Performance Analysis of the OGSA-DAI Software

Mike Jackson, Mario Antonioletti, Neil Chue Hong, Alastair Hume, Amy Krause, Tom Sugden and Martin Westhead

EPCC, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK

Abstract

This paper describes the experiences of the OGSA-DAI team in profiling and benchmarking components of the OGSA-DAI database access middleware built using the Open Grid Services Infrastructure and the emerging Database Access and Integration Services Global Grid Forum recommendations. The profiling approach and the tools used are described. A number of areas of concern are then analysed in detail. In particular, the analysis focuses on running database queries, creating Document Object Model objects, utilising Globus Toolkit Grid Security Infrastructure mechanisms, validating XML against XML Schema and inter-dependencies between third-party software used within OGSA-DAI.

1. Introduction

The *Open Grid Services Architecture – Data Access and Integration* (OGSA-DAI) project [1] started in February 2002. The first production release – Release 3.0 – was made in July 2003 shortly after the *Globus Toolkit 3.0* (GT3) release. Code development has been undertaken by two teams, at EPCC and IBM UK, with design input coming from the other project members.

The main goal of OGSA-DAI is to serve the UK e-Science community by providing middleware that allows controlled exposure of data resources to Grid environments. This then provides the base services required by higher-level data integration services, e.g. Distributed Query Processing (DQP) [2] and Data Federation services.

The OGSA-DAI project continues to develop its software – improving performance, providing support for additional data resources, tackling data integration issues and moving towards inter-operability with developing Web and Grid standards.

1.1 OGSA-DAI and the Grid Data Service

Currently, OGSA-DAI extends the Open Grid Services Infrastructure (OGSI) [3] by defining portTypes which are used to construct a set of base data services that provide uniform access to databases. The key service is the *Grid Data Service* (GDS). The GDS represents a database access end-point for a client and maintains the client's session with that database. Clients invoke database functionality by submitting *GDS-Perform documents* – XML documents which specify the actions a client wants performed on a database (e.g. queries or updates). The implementation of a GDS is embodied in the *GDS Engine* which parses and executes GDS-Perform documents, manages database connections via third-party drivers, and constructs *GDS-Response documents* – XML documents holding the results of the client's actions (e.g. query results or update counts).

1.2 OGSA-DAI and Performance

One of the challenges faced by the OGSA-DAI project has been to produce software which efficiently handles database-related requests. Application benchmarking [4][5] has already been reported by other groups. This paper focuses on profiling undertaken by the OGSA-DAI team itself. In contrast to the work of [4][5], which focuses on data access scenarios and scalability from an applications point of view, the profiling described in paper was undertaken with the aim of tuning the performance time of OGSA-DAI – identifying limitations and bottlenecks in the implementation – rather than with the aim of identifying fundamental architectural and design flaws. Both methodology and results may be useful to other groups undertaking similar types of work.

2. Profiling OGSA-DAI

The GDS is the service through which clients access a database. Profiling activity therefore focused on this service. In particular, the *Perform* operation through which GDS-Perform documents are submitted by and GDS-Response

documents returned to clients. The following profiling was undertaken:

- Identifying bottlenecks in the execution of GDS-Perform documents.
- Identifying overheads incurred by security.
- Examining the validation of GDS-Perform documents against their XML Schema.

OGSA-DAI Release 3.0.2 was profiled. This was deployed on an Apache Tomcat 4.1.29 / Globus Toolkit 3.0.2 (GT3) stack running on a Redhat Linux 9 distribution on a dual 2.4 GHz Pentium IV Xeon processor Intel machine with 2 Gb of memory. A 10,000 row OGSA-DAI `littleblackbook` MySQL database table – distributed with OGSA-DAI – was used, with the MySQL Connector/J 2.0.14 driver [6] managing OGSA-DAI-MySQL communications. During profiling, Tomcat was shut-down and re-started between repeated iterations to minimise the risk of caching effects within GT3 and OGSA-DAI.

2.1 Profiling Tools

A number of useful profiling tools were used:

- `System.currentTimeMillis` – a Java method that returns the current system time which can then be output to a console or a file.
- Apache Log4J [7] – a collection of classes for software logging. Logging statements, which are used to log messages to a file, can be added to a program. A developer can assign different priority levels to messages. Log messages can include elapsed execution time in milli-seconds, priority level, the name of class from which the message originates, the thread number and any developer-specific information.
- Borland Optimizeit [8] and EJ-Enterprises JProfiler [9] – method call visualisers which monitor CPU load and memory usage. These can be hooked into Tomcat and support server-side performance analysis. Optimizeit records method calls which can be visualised later and visualises threads in separate displays. JProfiler, in contrast, visualises in real time and has an integrated thread display.

