Using Trace Analysis For Improving Performance in COTS Systems

Erik Putrycz
National Research Council of Canada
Software Engineering Group
erik.putrycz@nrc-cnrc.gc.ca

Abstract

With Commercial off-the-shelf (COTS) based systems, developers are focused on "glue" code for integrating all the components to create applications. Existing tools for analyzing performance are not sufficient anymore with large systems. This paper describes new methods and tools for improving performance in COTS-based systems by analyzing the execution trace. The results of the analysis help the developer to tune his/her application and make the best usage of the underlying COTS components. A visualization tool integrates the analysis results with Eclipse, a major Java open-source IDE.

1 Introduction

In software development, when performance execution criteria are not met, developers have to find answers to several questions, for instance: Why is a particular program slow? How is the execution time spent? How can the performance be improved?

To answer these questions, compilers are often bundled with tools, called *profilers*, to analyze the execution. These tools help developers to locate *hot spots*: methods where the most of the execution time is spent.

These tools don't meet the challenges of modern software because software is often built with components of different vendors. As a consequence, understanding where time is spent is not sufficient. If the hot spots are located in the external components, on whose source code the developers have no control of, then an in depth analysis is required to understand the role of these hot spots in the execution and finally figure out how to improve the developer's source code to make the best usage of the external components.

This paper presents methods for analyzing performance in COTS-based systems using an execution trace. In section 2, the context of this work is introduced. COTS-based systems and software profiling are defined and current tools and profiling methods are presented. Section 3 presents a solution for analyzing performance in COTS-based systems. This method has been exercised on an existing application and the results are in section 4 while section 5 presents related work.

2 Context

In this section, the concept of COTS-based systems and software profiling are introduced. The capabilities of existing tools are presented and their limitations for COTS-based systems are discussed.

2.1 COTS Software

For building large applications, developers and customers are more and more considering assembling Commercial-Off-The-Shelf software, instead of developing their own applications. A COTS product can be defined by, a product that is [10]:

Copyright © 2004 National Research Council of Canada. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1. sold, leased or licensed or the general public;
2. offered by a vendor trying to profit from it;
3. supported and evolved by the vendor (who retains the intellectual property rights);
4. available in multiple, identical copies;
5. and used without internal modification by a consumer.

For this research, COTS products are considered to be software libraries, end-user products, or frameworks, where a framework is defined as "a set of cooperating classes, some of which may be abstract, that make up a reusable design for a specific class of software"[15]; and source code access is not required. The different COTS products involved are assumed to be glued together using a programming language that supports data collection (Section 3.1).

A COTS-based system is defined as an end-user system built using one or many COTS products. In such system, the developer does not write low level code, only the business logic and glue code that tailor and integrate all the COTS products.

2.2 Software profiling

In order to improve the performance of applications, *profilers* help to locate hot spots in the code. They identify which functions are called the most frequently. *Profiling* is defined as "The process of generating a statistical analysis of a program that shows processor time and the percentage of program execution time used by each procedure in the program"[3].

The usual profiling process is divided into three steps represented in Figure 1. The first step consists in enabling data collection on a target program during the execution. This is either through instrumentation code (for C/C++ for instance, detailed in Section 2.3) or using an execution environment extension (such as the Java JVMPI, Section 2.4). The execution then generates a data file that contains the execution trace or execution statistics. This output file can be used to generate reports with external tools.

Data collection for profiling can be achieved in different ways. Data can be collected from the execution environment or the code can be instrumented.

2.3 Code instrumentation

Code instrumentation consists in inserting additional code into the program in order to collect information about program behavior during program execution.

GProf [6] is the profiler for the GNU C/C++ compiler *gcc* and an example of code instrumentation. It is one of the first profilers to introduce several major concepts.

GProf requires programs to be compiled with a special option to add instrumentation. During the execution, *GProf* generates a file `gmon.out` that contains the execution trace.

The following reports can be generated using GProf analysis tool:

- the flat profile shows the total amount of time your program spent executing each function;
- the call graph shows how much time was spent in each function and its children;
- line by line profiling: histogram samples are assigned not to functions, but to individual lines of source code;
- annotated source listing: for each source, annotations show the number of times each line was called.

GProf is well adapted for non-COTS applications and provides some analysis functions such as cycle detection and source code annotations. The annotated source listing helps developers to take into account the results by linking them to the source code. Figure 2 presents an example of an annotated source code. In this example, the function was called twice, passing once through each branch of the if statement. The body of the do loop was executed a total of 26312 times (line 13). The while statement began execution 26312 times, once for each iteration through the loop. One of those times (the last time) it exited, while it branched back to the beginning of the loop 26311 times (line 14).

