A Performance Cost Evaluation of Aspect Weaving

Miguel García[1] **Francisco Ortin**[1] **David Llewellyn-Jones**[2] **Madjid Merabti**[2]

[1] Computer Science Department, University of Oviedo
Calvo Sotelo s/n, 33007, Oviedo, Spain
Email: `garciarmiguel@uniovi.es`, `ortin@lsi.uniovi.es`

[2] School of Computing & Mathematical Sciences, Liverpool John Moores University
Byrom Street, Liverpool L3 3AF, United Kingdom
Email: `D.Llewellyn-Jones@ljmu.ac.uk`, `M.Merabti@ljmu.ac.uk`

Abstract

Aspect-Oriented Software Development (AOSD) facilitates the modularisation of different crosscutting concerns in software development. In AOSD, aspect weaving is the composition mechanism that combines aspects and components in an aspect-oriented application. Aspect weaving can be performed statically, at load time or at runtime. These different kinds of weavers may entail a runtime performance and a memory consumption cost, compared to the classical object-oriented approach. Using the *Dynamic and Static Aspect Weaving* (DSAW) AOSD platform, we have implemented three different scenarios of security issues in distributed systems (access control / data flow, encryption of transmissions, and FTP client-server). These scenarios were developed in both the aspect-oriented and object-oriented paradigms in order to evaluate the cost introduced by static and dynamic aspect weavers. A detailed quantitative evaluation of runtime performance and memory consumption is presented.

Keywords: Aspect-oriented software development, runtime performance, memory consumption, aspect weaving, DSAW.

1 Introduction

The *Aspect-Oriented Software Development (AOSD)* (Irwin et al. 1997) paradigm allows developers to make good use of the *Separation of Concerns* (SoC) principle (Hürsch & Lopes 1995) when developing applications. AOSD offers a direct support to modularise different functionalities that cut across system software. The modularisation of crosscutting concerns prevents tangling of the application source code, making it easier to debug, maintain and modify (Parnas 1972). Typical examples of crosscutting concerns are persistence, authentication, logging and tracing (Ortin et al. 2004). The process of integrating aspects into the main application code is called *weaving* and an aspect *weaver* is the tool that performs it (Ackoff 1971). The weaving process can be performed statically (compile time or load time) or dynamically (at runtime). Dynamic weaving AOSD platforms offer a powerful mechanism to dynamically adapt running applications, modifying their functionality while the system is being executed (Popovici et al. 2002). Application concerns can be modified, inserted or removed without stopping the application execution. However, in some scenarios, the use of AOSD may involve an increase of runtime performance and memory consumption (Garcia et al. 2012).

There are specific scenarios where it is necessary to adapt running applications in response to runtime emerging requirements (Vinuesa et al. 2008), such as distributed systems security (Garcia et al. 2012). Distributed systems involve the interaction between disparate and independent entities working toward a common goal (Belapurkar et al. 2009). As the number and arrangement of these potentially mobile entities may change, these systems are commonly required to be flexible and scalable. Under these circumstances, security in distributed systems is a complex issue to be considered. The security concerns of distributed systems can be modularised using AOSD, becoming possible to adapt the security measures without compromising their global security, even when their sizes and arrangements change at runtime (Garcia et al. 2012).

Our objective is to compare the runtime performance and the memory consumption of aspect-oriented and object-oriented programming (OOP) paradigms, evaluating the cost of aspect weaving. For this purpose, we have assessed three different scenarios of distributed systems security, where both static and dynamic weaving is appropriate. These examples consider access control, data flow, and data encryption. The solutions based on AOSD were developed using the *Dynamic and Static Aspect Weaving* (DSAW) (Vinuesa et al. 2008) platform. DSAW is an AOSD platform that supports both static and dynamic weaving, allowing the modification of application concerns at runtime. By using this platform, it is possible to dynamically modify the flow, access and encryption of data dynamically. Therefore, the security measures of the distributed systems can be adapted when required, varying in size and arrangement. Following a statistically performance evaluation methodology (Georges et al. 2007), these implementations are quantitatively assessed. A comparison between both paradigms is presented to estimate the penalty introduced by the AOSD approach, compared to the OOP one.

The remainder of this paper is structured as follows. Section 2 describes static and dynamic weaving,

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: projects TIN2008-00276 *Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software*, and TIN2011-25978 *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages*.

and the DSAW platform. The implemented applications used in the assessment are presented in Section 3. In Section 4, we evaluate and discuss the results of both approaches. Section 6 presents the conclusions and future work.

