Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software

David Maplesden
Dept. of Computer Science
The University of Auckland
dmap001@aucklanduni.ac.nz

Ewan Tempero
Dept. of Computer Science
The University of Auckland
e.tempero@auckland.ac.nz

John Hosking
Faculty of Science
The University of Auckland
j.hosking@auckland.ac.nz

John C. Grundy
School of Software and Electrical Engineering
Swinburne University of Technology
jgrundy@swin.edu.au

ABSTRACT

The majority of existing application profiling techniques aggregate and report performance costs by method or calling context. Modern large-scale object-oriented applications consist of thousands of methods with complex calling patterns. Consequently, when profiled, their performance costs tend to be thinly distributed across many thousands of locations with few easily identifiable optimisation opportunities.

However experienced performance engineers know that there are repeated patterns of method calls in the execution of an application that are induced by the libraries, design patterns and coding idioms used in the software. Automatically identifying and aggregating costs over these patterns of method calls allows us to identify opportunities to improve performance based on optimising these patterns.

We have developed an analysis technique that is able to identify the entry point methods, which we call subsuming methods, of such patterns. Our offline analysis runs over previously collected runtime performance data structured in a calling context tree, such as produced by a large number of existing commercial and open source profilers.

We have evaluated our approach on the DaCapo benchmark suite, showing that our analysis significantly reduces the size and complexity of the runtime performance data set, facilitating its comprehension and interpretation. We also demonstrate, with a collection of case studies, that our analysis identifies new optimisation opportunities that can lead to significant performance improvements (from 20% to over 50% improvement in our case studies).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.

ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2688040.

General Terms

Performance,Measurement

Keywords

dynamic performance analysis, profiling, subsuming methods, runtime bloat

1. INTRODUCTION

Performance is a crucial and often elusive attribute for modern applications. Trends such as mobile application development, where resources are limited, cloud deployment, where running costs are directly impacted by software efficiency, and online solutions, where low latency response times are key, means that software performance analysis is often a vital part of software engineering today. Enabled by the increase in hardware capacity over the last three decades, the size and complexity of software has increased to a similar or even greater extent [13]. This growing scale of the software under development means that analysing and improving the performance of these systems has become increasingly difficult.

Many of the challenges faced when analysing the performance of modern large-scale systems are exacerbated by specific characteristics of object-oriented software. Following object-oriented principles tends to lead to applications with inter-procedural rather than intra-procedural control flow and a great number of methods. Additionally many object-oriented methodologies focus on developer productivity, producing maintainable and flexible software, and promoting componentisation and reuse. As a result most applications are built from reusable generalised frameworks and leverage established design patterns, making them very layered and complex.

For example, a Java service-oriented application might implement SOAP web services using the apache Axis web service framework, running in the apache Tomcat servlet engine and using the Hibernate persistence framework to access a relational database. This approach means that the handling of even the simplest request in these framework-based applications goes through many layers and will require hundreds, maybe thousands, of method calls to complete [17]. This excessive activity to achieve seemingly simple results is a problem that has become known as *runtime bloat* [26].

It makes the applications difficult to profile and it has led many large scale object-oriented applications to suffer from chronic performance problems [15].

Traditional application profilers provide method-centric feedback on where an application is consuming resources, in particular memory allocation and execution time. Therefore profiling the extremely complex runtime behaviour exhibited by these large-scale object-oriented applications typically reports resource costs that are thinly distributed across a large number of methods, and results in a massive dataset that is very difficult to interpret. This also means that compile time and dynamic runtime optimisation approaches struggle to mitigate runtime bloat because of the lack of easily identifiable optimisation targets [26].

This is the challenge that we are interested in: how can we provide more useful feedback on the performance of large-scale object-oriented applications so that it can be improved? How can we help software engineers to reduce runtime bloat?

Our key insight in this paper is that there are repeated patterns of method calls induced by the libraries and design idioms used in the implementation of the software. These repeated patterns represent coherent units of aggregation for the resource costs recorded by traditional profilers. We show that identifying and aggregating performance costs over these repeated patterns will facilitate a better understanding of the performance characteristics of the software and highlight new, high potential candidates for optimisation that would led to useful performance improvements.

One of our key goals is to *automatically* identify the key repeated patterns of method calls. It is not practical to manually detect these patterns in a large-scale application with complex runtime behaviour spanning thousands of methods.

Our approach to identifying these repeated patterns of method calls is to identify the key methods, which we call the *subsuming methods*, in the application that represent the entry point or root of these repeated patterns. The other methods we call the *subsumed* methods and we attribute their execution costs to their parent subsuming method.

The main contributions of this paper are:

- We introduce the concept of automatically identifying repeated patterns of method calls in an application profile and using them to aggregate performance costs.
- We describe *subsuming methods*, a specific technique for identifying the entry points to repeated patterns of method calls.
- We define a novel metric, minimum dominating method distance, used to help identify subsuming methods.
- We demonstrate that our approach can be applied efficiently to large scale software.
- We empirically evaluate our approach over standard benchmarks to characterise the typical attributes of subsuming methods.
- We demonstrate the utility of subsuming methods in several case studies.

The remainder of this paper is structured as follows. Section 2 motivates our work and presents background information. Section 3 covers related work. Section 4 describes our approach. Section 5 presents an evaluation using the DaCapo benchmark suite and several case studies. Section 6 discusses the results of our evaluation and areas of future work. We conclude in Section 7.

2. MOTIVATION AND BACKGROUND

Traditional profiling tools typically record measurements of execution cost per method call, both inclusive and exclusive of the cost of any methods they call. Usually of the most interest are the top exclusive cost methods, known as the *hot* methods. The cost measurements are usually captured with calling context information, that is, the hierarchy of active methods calls leading to the current call, and are aggregated in a calling context tree.

A calling context tree (CCT) records all distinct calling contexts of a program. Each node in the tree has a method label representing the method call at that node and has a child node for each unique method invoked from that calling context [3]. Therefore the method labels on the path from a node to the root of the tree describe a distinct calling context. For multithreaded programs containing multiple execution entry points a virtual root node is used to collate the multiple traces into a single tree. A CCT is an intermediate representation in the spectrum of data structures that trade off size for richness of information. It retains more information than either a flat method level aggregation of data (also known as a vertex profile), which discards all calling context information, or a dynamic call graph (also known as an edge profile), which retains only a single level of calling context information. It is more compact than a full call tree, which captures every unique method invocation separately but grows in size linearly with the number of method calls and hence is unbounded over time.

