In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

Semi-Automatic Instrumentation of Critical Distributed Applications: The eMIVA Toolkit

Markus Schmid[1], Jan Schaefer[1], Reinhold Kroeger[1], Ralf Heidger[2], and Marcus Thoss[3]

[1] University of Applied Sciences Wiesbaden
Distributed Systems Lab
Wiesbaden, Germany
{lastname}@informatik.fh-wiesbaden.de,
http://wwwvs.informatik.fh-wiesbaden.de
[2] Deutsche Flugsicherung GmbH (DFS)
Langen, Germany
ralf.heidger@dfs.de
[3] tang-IT Consulting GmbH
Wiesbaden, Germany
marcus.thoss@tang-it.com

Abstract. The approach presented in this paper aims to support developers by delivering methods and IDE-based tools for semi-automatic source code instrumentation that improve efficiency and consistency of the instrumentation process, and thus reduces the overall effort to be put into instrumentation. Our solution focuses on code generation for logging and performance monitoring purposes. The resulting toolkit provides methods for instrumentation support on different abstraction levels. In order to validate the approach, the DFS Deutsche Flugsicherung (German Air Navigation Services) realised the performance instrumentation of their critical, distributed radar tracking system PHOENIX with the help of the eMIVA[4] tools.

1 Introduction

In recent years, enterprise applications faced an ever growing complexity of business processes as well as an increase in the number of interacting hardware and software components. The ability to efficiently manage their IT infrastructure up to the application level is therefore critical to a company's success and results in rising importance of Service Level Management (SLM) technologies [10, 15, 17]. As a prerequisite for application management, monitoring and instrumentation techniques face growing interest. Depending on the criticality of an application, monitoring can either be based on statistical samples and values like averages and standard deviation, or can require monitoring of each request handled by the system, e.g. for validation or verification purposes. While most enterprise

[4] This project is supported by the German Federal Ministry of Education and Research under contract no. 1720X04

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

applications belong to the first category, air traffic control scenarios are an example for the second category. Here, even a statistically small number of slow requests may result in dangerous situations or fatal accidents.

In terms of performance monitoring, we distinguish black box and white box approaches. A black box approach neither allows any insight into the internal structure of the monitored component, nor observation of performance properties of internal subcomponents. Black box monitoring is non-intrusive (i.e., it does not require modification of the monitored system or source code), but often lacks the ability to deliver fine-grained measurements, as solely logs and system output can be used for an analysis. The ability to drill down into system internals is usually only given by using a white box approach. Here, either the application source code or the binary itself is modified by insertion of monitoring sensors. Besides being intrusive, white box monitoring is regularly more complex and harder to master, because a deep knowledge of the target system is required. In distributed settings, it is often necessary to interrelate monitored events between different components (cross-component measurements), e.g. for following a request, which, too, is not always possible with black box monitoring.

Talking about manual performance instrumentation, [9] states that the major challenges for a successful application instrumentation are (1) to correctly identify the relevant instrumentation points, (2) to perform the actual instrumentation in a consistent way, and (3) to draw a correct conclusion from the results that are obtained from the instrumentation. While (1) and (3) require a certain amount of domain knowledge, (2) is a possible source of errors that can be minimised by enhancing the developer's IDE with specialised instrumentation support features, as a manual instrumentation of large-scale software projects means a huge effort and imposes the danger of introducing errors. For (2), we therefore see great potential to increase instrumentation efficiency and correctness by instrumentation automation, which could help to establish performance-aware application design as an integral part of the application development process.

2 Related Approaches

Several approaches exist that aim at avoiding cumbersome manual application instrumentation. Aguilera et al. [3] propose a black box approach for performance debugging based only on observed communication traces between network nodes or application components in order to avoid any application- or middleware-level instrumentation. The gathered data is then mined to find performance bottlenecks in the system. A lot of effort is put on reconstructing relevant call-graphs or message flows with high probability of correctness. Fine-grained monitoring at the operating system level (Windows XP system calls) is used by Yuan et al. [19] for problem diagnosis, especially for identifying root causes of known problems. Magpie [5] provides a mechanism for fine-grained process monitoring. It is capable of correlating events from different sources (e.g., OS kernel, device drivers and applications) on a single machine that are correlated and filtered

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

according to application-specific "event schemas" and can then be mapped on high-level application-specific tasks.

