2015 IEEE/ACM International Conference on Software Engineering (ICSE 2015), Florence, Italy, 18-23 May 2015 © IEEE 2015

Performance Analysis using Subsuming Methods: An Industrial Case Study

David Maplesden*, Karl von Randow†, Ewan Tempero*, John Hosking* and John Grundy‡
*The University of Auckland, New Zealand.
dmap001@aucklanduni.ac.nz, e.tempero@auckland.ac.nz, j.hosking@auckland.ac.nz
†Cactuslab, New Zealand — karl@cactuslab.com
‡Swinburne University of Technology, Australia — jgrundy@swin.edu.au.

***Abstract*—Large-scale object-oriented applications consist of tens of thousands of methods and exhibit highly complex runtime behaviour that is difficult to analyse for performance. Typical performance analysis approaches that aggregate performance measures in a method-centric manner result in thinly distributed costs and few easily identifiable optimisation opportunities. Subsuming methods analysis is a new approach that aggregates performance costs across repeated patterns of method calls that occur in the application's runtime behaviour. This allows automatic identification of patterns that are expensive and represent practical optimisation opportunities. To evaluate the practicality of this analysis with a real world large-scale object-oriented application we completed a case study with the developers of `letterboxd.com` — a social network website for movie goers. Using the results of the analysis we were able to rapidly implement changes resulting in a 54.8% reduction in CPU load and an 49.6% reduction in average response time.**

***Index Terms*—Subsuming methods, Runtime bloat, Performance analysis, Object oriented software**

I. Introduction

Performance is a key attribute for the modern, cloud-based web applications that are in common use today. Inefficient software increases deployment costs (due to increased hardware requirements) and impacts the quality of the user experience, which is vital for vendors trying to attract and retain users in competitive markets.

Unfortunately many large scale applications suffer from poor performance [1]. Modern object-oriented applications are engineered for flexibility and maintainability, to improve developer productivity. They are built from existing frameworks with an emphasis on re-use. This results in software where the handling of each request passes through many layers and can require hundreds or thousands of method calls to complete [2], a problem known as *runtime bloat* [3]. This highly complex runtime behaviour is often difficult to analyse for performance.

The majority of performance analysis approaches used in practice collect and aggregate performance measures in a method-centric manner. But large-scale object-oriented applications may consist of tens of thousands of methods, resulting in thinly distributed costs and few easily identifiable optimisation opportunities.

Several sophisticated performance analysis approaches have been developed to tackle runtime bloat [2], [4]–[14]. However many are impractical for online production systems as they impose significant runtime overheads or unpalatable customisations.

In response to these challenges we have developed an efficient offline performance analysis called *Subsuming Methods Analysis* (SMA) [15]. The key idea behind SMA is that there are repeated patterns of methods calls in an application's runtime behaviour induced by the frameworks and design idioms used in the software. These repeated patterns represent identifiable bundles of code, spread across multiple methods, that consume significant resources, i.e. realistic optimisation opportunities. SMA operates on performance profiles that can be obtained using low overhead sampling based data collection and the offline analysis takes only a few minutes to complete, making it a practical approach to use in online production systems. In earlier work we conducted an evaluation of SMA based upon experiments and case studies using the DaCapo benchmark suite [16] and obtained promising results [15]. However the majority of the applications in this benchmark are relatively small utility applications.

To evaluate SMA with a real world large-scale object-oriented application we completed a case study with the developers[1] of Letterboxd [17] — a social network website for movie goers. Users rate and review films, keep a film diary and find other films to watch by following other members' recommendations. Letterboxd experiences significant load volumes, currently receiving over 3.6 million HTTP requests per day.

SMA helped us to readily identify a number of significant optimisation opportunities using only profiles gathered by low overhead sampling of the running application. These included unnecessary Hibernate session flushes, a problem with the distributed caching mechanism, and a third party library creating superfluous exceptions. We were able to implement simple changes for these areas resulting in a 54.8% reduction in CPU load on the application servers and an 49.6% reduction in average response time.

SMA was able to be applied rapidly in a real-world setting. It took less than a week of development effort to identify and implement the optimisations discussed in this paper. The entire case study was completed in less than four weeks.

The major contributions of this paper are:

[1]Letterboxd is developed by Cactuslab, New Zealand

1) Demonstrating how SMA facilitates the identification of optimisation opportunities in a real-world large-scale object-oriented application
2) Demonstrating that SMA can be effectively applied to a real-world system by using sampling-based profiling that is safe and has low overhead
3) An evaluation of the risks, benefits and lessons learned from applying SMA to a real-world application

The remainder of this paper is structured as follows: Section II contains technical information on SMA and the case study industrial environment. Section III outlines our methodology. Section IV presents our empirical results. Section V discusses the results from the case study, our experience using SMA and the lessons learned. Section VI summarises related work and we conclude in section VII.

II. Background

A. Subsuming Methods Analysis

The description we give here is summarised from a previous paper [15]. Traditional profiling tools typically record measurements of execution cost per method call, both inclusive and exclusive of the cost of any methods they call. The cost measurements are usually captured with calling context information and are aggregated in a data structure known as a calling context tree [18]. A calling context tree (CCT) records all distinct calling contexts of a program. Each node in the tree represents a method call and has a child node for each unique method that it invokes. Therefore the path from a node to the root of the tree represents a distinct calling context and the measurements stored at each node are the recorded costs for that calling context.

Our aim is to identify repeated patterns of method calls within the CCT over which we can aggregate performance costs. The intuition behind idea this is two-fold:

1) Consolidating costs within the CCT reduces the size and complexity of the tree, making it easier to interpret
2) A pattern of methods calls represents a greater range of behaviour than a single method and is therefore more likely to contain optimisation opportunities

Our approach to consolidating costs within the CCT is to identify the methods that are the most important from a performance standpoint and use these as the consolidation points within the tree, we call these the *subsuming* methods. All other methods we call *subsumed* methods and we attribute their costs to their parent node in the CCT. We aggregate the costs of subsumed methods recursively upwards until we reach a subsuming method. We call this cost the `induced cost` for the node as it represents the cost induced by the subsuming method at that node. Figure 1 illustrates the subsuming concept in an example CCT.