2.2 Analysis Method

The profiling method consisted of submitting GDS-Perform documents to a GDS and then analysing method call performance using both Optimizeit and JProfiler to identify potential bottlenecks and areas for further investigation. Apache Log4J statements were then added to the appropriate OGSA-DAI code so that information suitable for analysis could be gathered over repeated runs. These were assigned a high priority level and used a standard message prefix to facilitate their extraction from log files. This method relied upon the team having an intuition as to where potential bottlenecks might be occurring – the team considered methods relating to security or XML document validation and manipulation to be of particular concern.

3. Profiling the GDS::Perform operation – Server-side Perspective

The GDS::Perform operation was profiled by submitting the simplest possible GDS-Perform document – one requesting execution of an `N`-row SQL query "`SELECT * FROM littleblackbook WHERE id < N`" where N = [100 | 250 | 500 | 750 | 1000 | 2500 | 5000 | 7500 | 10000]. The times taken to complete the following activities were recorded:

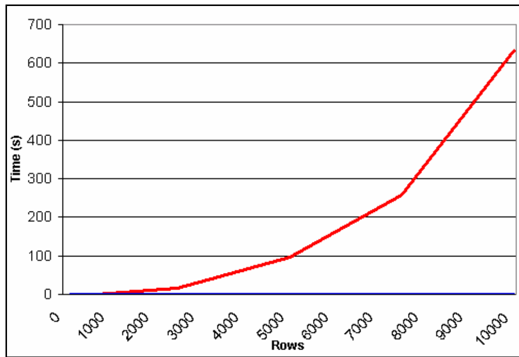
- Executing the GDS-Perform document – from the moment it is received from the GT3 infrastructure to the moment a GDS-Response document is handed over to GT3 to return to the client.
- Validating and parsing the GDS-Perform document.
- Loading a database driver, connecting to the database and configuring the connection.
- Using the driver to execute the query.
- Extracting the results from the database via the driver.
- Closing the database connection.
- Converting the results into a WebRowSet [10] XML representation and then building a GDS-Response document holding this WebRowSet representation.

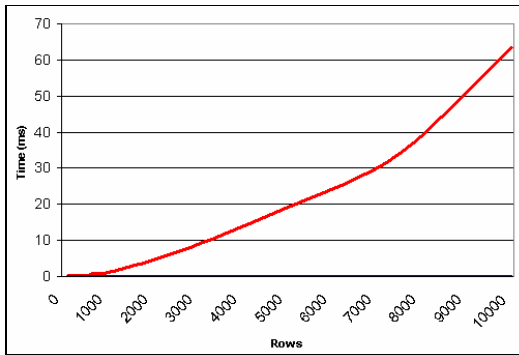
Profiling revealed that:

- 90% of the time taken by the Perform operation was spent within the GDS Engine.
- Validating a GDS-Perform document against its XML Schema took an average of 140 ms irrespective of the number of rows.
- Initialising the GDS Engine, loading a database driver, connecting to the database, running the query, and

extracting the results from the database took an average time of approximately 7 ms irrespective of number of rows.

- Post-operation clean-up within the GDS increased from 4 ms to 100 ms as the number of rows in the query increased. The reason for this is not known at this time.





Red lines indicate the times from the original analysis. Blue lines show the times recorded using a refactored GDS Engine adopting the performance enhancement of section 4.

Figure 1: Total GridDataService::Perform execution time and GDS Engine processing time per result row

Of most concern, however, was the fact that execution time degraded exponentially in relation to the number of rows in the query as shown in Figure 1. Studying the code invoked during the Perform operation revealed a number of suspects:

1. Code that prettifies the WebRowSet documents to enhance their readability – clearly this should not be the responsibility of a GDS.
2. Inefficient implementation structures including: numerous nested-ifs for handling database product-specific conditions and the distinction between database queries and database updates, case statements with large numbers of conditions, if statements within while loops, repeated array accesses and list size checks within loops.
3. Creation of multiple threads within the GDS Engine and the blocking of threads.
4. Java `StringBuffer` to `String` conversion.
5. Building XML documents using Document Object Model (DOM) [11] objects.