Other approaches apply a similar strategy for monitoring at the middleware layer. Pinpoint [6] is a system that focuses on automatic problem determination in large, dynamic Internet services. The system follows a white box approach and provides modified middleware components that are able to trace requests on their way through the system (cross-component).

Recently, configurable middleware instrumentation , i.e. without modification of applications and middleware framework, has been successfully applied for observing relevant application behaviour. Papers describe this technique being used for monitoring CORBA applications [11], or for Web Services [14].

In summary, the described indirect monitoring approaches are generally interesting from the data mining point of view. They have the advantage that they can be applied even if no source code is available, or if binary (e.g. [1]) and byte code instrumentation (e.g. [2]) – being quite expensive – are rejected. Interceptor-based instrumentation can be regarded in many cases as effective as direct application instrumentation, e.g. for SLM purposes.

For critical applications, like air traffic navigation software, statistical reasoning and reconstructing information using data mining principles is not sufficient. Here, during design and development, rigorous methods for ensuring required functional and non-functional characteristics (like performance) are necessary. It is well accepted that non-functional properties have to be taken into account from the very beginning of the application life cycle. For performance testing and debugging, full access to the application logic and status is often necessary. Even during operation, continuous monitoring of certain application characteristics may be legally required or a prerequisite for auditing procedures. Thus, in such a context direct application instrumentation is considered necessary.

Of course, the development of critical applications also has to follow economical rules. Thus, the objective of the eMIVA architecture and toolkit is to effectively support the life cycle of critical applications with respect to application instrumentation. This is achieved by enhancing the developer's IDE to support the instrumentation process and by associating an instrumentation model with the application during the application life cycle. From this instrumentation model, we semi-automatically generate the real instrumentation code in order to keep the effort for the developer low, and we can associate measured data to application design entities.

3 Application Response Measurement (ARM)

ARM (Application Response Measurement) [13] is am API standard issued by the ARM working group of The Open Group. It defines a vendor-neutral approach to measuring application performance through source code instrumentation. Originally, the ARM API was created by IBM and HP as a combined effort to consolidate their existing proprietary instrumentation and performance measurement solutions. It provides a host of Java interface definitions and C function

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

declaration, respectively, that map the ARM measurement model onto the languages. The following explanations focus on the C language binding. Central to the ARM model are the notions of *ARM transactions* and *ARM response times*. An ARM transaction is defined by two ARM API invocations, `arm_start()` and `arm_stop()`, embedded in the application source code (see figure 1). The actual implementation of the ARM methods is provided either by proprietary libraries from ARM vendors or by the Open Group ARM SDK, which is a rudimentary implementation available free of charge.

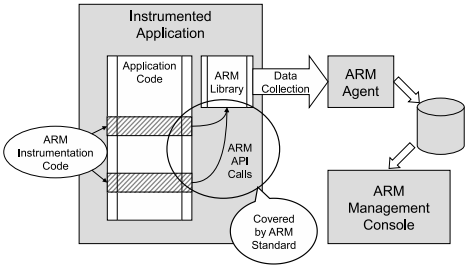

Fig. 1. ARM instrumented application

For non-trivial applications, performance models can reach a complexity that is similar to their associated application model. Therefore, the ARM model has been gradually enriched with elements supporting a more detailed and hierarchical modelling of the instrumentation points within the source code. It is possible to define reusable measurement types (via *Transaction Definitions*) and to attach contextual information to measurements (via *Identity and Context Properties* and *ARM Correlators*). For ARM correlators, an opaque byte array is optionally generated with an arm_start() call at runtime. This correlator serves as a token representing the generating ARM transaction. In order to convey to the ARM measurement environment that another ARM transaction should be modelled as having a parent-child relation to the token generation transaction, the correlator is provided as an argument to the arm_start() invocation of the child transaction. As of ARM 4.1, the transport of the correlator bytes from the parent to the child transaction's source code location – possibly across machine boundaries – is not covered by the ARM standard and has to be implemented by developers of the instrumented application.