Each subsuming method is the root of a subsumed subtree and represents a pattern consisting of itself and the subsumed methods it calls, either directly or transitively through other subsumed methods. The induced cost of a subsuming method is the sum of the induced costs for all the CCT nodes

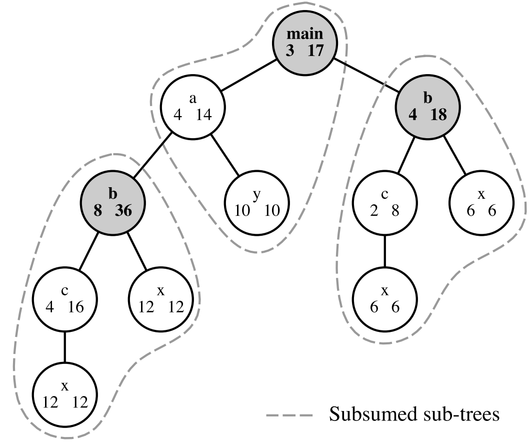

Fig. 1. Subsumed subtrees within a CCT

We have chosen *main* and *b* as subsuming methods. The numbers shown for each node are example exclusive and induced costs at that node.

associated with that method. As the exclusive cost of each node in the tree is consolidated into exactly one subsuming node the sum of the induced costs of all the subsuming methods equals the total cost of the CCT. Effectively the subsuming methods form a new way of partitioning the CCT at a coarser granularity than the natural method level partitioning.

In this case study we have considered two characteristics of methods to define a set of subsuming methods that give us interesting and useful results:

Methods that induce a limited range of runtime behaviour: These are uninteresting from a performance standpoint as they tend to be code too simple to optimise. We use the height of the method as a measure of the range of behaviour it induces. The height of a method is the maximum height of any sub-tree within the CCT with an instance of the method as its root. The trivial case is a leaf method that never calls any other method and therefore has a height of zero.

Methods called in a constrained manner: Specifically each call to the method can be traced back to a dominating method, a calling method responsible for its invocation.

We have used the distance from a method to its nearest dominating method as a measure of this characteristic (denoted dmd). The trivial case is when a method is only ever called from a single call site. That call site is the dominating method and dmd is 1.

Using the height and dmd attributes we can define a condition for identifying subsuming methods by specifying a bound on the minimum height and dmd a method must have to be considered subsuming. The results reported for this case study use a value of 10 as the bound, so only methods with a height and dmd greater than 10 are considered subsuming.

We experimented with other values for the bound and our results did not greatly change unless we set it very low (< 4) or very high (> 20). In the future we plan to investigate the impact of changing this bound with a variety of applications.

TABLE I
LETTERBOXD DEPLOYMENT PLATFORM

	Application Servers	Database Servers
CPU	Dual Hex-core Intel Xeon E5-2620 @ 2.00Ghz	
Memory	48GB RAM	64GB RAM
OS	Ubuntu 12.04 LTS	
Software	Apache 2.2 Java SE Runtime Environment 1.7.0_67 Java HotSpot 64-Bit Server VM Tomcat 7.0.54	PostgreSQL 9.3

B. The Industrial Setting

The application used in our industrial case study is Letterboxd, a social networking website for movie goers. It is a web application implemented in Java using popular and mature open frameworks in common industry use. It is deployed in the Tomcat servlet engine and uses JSP templates and the Hibernate ORM framework in front of a PostgreSQL database to implement the core server functionality. Extensive Javascript and CSS implement modern, sophisticated user interactions in the client browser. It is a complex application; extensive functionality manages many intertwined views and interactions. The Letterboxd codebase consists of over two thousand Java files containing several hundred thousand lines of code and over five hundred JSP templates. At runtime, including the open-source frameworks, many thousands of methods contribute to the server behaviour. Profiling runs sampled over 19 thousand unique methods. It is a mature application previously optimised for performance.

The Letterboxd production deployment is a typical high-availability multi-tiered architecture consisting of 4 physical servers, 2 application servers (app1 and app2) and 2 database servers (db1 and db2), with the details shown in Table I. Apache 2.2 is used on one of the Application servers to receive all incoming HTTP requests and load balance them across two Tomcat instances (one on each application server). All database requests initiated from the application servers are load balanced across the PostgreSQL instances on the two database servers using pgpool-II.

Letterboxd is a large scale web application which experiences significant load volumes. There are:

- 125,700 registered members
- 32,100 members who have been active in the last 30 days
- 12.07 million page views a month (per Google Analytics)
- 3.6 million HTTP requests per day (per our monitoring)

The load (shown in Figure 2) fluctuates in a regular daily and weekly pattern, with the peak times being during daylight hours in North America and weekends being busier than week days. This activity has created a large database of user generated content, consisting of:

- 55 GB of data
- 27.7 million films logged
- 1.58 million film reviews written

The database grows by more than 2GB each month.

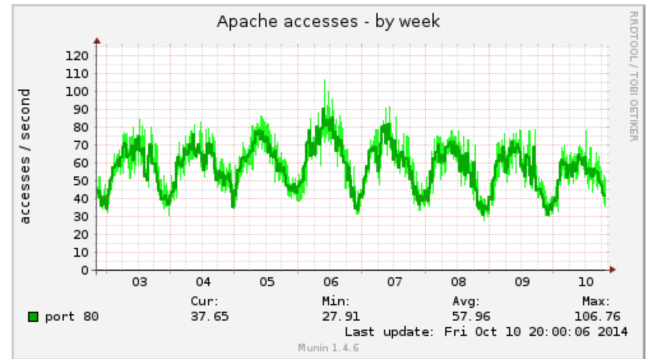

Fig. 2. HTTP requests received over a typical week

III. METHODOLOGY

Our case study aimed to investigate whether subsuming methods analysis was practical and useful to apply to a real-world large-scale industrial application, and to gain experience with the approach. To that end we applied an iterative methodology consisting of the following steps:

1) Establish performance baseline
2) Profile application
3) Analyse application performance
4) Implement improvements
5) Repeat baseline measurement

This approach allowed us to rapidly implement and measure several improvements in the limited time we had. We constrained the case study timeframe to four weeks to avoid impacting the regular development and release schedule for Letterboxd and to ensure we were testing performance changes over monitored periods that were only a few days apart.