Addressing points 1, 2 and 4 yielded only small constant improvements in execution time. To avoid diving into multi-threaded management issues it was decided to focus on 5. This revealed a significant overhead, the cause and solution of which are discussed in section 4.

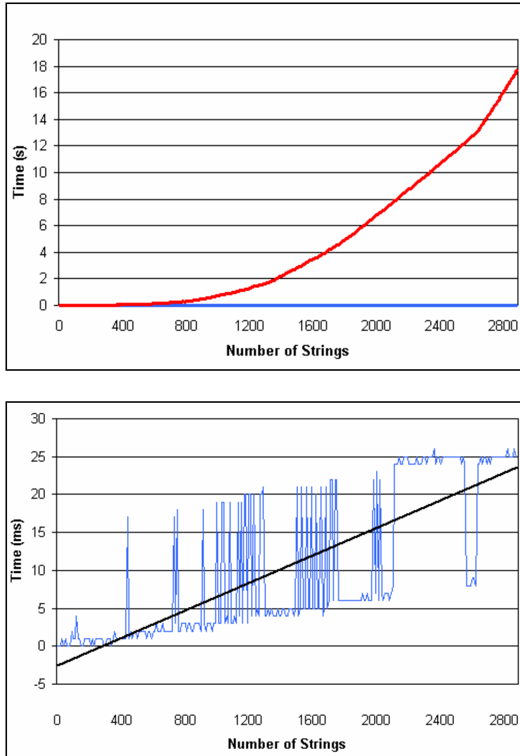
The analysis was performed again, but using a version of the GDS which adopted the solution of section 4. The results are shown in the graphs of Figure 1. The refactoring yielded a significant improvement in the performance of the Perform operation.

4. A DOM Deficiency

When building WebRowSet representations of query results, a DOM object is constructed. As each row is extracted from the database driver it is converted to XML and added to this DOM object. OGSA-DAI, like GT3, uses the Apache Xerces 2.4 [12] implementation of DOM (which, in turn, implements the Java 1.4 `org.w3c.dom` API). A performance problem arose from the use of a method, `appendData`, on the Xerces implementation of an `org.w3c.dom.Text` class, which appends a `String` to a DOM object. The first graph of Figure 2 plots the following:

- Time to append N `Strings` to a DOM object using the `Text.appendData` method.
- Time to append N `Strings` to a `StringBuffer`, using its `append` method, convert the `StringBuffer` to a `String`, and then append this `String` to a DOM object using a single `Text.appendData` call.

The graph shows that reducing reliance on the `Text.appendData` method leads to a significant reduction in execution time. The second graph of Figure 2 shows that the `StringBuffer.append` method does degrade as the number of `Strings` in the `StringBuffer` increases, but this degradation is both shallow and linear.



The upper graph shows the time taken to construct a DOM object consisting of a given number of `Strings` using repeated calls of `Text.appendData` compared to the use of a `StringBuffer` to collect together `Strings` before a single invocation of `Text.appendData`. The lower graph shows the performance degradation of using a `StringBuffer` to append `Strings`.

Figure 2: Appending DOM Objects and using `StringBuffer`

It should be noted however, that this time reduction comes at the expense of additional program logic. Checks must be made to ensure that the contents of the `StringBuffer` are flushed into the DOM object before any other object tries to access the DOM object. If this is not done then an incorrect view of the document being modelled may result. If such access could be frequent then using a `StringBuffer` and flushing regularly using `Text.appendData` may actually be less efficient than solely using `Text.appendData`. The solution to use should therefore be made on an application-specific case-by-case basis.

5. Profiling the GDS::Perform operation – Client-side Perspective

The GDS::Perform operation was profiled from a client's perspective by using `System.getTimeMillis` statements to time the invocation of the Perform operation by a client. A client which connected directly to the `littleblackbook` database via JDBC and using the MySQL Connector/J 2.0.14 driver was also profiled. In this client, the time was calculated from prior to the connection to the database to when the connection was closed after retrieving the query results.