Introducing ARM measurement technology into an application development and production cycle is often contrasted with using pre-existing ad-hoc or even elaborate custom measurement approaches using processor or operating system timers. The low overhead gained with a simplistic approach can hardly be achieved with an ARM implementation, the latter requiring at least one call into a shared library function including internal processing within the ARM library. On the other hand, a custom measurement solution is bound to evolve with the growing requirements of a complex distributed application, both regarding measurement data collection and back-end processing and analysis. The interchangeability of ARM shared library implementations and thus, of measurement processing and analysis facilities available permits the adaptation of overhead incurred through ARM.

Developers instrumenting an application using ARM are further encouraged to reduce the ARM related overhead by using features that were introduced with ARM 4.1. Assuming that the least overhead is generated by not performing

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

ARM calls at all, an *Instrumentation Control Interface* was specified. It can be queried by the application instrumentation code for granularity levels that should be applied when performing ARM API calls (e.g. at the lowest level, the application can decide to avoid ARM measurement calls completely).

4 The eMIVA-Toolkit

The eMIVA project aims to deliver methods and IDE-based tools for semi-automatic application instrumentation. The term eMIVA stands for efficient model-based instrumentation of distributed applications. The eMIVA tools assist developers by improving the efficiency and consistency of the instrumentation process, especially supporting developers without further knowledge of instrumentation technologies. The approach focuses on code-generation for logging and performance monitoring.

eMIVA provides a threefold instrumentation support on different abstraction levels: developers can instrument applications using either structured comments (resembling Javadoc [16] annotations), graphical IDE markers, or a UML model extension to mark performance-critical sections within their application. Based on this information, the eMIVA tools generate and inject instrumentation code for the target programming language into the application source code.

In terms of abstraction-level the *comment-based instrumentation* (see [18] for details) defines the lowest layer of instrumentation support that is offered by the eMIVA toolkit. Instrumentation comments are analysed by the eMIVA toolkit when a build process is started by the IDE. The toolkit then generates AspectJ [4] aspect files that contain the necessary instrumentation code. The AspectJ compiler weaves these aspect files into the Java source code in a separate build step, thereafter the actual Java byte code is generated by the standard Java compiler. A user interface allows the developer to choose the instrumentation architecture to be used throughout the build process.

eMIVA offers a *marker-based instrumentation* approach for developers who work with an eMIVA-enriched IDE. Using the marker-based instrumentation, a developer can place debugger-breakpoint-like graphical markers at any source code position the framework should create instrumentation code for. A marker references a logical position in the application source code and thus changes its position accordingly as the source file is edited.

Before the actual compilation of the source code starts, the eMIVA toolkit generates and injects instrumentation code into the application source code at the positions marked by the developer. Depending on the instrumentation architecture chosen, a marker offers different kinds of properties. The eMIVA model supports different instrumentation architectures simultaneously through marker-subclassing. This makes it possible to place logging markers into a source code file that already contains markers for ARM performance measurements.

When a build process is started, the eMIVA toolkit generates instrumentation code that later remains in the source file. Markers can be grouped according to business needs. Developers can modify, enable or disable single instrumentation

markers or marker groups as well as all markers in a source file or even the whole project. Code for disabled markers is removed from the source file.

To support (IDE-)external editing of source files as well as storage of projects in source control and versioning systems, eMIVA is capable of exporting model markers and their properties to either source code comments or a combination of comments and property files. Previously exported markers are automatically re-imported on opening a project in an eMIVA-enriched IDE and as a result the eMIVA model is updated accordingly.