For example the small program in Example 1 will generate the CCT shown in Figure 1. Table 1 shows the aggregated costs for each method assuming the per-call costs given in column one. The per-call costs are arbitrary numbers that we have assumed to complete the example.

The objective of performance profiling is to identify sections of source code that are performance critical, as these represent optimisation opportunities. The simplest approach to this is to identify the hot methods — methods with the highest exclusive cost. This is easily done by aggregating the performance data in the CCT for each method i.e. creating a flat method profile as in Table 1.

Example 1

```
 1:   void main() {                13:   void b() {
 2:       a();                     14:       c();
 3:       a();                     15:       c();
 4:       b();                     16:       x();
 5:   }                           17:       x();
 6:                                18:   }
 7:   void a() {                   19:
 8:       b();                     20:   void c() {
 9:       y();                     21:       x();
10:   }                           22:   }
11:                                23:
12:   void x() { ... }            24:   void y() { ... }
```

Table 1: Costs for Example 1

Method	Per-call	Invocations	Exclusive	Inclusive
main	3	1	3	71
a	2	2	4	50
b	4	3	12	54
c	1	6	6	24
x	3	12	36	36
y	5	2	10	10

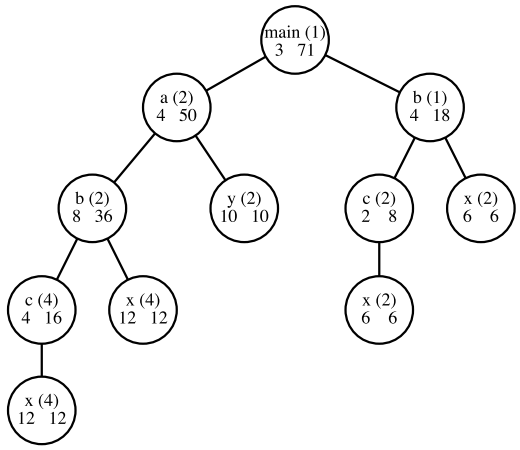

Figure 1: CCT for Example 1

The numbers shown for each node are the invocation count, exclusive cost, and inclusive cost for that node. Each node maintains the aggregated totals for a calling context.

Figure 2: Usages of formatDouble in fop

Unfortunately the list of hot methods often isn't useful because the identified methods are difficult to optimise or avoid. For example Table 2 shows the top ten hot methods from a case study (discussed in section 5.1.1) we completed on the `fop` benchmark from the DaCapo-9.12-bach benchmark suite [8]. The `fop` benchmark is a relatively small library used in many applications for applying XSL-FO stylesheets to XML documents to produce PDF or other formatted output. Only one of the top ten hot methods is actually a method in the `fop` codebase, all the others are support methods from the Java runtime library, methods that are heavily used and already well optimised. Also the method calls typically occur a large number of times within the CCT, each occurrence representing a different calling context in which the method was invoked. This makes it difficult to target the code that calls the method in order to avoid invoking the costly method. Finally, by the time we reach the fifth method we are considering methods using less than 2% of the total execution time, so even if we could remove this cost completely the benefit would be minor.

Apart from investigating hot methods, another common approach to identifying performance critical code is to perform a top-down search of the CCT, looking for a compact sub-tree that has a high total cost. The sub-tree should be compact so that it represents a constrained piece of dynamic behaviour that can be understood and optimised. If such a sub-tree even exists, finding it within a large CCT is difficult; nodes near the root of the CCT, which have high total costs, encompass very large sub-trees and nodes near the leaves of the CCT, which are compact, have low total costs.

Consequently traditional hot method analysis and top-down searches are often ineffective with a large CCT, and they can grow very large for object-oriented programs. The CCT generated by the relatively small `fop` benchmark contained over six hundred thousand nodes, real-world applications can create CCTs with tens of millions of nodes [10].

However there does exist within the `fop` benchmark a clear cut performance optimisation opportunity. A single method, `org.apache.xmlgraphics.ps.PSGenerator.formatDouble`, accounts for (by inclusive cost) over 26% of the total execution time. Figure 2 is a calling context ring chart [18, 1] visualising the full CCT with all the occurrences of this method and its sub-trees highlighted. The chart depicts the CCT as a series of concentric rings broken into segments, each segment representing a node in the CCT with the root node in the centre. Moving away from the middle each segment is divided into a new segment for each child node, so nodes deeper in the CCT are further away from the middle and leaf nodes in the CCT have no surrounding segments. The arc length covered by each segment is proportional to the inclusive cost of the associated calling context, so the more cost the sub-tree rooted at a node accounts for the longer the ring segment. Figure 2 illustrates the fact that `formatDouble` occurs in multiple locations deep within the CCT and in aggregate accounts for a significant amount of the total execution time.

The `formatDouble` method uses `java.text.NumberFormat` (Java's general purpose floating point number formatter) for producing a simple three decimal place format. It induces the same expensive pattern of method calls each time it is used, but that cost is distributed over a number of low-level string manipulation and floating decimal inspection methods. Once we have found this method it is easy to see it

Table 2: Top 10 hot methods in the fop benchmark

Method	Occ.	% Exc.	% Inc.
sun.misc.FloatingDecimal.dtoa	348	6.904	9.428
java.text.DigitList.set	374	5.266	6.166
java.text.DecimalFormat.subformat	374	3.123	5.614
org.apache.fop.fo.properties. PropertyMaker.findProperty	1501	2.471	11.675
sun.nio.cs.US_ASCII$Encoder.encode	568	1.795	1.796
sun.misc.FloatingDecimal.countBits	348	1.563	1.563
java.util.HashMap.hash	10663	1.512	3.534
java.lang.String.equals	4620	1.454	1.454
java.util.HashMap.getEntry	6081	1.348	4.950
java.lang.String.indexOf	3343	1.300	1.300

Occ. - is the number of occurrences of the method in the full CCT i.e. the number of distinct calling paths that lead to the method call

represents just the sort of opportunity we are looking for, a compact repeated pattern of method calls that accounts for a significant proportion of the overall cost, but finding it amongst the full CCT or deducing it from the list of low-level hot methods is difficult.