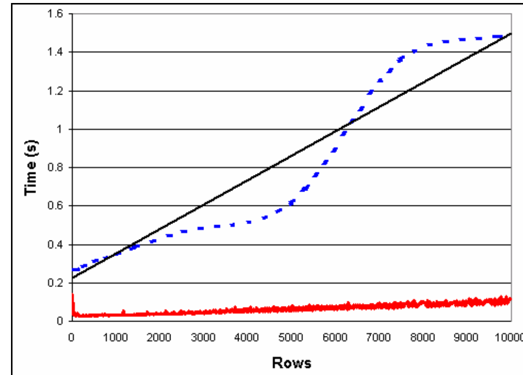


Figure 3: Comparing a direct JDBC connection (red) to OGSA-DAI (blue)

Figure 3 shows the results. The fact that OGSA-DAI incurs a higher round-trip time than JDBC is not a surprise, since OGSA-DAI also incurs overheads relating to the processing of the GDS-Perform document and building the GDS-Response documents in addition to GT3 and Tomcat overheads. However, within OGSA-DAI performance increases in relation to the number of rows at a steeper rate than the performance increase observed for a direct JDBC connection. It is likely that this is due to the overhead of processing and passing around the WebRowSet representation of the query results within OGSA-DAI. This is of concern if OGSA-DAI is to be a credible alternative to direct database connection solutions.

6. Security

This section describes investigations into the overheads incurred when enforcing various security configurations on the GDS. Four types of GDS were tested:

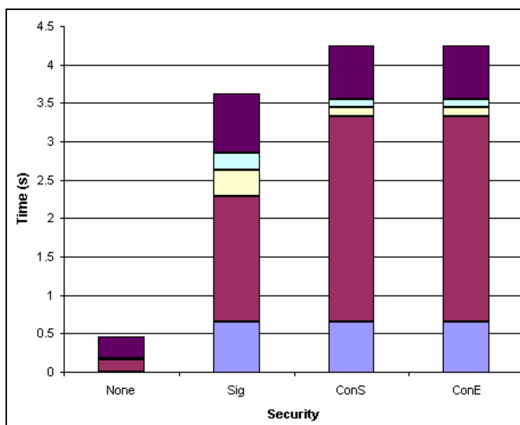
- GDS with no security (None).
- GDS which enforces GSI XML Signature (Sig).
- GDS which enforces GSI Secure Conversation with Message Signing (ConS).
- GDS which enforces GSI Secure Conversation with Message Encryption (ConE).

Profiling was performed from the perspective of the client and the server.

6.1 Client-side

At the client-side, three calls to the GDS FindServiceData operation were made followed by a call to Perform. This was done to identify different overheads in setting up security contexts between a client and server (establishing a security context is a characteristic of GSI Secure Conversation). Durations were calculated for:

- Calls to GT3 GSI modules to create a credential for the client based upon a user certificate and key.
- Each call to FindServiceData and Perform.



Security from the client-side showing the overheads (reading from bottom of the graph to the top) for creating/initialising a client credential, three consecutive calls to FindServiceData and a call to Perform.

Figure 4: Security from the client-side

From the results presented in Figure 4 the following points can be observed:

- Credential initialisation has a constant overhead of 650 ms regardless of security type.
- All calls to secure GDSs incur a longer round-trip time.
- The second and third calls to FindServiceData and the call to Perform are less for GSI Secure Conversation than for GSI XML Signature. This is unsurprising as GSI Secure Conversation expects a shared security context to exist between the client and the service. In contrast, no such context is present for GSI XML Signature.
- Use of a shared context by GSI Secure Conversation explains why the initial FindServiceData call is more costly for GSI Secure Conversation that that for GSI XML Signature. However, if one were to make numerous calls then this initial overhead may be recouped from the savings accrued from subsequent secure operation calls.
- Part of the overhead for the initial FindServiceData calls also includes GT3 initialising the GDS – hence the longer duration even when no security is present.
- For large queries/complex GDS-Perform documents, the security overhead would be subsumed within the cost of executing the GDS-Perform document as a whole.

6.2 Server-side

The following areas of the implementation of the GDS Perform operation were timed:

- Accessing client credentials using the GT3 infrastructure.
- Extracting the client's distinguished name – via GT3 infrastructure – if no credential is provided by the client or no distinguished name accessed from the credential.
- Mapping a client's distinguished name to a database user name and password.
- JDBC calls to connect to the database using this user name and password.

Figure 5 shows the security-related overheads in relation to the total time required to complete the Perform operation. The security-related overheads are also shown in more detail. The overhead for non-security related activities is constant regardless of the security enforced by the GDS. However, credential extraction incurs a greater overhead if no security is present (12ms) – since a number of security-related checks and attempts to get the credentials are made and fail – compared to when security is present (1-2ms).