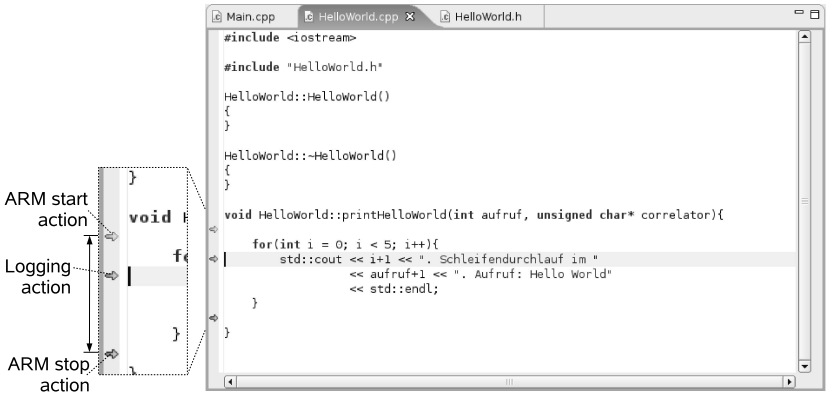

Fig. 2. Screenshot of an Eclipse editor with marker-based eMIVA Instrumentation

Figure 2 depicts eMIVA instrumentation markers displayed in an Eclipse editor. The icon colour of an instrumentation marker characterises the instrumentation point type. eMIVA currently offers measurement start markers (green), measurement stop markers (red), and logging markers (blue). The measurement markers are to be used in pairs to define a unit of work in the source code. Logging markers are independent of each other.

Instrumentation markers contain a number of general properties that may be extended by additional back-end-specific properties. For ARM measurements the `Level` and `Message` attributes are optional. If present, they are converted into ARM context properties. In addition ARM offers a number of additional properties to be set. General and ARM-specific properties are depicted in table 4.

eMIVA's *model-based instrumentation approach* follows the *Model Driven Architecture* (MDA) methodology. It targets developers of new applications, as it allows modelling performance and monitoring checks into the software during the design phase, which is a prerequisite for constant and efficient performance testing and debugging. Instrumentation points (resp. measurements) are defined in an UML diagram of the to be instrumented application. eMIVA contains a custom UML instrumentation profile with stereotypes for logging and measurements (called `Log` and `Measure`). This profile is independent of the *UML Profile*

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

Property	Description
`Active`	Set this marker active or inactive. This influences code generation for instrumentation points.
`Initialisation`	Initialisation code for the instrumentation framework used by this marker type will be generated here.
`Level`	Logging Level of this instrumentation point.
`Message`	Logging Message for this instrumentation point.
`Application name`	ARM Application Name.
`Transaction name`	Name of this Transaction Definition.
`Group`	Cluster ARM measurements into groups that can be enabled and disabled during application runtime.
`Export Correlator`	Name of the variable the correlator should be stored in for further processing.
`Import correlator`	Name of the variable a correlator should be picked from for initialising the current transaction.

Table 1. General and ARM-specific marker properties

for Schedulability, Performance, and Time (UML-SPT) [12], which lacks certain features (e.g. definition of correlated or unconstrained measurements).

The eMIVA stereotypes contain tagged values that allow adding more detail to instrumentation points (see figure 3). Logging points can contain log `Message` and `Level`, measurement points can also contain an `ID` and `CoopID`s, which are used to identify correlated measurements, and a value for identifying asynchronous measurements (`Asynch`). Once profile information has been added to a UML diagram, source code stubs can be generated using custom templates created for the *Model Driven Software Development* (MDSD) code generation framework *OpenArchitectureWare*[5] (OAW). In addition, the instrumentation point information extracted from the instrumented diagram is added to the eMIVA model, so that the instrumentation information can be processed by other eMIVA tools.

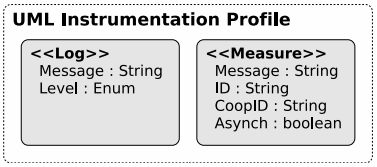

Fig. 3. eMIVA UML Instrumentation Profile

The overall architecture of the eMIVA toolkit is depicted in figure 4. The eMIVA code generator (depicted in the grey box) is separated into user interface, front-end and back-end components. The eMIVA GUI integrates with the IDE and allows to specify instrumentation and code generation-specific settings.