2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) (Irwin et al. 1997) is a concrete approach to implement the principle of *Separation of Concerns (SoC)* (Hürsch & Lopes 1995). AOSD facilitates the modularisation of different functionalities that cut across the entire system software (i.e., crosscutting concerns).

An *aspect* is a piece of code that cannot be encapsulated in a method or procedure, being scattered throughout the source code of an application. Common examples of aspects include transaction control, memory management, threading, persistence or logging (Hürsch & Lopes 1995).

With the classic object-oriented paradigm, crosscutting concerns in a system cannot be modularised as regular classes. AOSD handles this problem, allowing the separation of those concerns whose code is commonly tangled with the code of other classes. The major benefits of this approach are higher level of abstraction, concern reuse, higher legibility and improved software maintainability (Hürsch & Lopes 1995).

The final application is built by weaving the application aspects with the corresponding classes (Figure 1). The output code mixes the aspect code with the application functionality modularised in traditional classes. The aspect weaver performs this code processing, offering a higher level of modularisation to the programmer. As shown in Figure 1, the major difference between AOSD and OOP is that the object-oriented programmer has to decide where to place the code of the crosscutting concerns, whereas the aspect weaver automates this process. In this paper, we evaluate the cost of this automation in three different scenarios.

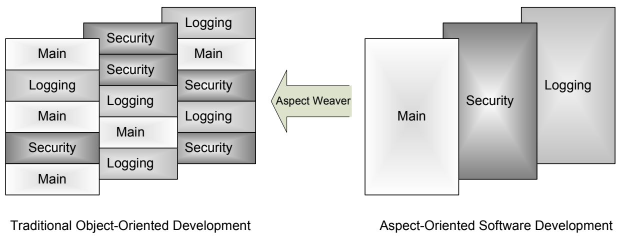

Figure 1: Traditional object-oriented development vs. aspect-oriented development.

2.1 Aspect Weaving

Once both the application components and the aspects are developed, it is necessary to build the final program (i.e., aspect weaving). The compiler of an object-oriented language receives the application source code and generates the executable file. In AOSD, the code (either source or binary) must also be processed by the aspect weaver to obtain the final program with full functionality. This process can be performed statically (at compile time or load time) or dynamically (at runtime).

Aspects should define the way they are related to application components, so that the aspect weaver can generate the final application by mixing the code of both kinds of modules. There are specific elements of the programming language semantics where the aspect code may be injected. These semantics elements are stable points of execution called *join-points* (Irwin et al. 1997). Therefore, it is necessary to describe the mapping between join-points and aspect code. For this purpose, *pointcuts* are defined as a set of join-points (usually using regular expressions) plus, optionally, some of the values in the execution context of those join-points (Kiczales et al. 2001).

2.1.1 Static Weaving

The majority of existing AOSD implementations provide static weavers. Static weavers combine the aspect and component functionality prior to application execution. This combination consists in inserting calls to *advice* in the components code. An advice is a method-like construct used to define the additional behaviour to be injected in the join-points expressed by a pointcut (Kiczales et al. 2001). An advice is the part of an aspect that modularises the code of the crosscutting concern.

This type of weaving commonly causes little performance penalty because all the code is combined and statically optimized before its execution. Since the application is woven before its execution, when a new aspect is required at runtime the application should be stopped, recompiled, rewoven and restarted, losing the non-persistent state of the process. There are scenarios where running applications require the dynamic addition, deletion or modification of aspects, and hence a dynamic weaving approach is more suitable (Ortin & Cueva 2004).

2.1.2 Dynamic Weaving

There are applications that need to be adapted at runtime in response to changes in their execution environment (Popovici et al. 2002, Zinky et al. 1997, Ségura-Devillechaise et al. 2003). An example is the so-called *autonomic software*; these systems should be able to repair, manage, optimise or recover themselves (Kephart & Chess 2003).

In the case of dynamic weaving, the program is compiled in the traditional way and an executable file is obtained. This program does not need to foresee which modules may be adapted at runtime. When the running program needs to be modified, it can be dynamically woven with new aspects that adapt the behaviour of the application.

The main advantage of this kind of weaving is that it supports the dynamic adaptation of programs, plus the modularisation of the different application concerns. Therefore, the resulting code is more adaptable and reusable, and both the aspects and the basic functionality can evolve independently (Pinto et al. 2001). However, this dynamic adaptation commonly entails a runtime performance cost (Böllert 1999).