This is a classic example of the type of runtime bloat experienced by many large-scale object oriented applications. The use of a conveniently available and powerful generic library routine has a significant performance impact that is later difficult to detect amongst the mass of performance data produced when the application is profiled. We were able to refactor this method to use a much more specialised approach that drastically reduced its relative cost, improving the overall execution cost of the benchmark by 22%. Our aim is to help identify these types of opportunities.

3. RELATED WORK

There is a significant body of work into investigating software performance that we cannot adequately describe here due to space limitations. This includes extensive research into model-based performance prediction methods that are complementary to our empirical performance analysis approach and many papers on novel performance data collection (profiling) approaches that are applicable to object-oriented software (e.g. [4, 6, 7, 20]). Typically these data collection approaches are either striving for lower overheads, better accuracy or better contextual information (e.g. argument, loop or data centric profiling). By contrast our work is focussed on improving the analysis rather than the collection of performance data.

The most closely related work to ours in terms of its motivation is the existing research into runtime bloat [26]. Generally they have focussed on memory bloat (excessive memory use)(e.g. [9, 5]) or they have taken a data-flow centric approach [16, 17], looking for patterns of inefficiently created or used data structures, collections and objects [22, 25, 27, 24, 28, 19]. This includes approaches specifically looking at the problem of object churn, that is the creation of a large number of short-lived objects [11, 12]. In contrast, we investigate a control-flow centric approach, searching for repeated inefficient patterns of method calls.

Also related are approaches to aggregating calling context tree summarised performance data [14, 23]. 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.

4. SUBSUMING METHODS

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 in this way reduces the size and complexity of the tree, making it easier to interpret to discover performance bottlenecks.
2. The traditional unit of aggregation, individual methods, often identifies bottlenecks that are difficult to optimise. By contrast a pattern of methods calls is more likely to encapsulate a range of behaviour that contains optimisation opportunities.

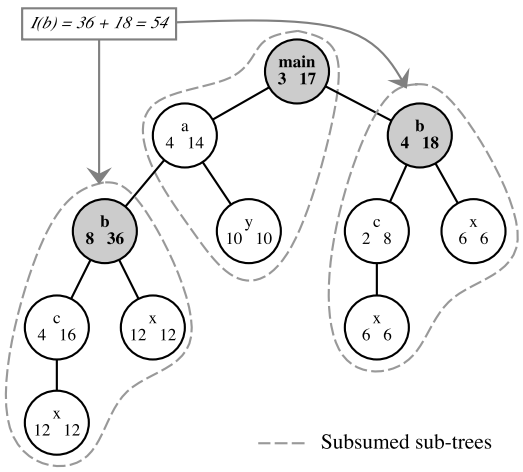

Figure 3: Subsumed subtrees for Example 1

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

Our approach to consolidating costs within the CCT is to identify the methods that are the most interesting from a performance standpoint and use these to identify 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 attribute costs recursively upwards until we find a node labeled with a subsuming method, where the costs are aggregated. We call this cost the `induced cost` for the node as it represents the cost induced by the subsuming method at that node. Figure 3 illustrates the subsuming concept using the CCT from our earlier example. Here we have chosen methods *b* and *main* as our subsuming methods (the method at the root of the CCT always becomes a subsuming method).

Each node labeled with a subsuming method becomes the root of a subsumed subtree and represents a pattern or block of code consisting of itself and the nodes it subsumes, either directly or transitively through other subsumed nodes. The induced cost of a subsuming method is the sum of the induced costs for all the nodes in the tree labeled with that method. In our example the total induced cost for method *b* is $36 + 18 = 54$. As the exclusive cost of each CCT node is consolidated into exactly one subsuming node (and each node is labeled with exactly one method) 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 initial method level partitioning.

This approach can be used to subsume any type of resource cost recorded in the CCT, or multiple costs at once. Typically these costs are execution time, invocation count or memory allocations but our approach can be applied to any recorded cost value.

More formally: Let V be the set of nodes in the CCT. Let M be the set of methods of the application, used as labels of nodes in the CCT. Let $S \subseteq M$ be the set of subsuming methods. Let function $l(v) : V \to M$ denote the method label of a node v and function $c(v) : V \to \mathbb{R}$ denote the cost.

Then we define $nodes(m) = \{v : v \in V, l(v) = m\}$ i.e. the subset of V that have the label m, and the induced cost for a node $v \in V$:

$$i(v) = c(v) + \sum_{c \in child(v)} \begin{cases} 0 & c \in S \\ i(c) & c \notin S \end{cases}$$

and the induced cost for a method $m \in S$:

$$I(m) = \sum_{v \in nodes(m)} i(v)$$

4.1 Identifying Subsuming Methods

Our approach is based upon identifying a subset of the methods in an application that are interesting from a performance standpoint i.e. the subsuming methods. Using different sets of subsuming methods leads to different results and potentially different insights into application performance. In this paper we have considered two characteristics of methods in order to define a set of subsuming methods that give us interesting and useful results. We are confident that there are many other approaches to selecting the subsuming methods that would also be effective. We discuss some of these in section 6. The two characteristics are:

Methods that induce only a very limited range of behaviour at runtime. These methods are not interesting from a performance standpoint because they tend to be simple code that is difficult to optimise. We have used 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 rooted at a node labeled by the method. For example the height of method `a` from our earlier example is 3 (see Figure 4). The trivial case is a leaf method that never calls any other method and therefore has a height of zero.