The eMIVA front-end component consists of a small programming-language independent core that is extended by programming-language dependent cores. The core controls the eMIVA model, that holds details on the actual instrumentation points that have been identified, so that they can be processed by eMIVA. The programming-language dependent cores control the interaction with the IDE, in case of comment-based instrumentation this means the invocation of

[5] http://www.openarchitectureware.org

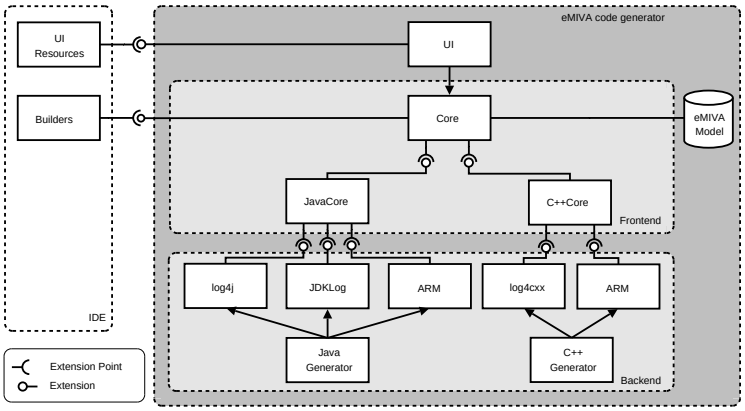

Fig. 4. The eMIVA toolkit: overall architecture

an aspect-compiler during build, in case of marker-based instrumentation the generation or removal of instrumentation code within an editor buffer. In order to enhance eMIVA by adding support of an additional programming language, a developer simply adds a language-dependent core enhancement component for the desired language to the eMIVA front-end.

The front-end is complemented by the eMIVA back-end, which hosts the instrumentation architecture plug-ins available. These plug-ins inform the framework about marker types and supply code generation hooks for the front-end. In order to enhance eMIVA with an additional instrumentation architecture, it is sufficient to provide an additional instrumentation plug-in. As instrumentation architectures do necessarily provide programming language dependent interfaces, the plug-ins extend a common language-dependent generator component that encapsulates generic functionality like e.g. adding `import` or `include` statements to a certain class. This plug-in-mechanism allowed us to extend eMIVA with a back-end for the proprietary logging API used by the DFS.

Support for more complex architectures may then be externalised to eMIVA helper classes that are automatically generated during build and linked to the application code. A good example for such a scenario is the ARM instrumentation plug-in: The ARM API requires a certain amount of initialisation code as well as a management-facility for references to running transactions. As an ARM transaction might be started in one method of an application and stopped in a different method, storing of references in local variables is not sufficient. Therefore the eMIVA ARM helper classes do provide an ARM transaction stack that handles the mapping of ARM start and stop calls.

Currently our Eclipse-based eMIVA implementation supports Java and C++ as programming languages and the ARM 4.0/4.1 API as well as log4j, JDK-Logging, log4cxx and the proprietary logging facility of the DFS.

The model-based instrumentation is currently only available for Java applications. It uses the Eclipse-based OAW code generation framework for UML model

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

processing. Model-based instrumentation is in principle possible without using the eMIVA code generation back-end, as OAW already contains the complete functionality required for model-to-code processing. However, for being able to combine instrumentation points defined in a UML model with points manually defined in the generated source code afterwards, the OAW workflow has to integrate with the eMIVA code generation back-end. The OAW workflow engine uses our instrumentation profile and an existing UML application model as input (as EMF XMI files) to create source code stubs and the necessary instrumentation information for the eMIVA instrumentation model. Thus, the UML-defined instrumentation points are added to the model, once they have been processed by the OAW workflow. Instrumentation code is mostly imported from pre-written templates, which are shared between the eMIVA and OAW code generators.

5 Sample Application: The PHOENIX Radar Tracking System

PHOENIX [7, 8] is an efficient distributed, client-server-based multi-radar tracking and air-situation display system, running on Linux. It is designed, developed and maintained solely by DFS. PHOENIX is today the primary tower radar data processing system (RDPS) used by DFS and will become the fall back system for air-situation display at all DFS locations. An RDPS is responsible for processing incoming radar data (plots), which include measurement noise, into stable trajectories named tracks. Tracks contain additional data like climb/descent rate and other important information required for an air traffic controller display.

Being used by air traffic controllers, the radar tracking system is a very critical application. Besides availability requirements, the overall data processing time of the system is of special interest. Taking the velocity of an aircraft into account, the delay between radar echo and screen display of the respective track has to stay within a certain time range in order to provide accurate information for the air traffic controller.

