 2, No. 4, pp. 362–378, 2008.

[17] B. S. Mitchell, "A heuristic search approach to solving the software clustering problem," Ph.D Thesis, Drexel University, March 2002.