2.2 Dynamic and Static Aspect Weaver

Existing dynamic weaving tools such as AOP/ST, PROSE, DAOP, JAC, CLAW, LOOM.NET, JAsCo or DSAW (Vinuesa et al. 2008) can be used to adapt running applications to new requirements, not foreseen at design time. Since we want to evaluate the cost of both static and dynamic weaving, a platform that supports both approaches may facilitate our work. That was the main reason why we selected DSAW (Vinuesa & Ortin 2004, Ortin et al. 2011), an

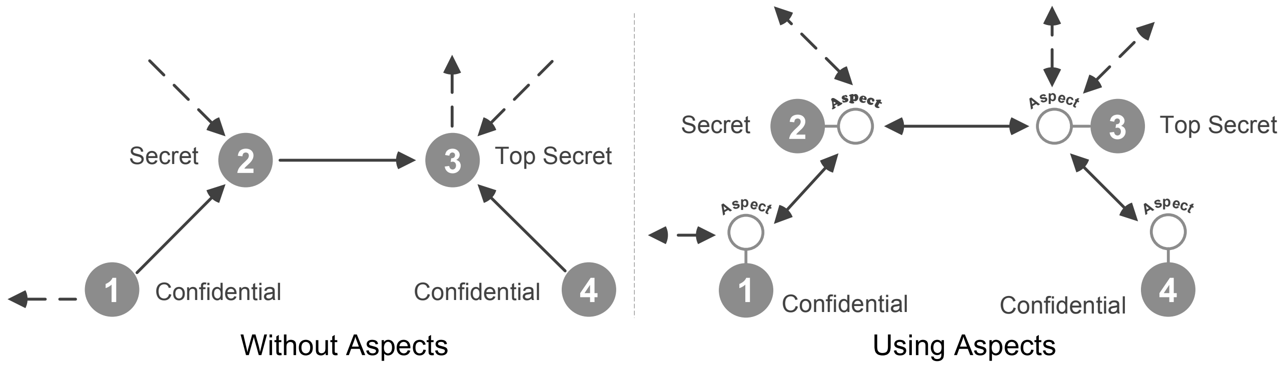

Figure 2: Distributed system with different authorisation levels.

aspect-oriented software development platform that supports homogeneous static and dynamic weaving. Its main features are:

1. Full dynamic weaving: DSAW instruments applications enabling all the application *join-points*. Since the adaptive JIT compiler of the CLR optimizes the code introduced by DSAW, the penalization is not significant. Then, DSAW allows runtime (un)weaving of aspects, even at join-points that were not woven before the application was executed.

2. Platform independence: It is designed over the .NET virtual machine reference standard (ECMA 2005). As a result, any .NET application can be run over DSAW.

3. Language independence: DSAW performs the adaptation applications at the virtual machine level, instrumenting the Intermediate Language (IL) of executable files and libraries. Any .NET high-level programming language can be used to program application components and aspects.

4. Weave-time independence: Aspect and component implementations do not depend on the type of weaving to be performed. Therefore, changing from dynamic to static, and vice versa, is a straightforward task.

5. Wide range of join-points: DSAW offers a wide and flexible set of join-points to facilitate the adaptation of applications for both dynamic and static scenarios.

3 Use cases

The objective of this paper is to compare runtime performance and memory consumption of the OOP and AOSD approaches. In order to do this, we have used both the static and dynamic weavers of DSAW, developing security issues of distributed systems (Garcia et al. 2012). DSAW has been used to implement two specific scenarios: access control / data flow and encryption of transmissions. A third scenario taking an existing real application, a FTP client and server, has also been used in our experiments. Details of these implementations are presented in (Garcia et al. 2012). These three scenarios were developed using both AOSD and the traditional OOP paradigm.

In the first scenario, we tackle the vulnerabilities caused by the flow of data through a network. Each node in the network has an authorization level. The security policy of the distributed system dictates that a node with an authorization level can only send and receive information from those nodes with greater or equal authorization level (NCSC 1990). The left part of Figure 2 shows an example. Nodes 1 and 4 can send information to any other node because the *confidential* level is the lowest one. Node 2 can only send information to node 3, since the *secret* authorization level is lower than *top secret*. Finally, node 3 cannot send information to anyone because it has the highest authorization level.