Figure 1: Profiling process overview

```
1                 ulg updcrc(s, n)
2                     uch *s;
3                     unsigned n;
4             2 ->{
5                     register ulg c;
6                     static ulg crc = (ulg)0xffffffffL;
7
8             2 ->    if (s == NULL) {
9             1 ->    c = 0xffffffffL;
10            1 ->    } else {
11            1 ->    c = crc;
12            1 ->        if (n) do {
13        26312 ->            c = crc_32_tab[...];
14 26312,1,26311 ->        } while (--n);
15                    }
16            2 ->    crc = c;
17            2 ->    return c ^ 0xffffffffL;
18            2 ->}
```

Figure 2: Example of GProf line annotation

For large COTS-based systems, GProf is not suitable. Reports are based on time spent in each method and no information is given concerning the use of underlying component (libraries with C/C++). GProf is thus not able to trace issue that are located outside the developer's source code because one cannot instrument the COTS components.

With aspect-oriented programming, an aspect can monitor, in addition to method entry and exit, the methods parameters. For instance, AspectJ *pointcuts* [7] allow to execute aspects on any method execution and each aspect can access the method parameters.

2.4 Profiling with Virtual machines

Another possibility is to gather execution information from the execution environment. Most virtual machines (e.g. .NET or Java) offer an interface for external profilers and a component can be "plugged" to the virtual machine to receive execution events such as method entry and exist.

The Java Virtual Machine (JVM) specifications include a mechanism for plugging profilers called the *Java Virtual Machine Profiling Interface* (JVMPI). The following set of events can be traced using this interface:

- method events: method enter and exit;
- object allocation, move, and free;
- garbage collection start and finish;
- thread start and end;
- class load and unload;
- JVM initialization and shutdown.

Also, the JVMPI can generate heap and objects dumps.

The JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent. The virtual machine notifies the profiler agent of various events corresponding to heap allocation, method calls, etc. The profiler agent issues control requests through the JVMPI. For example, the profiler agent can turn on/off a specific event notification based on the needs of the profiler front-end.

Any Java program can be profiled without special compilation or instrumentation. The

mechanism for plugging profilers to the Java virtual machine allows vendors to implement a profiler.

The Java virtual machine ships with a JVPMI agent called *HPROF*, and several other commercial profiling packages are available for the Java Virtual Machine such as *Borland OptimizeIt*, *Quest's JProbe* or *IBM's Hyades*.

In addition to CPU and heap profiling, *Borland OptimizeIt* and *IBM Hyades* include filters which group a certain number of classes using simple string matching. These filter groups can be associated with COTS components.

The underlying methods of existing profilers are similar and rely on execution time per method for all reports, but differ in their interface.

Hyades capabilities are detailled in Section 3.1.

2.5 Conclusions

The two main mechanisms for enabling profiling are instrumentation or virtual machine monitoring. Instrumentation requires source code access and is not suited for COTS components.

With virtual machines and monitoring interfaces such as the JVMPI, it is possible to access low-level execution events without the source code access.

In COTS-based systems, with a large number of classes, collecting and interpreting performance information are difficult and most profilers don't scale because analysis are done in memory.

In addition, the profiling results are project and platform dependent and difficult to interpret.

3 Performance analysis for COTS-based systems

In this section, a solution for analyzing performance in COTS-based systems is presented. The solution is illustrated with its Java implementation, although it can be applied to any programming language or platform.

The main assumption for performance analysis in COTS-based systems is that a developer can improve performance by changing the way he is using the external components but has no control on the source code of the COTS products. This means that the primary concern with COTS-based systems is not the absolute performance in terms of CPU usage and memory, but the *relative performance* of one component to another.

To understand interactions between several components, a low-level execution trace is analyzed to report on the external components usage.

The different steps of this analysis are displayed in Figure 3. The first step consists in collecting data (Section 3.1). The low-level data gathered is used to construct an execution trace (Section 3.2). Using this execution trace, statistics are calculated on the external components usage (Section 3.3). The statistics are analyzed using clustering to offer a platform and project independent way to interpret results with severity levels (Section 3.4). And finally, the results can be displayed within the source code in Eclipse (Section 3.5).

3.1 Data collection using the Hyades project

Hyades provides an open source platform for Automated Software Quality (ASQ) tools, and a range of open source reference implementations of ASQ tooling for testing, tracing and monitoring software systems.

Hyades offers an environment for managing the execution of distributed programs. The environment consists of

- agents executed on each host and responsible for starting and stopping programs and, bridging the JVMPI agent with the Eclipse plug-ins;
- a JVMPI interface (Section 2.4) that provides JVM monitoring to Hyades;
- several Eclipse plug-ins for testing, launching programs and profiling.