A. Establish performance baseline

Many different metrics can be used to measure the performance for a web application. We chose to focus on CPU load and HTTP request response time. These generally correlate well with the usual performance optimisation goals of reducing the use of hardware resources and improving user experience. We performed our performance measurements over a 12 hour window from 2am to 2pm NZDT (NZ Daylight Time). This is when Letterboxd experiences the highest request volumes (2am to 2pm NZDT equates to 9am to 9pm EDT in the US). We also wanted to avoid the off-peak times when several scheduled tasks are run in the background on the Letterboxd servers so we would only be measuring request workload.

To measure the CPU load we used the command-line utility `top` to record samples on each application server every 3 seconds for the duration of the 12 hour run. This gave us several performance measures but in particular it reported the %CPU being used by the main Letterboxd Java process. When run in sampling mode the %CPU reported by `top` is the percentage of CPU being used since the last sample was taken, so the sampling rate does not effect the coverage of the CPU monitoring but does effect results granularity. Note that %CPU is reported relative to a single core i.e. 100% CPU means one

TABLE II
THREAD STATE CLASSIFICATIONS

State	Description
BLOCKED	Thread blocked waiting for a monitor lock
WAITING	Thread in a waiting state due to a call to Object.wait(), Thread.sleep() or Thread.join()
IO_WAIT	A runnable thread performing IO communication
RUNNABLE	All other runnable threads

entire CPU core is being used. The theoretical maximum CPU for the Letterboxd servers is 1200%.

To measure the HTTP request response times we enabled Apache HTTP access logging, which we configured to record the request path, the time the request was completed, and time time taken for every HTTP request received by Letterboxd. These logs also allowed us to verify that request rates were similar across different monitoring runs.

B. Profiling approach

The scale of the deployment means that Letterboxd faces challenges common to many real-world applications — it is expensive to duplicate the production environment to create a dedicated test environment, and it is difficult to simulate the load from the production environment to create realistic test scenarios. Even if a realistic test environment was created it would be a significant on-going effort and expense to maintain it. Therefore we used an approach that allowed us to gather profiling data directly from the production environment. This greatly improves the practicality of the approach and allowed us to gather data quickly and iterate rapidly. However it did limit the profiling approaches we could use. Our approach needed to be unobtrusive — so as to have a minimal impact, and safe — so as not to risk the stability of the system.

We chose to build a statistical profile of the application's activity by sampling the JVM's thread activity and processing this offline into a calling context profile. Although we developed our approach independently our profiler is a simpler version of Altman et al's [19]. We used the built-in support provided by the JVM to trigger a full thread dump once every second over a period of several hours. Each thread dump contains a complete snapshot of all the activity in the JVM, represented by a complete call stack for each live thread. These samples are then aggregated to build a statistical profile of where the JVM is spending its time.

We classify each individual thread as to the state it is in, such as runnable, blocked, waiting for IO etc (referred to as the *Wait State* by Altman et al), so as to be able to build a profile of the activity for a particular state. This is useful as typically a high proportion of threads are frequently idle, waiting for some event e.g. the next HTTP request to arrive or the next scheduled task to begin. These idle threads are typically uninteresting from a performance analysis point of view, though some are helpful to understand certain performance characteristics e.g. threads waiting for a database query. To classify threads we used the states defined in Table II and we generally classified each thread according to the state it was reported with by the JVM. The only distinction we made was to distinguish runnable threads performing IO communication as being in an IO_WAIT state.

We repeated this profiling step after each implemented improvement to evaluate the impact the improvement had on the application's behaviour.

C. Analyse application performance

We first analysed the baseline profile using traditional CCT tree and hot method views and attempted to understand the application's runtime behaviour and identify optimisation opportunities. We subsequently applied subsuming methods analysis to find the top induced cost methods and top inclusive cost subsumed methods. Each of these we investigated to identify optimisation opportunities. Later iterations of the analysis phase, for the profiles taken after each implemented improvement, focussed on verifying the improvement had caused the expected change in behaviour and that no unexpected changes had occurred.

D. Implement improvements

From our performance analysis we identified optimisation opportunities and implemented improvements for these. One of the interesting aspects of the case study was that not all of these improvements involved changes to the application's source code. Some were addressed via upgrades to 3rd party libraries or configuration changes in the application. In all cases the necessary changes were suggested by the primary author of this paper, who is an experienced Java software engineer but is not a developer of the Letterboxd application and was not familiar with the application before the case study began. The changes were then implemented in collaboration with the Letterboxd development team.

E. Repeat baseline measurements

After each implemented improvement we repeated the performance measurements over the same 12 hour period, 2am to 2pm NZDT. This allowed us to compare the CPU loads and response times to evaluate the effectiveness of the improvements. Not only did we track the response times of all HTTP requests but we tracked the response times for particular types of requests that the profiling had indicated should be the most impacted by the performance improvements made.

IV. RESULTS

We present some selected results from our performance monitoring and performance analysis. We have chosen to present the results of the traditional performance analysis in some detail, to demonstrate the challenges presented by the traditional performance views.

A. Performance Baseline

Our performance baseline results were gathered on a normal (non-holiday) weekday in early October 2014. Table III summarises the recorded response time results, grouped by the request path. Wildcards in request paths represent locations filled by parameters such as the logged in user's username or the

TABLE III
BASELINE RESPONSE TIMES

Request Path	Count	Response Time (ms)			
		Mean	Median	90th	99th
/*/film/*	165,935	318	280	540	920
/*/list/*	49,595	838	490	2050	3570
/film/*/	48,536	749	770	1120	1570
/*/films/*	32,229	608	430	1380	1970
/*/rss/	24,546	483	370	1090	1620
/search/*	8,213	1924	1790	3340	4970
All	2,284,931	189	20	520	1790
Significant*	1,045,885	389	210	940	2530

*Significant requests are all requests except those to /ajax/letterboxd-metadata/

name of a film or list. One feature of these results specific to the Letterboxd application is the high proportion (over 54%) of all requests for `/ajax/letterboxd-metadata/`. These are trivial requests serviced very efficiently by the application and are background requests handled asynchronously from the client so do not impact the user experience. As there are so many of them, they skew the overall results, particularly the median and percentile values. The statistics in the row labeled Significant in Table III exclude these trivial requests.