The traditional implementation only considers one-to-one relationships (Lang & Schreiner 2002), implying restrictions on data flow in point-to-point networks with changing topologies. For example, nodes 1 and 4 in Figure 2 have the same access level, but they cannot exchange information because node 3 cannot relay messages to nodes 2 and 4.

We have used the DSAW static weaver to implement a distributed system with this security policy, which guarantees the secure transmission of information over changing topologies, tagging data with the authorization levels of nodes. Applications are built relying on the classical *send* and *receive* operations, and aspects intercept these two messages to include the following functionalities:

1. Encryption of information to avoid unauthorized access to it.

2. Authentication to grant the user the appropriate authorization level.

3. Data tagging to determine how information flows across the network and to control the access to it.

As shown in the right part of Figure 2, all nodes can now exchange information between them regardless of their authorization level, because aspects control the data flow and restrict the access to data. As a result, nodes 1 and 4 can securely exchange data through nodes 2 and 3.

The second scenario is based on distributed systems made up by mobile devices, where network topologies and communication channels may dynamically change. If the user is connected to a distributed system and it is detected that the communication channel is not secure any more, encryption of transmissions may be required. Therefore, a dynamic encrypting aspect is woven with the application that uses the distributed system, while the system is running. The aspect is even able to forward the channel to another secure one if the mobile device allows it. Any kind of encryption or forwarding aspect can be woven at runtime, because the DSAW dynamic weaver does not impose any coupling between aspects and components. Finally, if the mobile device returns to a trusted environment, the encryption aspect is unwoven to avoid the unnecessary overhead of encryption.

The last scenario is a common client-server FTP environment[1]. We have added dynamic aspect weaving to the existing client and server applications in order to cipher all messages exchanged between them, when a more secure communication is needed. It is feasible to cipher the channel when critical information is exchanged, e.g. during the client login process, and to use the default channel when the exchanged information is not so important. In a standard client-server FTP communication, the information is sent and received directly. On the other hand, in an enhanced scenario where cipher is enabled using aspects, all the information passes through the dynamically woven aspect, responsible for encryption and decryption. Before either the server or the client sends a FTP command, the aspect encrypts the message; and just after a FTP command is received, the same aspect decrypts it. Thus, the exchanged information travels ciphered using the same channel, transparently to both the client and the server. If the aspect is then unwoven, the information flows as it does in the original scenario.

4 Evaluation

In this section, we present an assessment that compares runtime performance and memory consumption of the DSAW platform and the traditional OOP. The first subsection outlines the experimental methodology employed, describing the hardware, programming language and the scenarios used. For each use case described in the previous section, we present data of the runtime performance and memory consumption when using the AOSD and OOP paradigms. Finally, we present a discussion of the measurements obtained.

4.1 Methodology

We have implemented in DSAW the three use cases described in Section 3, using the aspect-oriented paradigm to modularise the crosscutting concerns in the applications. The first scenario (access control and data flow) is composed of three single nodes, and the second one (encryption) uses two single nodes and two encryption/decryption nodes –implementation details are presented in (Garcia et al. 2012). In order to compare this approach with OOP, we have also implemented the crosscutting concerns tangling their code with the rest of modules in the application, following the conventional approach of object-orientation. The two different versions of the three use cases vary in the way crosscutting concerns are tangled: by a human or by an aspect weaver. The objective is to measure the memory consumption and runtime performance cost of DSAW static and dynamic weaving. All the applications were developed in the C# programming language.

Regarding data analysis, we have followed the methodology proposed by Georges (Georges et al. 2007) to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT compilation. In this methodology, the start-up performance measures how quickly a system can run a relatively short-running application. To measure start-up performance, a two step methodology is used:

1. We measure the elapsed execution time of running multiple times the same program. This results in p (we have taken $p = 30$) measurements x_i with $1 \leq i \leq p$.

2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is computed using the *Student's t*-distribution because we took $p = 30$ (Lilja 2000). Therefore, we compute the confidence interval $[c_1, c_2]$ as:

$$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

Being $\overline{x}$ the arithmetic mean of the x_i measurements, $\alpha = 0.05$ (95%), s the standard deviation of the x_i measurements, and $t_{1-\alpha/2;p-1}$ defined such that a random variable T, that follows the *Student's t*-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1 - \alpha/2$.

The data provided is the mean of the 95% confidence interval.