More formally the height $h(v)$ of any node $v \in V$ is:

$$h(v) = \begin{cases} 0 & |child(v)| = 0 \\ \max_{c \in child(v)}(1 + h(c)) & |child(v)| > 0 \end{cases}$$

and the height $H(m)$ of any method $m \in M$ is:

$$H(m) = \max_{v \in nodes(m)} h(v)$$

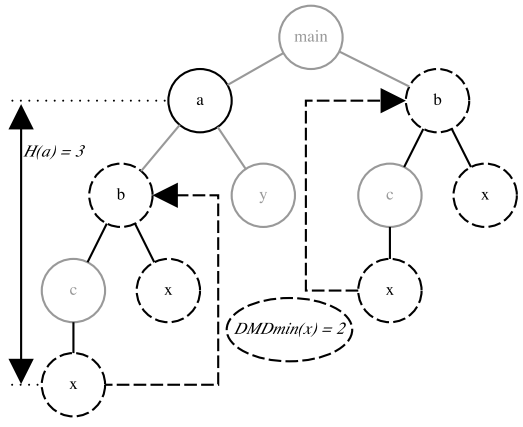

Figure 4: CCT for Example 1 showing height and DMD_{min}

Methods that are called in a very constrained set of circumstances. Specifically each call to the method can be traced back to a nearby *dominating* method, a distinct calling method responsible for its invocation. A dominating method appears in the call stack for every invocation of the dominated method i.e. if a method p dominates a method m, then any call to m is preceded by a call to p and followed by the return of that same call to p. Whilst potentially interesting from a performance standpoint these dominated methods are generally less interesting than the dominating method, as the dominating method is always invoked shortly before and encapsulates their invocation.

We have used the distance from a method to its nearest dominating method as a measure of this characteristic. The trivial case is when a method is only ever called from a single call site i.e. every occurrence of the method within the CCT has the same parent method. In this case that single parent will be the dominating method and the dominating method distance will be 1.

Formally for methods p and m, we say p is a dominating method of m (or p dominates m) if $p \neq m$ and $\forall v \in nodes(m)$ there exists a node $n \in V$ such that $l(n) = p$ and n is an ancestor of v in the CCT. By definition this means that the method label assigned to the root node dominates all other methods. Hence every method except the root method is guaranteed to have at least one dominating method i.e. the root method.

We define a distance function for the dominating method p of a method m as:

$$dmd(p, m) = \max_{v \in nodes(m)} d(p, v)$$

where $d(p, v)$ is the length of the path from node v to the first ancestor node n such that $l(n) = p$ (such an ancestor node must exist otherwise, by definition, p does not dominate m). We call this the dominating method distance (DMD) for m to p.

The minimum DMD for a given method m then is:

$$dmd_{min}(m) = \min_{p \in M} dmd(p, m)$$

i.e. the smallest DMD amongst all the dominating methods of m. Figure 4 illustrates the minimum DMD for method x from Example 1. Table 3 lists the dominating method and dmd_{min} for each method from Example 1.

Table 3: Height and dmd_{min} for Example 1

Method	Height	Dominating Method	dmd_{min}
main	4	–	–
a	3	main	1
b	2	main	2
c	1	b	1
x	0	b	2
y	0	a	1

Using our height and dmd_{min} attributes we can define a condition for identifying subsuming methods by specifying a bound on the minimum height and/or dmd_{min} a method must have in order to be considered subsuming i.e. we can define that all subsuming methods must have a height greater than H_{bound} and dmd_{min} greater than D_{bound}:

$$S = \{m : m \in M, H(m) > H_{bound}, dmd_{min}(m) > D_{bound}\}$$

It is also straightforward to efficiently implement an interactive analysis where these bounds can be dynamically changed as we need to only recalculate the induced costs after changing these bounds.

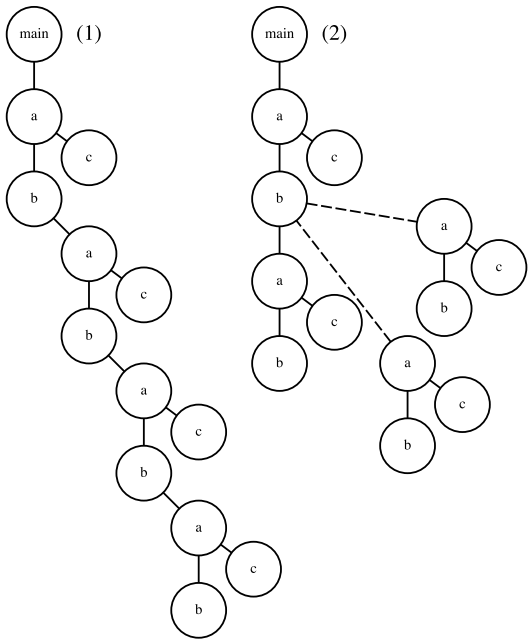

Figure 5: CCT before (1) and after (2) the recursive adjustment

4.2 Adjusting for recursive calls

Our discussion so far has ignored the impact of recursive calling patterns on the CCT and its subsequent analysis. Even conceptually simple recursive algorithms can result in very deeply nested call paths. These create subtrees with very large heights and dominating method distances that we would like to reduce to a single representative repetition of the recursion.

To achieve this we identify and reduce these recursive call paths in the CCT before we perform our height and DMD calculations. That way we can calculate the height and DMD in the adjusted tree to ensure our subsuming characterisation appropriately adjusts for recursion. We then perform our induced cost calculation on the original tree to ensure we aggregate the full costs from the tree. To build the adjusted tree we traverse the original tree from the root node downwards calculating, at each node, the ancestor node that we want to use as the adjusted parent. We perform an in-order traversal of the original tree to ensure that all ancestor nodes of the current node have already been added to the adjusted tree. Figure 5 illustrates this recursive adjustment with a simple example. There is a recursive path from method `a` to method `b` back to method `a`. We find each repetition of the sequence after the first and insert it as a child of that first sequence. The exact algorithm we use is detailed in the next section.

4.3 Implementation

We have developed a tool which implements our subsuming methods analysis. The tool takes as input a CCT representing a captured execution profile. We represent a CCT using a typical tree data structure made up of Nodes as detailed in Listing 1. Each Node has a Method label, a cost, a link to its parent and a list of children, all of which are populated when the CCT is input. Each Method has a populated list of the Nodes it is associated with, i.e. for all Nodes n it is true that $n.method.nodes.contains(n)$. The remaining fields in the Node and Method data structures are the ones that we calculate as part of our analysis. There is a Method object for each distinct method used by the application at runtime. In practice our Method object has fields for the method's name, owning class and signature, but these are not important for the description of the algorithms that follow. The root Node of the tree has a `null` parent.