PHOENIX is realised as a LAN-based client-server system and consists of up to 30 concurrent processes. Central component is the PHOENIX d-mrts multi-radar track server, which is connected to ASTERIX radar data sources via UDP. One or more controller working positions (CWPs) run the display system (CWP process), which is fed with track data from d-mrts via a message distribution server process (d-msg). The CWP provides an air situation image for the air traffic controller.

The main goal of the ARM performance instrumentation of PHOENIX was to determine precisely the amount of time required by the data flow through the whole system. For this, the d-mrts (plot to track processing), d-msg (linkage of additional data) and CWP (display of tracks) processes have been instrumented. Being a critical application, determining statistical information, like average response time and standard deviation is not sufficient. For verification and validation purposes, individual measurements have to be recorded as well.

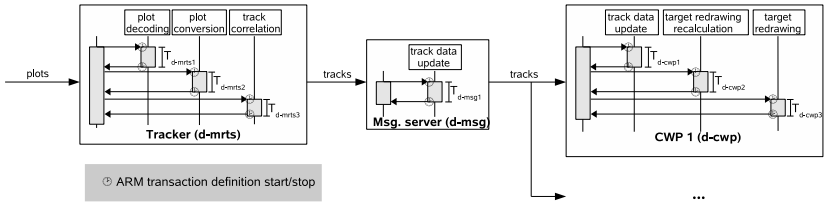

Fig. 5. PHOENIX data flow and respective ARM transaction definitions

The fundamental data flow across the instrumented processes for the measurement scenarios in the distributed test environment is shown in figure 5. Processing of incoming plots starts in the tracker with the decoding of the ASTERIX radar data format to an internal format. After decoding the plot-to-track processing starts, which includes the conversion and correlation with Kalman filtering. Then the generated or updated tracks are forwarded to the message server, which adds additional data to each track and sends them to the CWPs for display. As depicted in figure 5, several ARM response time measurement points have been identified in the corresponding functions to reflect these processing steps.

The ARM instrumentation of the PHOENIX sources has been realised with the help of the marker-based instrumentation of the eMIVA toolkit. For this purpose the existing PHOENIX sources were imported into the C/C++ development environment of the Eclipse IDE.

The performance of the instrumented PHOENIX system was in depth evaluated with test data in three different scenarios in a 120 cwp-machine test installation. One scenario used 30 minutes of recorded live radar data of the whole German air space at a busy day (∼700 tracks). The other scenarios were based on artificial data provided by a DFS testing tool, which creates large amounts of flights for full system load tests.

During the tests all mean response times were below 1ms, the maximum response time of each of the processes was below 13ms. With these values the DFS performance requirements would have been fulfilled easily but measurement results showed that the actual overall system response time was much longer (average 600ms). Therefore the whole PHOENIX processing chain was investigated, taking the inter-process communication overhead into account.

It was discovered, that significant response time overheads were caused by queueing effects in the tracker and msg-server components of the system, that could be eliminated as a result of the measurements. In addition, under high load the CWP eventually switches to a different mode (called MPA mode). In this mode all received track updates are collected but not immediately processed to display. In MPA mode, processing for display happens after a configurable period of time (5000ms in these scenarios), such that display performance problems can be circumvented. After the load for the CWP falls under a defined limit the CWP switches back to normal mode. Mode changes caused additional overhead.

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

Recapitulating the findings of the evaluation of the PHOENIX instrumentation a validation of DFS requirements could be achieved. Measurements prove that less than one second is needed from radar data reception to the display of the updated track information on the screen. In addition, the instrumentation provided a very useful and detailed performance view on the internals of the PHOENIX processing steps.

The measurements also identified relevant starting points for more detailed investigations for future performance improvements. Based on the results, a more detailed investigation of the CWP was carried out and the overload state problem caused by the working mode change could be solved. The DFS intends to build up a semi-automatic ARM test suite for regression tests.

6 Summary

We discussed the need for comprehensive performance monitoring facilities of critical applications. Detailed application monitoring often requires an expensive manual source or binary code instrumentation. Manual instrumentation is not only costly, but also error-prone, therefore automation of instrumentation offers a great potential for optimisation and cost reduction.