Figure 3 shows the average request arrival rate (bottom), response time(top) and CPU load for app1 and app2 (middle two curves) over the data gathering period. The request arrival rate and response time are calculated using only the Significant requests (therefore excluding the requests to `/ajax/letterboxd-metadata/`) and using rolling 5 minute averages i.e. the average number of requests received per minute and the average response time per request for the prior 5 minute period. To improve readability no scale is shown on the graph for the request arrival rate. It is plotted with a linear scale starting from 0, demonstrating the very regular nature of the incoming request load. The CPU load is shown for each of the two front-end application servers, app1 and app2. Over the 12 hour period Letterboxd averaged 1453 significant requests per minute, 3174 total requests per minute, 389ms response time for significant requests and CPU load of 239.7% on app1 and 219.5% on app2.

B. Traditional CCT Performance Analysis

For our baseline profile we gathered thread dump samples at 1 second intervals over a 12 hour period of normal activity and processed this offline into a calling context tree profile. This resulted in a large CCT containing 480,785 nodes, 11,732 unique methods and a maximum depth of 273.

We first analysed the generated CCT using the normal views currently in widespread use in industry. The CCT represents a statistical profile of where all the threads in the JVM spend their time. As is commonly the case with these types of CCTs the overall results are overwhelming skewed towards those locations in the code where idle threads wait for incoming requests to process. To get meaningful results we used the standard practice of limiting the analysis to a subset of the total profile, we did this in two ways:

Analyse only the recorded runnable time — This excludes all thread activity in BLOCKED, WAITING or IO_WAIT states, effectively excluding all currently idle threads. This generally correlates well with JVM activity contributing to CPU load, thus is suitable for our aim of reducing CPU load.

Analyse only the subtree rooted at Tomcat's `CoyoteAdapter.service()` method — This method is from the Apache Tomcat servlet container processing loop and encapsulates all activity that is directly processing an incoming HTTP request. We refer to this as the profiled *request service time*. It generally correlates well with the elapsed time experienced by users of the web application. It is a non-trivial subset of the full CCT, containing 476,296 nodes (over 99% of the full CCT) and accounts for over 90% of all runnable time. Importantly though it also includes significant amounts of non-runnable activity, most notably IO_WAIT time from threads communicating with the database. In fact over 65% of the activity in this subtree is database related IO_WAIT time.

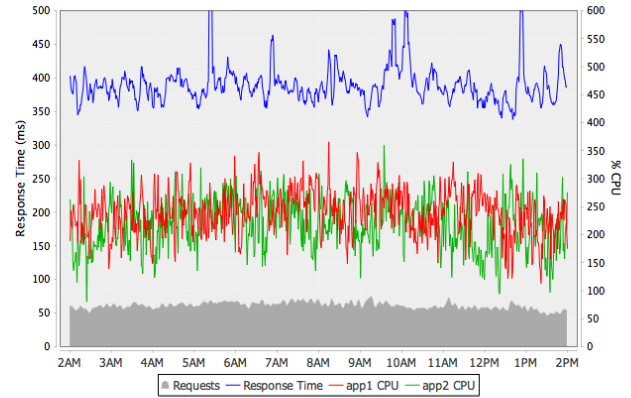

Fig. 3. Baseline — Request Rate, Response Time and CPU

1) Runnable Time — Exclusive Cost: The top runnable time exclusive cost methods (colloquially known as the *hot* methods) are shown in Table IV. This list exhibits all the

TABLE IV
TOP 20 METHODS — RUNNABLE TIME, EXCLUSIVE COST

Method	% Cost
sun.reflect.DelegatingMethodAccessorImpl.invoke()	11.470
java.lang.System.identityHashCode()	5.939
java.lang.Throwable.fillInStackTrace()	5.369
org.hibernate.event.internal.AbstractVisitor.processValue()	4.251
java.lang.String.intern()	4.147
java.lang.Object.hashCode()	3.044
org.hibernate.type.AbstractStandardBasicType.isEqual()	2.749
java.lang.UNIXProcess.waitForProcessExit()	2.569
org.hibernate.type.TypeHelper.findDirty()	2.517
java.lang.UNIXProcess.forkAndExec()	2.470
sun.misc.Unsafe.unpark()	1.932
org.hibernate.internal.util.compare.EqualsHelper.equals()	1.715
sun.misc.Unsafe.park()	1.500
org.hibernate.engine.internal.Collections.processReachableCollection()	1.478
org.apache.log4j.Category.getEffectiveLevel()	1.255
java.lang.Long.toString()	1.193
(redacted for security reasons)	1.136
java.lang.Object.<init>()	1.125
java.util.HashMap.hash()	1.065
org.postgresql.jdbc2.AbstractJdbc2Statement.killTimer()	1.052

TABLE V
TOP 20 METHODS — RUNNABLE TIME, INCLUSIVE COST

Method	% Cost
java.lang.Thread.run()	99.404
java.util.concurrent.ThreadPoolExecutor.runWorker()	97.732
java.util.concurrent.ThreadPoolExecutor$Worker.run()	97.732
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run()	91.520
org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.run()	91.515
org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process()	91.511
org.apache.coyote.ajp.AjpProcessor.process()	91.511
org.apache.catalina.connector.CoyoteAdapter.service()	91.437
org.apache.catalina.core.StandardEngineValve.invoke()	91.055
org.apache.catalina.core.StandardHostValve.invoke()	91.046
org.apache.catalina.valves.ErrorReportValve.invoke()	91.046
org.apache.catalina.authenticator.AuthenticatorBase.invoke()	90.519
org.apache.catalina.core.StandardContextValve.invoke()	90.474
org.apache.catalina.core.StandardWrapperValve.invoke()	90.465
org.apache.catalina.core.ApplicationFilterChain.doFilter()	90.392
org.apache.catalina.core.ApplicationFilterChain.internalDoFilter()	90.392
com.xk72.webparts.RequestCharacterEncodingFilter.doFilter()	90.372
com.xk72.webparts.multipart.MultipartHttpFilter.doFilter()	90.288
com.xk72.webparts.csrf.CSRFFilter.doFilter()	90.266
com.cactuslab.supermodel.web.SupermodelFilter.doFilter()	90.184