To measure runtime performance, we have instrumented the code with hooks that registers the value of high-precision time counters provided by the Windows 7 operating system. This instrumentation calls the native function `QueryPerformanceCounter` of the `kernel32.dll` library. This function returns the execution time measured by the operating system Performance and Reliability Monitor (MicrosoftTechnet 2012). We measured the difference between the beginning and the end of exchanging a set of messages to obtain the total execution time. Tests were made with different message sizes.

For memory consumption, we measured the maximum size of working set memory used by the process (the `PeakWorkingSet` property). The working set of a process is the number of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including instructions from the process modules and the system libraries.

These implementations have been compared using the .NET Framework 2.0 build 50727 for 32 bits, over a Windows 7 x64 operating system. All tests have been carried out on a lightly loaded 2.13GHz Intel Core 2 Duo system with 4GB of RAM.

4.2 Evaluation

To evaluate the cost of static and dynamic weaving in DSAW, we have developed the three proposed scenarios using the traditional object-oriented programming paradigm. Using object-orientation, we have extended the implementations with access control and data flow security measures in the first use case, and encryption in the other two scenarios. These same functionalities were also developed as separate aspects, using the AOSD paradigm.

In the control access and data flow scenario, we have used static weaving to inject the aspect in the original system. In the encryption and FTP use cases, the DSAW dynamic weaver was employed. Following the start-up methodology presented in Section 4.1, we measured the influence of the number of messages on the performance and memory penalties. Since both penalties remained constant, we used a fixed number

[1]We have used the FTP.Net client (`http://ftpnet.sourceforge.net`) and the SimpleFTP Server (`http://www.tudra.net/wp/2007/10/15/simpleftp-server`).

of 6,000 messages for the first and third applications, and 100,000 messages for the second one.

Figures 3 to 5 show the runtime performance penalty in the three scenarios. For each application, we increase the number of words contained in the messages in order to analyse the influence of message sizes in runtime performance. Performance penalties are calculated relative to the corresponding OOP implementation. Values are the difference between the DSAW and OOP execution times, divided by the value of the OOP implementation (expressed in percentage form) for each message size (expressed in number of words).

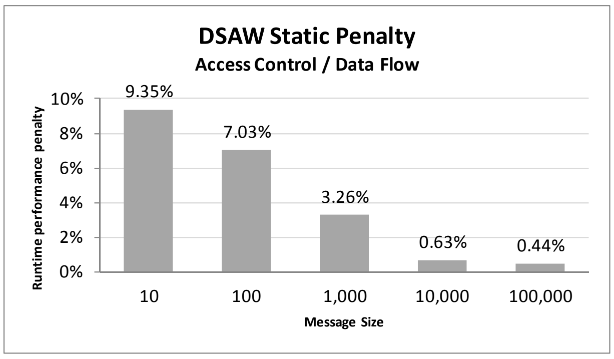

Figure 3: Runtime performance penalty of the Access Control / Data Flow application.

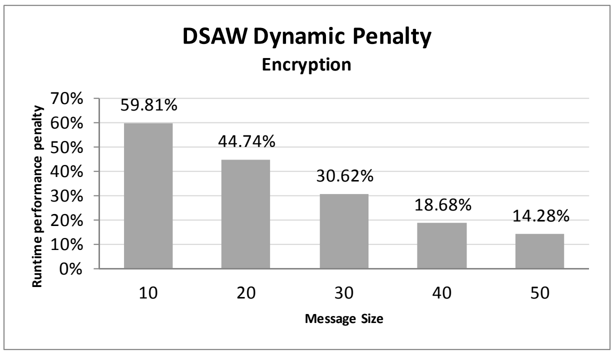

Figure 4: Runtime performance penalty of the Encryption application.

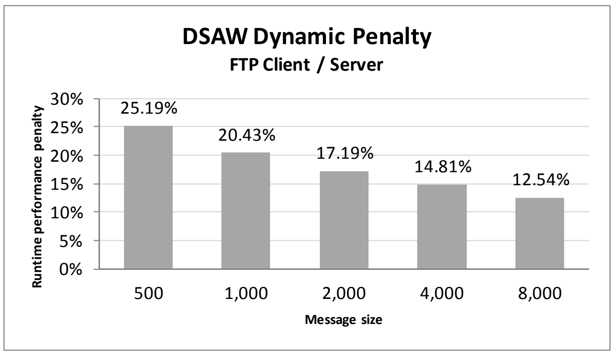

Figure 5: Runtime performance penalty of the FTP application.