Several approaches exist that aim at alleviating a developer's instrumentation effort, by reusing instrumented operating system and middleware layers and therefore can only indirectly capture the business state of an application with high probability. For this reason the paper introduced the eMIVA toolkit, which aims at supporting developers during application instrumentation. eMIVA integrates with the developers IDE and provides a threefold approach (targeting different abstraction layers) for semi-automatic instrumentation.

DFS implemented an ARM-based performance instrumentation of their critical radar tracking application, using an eMIVA tools implementation based on the Eclipse C/C++ development environment CDT. During this instrumentation, DFS verified the functionality of the eMIVA tools and helped to improve their stability. The overall instrumentation process took approximately one man-month and the analysis of the collected data one additional man-month. Results show, that the eMIVA tools meet the requirements for a source code instrumentation of standard applications like validation of performance constraints and detection of performance-critical situations. However, due to the middleware complexity of PHOENIX, at some points more flexibility is required than eMIVA can provide currently, e.g. definition of entangled transactions. Therefore DFS currently works on an adaptation of the instrumentation to better meet PHOENIX-specific characteristics. Based on experience gained in the project, tang-IT discussed requirements for measuring of asynchronous processes in the ARM working group. These will be met by the upcoming ARM 4.1 specification.

Future work will enhance the model-based eMIVA instrumentation by means for the representation of performance measurement results in the UML model of an application. Primary goal of this enhancement is the establishment of a round-trip methodology for performance-aware application design. A second focus of

In: Proceedings of the 14th Annual Workshop of HP Software University Association, Hosted by the Leibnitz Computing Center and the Munich Network Management Team, July 08 - 11, 2007

our future work is the identification and classification of high-level business transactions based on performance monitoring results of underlying services. Such a mechanism could significantly alleviate business transaction-related accounting of service usage.

References

1. Dynamic instrumentation of an executable program, Oct 2002. Patent US 2002/0152455 A1.
2. Flexible and extensible Java bytecode instrumentation system, Jul 2003. Patent WO 03/062986 A1.
3. M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP'03)*, 2003.
4. The AspectJ Team. *The AspectJ Programming Guide*, 2003.
5. Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2004.
6. Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the Int. Conf. on Dependable Systems and Networks*. IEEE, 2002.
7. Ralf Heidger. A distributed system architecture for scalable sensor data processing ATC systems. In *2nd Int. Workshop on Intelligent Transportation (WIT 2005), Proceedings*, 2005.
8. Ralf Heidger. A new generation RDP fallback system in the DFS. *European Journal of Navigation*, Sept. 2006, 2006.
9. Mark W. Johnson. Monitoring and Diagnosing Applications with ARM 4.0. In *Int. CMG Conf.*, pages 473–484. Computer Measurement Group, 2004.
10. Lundy Lewis. *Service Level Management for Enterprise Networks*. Artech House Publishers, 1999.
11. Priya Narasimhan, Louise E. Moser, and P.m. Melliar-Smith. Using Interceptors to Enhance CORBA. *Computer*, 32(7):62–68, 1999.
12. Object Management Group. *UML Profile for Schedulability, Performance, and Time*, 2005. http://www.omg.org/cgi-bin/apps/doc?formal/05-01-02.pdf.
13. The Open Group. *Application Response Measurement - ARM*, 2006.
14. Jan Schaefer. An Approach for Fine-Grained Web Service Performance Monitoring. In *Distributed Applications and Interoperable Systems : 6th IFIP WG 6.1 Int. Conf., DAIS 2006, Proceedings*. IFIP, Springer, June 2006.
15. Rick Sturm, Wayne Morris, and Mary Jander. *Foundations of Service Level Management*. SAMS Publishing, April 2000.
16. Sun Microsystems. *javadoc - The Java API Documentation Generator*, 2002.
17. D. Verma. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.
18. Christoph Weimer. IDE-gestuetzte Generierung von Quellcode zur Instrumentierung von Anwendungen, April 2005. in German.
19. Ch. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *Proc. EuroSys2006*, 2006.