TABLE VI
TOP 20 METHODS — REQUEST SERVICE TIME, EXCLUSIVE COST

Method	% Cost
java.net.SocketInputStream.socketRead0()	62.128
sun.reflect.DelegatingMethodAccessorImpl.invoke()	4.123
java.lang.System.identityHashCode()	2.126
java.lang.Throwable.fillInStackTrace()	1.930
sun.nio.ch.EPollArrayWrapper.epollWait()	1.774
org.hibernate.event.internal.AbstractVisitor.processValue()	1.528
java.lang.String.intern()	1.491
java.net.SocketOutputStream.socketWrite0()	1.098
java.lang.Object.hashCode()	1.092
org.hibernate.type.AbstractStandardBasicType.isEqual()	0.988
org.hibernate.type.TypeHelper.findDirty()	0.904
java.lang.UNIXProcess.forkAndExec()	0.888
java.io.FileInputStream.readBytes()	0.651
org.hibernate.internal.util.compare.EqualsHelper.equals()	0.616
org.hibernate.engine.internal.Collections.processReachableCollection()	0.531
org.apache.log4j.Category.getEffectiveLevel()	0.437
java.lang.Long.toString()	0.426
(redacted for security reasons)	0.408
java.lang.Object.<init>()	0.404
java.lang.Object.wait()	0.383

classic challenges facing engineers attempting to optimise a large object-oriented application. All of the methods listed are either from the JVM implementation or one of the 3rd party frameworks being utilised by the application. Not a single method from the application under test is on the list. The top method from the Letterboxd code base is 65th on the list with an exclusive cost of only 0.145%. By definition the exclusive cost of the method is the time spent in that one method, meaning the potential for optimising that time is limited to the code in the method. Generally the methods are either trivial or highly optimised already. We can see that Java reflection is being frequently used, which is normal for a Java web application leveraging both Hibernate and JSP templates. There is a reasonable amount of Hibernate activity, again this is expected for a data-driven application. More interesting is the high placement of *java.lang.Throwable.fillInStackTrace*, perhaps indicating an unusually high number of exceptions being thrown. If we can track down where they are being thrown they might be able to be optimised. Even so, it still only accounts for just over 5% of the runnable activity.

2) Runnable time — Inclusive Cost: The top runnable time inclusive cost methods are shown in Table V. These are typical of a Java web application running in the Tomcat servlet engine. All of the methods listed are those that make up the main processing path for incoming HTTP requests.

3) Request Service Time — Exclusive Cost: The top request service time exclusive cost method are shown in Table VI. These are very similar to the runnable time hot methods except that they also include several IO related methods. The most notable difference is the `socketRead` method and its large 62% exclusive cost. This indicates a large proportion of our time is spent waiting for a response from a remote server, almost certainly the database. This is again unsurprising but worth investigating the calling methods to see where the communication is being initiated from.

4) Request Service Time — Inclusive Cost: The top request service time inclusive cost methods are identical to the top runnable time inclusive cost methods.

C. Subsuming Methods Analysis

The analysis identified 345 subsuming methods (from the original 11732 unique methods). The top runnable time induced cost subsuming methods are shown in Table VII. The notable difference between these methods and the top runnable time hot methods is each of these represents a non-trivial subtree of the CCT called from multiple locations. They naturally represent a wider range of behaviour than an individual hot method and therefore contain more optimisation opportunities. We were able to implement a number of optimisations by investigating these methods in more detail.

TABLE VII
TOP 20 SUBSUMING METHODS — RUNNABLE TIME, INDUCED COST

	Method	% Cost
1	org.hib*.AbstractFlushingEventListener.flushEverythingToExecutions()	28.522
2	java.lang.Thread.run()	5.853
3	org.apache.catalina.core.ApplicationFilterChain.doFilter()	5.546
4	org.hibernate.engine.internal.Cascade.cascade()	5.471
5	com.xk72.util.JexlExpressionEvaluator.evaluateStringWithExpressions()	4.343
6	com.cactuslab.supermodel.out.DefaultOutputFormat.format()	2.883
7	org.hibernate.engine.internal.StatefulPersistenceContext.addEntity()	2.846
8	org.apache.jsp.tag.webfilm_002dposter_tag.doTag()	2.706
9	org.hibernate.internal.SessionImpl.list()	2.473
10	org.hibernate.internal.SessionImpl.initializeCollection()	2.067
11	com.xk72.webparts.actions.RootActionContext.doAction()	2.003
12	org.jboss.marshalling.AbstractMarshaller.writeObject()	1.906
13	com.cactuslab.supermodel.beans.SBeanAccessor.get()	1.642
14	org.apache.tomcat.jdbc.pool.DisposableConnectionFacade.invoke()	1.552
15	com.cactuslab.supermodel.routing.AbstractRouteRouter.formatCurrent()	1.318
16	org.hibernate.engine.spi.CollectionEntry.postInitialize()	1.268
17	org.hibernate.internal.SessionImpl.flush()	1.217
18	org.hibernate.internal.SessionImpl.fireLoad()	1.124
19	com.xk72.webparts.user.DefaultAbstractUserHome.findUser()	1.084
20	com.cactuslab.supermodel.beans.SBeanManager.uid()	1.061

TABLE VIII
SUMMARY OF RESULTS

	All Requests (ms)					Significant Requests (ms)					CPU (%)		
	Count	Mean	Median	90th	99th	Count	Mean	Median	90th	99th	app1	app2	Norm.*
Baseline	2,284,931	189	20	520	1790	1,045,885	389	210	940	2530	239.7	219.5	219.5
Session Flush fix	2,267,151	160	20	460	1290	1,003,294	339	210	800	1650	172.3	167.0	169.1
Cache fix	2,254,251	117	10	330	1000	1,008,173	243	140	570	1380	121.9	125.2	122.5
DB Query fix	2,769,009	102	10	260	910	1,303,464	201	100	470	1260	147.1	151.1	114.4
Exceptions fix	2,269,720	105	20	270	900	1,130,374	196	90	440	1230	110.4	114.1	99.3
Overall Improvement		44.4%	0%	43.9%	49.7%		49.6%	57.1%	53.2%	51.4%			54.8%

* Normalised CPU is the average CPU across app1 and app2 per one million significant requests i.e. $CPU_{Norm} = \frac{(CPU_{app1}+CPU_{app2})\times 1{,}000{,}000}{2\times SignificantRequestCount}$

1) Unnecessary Hibernate Session Flushing: Method 1 encapsulates the Hibernate framework's implementation for flushing the current session, the process that synchronises the in-memory state of the current session with the database. Method 4 is also part of the session flushing mechanism. Together they accounted for 34% of the application's runnable time. This was anomalous given Letterboxd performs vastly more reads from the database than writes. Inspecting the behaviour of the associated subsumed sub-tree reveals most of the time was spent dirty-checking objects associated with the current session. Very little time was spent synchronising changes. This implies that often sessions were being flushed that did not contain any changed objects.