Concerning the memory consumption, Tables 1 to 3 show the memory usage in the three scenarios: access control / data flow (OOP vs. DSAW static), encryption (OOP vs. DSAW dynamic) and FTP client-server (OOP vs. DSAW dynamic). Memory consumption is expressed in KBytes, and message size in number of words per message.

4.3 Discussion

After presenting the data in Figures 3 to 5 and Tables 1 to 3, three issues are highlighted. The first one is related to the runtime performance cost of weaving. In all scenarios, the performance costs decrease as the size of messages increases. In the first application, the performance cost of static weaving varies from 9.35%, with the minimum message size, to 0.44%, when messages are 10,000 times greater. In the second scenario, the cost decreases from 59.81% (10 words per message) to 14.28% (50 words per message). Finally, for the FTP application shows a performance penalty of 25.19% for the smallest message, while this penalty drops to 12.54% when the message size is multiplied by 16. Therefore, runtime performance penalty grows as the size of the message drops. This dependency is caused by the number of intercepted joinpoints executed, which remains constant in each experiment. For bigger messages, the overall execution time rises, but the injection code executed (the number of joinpoits) stays the same. The FTP application shows a smaller dependency on the message size. This is because the application executes 10 commands (such as creating, changing and erasing directories), and only one is file (message) transmission.

The second discussion is the different performance penalties depending on the type of weaving. In the static scenario, runtime performance penalty is between 9.35% and 0.44%. When the size of messages is significantly high, the cost of static weaving is almost negligible. However, the cost of dynamic weaving is more notable. The encryption application showed a performance penalty between 59.81% and 14.28%, and 25.19% and 12.54% in the case of FTP. This higher performance cost of dynamic weaving is caused by different factors. First, the execution of the dynamic weaver is included in the overall execution time. Second, the runtime examination of joinpoint registration (checking whether there are aspects waiting for a joinpoit to be executed) also implies a performance price. Finally, when a joinpoint is reached, registered aspects are called by means of an indirection (a reference); whereas static weaving simply tangles the code, enabling the optimizations performed by the JIT compiler (Redondo et al. 2008, Ortin et al. 2009).

The last issue is related to memory consumption. In every scenario, the memory consumption penalty is not affected by the size of messages: standard deviations of the three applications where 0.14%, 0.34% and 0.26%, respectively. The static weaving technique has shown an average memory consumption increase of 2%. This average cost augments to 60% and 47% when dynamic weaving is used in the two last scenarios. This difference is due to the additional code and the registered aspects per joinpoint table implemented by the dynamic weaver to allow dynamic adaptation of components. Therefore, the cost of dynamic weaving examples has been higher than static ones, for both runtime performance and memory consumption.

5 Related Work

There are some existing works that compare the runtime performance of different AOSD platforms (Vinuesa et al. 2008, Bijker 2005, Vanderperren & Suvée

Message size	10	100	1,000	10,000	100,000
Object-Oriented	28,001	28,112	29,532	38,272	51,788
DSAW Static	28,735	28,859	30,387	39,400	53,204

Table 1: Memory consumption (KBs) of the Access Control / Data Flow application.

Message size	10	20	30	40	50
Object-Oriented	27,408	27,440	27,524	27,517	27,650
DSAW Dynamic	43,880	43,880	44,187	43,980	44,148

Table 2: Memory consumption (KBs) of the Encryption application.

Message size	500	1,000	2,000	4,000	8,000
Object-Oriented	33,708	33,799	33,799	33,832	33,764
DSAW Dynamic	49,484	49,600	49,768	49,772	49,716

Table 3: Memory consumption (KBs) of the FTP application.

2004), but there are not many that compare AOSD implementations with equivalent OOP versions. Hilsdale and Hugunin (Hilsdale & Hugunin 2004) assess both run-time and compile-time performance of AspectJ, introducing a simple logging policy as an aspect in a benchmark. In this experiment, the aspect is woven statically. It logs all the method entries of the XSLTMark benchmark. The obtained results are fairly similar to the results presented in this work. The runtime performance cost of the AOSD version compared to the hand-coded one is around 3%. Moreover, the authors present a comparison between the original application (without any modification) and the AOSD version (with logging) to evaluate the overhead introduced by the logging aspect. Using different versions of the aspect code, this overhead is greatly reduced from 2,500% to 22%.