Algorithm 1 Data-structures

```
class Tree
    Node root
    List<Method> methods

class Node
    Method method
    integer cost
    Node parent
    List<Node> children
    Node adjustedParent
    List<Node> adjustedChildren
    integer height
    integer induced

class Method
    List<Node> nodes
    integer maxHeight
    integer minDMD
    integer induced
```

This input data can be obtained from a number of different open-source or commercial profilers that capture CCT structured profiles. We have implemented adapters for the popular commercial tool JProfiler and the open-source profiler JP2. We have also implemented a tool which will parse a series of thread dumps captured from a running JVM and build from them a rough statistical CCT profile. The details of these adapters are not interesting for the current discussion except to highlight the fact that our approach can be used in conjunction with a number of existing profiling tools and frameworks for a variety of languages and platforms.

The first algorithm we apply is the recursive reduction process shown in Listing 2. It traverses the CCT finding and then linking each node with its adjusted parent. To find the adjusted parent for a node we scan back through its ancestor nodes looking for the last two nodes with the same method label as the current node. We then compare the two subpaths, from the current node to the most recent occurrence of the same method and from the most recent to the second most recent occurrence of the method, and if they match we set the adjusted parent to be the parent of the most recent occurrence of the method, effectively removing the last repetition of the recursion. We do this scanning process not in the original tree but in the adjusted tree, meaning that earlier repeats of the recursion have already been removed, therefore all consecutive repetitions of the same recursion are reduced to a single occurrence.

The result of this algorithm is the adjusted tree with repeated recursive call patterns reduced to a single instance

Algorithm 2 Reduce Recursive Paths

```
function reduceRecursivePaths(Node n)
    n.adjustedParent ← findAdjustedParent(n)
    n.adjustedParent.adjustedChildren.add(n)
    for all Node c in n.children do
        reduceRecursivePaths(n)

function findAdjustedParent(Node current)
    Node match1 ← null, match2 ← null
    Node n ← current.parent
    while n ≠ null & match2 = null do
        if n.method = current.method then
            if match1 = null then
                match1 ← n
            else
                match2 ← n
        n ← n.adjustedParent
    if match1 = null||match2 = null then
        return current.parent
    Node n1 ← current.parent
    Node n2 ← match1.adjustedParent
    while n1 ≠ match1 & n2 ≠ match2 do
        if n1.method ≠ n2.method then
            return current.parent
        n1 ← n1.adjustedParent
        n2 ← n2.adjustedParent
    if n1 ≠ match1||n2 ≠ match2 then
        return current.parent
    return match1.adjustedParent
```

as illustrated earlier in Figure 5. Note that every node from the original tree is included in the adjusted tree, so they contain the same number of nodes, all that has happened is that some nodes have been moved so they are inserted as the child of a node that was previously an ancestor higher in the tree, potentially reducing the overall height of the tree.

Once we have built our adjusted tree we use it to calculate the height of each node. This is a simple traversal (Listing 3) where we recursively calculate the height of each child and set our height to be one more than the maximum child's height. We also update the maximum height of the method associated with each node.

Next we calculate the minimum DMD for each method using the algorithm in Listing 4.

This is the most complex algorithm in our approach. The basic idea is to calculate the DMD for each method m to each of its dominating methods and find the smallest of these values to be the minimum DMD for m. However finding all the dominating methods for a method may be costly, so we take advantage of the fact that each dominating method must exist on the call path for every node n that is labeled with m. This means we can choose any node labeled with m and we only need check the methods that are the labels of its ancestors in the CCT. In our implementation the method `minDMD` takes the first node labeled with m and walks back up the ancestor nodes to the root of the tree using the associated methods as candidate dominating methods. The method `DMD` calculates the dominating method distance from m to p and returns ∞ if p does not in fact dominate m.

Once we have the height and minimum DMD calculated for each method we can traverse the original tree and calculate the induced costs (Listing 5). For each node we calculate the induced cost of each child node and add it to the current node's induced cost if the child is labeled with a subsumed method. We also add the cost to the aggregated induced cost for the associated method.

Algorithm 3 Calculate Height

```
function calculateHeight(Node v)
    v.height ← 0
    for all Node c in v.adjustedChildren do
        calculateHeight(c)
        if c.height + 1 > v.height then
            v.height ← c.height + 1
    if v.height > v.method.maxHeight then
        v.method.maxHeight ← v.height
```

Algorithm 4 Calculate Minimum DMD

```
function minDMD(Method m)
    m.minDMD ← ∞
    Node n ← m.nodes.getFirst()
    n ← n.adjustedParent
    while n ≠ null do
        integer dist ← dmd(n.method, m)
        if dist < m.minDMD then
            m.minDMD ← dist
        n ← n.adjustedParent

function dmd(Method p, Method m)
    integer dmd ← 0
    for all Node n in m.nodes do
        integer dist ← distance(p, n)
        if dist > dmd then
            dmd ← dist
    return dmd

function distance(Method m, Node n)
    integer dist ← 0
    while true do
        dist ← dist + 1
        n ← n.adjustedParent
        if n = null then
            return ∞
        if n.method = m then
            return dist
```

Algorithm 5 Calculate Induced Cost

```
function calculateInducedCost(Node n)
    n.induced ← n.cost
    for all Node c in n.children do
        calculateInducedCost(c)
        if isSubsumedMethod(c.method) then
            n.induced ← n.induced + c.induced
    n.method.induced ← n.method.induced + n.induced

function isSubsumedMethod(Method m)
    return (m = java.lang.Method.invoke())||
((m.maxHeight ≤ H_bound)&(m.minDMD ≤ D_bound))
```

As mentioned earlier the basis of our `isSubsumedMethod` check is a simple test of the already calculated maximum height and minimum DMD values against user defined constant values. The only extension to this that we have added is special handling for `java.lang.Method.invoke()` so that it is always marked as a subsumed method. Many Java libraries and frameworks make extensive use of reflection for various functionality. This means that `invoke` typically occurs in a large number of different contexts within a CCT, resulting in a large minimum DMD, and usually has a non-trivial height, as the methods being called via reflection are often non-trivial. Therefore without adding it as a special case it will often be listed as a subsuming method when in fact it is rarely interesting from a performance standpoint.