We looked at the calling methods and found there was a single code path inducing over 96% of the time spent flushing sessions. This was a location in the code where an explicit transaction commit was being performed after loading specific type of object. With Hibernate using its default session auto flush mode, the explicit transaction commit was triggering a flush of the session. Upon review of the code with the Letterboxd developers it was apparent that the explicit transaction commit at that location was unnecessary, and so we were able to trivially remove it.

This is an excellent example of the type of opportunity that subsuming methods analysis highlights that are otherwise difficult to find. The subsumed sub-tree for method 1 occurred in 514 different locations in the full CCT, all but one of these locations having a depth in the tree of at least 68. This means it would have been difficult to manually find enough of these locations in the CCT to notice the repeated pattern. The subsumed sub-tree was made up of 139 nodes with a height of 41, meaning it was a non-trivial pattern whose cost was dispersed over a large number of methods, many of them well removed from the sub-tree's root method. Hence it would also have been difficult to identify this root method from the traditional list of hot methods.

Our profile estimated that the removed code branch accounted for approximately 32% of the runnable time in the application and 13% of the total time spent servicing HTTP requests. When we repeated the baseline performance measurements after applying the change (Session Flush fix in Table VIII) we found the CPU usage had in fact dropped by 23% and the average response time for significant requests by 13%.

2) Ineffective Caches: Method 9 is the main query API for accessing persistent storage via the Hibernate framework. It was not surprising such a pivotal part of the application framework would show up in our analysis, however over 60% of the time `list()` was called from two locations that were protected by in-memory caches. This indicated that these caches were not working effectively. Additionally Method 12 is the core routine of the communication mechanism used by the distributed caching implementation. Checking the callers to this routine showed it was triggered to replicate cache invalidation messages, but was being called surprisingly often. Cached data in Letterboxd changes very rarely, so there should be no need for frequent cache invalidation.

It is normal distributed cache behaviour to replicate invalidations, it assumes any put into a local cache is a new value and needs to invalidate the cached values on other nodes. The problem was, with two nodes in the Letterboxd cluster, each was taking turns putting the same value from the underlying database into its local cache and invalidating the other node. For communication efficiency the caches were not propagating the cached values, just the cache key being invalidated, so the remote nodes had no way of detecting that the new value was in fact the same as their existing value. There is an alternative method on the distributed cache API that is designed to be used when populating the cache from a shared data store and does not invalidate remote caches. It was straight-forward to update the Letterboxd cache mechanism to use this more appropriate API.

This is an example of a problem that only manifested due to the clustered production environment and did not show up in normal development testing. In any single node deployment, there are no replicated invalidations, and using the cache's simple `put()` API worked perfectly well.

Subsuming methods analysis provided us with several clues that led us to rapidly identify that the caching mechanism was not working effectively. By contrast there was little chance of deducing this from the traditional CCT performance views. The key methods of interest were the relevant (cache-protected) callers of `SessionImpl.list()`, that is `AbstractDefaultSlugSpace.doFind()` and `BaseIMObjectHome.getAllByCode()`. There were over 2000 occurrences of these methods distributed

throughout the full CCT making it very unlikely a manual search of the tree would have revealed their true cost.

A new profile (taken after the session flush fix had been applied) estimated that approximately 23% of the runnable time in the application was shielded by these now correct cache mechanisms, and a further 3% of all runnable time had been spent in the distributed cache communication mechanism. The profile also estimated around 30% of the total time response time was now shielded by the caches. We again repeated our baseline performance measurements after fixing the caching mechanism (Cache fix in Table VIII) and found the CPU usage had dropped by a further 28% and the average response time for significant requests by a further 28%.

3) Improving Database Query Efficiency: The next improvement we applied consisted of two changes suggested by the breakdown of other activity in method 9. Nearly 17% of the total time spent in `list()` was spent retrieving result set meta data. We discovered this was induced by a database query monitoring system that was using the result set meta data to record basic statistics about the executed queries. The expensive call was an optional part of the JDBC API that had only been newly implemented in the particular version of the JDBC driver used by Letterboxd. A later version of the driver included a more efficient implementation that cached the meta data, so we simply upgraded the JDBC driver. Additionally around 10% of the runnable time in `list()` was being spent preparing statements. We configured Tomcat to use a prepared statement cache to reduce this time. Again these changes were ones that were quickly deduced by the information provided by the subsuming methods analysis and would have been more difficult to discover using the traditional views of the CCT profile.

Another new profile taken after the cache fixes estimated that a further 2.1% of runnable time and 15% of total response time should be removed by these improvements. When we repeated the baseline performance measurements (DB Query fix in Table VIII) we found the CPU usage had dropped by 6.6% and the average response time for significant requests by 17%. We believe the improvements were better than we had estimated because the reused prepared statements avoided other one-off query initialisation costs outside of the direct call to `prepareStatement`.

4) Superfluous Exceptions: The final improvement we implemented was the elimination of superfluous exceptions from the JEXL and Apache `commons-beanutils` libraries.

Method 5 is the key entry point into the JEXL library, which is used by Letterboxd to evaluate attributes from configuration templates at runtime. Inspecting the behaviour of the subsumed sub-tree shows much of the induced cost of this method was caused by the creation of exceptions. In the JEXL library, the interpreter can be run in either strict or non-strict mode. Some evaluation conditions that cause exceptions in strict mode simply return `null` in non-strict mode. However the JEXL interpreter was always constructing the exception and then deciding at a later point in the code whether to throw the exception or not. It was a simple change to avoid the construction of the exceptions in non-strict mode.

Method 13 is a utility method used by Letterboxd to leverage the Apache `commons-beanutils` library to retrieve properties of bean-like objects via reflection. A large proportion of its subsumed activity was spent constructing exceptions. Internally the library caches the discovered getter method used for each bean property to avoid the cost of introspecting the class on each call. However when a bean property had no suitable getter method this fact was not cached, so calls to the library for unrecognised bean properties resulted in an expensive class introspection and the raising of an exception on every call. It was another simple change to cache the fact that a particular bean property had no suitable getter method.