The usage of the Hyades environment for this work is described in Figure 4. On a server host, the target program is run on a JVM with the Hyades JVMPI interface. The low-level execution trace is collected on the developer

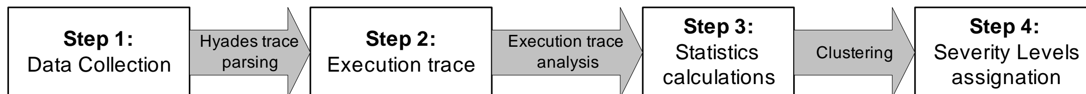

Figure 3: Analysis process

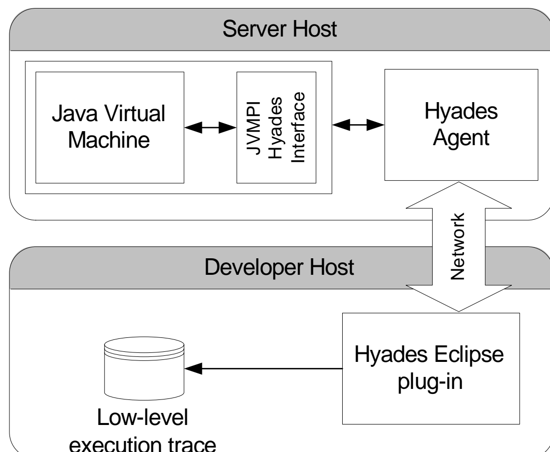

Figure 4: Hyades Profiling Infrastructure

workstation with the Eclipse plug-ins through the Hyades agent. This way, the profiler stays uninstrusive during the target execution.

In the low-level execution trace, methods' start time, end time, thread context, class information and caller identifier are recorded. Figure 5 presents an example of the low-level execution trace. This example contains a class definition (lines 1-2), method definitions (lines 3-4 and 9-10) and several method events.

The Hyades profiler can filter any set of method/calls/classes included in the trace. This feature is used to collect in the execution only the relevant components. For example, in a database centric application, only database drivers (e.g. `org.mysql.jdbc.*`) and the developer's code are selected and Java SDK methods (`java.*`) are excluded. This filtering reduces significantly the size of the trace captured and enables collection over a long execution period.

Hyades currently supports only the Java Virtual Machine, but it is possible to implement similar distributed monitoring and filtering for other virtual machines, if they offer a monitoring interface (e.g. .NET has a monitoring interface similar to JVMPI). In the case of a programming language compiled to native machine code, - if the source code is available - compilation with instrumentation (c.f. Section 2.3) enables the collection of the low-level trace.

3.2 Execution trace construction

Using the low-level data collected with Hyades, an execution trace is built. The thread context and execution context information allow the reconstruction of an execution trace containing the callers and callee of each method within the selected components. Figure 6 represents how the execution trace is constructed using the low-level data collected. em_n denotes one execution of the method m_n. During the execution of method m_1, methods m_2,m_3 and m_4 were executed in the same thread. m_1 is consequently considered the caller of m_2,m_3 and m_4. The execution trace for a thread is a list of method execution couples (*caller*,*callee*). In this case, it contains $((em_1, em_2), (em_1, em_3), (em_1, em_4))$.

In multi-threaded environments, resource accesses are often delegated to another thread. Patterns such as *Thread-Specific Storage* or *Leader/Followers* [14] are commonly used in existing systems and they rely on a multi-thread resource management. If access to resources on a thread is the cause of a hot spot in another thread, then an execution trace *per thread* wouldn't link the events in one thread to the resource accesses. To solve that problem, a *cross-thread execution trace* is built by considering all events on the same thread. Figure 7 represents how this trace is built. On *thread 1*, methods em_1 and em_2 are executed (for instance the database calls). On the *thread 2*, methods em_4 and em_5 are executed during method m_3. When all events are considered on the same thread, method execution em_3 is the *cause* of em_2, em_4 and em_5. The *cross-thread execution trace* does not guarantee the

```
<classDef threadIdRef="29" name="org.jboss.util.TimedCachePolicy" sourceName="TimedCachePolicy.java"
                                            classId="6395" objIdRef="6396" time="1075221015.182855844"/>
<methodDef name="get" signature="(Ljava/lang/Object;)Ljava/lang/Object;" startLineNumber="158"
                                                 endLineNumber="172" methodId="6388"  classIdRef="6395"/>
<methodEntry threadIdRef="29" time="1075221015.182743549" methodIdRef="6388" classIdRef="6395"
ticket="1446" stackDepth="23"/>
<classDef threadIdRef="29" name="org.jboss.security.SimplePrincipal" sourceName="SimplePrincipal.java"
                                            classId="6421" objIdRef="6422"  time="1075221015.183053970"/>
<methodDef name="hashCode" signature="()I" startLineNumber="46" endLineNumber="46" methodId="6420"
                                                                                     classIdRef="6421"/>
<methodEntry threadIdRef="29" time="1075221015.183023214" methodIdRef="6420" classIdRef="6421"
                                                                         ticket="1448" stackDepth="25"/>
<methodExit threadIdRef="29" methodIdRef="6420" classIdRef="6421" ticket="1448"
                                                    time="1075221015.183115005" overhead="0.000087949"/>
```