4.4 Implementation Efficiency

An important aspect of our approach is its efficiency. We are able to practically apply our approach to captured calling context tree profiles generated from large scale applications.

In terms of required space all the algorithms from the previous section operate on precisely the data structures we outlined without requiring anything more than the small constant number of variables they declare. For our data structures the space used is a constant amount of space for each node and method record plus the space required for the lists of child references. As each of these (the number of nodes, methods and total number of children) is bounded by the number of nodes in the CCT the space required for our representation is proportional to the size of the CCT.

Of course a number of the algorithms are recursive in nature and may generate a number of call stack frames but the recursion depth is bounded by the height of the CCT. Therefore the overall space requirements for processing a CCT with n nodes and a height of h is $O(n + h)$, which is equivalent to $O(n)$ given that $h \leq n$.

The recursive adjustment algorithm traverses every node in the tree but the calculation done at each node in findAdjustedParent uses only two loops each of which iterates over, at most, the ancestors of the current node. Hence findAdjustedParent has a worst case time proportional to the height of the original CCT and, since it is called once for each node in the CCT, the cost of building the adjusted tree is $O(n \times h)$. Given that for all but the most extremely unbalanced trees (which are very unlikely for CCTs of non-trivial applications) h is proportional to $\log n$ the average cost is $O(n \log n)$.

The height calculation is a simple constant time calculation for each node in the tree, so $O(n)$.

The minimum DMD calculation is the most expensive algorithm we undertake. For a CCT with n nodes, a height of h and m methods:

> `minDMD` is called m times.
> The loop in `minDMD` iterates at most h times.
> $\Longrightarrow$ `DMD` is called at most h times for each method m.
> Now `DMD` calls `distance` once for each node labeled with m
> $\Longrightarrow$ in total `distance` is called at most h times for each node in the CCT
> $\Longrightarrow$ `distance` is called at most $h \times n$ times.
> The method `distance` is also $O(h)$ as the number of iterations for the loop it uses is bounded by the height of the CCT
> $\Longrightarrow$ entire algorithm is $O(h \times h \times n)$.
> As before h is proportional to $\log n$ for all but the most unbalanced trees so the minimum DMD calculation is $O(n \log^2 n)$

Finally, assuming the `isSubsumedMethod` check is a constant time operation, the induced cost calculation, is a simple constant time calculation for each node in the CCT and is therefore $O(n)$. In our case the `isSubsumedMethod` check simply compares the already calculated height and minimum DMD to constant values.

5. EVALUATION

In order to evaluate our subsuming methods approach we have captured CCT profiles of the 14 benchmark applications in the DaCapo-9.12-bach suite [8] and applied our subsuming methods analysis to those captured profiles. Using the results of those experiments we have undertaken:

- A study of the characteristics of subsuming methods
- An empirical evaluation of the analysis time required
- Three detailed case studies that describe real problems found by our tool in the benchmark applications

All benchmarks were run with their default input size. All experiments were run on a quad-core 2.4 GHz Intel Core i7 with 8 GB 1600 MHz DDR3 memory running Mac OS X 10.9.3. We used Java SE Runtime Environment (build 1.7.0_40-b43) with the HotSpot 64-Bit Server VM (build 24.0-b56, mixed mode).

To capture the CCT profiles for each benchmark we used the open-source JP2 profiler developed at the University of Lugano [20, 21]. Our subsuming methods analysis can be applied to any CCT structured profile data but the JP2 profiler appealed to us for our experiments because it measures execution cost in terms of a platform independent metric, namely the number of bytecode instructions executed, which makes the captured profiles accurate, portable, comparable and largely reproducible. The only reason for variation in the profiles across multiple runs is non-determinism in the application or in the JVM (often related to thread scheduling in multi-threaded applications).

For the majority of the benchmarks we ran JP2 with the DaCapo-9.12-bach suite in the fashion outlined in the most recent JP2 paper [20], only adding our own custom 'dumper' which is used at program exit to serialise the captured JP2 profile to the binary CCT format our tool takes as an input. The framework activates the JP2 profiling using a callback mechanism that the DaCapo benchmark harness provides, so that the captured JP2 profiles include only the benchmark application code and not the benchmark harness. However we found that this approach only activated profiling for the thread that actually called the benchmark harness callback and any threads it subsequently created. For the client/server benchmarks (`tomcat`, `tradebeans` and `tradesoap`) and the benchmarks with background worker threads that were initialised before the main benchmark starts (`eclipse` and `xalan`) using the benchmark harness callback meant the captured profile included only a small fraction of the actual benchmark activity. Therefore for these 5 benchmarks we used our own wrapper which activated profiling for the entire run of the benchmark.

The results of our experiments on the DaCapo suite are summarised in Table 4. We ran and analysed each benchmark 5 times. Because of the very low variation between

Table 4: Results for DaCapo benchmarks when Subsuming Methods have height and $DMD_{min} > 4$

Benchmark	Instr Count (millions)	CCT Node Count			Method Count			$S(e)$	$S(i)$	$S(*)$	Analysis Time (ms)
		All	Subsuming	Ratio	All	Sub.	Ratio				
avrora	8393.98	176646.6	7158.0	4.05%	2189.0	74.0	3.38%	2	4	16	839.2
batik	2413.90	573887.0	48218.4	8.40%	6616.0	416.8	6.30%	0	2	18	3870.4
fop	863.41	628750.8	71429.8	11.36%	6709.4	345.0	5.14%	5	1	14	4512.4
luindex	2767.65	207279.0	7186.6	3.47%	2667.0	162.0	6.07%	3	1	16	980.0
lusearch	8121.91	59987.2	3850.2	6.42%	1726.0	73.0	4.23%	3	3	15	482.6
pmd	2272.56	4845777.8	1226270.6	25.31%	4573.2	266.2	5.82%	7	2	12	38424.4
sunflow	49626.02	299438.2	7666.4	2.56%	2341.6	106.0	4.53%	0	4	16	1087.2
xalan	8556.75	439529.6	38988.8	8.87%	4506.4	278.0	6.17%	1	5	15	2742.6
tradebeans	22799.24	8024512.8	1117675.8	13.93%	29111.6	2465.2	8.47%	6	4	10	52801.4
tradesoap	24768.51	8693078.0	1229925.6	14.15%	29913.8	2553.2	8.54%	4	1	15	62061.4
h2	12745.53	138520.2	17010.6	12.28%	1969.8	102.2	5.19%	4	3	13	1171.4
jython	12289.34	18982647.8	2667036.6	14.05%	5794.4	441.0	7.61%	7	1	12	215782.6
eclipse	67468.64	20670484.0	2942069.0	14.23%	16793.2	1841.0	10.96%	2	2	16	195120.6
tomcat	4067.95	2623250.6	357123.6	13.61%	13494.8	1025.2	7.60%	2	3	15	13823.2
Minimum				2.56%			3.38%	0	1	10	
Lower Quartile				6.91%			5.15%	2	1.25	13.25	
Median				**11.82%**			**6.12%**	3	2.5	**15**	
Upper Quartile				14.02%			7.61%	4.75	3.75	16	
Maximum				25.31%			10.96%	7	5	18	