These were optimisations that were suggested by the traditional views of the CCT profile, due to the fact that excessive exceptions were apparent in the methods in Table IV. However it would have required navigating the calling hierarchy of `fillInStackTrace()` to discover the contributing locations. By contrast these locations were readily apparent in our list of top subsuming methods.

A final profile taken after the database query improvements estimated that 12% of runnable time and 3.5% of total response time should be saved by these improvements. The final baseline performance measurements (Exceptions fix in Table VIII) showed the CPU usage had dropped by 13% and the average response time for significant requests by 2.5%.

D. Other Opportunities

We also discovered several other optimisation opportunities requiring more substantial changes that have been added to the future development plan for Letterboxd.

Method 6 is a utility method used by Letterboxd to apply an HTML tidy function to an HTML snippet to be injected into a page. It is used to guarantee the formatting of HTML encoded comments supplied by users who provide film reviews or comments. Upon seeing the method listed as a performance bottleneck the Letterboxd developers immediately identified it as an area that could be improved.

Method 7 is the method used in the Hibernate framework to register a newly hydrated persistent object with the current session. The cost it is incurring could be reduced by reducing the number of individual persistent objects loaded by hibernate. Letterboxd currently has a very fine-grained persistent object model, in a number of places small child objects could instead be modelled using hibernate components which would significantly reduce the number of individual persistent objects.

Method 8 is the manifestation of a JSP tag used to render a film poster in a JSP template. This is a very frequently used tag that implements a wide range of behaviours specified by different attributes on the tag. However many of those features are very rarely used so it could be optimised by reducing the range of supported behaviours and implementing new separate JSP tags for when those features are wanted.

TABLE IX
FINAL RESPONSE TIMES — SELECTED PATHS

Request Path	Count	Response Time (ms)				Original	Decrease
		Mean	Median	90th	99th	Mean	
/*/film/*	236173	175	120	370	740	318	45.0%
/*/list/*	68548	182	140	330	1050	838	78.3%
/film/*/	51652	387	380	610	980	749	48.3%
/*/films/*	41935	194	120	450	640	608	68.1%
/*/rss/	28326	185	120	480	790	483	61.7%
/search/*	9424	657	650	980	1660	1924	65.9%

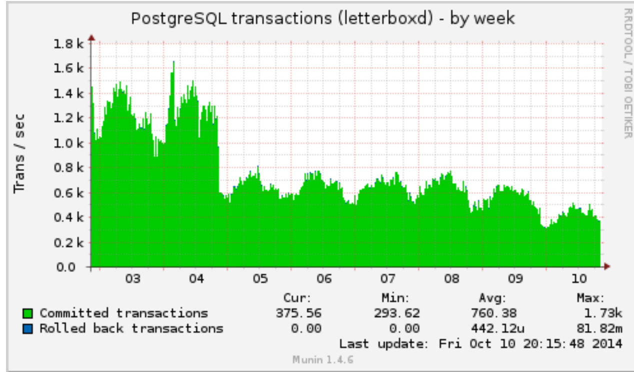

Fig. 4. Database transactions

E. Results Summary

The overall results are summarised in Table VIII. In total there has been a 49.6% drop in average response time for significant requests and a 54.8% drop in normalised CPU load. Table IX shows the response time statistics from our final experiment for several key request paths. All request paths we monitored showed an improvement in average response time of at least 35%, several of the most commonly used paths showed improvements of over 65%.

Besides our experimental measurements there are several other indicators of the performance improvements' impact. One particularly striking improvement is in the number of executed database transactions. Figure 4 shows the database transaction execution rate over the week we performed our testing. The first dramatic drop late on the 4th is when we implemented the distributed cache fix, avoiding the constant reloading of data from the database. The second drop late on the 9th was when we upgraded the JDBC driver, avoiding the result set meta data queries.

V. LESSONS LEARNED

Our aim in this case study has been to evaluate and gain experience with applying subsuming methods analysis to a real world large-scale object-oriented application. We have found subsuming methods analysis:

1) Practical to apply to a real-world large-scale application
2) Helped identify useful optimisation opportunities
3) Facilitated significant improvements in performance

It is difficult to quantify precisely how much subsuming methods analysis assisted the discovery of these optimisation opportunities. The feedback from the engineers at our industry partner was that it was very helpful. SMA provides novel and interesting views of the profiling data. The top induced cost subsuming methods helped illuminate new optimisation opportunities and the top inclusive cost subsuming methods (a view that space limitations precluded us discussing) helped in forming an overall understanding of costs. A key lesson is that subsuming methods analysis nicely complements existing performance views. It enhances rather than attempting to replace existing approaches and adds value when used in conjunction with the traditional hot method and CCT views.

Subsuming methods analysis can be used with any profiling approach that produces a CCT profile, making it applicable in wide range of performance investigations. After evaluating several different commercial and open source profilers we successfully adopted a safe and low overhead stack sampling profiling approach. We chose this approach because it was simple to apply and used only existing standard sampling mechanisms. The only requirement to apply the approach is that there is sufficient disk space to record the stack samples. Such a sampling approach in Java applications can give inaccurate results [20] because samples occur only at specific yield points placed by the JVM. It is our conjecture that the aggregation performed by SMA helps overcome this inaccuracy i.e. whilst the list of hot methods may not be accurate the list of subsuming methods is. The correlation we saw between the predicted and measured performance improvements we made gives some evidence to support this.

The application in our case study, Letterboxd, is a representative example of a modern, large-scale object-oriented web application. The deployment environment it operates in and the load it experiences are non-trivial. Therefore we are confident that the successful experience we had applying subsuming methods analysis to Letterboxd would extend to other real-world applications. Letterboxd is a Java application, but the general approach can be applied with other technology platforms. .NET applications in particular can be profiled and analysed in a similar manner.

Subsuming methods analysis has some limitations. It did not provide any assistance with how an identified bottleneck might be addressed — all the implemented optimisations had to be developed by software engineers. There were also some methods (methods 2 and 3 from Table VII) in the list of top induced cost methods that did not represent realistic optimisation opportunities, they were in effect false positives. Each of these are areas of potential future work. We are also interested in evaluating false negatives, perhaps by analysing an application with known or deliberately introduced bottlenecks.

VI. RELATED WORK

There is an extensive range of modern profiling tools and academic research into software performance that we are unable to cover in detail here.