Figure 5: Hyades low-level trace data example

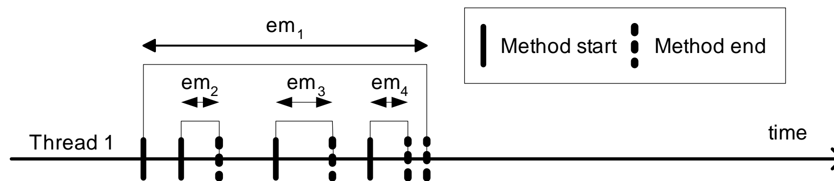

Figure 6: Execution trace construction

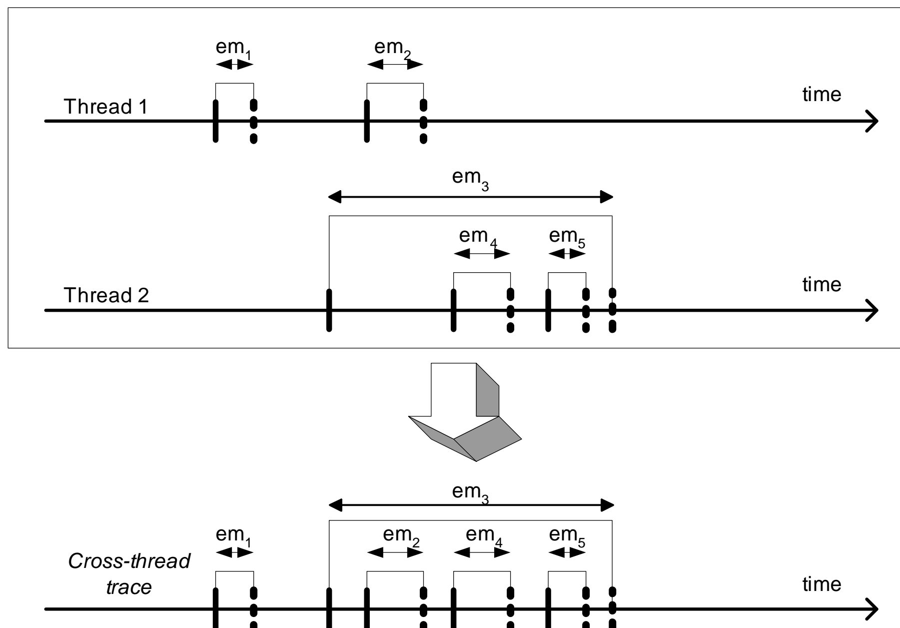

Figure 7: Cross-thread execution trace construction

accuracy of the findings but provides a better understanding of the situation.

An execution trace is constructed for a pair of component(C_n, C_m). Callers considered are from C_n and callees from C_m. For instance, if the developer's code (located in component C_1) is using components C_2 and C_3 then execution traces are calculated for (C_1, C_2) and (C_1, C_3). Depending on the pair of components selected, either the execution trace *per thread* or the cross-thread execution trace is chosen.

3.3 Statistical analysis

Statistics are calculated using the execution trace for a pair of components (C_n, C_m). Figure 8 is an example of statistics construction for (C_1, C_2). $e_1 m_1(C_1)$ denotes the execution number 1 of method m_1 from component C_1. Statistics on the number of calls and total time spent in other components are calculated on a low granularity. In this example, the number of calls, variance (in time spent) and total time spent by $e_1 m_1(C_1)$ in $m_2(C_2)$ and $m_3(C_3)$ are calculated. Using these statistics, the external hot-spots - methods in external components were most time is spent - are identified.

3.4 Severity levels assignations

The results obtained using the statistical analysis are still difficult to interpret because they are platform and project dependent. To ease their interpretation, a data mining technique called *clustering* is used to calculate a platform and project independent *severity level scale*.

Clustering is the process of grouping similar data together based on a given similarity or distance metric [12]. The main objective of the clustering is to determine similarities/groupings among cases that may reveal new information to the researcher. For a given data set, clustering can be performed on rows or columns using a choice of algorithm.