Regarding the use of AOSD to implement and adapt security measures, Viega (Viega et al. 2001) proposes an extension of the C programming language to support aspects. This extension allows the definition of security policies apart from the application code. AspectJ has also been used for security issues. Huang (Huang et al. 2004) presents a generic and reusable library to introduce security mechanisms in Java developments. This library provides reusable and generic aspects in AspectJ, as practical software components, and a prototype implementation of a common security-relative API for AOSD. Kim and Lee (Taeho & Hongchul 2008) also use AspectJ to discuss how authorisation capabilities could be added to existing well-structured, object-oriented systems. In these works, neither run-time performance nor memory consumption is assessed.

6 Conclusions

Aspect-Oriented Software Development allows the separation of crosscutting concerns in software development. The modularisation of different application concerns provides higher level of abstraction, concern reuse, higher legibility and improved software maintainability. However, in some scenarios, the use of AOSD may involve an increase of runtime performance and memory consumption. This paper presents a comparison of runtime performance and memory consumption between three different applications developed using AOSD and the classical object-oriented approach. All the applications were developed in the DSAW platform and using the C# programming language. The three applications apply security measures to distributed systems: access control and data flow, communications encryption, and FTP client-server. The first application is statically woven, whereas the two last ones require dynamic weaving.

The assessment of runtime performance has shown that the DSAW static weaver have entailed a performance cost of 9.35%, compared to the traditional object-oriented development. When the size of the messages increases, the performance cost decreases to values near to zero. In the dynamic weaving scenarios, runtime performance penalties were 59.81% and 25.19%, dropping to values around 13% when message sizes grow. Different factors in the dynamic weaving technique implemented by DSAW causes this performance increase compared to static weaving. In all the scenarios, memory consumption has not depended on the size of messages. The memory usage penalty of static weaving was around 2%, and 60% and 47% in the case of dynamic weaving.

We plan to apply this evaluation methodology to commercial aspect-oriented applications developed in other platforms such as AspectJ, Spring AOP, JAsCo or JBoss AOP. Regarding the use of AOSD in distributed systems security, future work will be focused on applying the AOSD approach to develop other security measures such as *Intrusion Detection* or *Load Balancers*. A more involved question concerns how our approach can be suitably generalised in distributed "systems-of-systems" scenarios (Ackoff 1971). We aim to address this challenge in the future by considering more flexible means of defining pointcuts, such as by allowing pointcuts to be defined in a reactive manner, taking into account the results of analysis of multiple interacting applications within a network, and using different techniques together with our approach (Söldner et al. 2008, Tanter et al. 2009).

The current documentation and implementation of this work can be freely downloaded from (DSAW 2010).

References

Ackoff, R. (1971), 'Towards a system of systems concepts', *Management Science* **17**(11), 661–671.

Belapurkar, A., Chakrabarti, A., Ponnapalli, H., Varadarajan, N., Padmanabhuni, S. & Sundarrajan, S. (2009), *Distributed Systems Security: Issues, Processes and Solutions*, Wiley.

Bijker, R. (2005), Performance effects of aspect oriented programming, *in* '3rd Twente Student Conference on IT, Enschede June'.

Böllert, K. (1999), On weaving aspects, *in* 'Workshop on Object-Oriented Technology', Springer-Verlag, p. 302.

DSAW (2010), 'Using DSAW in distributed system security systems', `http://www.reflection.uniovi.es/dsaw/download/2010/ietsw/`.

ECMA (2005), 'TG3. Common Language Infrastructure (CLI). Standard ECMA-335'.

Garcia, M., Llewellyn-Jones, D., Ortin, F. & Merabti, M. (2012), 'Applying dynamic separation of aspects to distributed systems security: a case study', *Software, IET* **6**(3), 231–248.

Georges, A., Buytaert, D. & Eeckhout, L. (2007), 'Statistically rigorous Java performance evaluation', *ACM SIGPLAN Notices* **42**(10), 57–76.

Hilsdale, E. & Hugunin, J. (2004), Advice weaving in AspectJ, *in* 'International Conference on Aspect-Oriented Software Development (AOSD)', ACM, pp. 26–35.

Huang, M., Wang, C. & Zhang, L. (2004), Toward a reusable and generic security aspect library, *in* 'AOSD Technology for Application-Level Security (AOSDSEC)', Vol. 4, pp. 5–10.