The majority of Java profiling tools (e.g. [21]–[25]) produce a form of CCT profile and allow the investigation of the top inclusive time and exclusive time methods. They support a wide range of data collection options typically based around

dynamic byte code instrumentation or stack sampling. Most have modern interactive UIs allowing the browsing, searching and filtering of results. None support the automatic identification of expensive repeated patterns of method calls.

The most closely related work to ours in terms of its motivation is the existing research into runtime bloat [3]. Generally they have focussed on memory bloat (excessive memory use)(e.g. [4], [5]) or they have taken a data-flow centric approach [2], [6], looking for patterns of inefficiently created or used data structures, collections and objects [7]–[12]. This includes approaches specifically looking at the problem of object churn, that is the creation of many short-lived objects [13], [14]. In contrast, we investigate a control-flow centric approach, searching for repeated inefficient patterns of method calls. Additionally many of these approaches rely on detailed instrumentation and data collection that adds significant runtime overhead to the system. Subsuming methods analysis can be practically applied with low-overhead profiling.

Also related are approaches to aggregating calling context tree summarised performance data [26], [27]. These are based on grouping by package and class name, aggregating methods below a certain cost threshold into the calling method or the manual specification of aggregation groupings. None of these approaches attempt to *automatically* detect repeated patterns of method calls.

VII. Conclusion

Our experience in this case study has been that subsuming methods analysis is easy to apply to a real-world large-scale application and greatly assisted us in identifying new optimisation opportunities that led to real improvements. We were able to successfully combine subsuming methods analysis with a profiling approach that was safe and practical for use in an online production environment. This allowed us to rapidly identify and implement optimisations that led to a 49.6% drop in average response time and a 54.8% drop in CPU load for the application.

Acknowledgment

David Maplesden is supported by a University of Auckland Doctoral Scholarship and the John Butcher One-Tick Scholarship for Postgraduate Study in Computer Science.

References

[1] N. Mitchell, E. Schonberg, and G. Sevitsky, "Four Trends Leading to Java Runtime Bloat," *IEEE Software*, vol. 27, no. 1, pp. 56–63, 2010.

[2] N. Mitchell, G. Sevitsky, and H. Srinivasan, "Modeling Runtime Behavior in Framework-Based Applications," *Lecture Notes in Computer Science*, vol. 4067, pp. 429–451, 2006.

[3] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software Bloat Analysis: Finding , Removing , and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications," *Proc. of the FSE/SDP Workshop on the Future of Software Engineering Research - FoSER 2010*, pp. 421–425, 2010.

[4] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy, "Patterns of Memory Inefficiency," *Lecture Notes in Computer Science*, vol. 6813, pp. 383–407, 2011.

[5] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, Recycle to De-bloat Software," *Lecture Notes in Computer Science*, vol. 6813, pp. 408–432, 2011.

[6] N. Mitchell, G. Sevitsky, and H. Srinivasan, "The diary of a datum: an approach to modeling runtime complexity in framework-based applications," *Library-Centric Software Design - LCSD'05*, p. 85, 2005.

[7] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: Adaptive Selection of Collections," *Proc. of the 2009 ACM Conf. on Programming language design & implementation - PLDI '09*, pp. 408–418, 2009.

[8] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Finding low-utility data structures," *Proc. of the 2010 ACM Conf. on Programming language design & implementation - PLDI '10*, pp. 174–186, 2010.

[9] G. Xu and A. Rountev, "Detecting inefficiently-used containers to avoid bloat," *Proc. of the 2010 ACM Conf. on Programming language design & implementation - PLDI '10*, pp. 160–173, 2010.

[10] G. Xu, "Finding reusable data structures," *Proc. of the ACM Intl. Conf. on Object oriented programming systems languages & applications - OOPSLA '12*, p. 1017, 2012.

[11] D. Yan, G. Xu, and A. Rountev, "Uncovering performance problems in Java applications with reference propagation profiling," *Proc. of the 2012 34th Intl. Conf. on Software Engineering (ICSE)*, pp. 134–144, 2012.

[12] K. Nguyen and G. Xu, "Cachetor: detecting cacheable data to remove bloat," *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, pp. 268–278, 2013.

[13] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," *Proc. of the 2007 Intl. symposium on Software testing and analysis - ISSTA '07*, pp. 118–128, 2007.

[14] ——, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," *Proc. of the 16th ACM SIGSOFT Intl. Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, pp. 59–70, 2008.

[15] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy, "Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software," *Proc. of the 6th ACM/SPEC Intl. Conf. on Performance Engineering - ICPE '15*, p. to appear, 2015. [Online]. Available: https://www.cs.auckland.ac.nz/~dmap001/subsuming/sma.pdf

[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," *Proc. of the 21st annual ACM Conf. on Object-oriented programming systems, languages & applications - OOPSLA '06*, pp. 169–190, 2006.

[17] Letterboxd. [Online]. Available: http://letterboxd.com/

[18] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *Proc. of the ACM 1997 Conf. on Programming language design & implementation - PLDI '97*, pp. 85–96, 1997.

[19] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," *Proc. of the ACM Intl. Conf. on Object oriented programming systems languages & applications - OOPSLA '10*, pp. 739–753, 2010.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," *Proc. of the 2010 ACM Conf. on Programming language design & implementation - PLDI '10*, pp. 187–197, 2010.

[21] Yourkit. [Online]. Available: http://www.yourkit.com/

[22] Jprofiler. [Online]. Available: https://www.ej-technologies.com/products/jprofiler/overview.html

[23] Visualvm. [Online]. Available: https://visualvm.java.net/

[24] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini, "JP2: Call-site aware calling context profiling for the Java Virtual Machine," *Science of Computer Programming*, vol. 79, pp. 146–157, Jan. 2014.

[25] W. Binder, "Portable and accurate sampling profiling for Java," *Software: Practice and Experience*, vol. 36, no. 6, pp. 615–650, May 2006.

[26] S. Lin, F. Taʻıani, T. C. Ormerod, and L. J. Ball, "Towards Anomaly Comprehension: Using Structural Compression to Navigate Profiling Call-Trees," *Proc. of the 5th Intl. symposium on Software visualization - SOFTVIS '10*, pp. 103–112, 2010.

[27] K. Srinivas and H. Srinivasan, "Summarizing application performance from a components perspective," *Proc. of the 10th European software engineering Conf. held jointly with 13th ACM Intl. symposium on Foundations of software engineering - ESEC/FSE-13*, pp. 136–145, 2005.