The clustering method used is based on the Euclidean measure. Elements that have a low distance between each other in a Euclidean space with n-dimensions (where n is the number of attributes) are considered similar. The clusters are built for each relevant pair of component interactions using execution time, variance and number of calls (calculated above for each method execution). A cluster is characterized with an average execution time, an average execution count and an average variance. With this information, the clusters can be classified into:

- clusters with a high average execution time contain method executions that consume the most execution time.
- clusters with a high average invocation count help to find possible optimizations.

The variance shows the repeatability of a concern. Depending on the variance, these methods are either *potential hot spots* or *recurring hot spots*.

Severity levels are directly derived from the clusters. The clusters are sorted using their average execution time and the severity level 1 is given to the cluster with the highest average execution time.

3.5 Visualization in Eclipse

To help a developer take into account the analysis results, severity levels are displayed within the developer's IDE. This way, a developer can work with a single view combining the source code edition and performance visualization.

The implementation is an Eclipse plug-in that adds to the default Java editor different background colors on each method to display the severity levels. The developer can select in the plug-in options the pair of components he/she is interested into. The plug-in is detailed with an example in Section 4.5.

4 Results

The method and tools for analyzing performance in COTS systems have been applied to an actual implementation to prove their effectiveness. The target application is a web based J2EE application called *WitanWeb*.

4.1 WitanWeb

WitanWeb system supports the on-line management of refereed submissions. Authors submit their proposals for consideration, referees

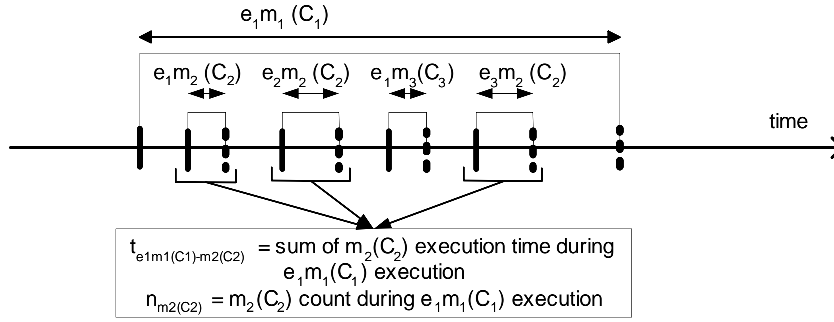

Figure 8: Statistics construction for (C_1, C_2) interactions

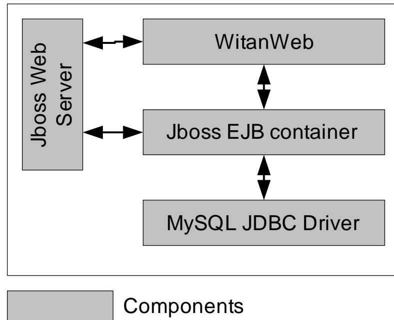

Figure 9: WitanWeb application structure

provide reports on the proposals, and committee members make decisions as to the action to take on each proposal. All this is done in a collaborative fashion using a standard Web browser to access the system.

WitanWeb was developed to be compliant with the J2EE 1.3 standard and can be run on any platform that supports this standard. Persistent data and business processes are using Enterprise Java Beans (EJB) 2.0 [4]. The presentation layer is built entirely around Java Server Pages 2.2. The structure of the application is represented in Figure 9.

The application can be configured to run on any J2EE compliant platform and can use any JDBC compliant database for persistent storage. A high degree of configurability to different refereeing processes can be achieved by modifying the JSP pages.

4.2 Execution scenario

In this scenario, WitanWeb is using JBoss, an open source J2EE server. The database is MySQL. Both were running on the same host (on a Pentium 4 1.8 GHz) with Linux Suse 9 and Java 1.4.2.

The trace analysis methods have been applied on a performance issue with WitanWeb on the generation of the refereing progress summary page. This page required 9 minutes to be generated.

The different components involved were:

- C_{nrc}: the WitanWeb code (the developer's code in this scenario);
- C_{mysql}: the MySQL JDBC driver used for database accesses;
- C_{jboss}: JBoss, the J2EE application server.

For the data collection step described in Section 3.1, JBoss startup scripts have been modified to enable the Hyades JVMPI profiler.

To reduce the low-level trace size, filters have been set to capture execution events for the following components:

- `com.mysql.*` for C_{mysql};
- `org.jboss.*` for C_{jboss};
- `nrc.witan.*` for C_{nrc}.

Low-level JDK methods were excluded with filters on `com.sun.*` and `java.*`.

As explained in 3.1, The Hyades Eclipse plug-in was run on another host for capturing and storing remotely the low-level execution trace. The low-level trace was captured

during the whole execution of the web page (9 minutes).

4.3 Performance analysis tools

The full trace consists of 870783 method execution events, 717 unique methods and 201 classes.