$S(e)$ – The number of the top 20 subsuming methods that were also in the top 20 exclusive cost methods
$S(i)$ – The number of the top 20 subsuming methods that were also in the top 20 inclusive cost methods
$S(*)$ – The number of the top 20 subsuming methods that **did not** appear in either the top 20 exclusive or inclusive cost methods

each run our table only lists the average measurements across the 5 runs for each benchmark.[1] We characterised all methods with a height and minimum DMD greater than four as subsuming methods. We chose to use four as our threshold as it represents a relatively small distance in the CCT that can be readily visualised but still allows us to subsume a significant proportion of methods. We have also experimented with other small values and they generally return very similar results. The proportion of methods subsumed slowly increases as the threshold increases in an unsurprising manner.

The results show that:

- Across the benchmarks a median of 6.12% of all methods were subsuming methods
- The median size of the subsuming CCT was 11.82% of the size of the full CCT
- The median value for $S(*)$ was 15.

$S(*)$ measures the number of the top 20 subsuming methods (by induced cost) that did **not** appear in either the top 20 inclusive cost or the top 20 exclusive cost methods i.e. they represent code locations not directly highlighted by traditional measures. A median value of 15 implies 75% of the top 20 subsuming methods represented new potential optimisation opportunities. The compression results (only 6.12% of all methods and 11.82% of all CCT nodes were subsuming) demonstrate that subsuming methods analysis produces a CCT that is greatly reduced in size and has far fewer unique methods to inspect. We feel this substantially eases the task of interpreting the performance data.

We recorded the time taken to complete the analysis for each of our benchmarks. Note that this was the offline analysis time, it did not include the time taken to execute and profile each benchmark, only the time taken to parse and load the previously recorded profile data and apply our subsuming methods analysis. In section 4.4 we analysed the efficiency of our analysis and showed that it was practical to apply our analysis to large sets of performance data. Here we have the empirical data to support that claim, even the largest of our data sets was able to be analysed in under 4 minutes. The graph in Figure 6 shows the relationship between the number of nodes in the CCT and the required analysis time. The fit line we have plotted is of the form $y = a + bx\log^2(x)$ where a and b are constants and x and y are the CCT node count and analysis time respectively. The

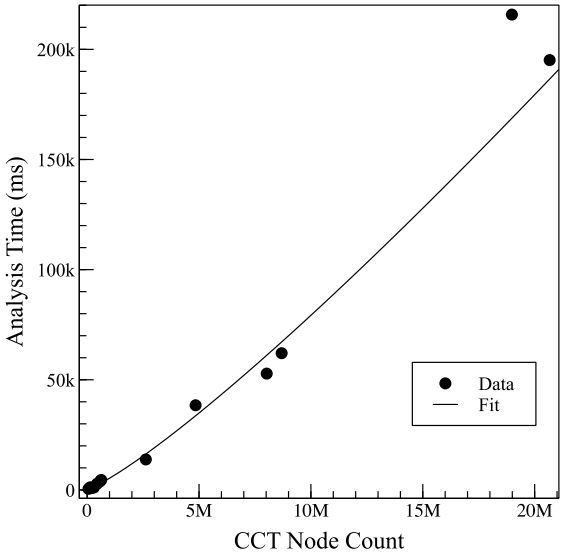

Figure 6: Analysis Time vs CCT Size

[1] `https://www.cs.auckland.ac.nz/~dmap001/subsuming` has more details and complete results.

fit line indicates that the measured analysis time is very close to our theoretical complexity of $O(n \log^2 n)$.

5.1 Case Studies

We have undertaken a number of case studies to demonstrate the utility of using subsuming methods in performance analysis. Each of these case studies is based on one of the individual benchmarks from the DaCapo suite. Due to space limitations we have only been able to present a limited number of the case studies here. All of the analysis and implementation of improvements for all of the case studies was completed within a single week on code bases with which we were unfamiliar, highlighting the fact that the subsuming methods approach rapidly facilitated useful improvements in the benchmark code.

5.1.1 Case Study: fop

Our first case study, introduced earlier in section 2, is on the `fop` benchmark. As we described earlier the method `org.apache.xmlgraphics.ps.PSGenerator.formatDouble` accounts for over 26% of the total cost of the benchmark. This method uses a `java.text.NumberFormat` to convert a double into a Java string with a 3 decimal place format. Having identified this costly method we have been able to implement a highly customised version that performs the same transformation much more efficiently, leading to an overall improvement in the benchmark of 22%.

The difficult part of this optimisation was identifying the opportunity i.e. that `PSGenerator.formatDouble` was inefficiently using Java's general purpose number formatting library. We found this opportunity almost immediately using subsuming methods. The top subsuming method in our analysis, with an induced cost of 13.6% (and inclusive cost of over 26%), is `java.text.DecimalFormat.format`. Inspecting the callers of this method we find that it is being called over 99% of the time by `java.text.NumberFormat.format` which in turn is being called over 98% of the time by `PSGenerator.formatDouble`. In short it takes only a brief time inspecting the top subsuming method in the benchmark to highlight the `PSGenerator.formatDouble` method. All that remains is to inspect the source code for the `PSGenerator` class to identify the opportunity that exists to fix the inefficient formatting code.