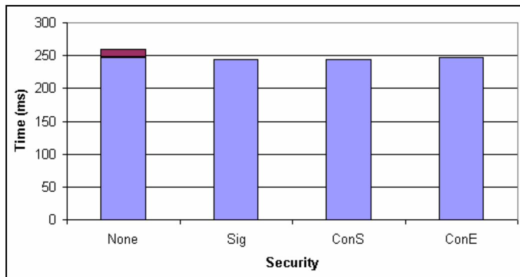
Figure 6 shows that security has no effect when connecting to the database. Costs incurred in mapping a client's distinguished name to a database user name and password, connecting to the database and other overheads are relatively constant.

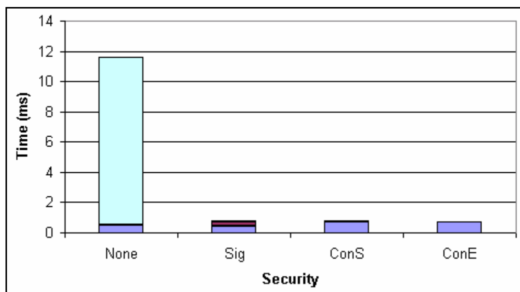
Again, this demonstrates that security overheads are of the order of a few milli-seconds and therefore do not substantially degrade OGSA-DAI performance.

7. Validating Against XML Schema

As described in section 3, validating the GDS-Perform document against its XML Schema takes approximately 140 ms. This

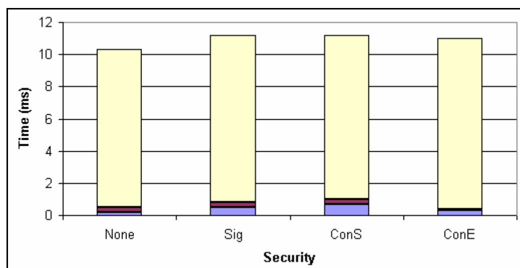
validation is performed by the `parse` method of the Xerces 2.4 class `org.apache.xerces.parser.DOMParser`. During initial analysis, however, a validation time of 300 ms was evident. Investigating this discrepancy revealed a dependency between the above `DOMParser.parse` method and the JDBC `java.sql.DriverManager.getConnection` method.





The upper graph shows the overheads for security-related as a proportion of the time taken to complete the GDS::Perform operation. The lower graph shows the overheads for running a security check and getting the client credentials (or, for the case of None, failing to get the credentials)

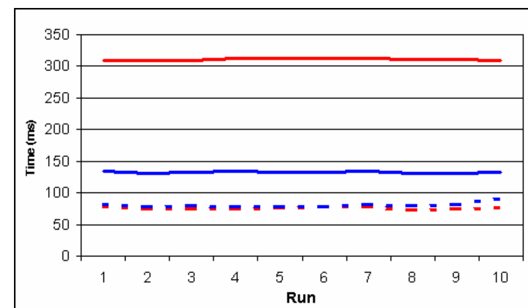
Figure 5: Security from the server-side



Security from the server-side showing the overheads (reading from bottom of the graph to the top) for mapping a user credential to a database user name and password, other connection overheads and creating a database connection.

Figure 6: Security and database connection

The results, shown in Figure 7, reveal that when a call to `DriverManager.getConnection` occurs after a call to `DOMParser.parse` to validate a GDS-Perform document then the time to parse the document takes 300 ms. However if `DriverManager.getConnection` has already been invoked prior to the call to `DOMParser.parse` then the validation only takes 140 ms. This implies a dependency between the implementation of `java.sql.DriverManager`, the MySQL Connector/J 2.0.14 driver (invoked by `DriverManager`) or Xerces `DOMParser.parse`. The exact nature of this dependency is unknown at present but is most likely to be the loading of some shared class.



Average, over ten runs with ten GDSs, of durations of calls to `DriverManager.getConnection` (dashed) and `DOMParser.parse` (solid) where `getConnection` is called before (blue) or after (red) `parse`.