To analyze the low-level Hyades trace (about 2 gigabytes), all data was transferred to a database. The database is Microsoft MSDE (chosen for its ease of use and query capabilities) running on a Pentium 4 2.8 GHz with Windows XP. Java and the Hibernate framework [2] helped to implement the analysis process described in Section 3 without requiring low-level database code. All analysis was done on the same host which JBoss was running on.

BioMiner software [5], a product developer by the Integrated Reasoning group of NRC, was used for experimenting with several clustering methods and calculating the results. One of the key advantages of the software is that all available forms of data processing and analysis functionalities are integrated into one environment. The data processing and data analysis modules consist of a collection of algorithms and tools to support data mining research activities in an interactive and iterative manner. Since then, BioMiner has been applied to knowledge discovery processes in several bio-medical research.

4.4 Analysis results

The analysis focused on the following component interactions:

- (C_{nrc}, C_{nrc}): all interactions within the WitanWeb code;
- (C_{nrc}, C_{jboss}): interactions between WitanWeb and JBoss;
- (C_{nrc}, C_{mysql}): interactions between WitanWeb and MySQL driver.

The Hyades low-level trace parsing and transfer to the MSDE database required 23 minutes. The execution trace construction (described in Section 3.2) took 4 hours, mostly spent in database accesses. The statistics calculation and clustering duration (Sections 3.3 and 3.4) was 32 minutes.

Tables 1, 2 and 3 represent the clusters characteristics for each component pair.

For a component pair, 10 clusters have been constructed to create 10 severity levels. On each table, *Methods* is the number of methods in a cluster, *Total time* is the average total time of a cluster , *Variance* is the average variance and *Inv Count* is the average invocation count. Globally, the variance is very low, which means that most methods had a constant behavior during the execution. The clustering has been effective and helped to group the methods in a consistent way and highlight the hot spots.

(C_{nrc}, C_{nrc}) interactions statistics showed the main issue on the refereing progress summary generation. An important amount of time was spent in *hashcode* and *equals* methods on an object called *PaperKey*. By adding a caching mechanism to these methods, it was possible to reduce the generation from 9 to 4 minutes.

The component pair (C_{nrc}, C_{mysql}) analysis used the cross-thread execution trace. This execution trace contained few interactions because JBoss database cache pre-loading reduced the database activity during the execution. The severity levels and statistics showed a design issue with the *EJB Entity Beans* and collections (when an item was updated, complete collections were written to the database every time). This required the refactoring of several classes.

4.5 Eclipse visualization plug-in

Figure 10 is a screenshot of the Eclipse plugin that displays the analysis results within the source code view. In this example, the component pair considered is (C_{nrc}, C_{nrc}). Levels of red are used in the background of the methods to display their severity level. The hover (text box that is activated when the mouse stays over an item during a certain amount of time) displays more detailed information: the severity levels (for each invocation in the execution tree), the caller, the invocation count and average invocation time. This example shows the severity levels for the issue explained

Cluster Label	Methods	Variance	Invocation Count	Total time
nrc-nrc-C1	266	5.65602E-34	29.73308271	10.34588112
nrc-nrc-C2	740	0	2	0.111308956
nrc-nrc-C3	1052	0	1	0.056682472
nrc-nrc-C4	4	0	1	0.364279985
nrc-nrc-C5	261	1.56172E-34	12	0.680312025
nrc-nrc-C6	1	3.36E-33	12	4.371248244
nrc-nrc-C7	8	0	1	2.805575847
nrc-nrc-C8	4	0	2.5	0.598315477
nrc-nrc-C9	21	0	2	0.190541608
nrc-nrc-C10	20	0	1	0.012231708

Table 1: (C_{nrc}, C_{nrc}) method clusters

Cluster Label	Methods	Variance	Invocation Count	Total time
nrc-jboss-C1	1	0	1	2.805575847
nrc-jboss-C2	1	1.27E-35	19	0.230325467
nrc-jboss-C3	1	0	1	0.364307881
nrc-jboss-C4	231	0	1	0.054860757
nrc-jboss-C5	1	0	1	0.155697346
nrc-jboss-C6	1	0	4	0.048489572
nrc-jboss-C7	23	0	1	0.062911054
nrc-jboss-C8	7	0	1	0.083628961
nrc-jboss-C9	8	0	1	0.025039196

Table 2: (C_{nrc}, C_{jboss}) method clusters

Cluster Label	Methods	Variance	Invocation Count	Total time
nrc-mysql-C1	5	0	1	0.062455893
nrc-mysql-C2	4	0	1	2.504304409
nrc-mysql-C3	1	7.11E-28	608	1522.617081
nrc-mysql-C4	4	0	1	0.025039196
nrc-mysql-C5	1	0	14	39.27806186
nrc-mysql-C6	1	7.11E-28	609	1525.121385
nrc-mysql-C7	3	0	1	2.805575847
nrc-mysql-C8	1	0	13	36.47248601