Hürsch, W. & Lopes, C. (1995), 'Separation of concerns', *Northeastern University, February* .

Irwin, J., Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. & Loingtier, J. (1997), Aspect-oriented programming, *in* 'European Conference on Object-Oriented Programming (ECOOP)', pp. 220–242.

Kephart, J. & Chess, D. (2003), 'The vision of autonomic computing', *Computer* **36**(1), 41–50.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. (2001), An overview of AspectJ, *in* 'European Conference on Object-Oriented Programming (ECOOP)', Springer, pp. 327–354.

Lang, U. & Schreiner, R. (2002), *Developing secure distributed systems with CORBA*, Artech House Publishers.

Lilja, D. (2000), *Measuring computer performance: a practitioner's guide*, Cambridge University Press.

MicrosoftTechnet (2012), 'Windows server techcenter: Windows performance monitor'.

NCSC, N. C. S. C. (1990), 'Trusted network interpretation environments guideline'.

Ortin, F. & Cueva, J. M. (2004), 'Dynamic adaptation of application aspects', *Journal of Systems and Software* **71**(3), 229–243.

Ortin, F., Lopez, B. & Perez-Schofield, J. (2004), 'Separating adaptable persistence attributes through computational reflection', *Software, IEEE* **21**(6), 41–49.

Ortin, F., Redondo, J. M. & Baltasar García Perez-Schofield, J. (2009), 'Efficient virtual machine support of runtime structural reflection', *Science of Computer Programming* **74**(10), 836–860.

Ortin, F., Vinuesa, L. & Felix, J. (2011), 'The DSAW Aspect-Oriented Software Development Platform', *International Journal of Software Engineering and Knowledge Engineering* **21**(7), 891.

Parnas, D. (1972), 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM* **15**(12), 1053–1058.

Pinto, M., Amor, M., Fuentes, L. & Troya, J. M. (2001), Run-time coordination of components: Design patterns vs. component-aspect based platforms, *in* L. Bergmans, M. Glandrup, J. Brichau & S. Clarke, eds, 'Workshop on Advanced Separation of Concerns (ECOOP)'.

Popovici, A., Gross, T. & Alonso, G. (2002), Dynamic weaving for aspect-oriented programming, *in* 'International Conference on Aspect-Oriented Software Development', ACM Press, pp. 141–147.

Redondo, J. M., Ortin, F. & Cueva, J. M. (2008), 'Optimizing reflective primitives of dynamic languages', *International Journal of Software Engineering and Knowledge Engineering* **18**(6), 759–783.

Ségura-Devillechaise, M., Menaud, J., Muller, G. & Lawall, J. (2003), Web cache prefetching as an aspect: towards a dynamic-weaving based solution, *in* 'International Conference on Aspect-Oriented Doftware Development (AOSD)', ACM, p. 119.

Söldner, G., Schober, S., Schröder-Preikschat, W. & Kapitza, R. (2008), 'AOCI: Weaving components in a distributed environment', *On the Move to Meaningful Internet Systems: OTM 2008* pp. 535–552.

Taeho, K. & Hongchul, L. (2008), 'Establishment of a security system using aspect oriented programming', *2008 International Conference on Control, Automation and Systems* pp. 863–866.

Tanter, É., Fabry, J., Douence, R., Noyé, J. & Südholt, M. (2009), Expressive scoping of distributed aspects, *in* 'ACM international Conference on Aspect-Oriented Software Development (AOSD)', ACM, pp. 27–38.

Vanderperren, W. & Suvée, D. (2004), Optimizing JAsCo dynamic AOP through Hotswap and Jutta, *in* 'AOSD Workshop on Dynamic Aspects', Vol. 3.

Viega, J., Bloch, J. & Chandra, P. (2001), 'Applying aspect-oriented programming to security', *Cutter IT Journal* **14**(2), 31–39.

Vinuesa, L. & Ortin, F. (2004), 'A Dynamic Aspect Weaver over the. NET Platform', *Metainformatics* **3002**, 197–212.

Vinuesa, L., Ortin, F., Felix, J. M. & Alvarez, F. (2008), DSAW - a dynamic and static aspect weaving platform., *in* 'International Conference on Software and Data Technologies (ICSOFT)', INSTICC Press, pp. 55–62.

Zinky, J., Bakken, D. & Schantz, R. (1997), 'Architectural support for quality of service for CORBA objects', *Theory and Practice of Object Systems* **3**(1), 55–73.