Figure 7: `DOMParser.parse` and `DriverManager.getConnection` dependency

7.1 Another Implicit Security Dependency

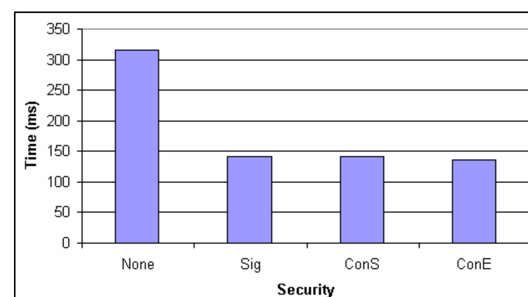


Figure 8: Security and `DOMParser.parse` duration

Figure 8 reveals another dependency involving `DOMParser.parse`. Provided that there is no prior invocation of `DriverManager.getConnection` then the invocation of `DOMParser.parse` to parse

a GDS-Perform document takes on average 300 ms if no security is applied while it only takes 140 ms if security is applied. This is due to the fact that GT3 uses server-side XML files to specify security configurations and these are loaded and parsed when secure services are first created – the classes for parsing XML files have therefore already been loaded when `DOMParser.parse` is called to validate a GDS-Perform document.

8. Conclusions

This paper has described the experiences of the OGSA-DAI team undertaking a performance tuning exercise of OGSA-DAI. OGSA-DAI offers significant functionality over direct connection database technologies such as JDBC, for example data transformation, compression and delivery. However, it is vital that when it comes to standard database access scenarios that *are* supported by existing direct access technologies that OGSA-DAI should be able to compete in terms of performance. The team's experiences raise a number of issues:

- Graphical profilers are useful for identifying potential performance suspects while Log4J statements can support the collection of repeated data for formal analysis.
- Performance hits can occur in unsuspected places, for example the `Text.appendData` operation. Third-party code should be subject to performance testing before inclusion in performance-critical software.
- When relying upon software from multiple third-parties, unanticipated dependencies and performance-related behaviours may arise, often from the loading of shared classes.
- Both OGSA-DAI and GT3.0.2 security overheads were far less heavy than expected indicating that secure client-service communications are realisable without a significant degradation of performance – any performance degradation is small compared to the cost of executing application-specific functionalities.

Directions for future investigation could include:

- Performing an analysis of XML:DB-related activities e.g. XPath and XUpdate.
- Profiling performance using very large data sets, especially retrieval of such data sets.
- Investigating why cleaning up after a query increases its duration gradually in response to an increase in the number of rows in a query result.
- Reducing where possible the overheads within OGSA-DAI of constructing, processing and transporting WebRowSet representations of query results.
- Investigating the inter-dependence of Xerces and JDBC classes.

Acknowledgements

This work is supported by the UK e-Science Grid Core Programme, whose support we are pleased to acknowledge. We also gratefully acknowledge the input of our past and present partners and contributors to the OGSA-DAI project including: IBM UK, University of Manchester, University of Newcastle, Oracle UK.

References

[1] OGSA-DAI Project. See http://www.ogsadai.org.uk.
[2] DQP Project. See http://www.ogsadai.org.uk/dqp.
[3] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Snelling, D., Vanderbilt, P. *Open Grid Services Infrastructure version 1.0 (Draft 33).* Open Grid Services Infrastructure WG, Global Grid Forum, June 27th 2003. See http://www.gridforum.org/ogsi-wg/.
[4] Deelman, E., Singh, G., Atkinson, M.P., Chervenek, A., Chue Hong, N.P., Kesselman, C., Patil, S., Pearlman, L., Sui, M-H., *Grid-Based Metadata Services*. Submitted to 16th International Conference on Scientific and Statistical Database Management, June 2004.
[5] Kodeboyina, D., Plale, B., Experiences *with OGSA-DAI: Portlet Access and Benchmark.* Designing and Building Grid Services Workshop, October 8th 2003.
[6] MySQL Database Driver. See http://dev.mysql.com/downloads/connector/j/2.0.html.
[7] Apache Log4J. See http://jakarta.apache.org/log4j/.
[8] Borland Optimizeit. See http://www.borland.com/.
[9] EJ-Enterprises JProfiler. See http://www.ej-technologies.com/products/jprofiler/overview.html/.
[10] JSR 114: JDBC Rowset Implementations. See http://www.jcp.org/en/jsr/detail?id=114/.
[11] W3C Document Object Model. See http://www.w3.org/TR/DOM-Level-2-Core/core.html/.

[12] Apache Xerces 2.4. See http://xml.apache.org/xerces2-j/releases.html/.