Table 3: (C_{nrc}, C_{mysql}) method clusters

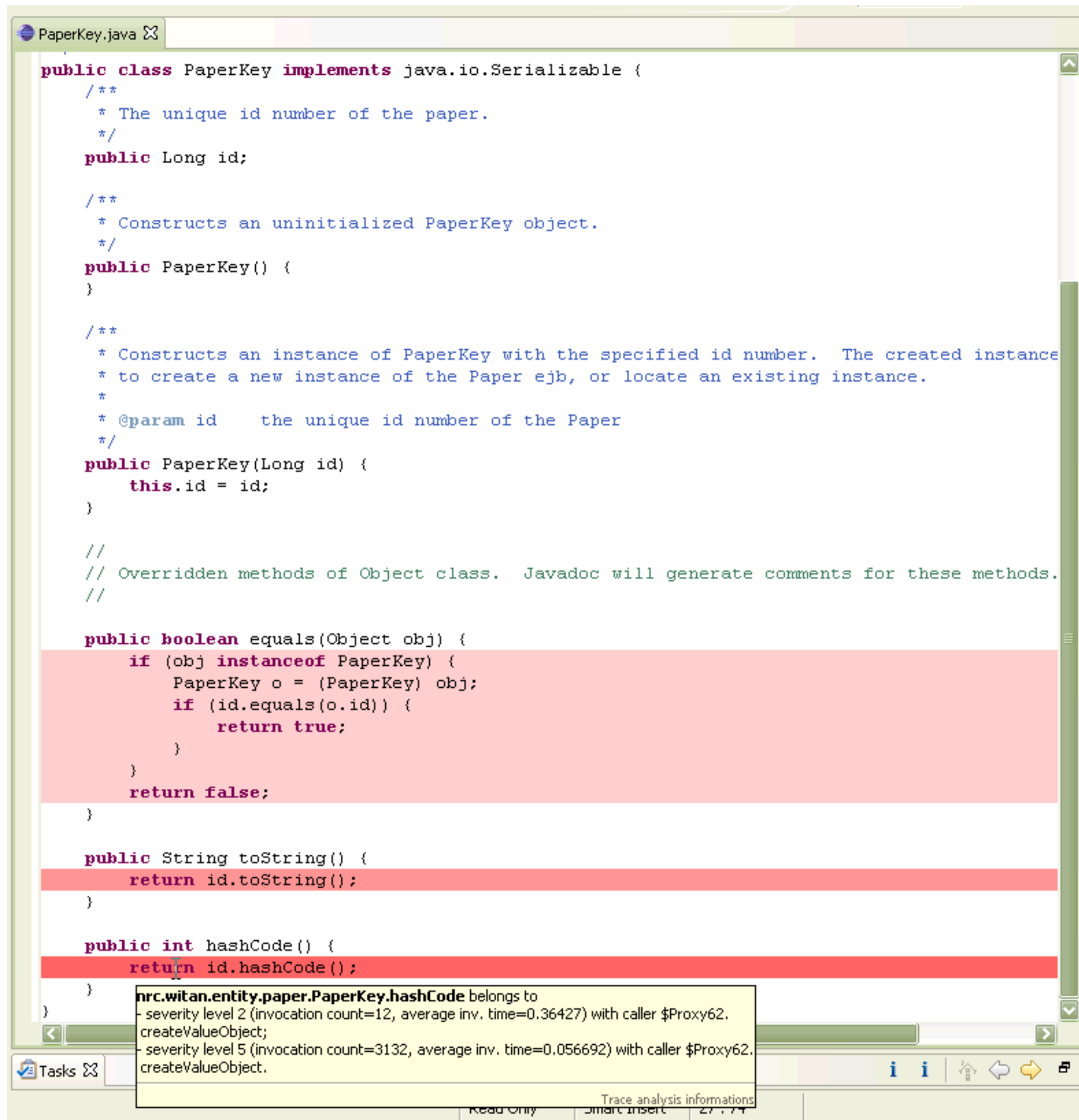

Figure 10: Eclipse plug-in screenshot

above with *hashcode* and *equals* methods in *PaperKey*.

5 Related work

In this section, similar work related to profiling and autonomic computing are presented.

5.1 Profiling

Profiling has been mostly exercised at the compiler level: algorithms and prototypes are used to position code based on an execution profile. With positioning procedures and basic blocks optimally in the compiled program, the page misses can be reduced[11]. S. Rubin, R. Bodk and T. Chilimbi [13] present a profile-analysis component for optimizing the data layout (fields within objects, objects within objects and objects within the heap) based on a objects trace from a program execution. These techniques are complementary to this work as they target the way the code is compiled.

C. Krintz presents an approach in [8] for coupling online and offline performance information to improve execution of Java programs. The execution of Java programs is improved using optimizations computed with the profiling data and also dynamic monitoring of the application execution. Optimizations are done with on-the-fly compilation of several methods and inlining.

Other metrics than CPU are being used by Scott McFarling [9] to improve code layout. The profiles are obtained from large desktop applications in real-world situations. Memory and disk usage profiles are used to reduce the kernel paging and memory usage by optimizing the compilation.

5.2 Autonomic computing

Autonomic computing aims to make systems self-managing.

Trace analysis is used to locate issues. IBM's autonomic component [16] correlates logs from several applications to diagnose WebSphere issues.

Trace analysis is often used for tuning bases. Microsoft Research developed several tools for self-tuning databases. The index tuning wizard of SQL Server 2000 [1] analyzes the execution trace of SQL queries to determine the best indexes for tables. It recommends the right mix of indexes and indexed views for a given workload of queries and updates.

6 Conclusions

As the size and the prevalence of COTS-based applications is growing, existing methods to analyze performance don't scale and are not appropriate anymore. They rely on time/method statistics which are not sufficient for COTS-based systems: a developer has no control on the external components source code and he/she can solve hot spots by making the best usage of the external components.

This article presents a novel approach for analyzing performance in COTS-based systems. This approach relies on a low-level trace analysis to understand the interactions between the components involved. An execution trace is constructed using data collection (with the IBM Hyades tools) and statistics are calculated on the usage of the external components by the developer's code. Clustering methods synthesize statistics obtained from the execution trace by attributing a project and platform independent scale, called severity levels, that shows the usage of one component by another. Finally, a plug-in for Eclipse offers a view of the performance integrated within the source code editor. The method has been used to solve performance issues with WitanWeb, a web application based on J2EE.

In the future, the analysis will be extended toward an advisor, a list of suggestions to improve the performance. Also, a knowledge database framework that allows application specific trace analysis methods (e.g analyzing a database driver's usage to suggest better optimization parameters) is in the process of being constructed.

Acknowledgements

The author would like to thank, the Software Engineering Group of IIT for all their help and constructive comments on this article; the

Integrated Reasoning Group of IIT, particularly Dr. A Famili for all help on data mining and J. Ouyang for expertise and advice with BioMiner.

About the Author

Erik Putrycz is a research associate at the Software Engineering group of the National Research Council of Canada, where he is currently working since May 2003. He received his Ph.D. in December 2002 from Institut National des Telecommunications (Evry, France). His main research interest is COTS software. He worked before on middleware, load balancing and resource management for large scale networks. His Internet address is erik.putrycz@nrc-cnrc.gc.ca.

References

[1] Kollar L. Agrawal S., Chaudhuri S. and Narasayya V. Index tuning wizard for microsoft sql server 2000. White Paper.

[2] Hibernate community. HIBERNATE - Relational Persistence for Idiomatic Java. http://www.hibernate.org, 2004.

[3] IBM Corporation. *C and C++ Compilers manual*, 1998.

[4] L. DeMichiel, L. Yalinalp, and S. Krishnan. *Java 2 Platform Enterprise Edition Specifications, v2.0.* Sun Microsystems, 1999.

[5] A.F. Famili and J. Ouyang. Data mining: understanding data and disease modeling. *Applied Informatics*, pages 32–37, 2003.

[6] J. Fenlason and R. Stallman. Gnu gprof, the gnu profiler. http://www.gnu.org/, November 1998.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[8] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the international symposium on Code generation and optimization*, pages 69–78. IEEE Computer Society, 2003.

[9] Scott McFarling. Reality-based optimization. In *Proceedings of the international symposium on Code generation and optimization*, pages 59–68. IEEE Computer Society, 2003.

[10] B.C. Meyers and P. Oberndorf. *Managing Software Acquisition.* SEI Series in Software Engineering. Addison Wesley, 2001.

[11] K. Pettis and R.C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990.

[12] R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann, 1993.

[13] S. Rubin, R. Bodk, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, Portland, Oregon, USA, 2002. ACM Press.

[14] D. Schmidt, M. Stal, H. Rohnert, and F. Bushmann. *Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns.* John Wiley and Sons Ltd, 2000.

[15] C. Szyperski. *Component Software Beyond Object Oriented Programming.* Addison Wesley / ACM Press, New York, second edition edition, 2002.

[16] B. Topol and B. Ogle. Automating problem determination: A first step toward self-healing computing systems. IBM White Papers.