5.1.2 Case Study: h2

The second case study is the `h2` benchmark that runs a series of SQL load tests via JDBC against the H2 pure Java relational database implementation.

The second highest subsuming method by induced time in the benchmark is `org.h2.jdbc.JdbcResultSet.getString`, with an induced cost of nearly 16% and inclusive cost of over 25%. This method is called repeatedly whilst processing the results of SQL queries to retrieve the individual values for each row and column. It performs the relatively simple task of retrieving the Java object value at a particular index in the result set and converting it to a Java string. There are two major inefficiencies in the implementation that we were able to fix. Firstly it performs a very expensive validity check (including testing whether the associated database connection is still valid) on every call, even though the results for the current row have already been retrieved into memory and accessing them does not require any use of the underlying database connection. Secondly the string conversion for date and timestamp columns relies on an inefficient `java.sql.Timestamp.toString` implementation. More than 40% of the method time is spent in `Timestamp.toString` even though less than 10% of the columns in the database are dates or timestamps. We were able to fix both of these issues by patching the H2 code to avoid the expensive validity check and the call to `Timestamp.toString`, which we replaced with our own string conversion routine. These fixes combined to reduce the cost of `JdbcResultSet.getString` by 66% and the overall cost of the benchmark by 17%.

5.1.3 Case Study: tomcat

Our final case study is on the `tomcat` benchmark. Tomcat is a popular Java HTTP web server and servlet engine implementation. Using subsuming methods analysis highlights two significant optimisation opportunities in the benchmark, though neither is an inefficiency in the Tomcat server codebase. These are real optimisation opportunities in terms of the benchmark itself, but are probably atypical in terms of how Tomcat is used in other scenarios. Nevertheless they are real problems in this benchmark that we were able to find and fix and the fact that they may be atypical implies that the `tomcat` benchmark itself may not be a great representation of a typical Tomcat server deployment.

The first opportunity is `org.dacapo.tomcat.Page.fetch`, which is the implementation of a HTTP client that drives the benchmark by sending a series of HTTP requests to the Tomcat server. Three of the top four subsuming methods by induced cost are being called by this method. In total the `Page.fetch` method has an inclusive cost of 49% of the entire benchmark. Checking the implementation reveals some major inefficiencies in processing the received HTTP responses. The responses (received initially as a stream of bytes) are converted to a Java string, this string is then converted back into a byte array to calculate an MD5 checksum for the response and compare it to an expected result. The string version of the response is then also formatted with platform specific line endings (using a regular expression search and replace) in preparation to be written to a local log file, but the response is only actually written to the log file in exceptional circumstances (such as when the checksum doesn't match the expected result). We changed the implementation to calculate the MD5 checksum directly from the received bytes and only create the string encoded response with platform specific line endings when actually necessary. This reduces the cost of `Page.fetch` by over 65%.

The second optimisation opportunity we identified from inspecting the list of top subsuming methods when sorted by inclusive time. As we briefly discussed in section 2 when working with the full list of methods (i.e. not filtered to just the subsuming methods) the list of top inclusive time methods is rarely helpful. It is naturally dominated by the methods near the root of the CCT and in large-scale object-oriented applications with large CCTs and deep calling hierarchies there are a large number of framework methods that come to dominate the list. When filtered to just the subsuming methods however, because one of our criteria for subsuming methods (namely requiring a non-trivial minimum DMD) naturally excludes methods used in a predictable fashion, the list better provides a succinct overview of the most costly interesting sub-trees in the CCT.

In the case of the `tomcat` benchmark the list allows to quickly find that `JspCompilationContext.compile`, which

is the seventh method on the list, accounts for over 28% of the cost of the benchmark (`Page.fetch`, at 49%, was second on the list behind only `java.lang.Thread.run`). We were able to quickly confirm that this represented the time the benchmark was spending compiling requested JSP resources first into Java servlets and then into Java bytecode. However JSP resources can be precompiled. Doing this reduces the time being spent in the jasper compiler during the benchmark by 89%. The time was not completely eliminated because there was one JSP property group defined by Tomcat's example web application which relied on runtime properties and therefore triggered a recompilation of the associated JSP.

Neither of these optimisations were problems in the Tomcat server implementation, which is a mature and extensively tuned application, but they were both inefficiencies in the profiled benchmark that our subsuming methods approach allowed us to rapidly find and address. The combination of the two improvements produced a cost reduction in the benchmark of over 57%.

5.1.4 Summary of Case Studies

In our case studies we have been able to demonstrate the ability of subsuming methods analysis to aid in the identification of patterns of expensive methods calls that can then be optimised. Each case study highlighted a unique set of opportunities that were able to be addressed in different ways. For `fop` we were able to replace a powerful but inefficient generic library with a highly specialised approach. In the `h2` benchmark we removed unnecessary work (the validity checks) and optimised a poor `toString` implementation. In the `tomcat` benchmark we again removed unnecessary work in the client harness and avoided the cost of compiling JSP pages at runtime by precompiling them. All of these opportunities were readily found by investigating the results of the subsuming methods analysis.

6. DISCUSSION

Subsuming methods is a novel idea that provides additional insight into the runtime behaviour and performance of object-oriented applications. It enables us to discover new optimisation opportunities that are not apparent by inspecting the hot methods of an application. We regard it as a complementary approach, it provides the most benefit when used in conjunction with existing approaches to interpreting performance data (such as hot methods, calling context ring charts [18, 1] and using multiple context sensitive views [2]). It has the advantage of being efficient to apply as an offline analysis over data that can be obtained using a range of profiling tools. This means the approach is applicable in a range of performance investigations, from execution time and memory allocation profiling to more detailed analyses such as memory access patterns and CPU pipeline stalling instructions. It is also applicable in a range of runtime contexts, from detailed experimental performance analysis in dedicated test environments to the analysis of low overhead sampling profiles from production systems.

In section 4.1 we presented a particular approach for identifying subsuming methods, but there are other techniques that would complement or enhance our current approach. For example we are interested in using more sophisticated static analysis techniques, such as cyclomatic complexity measures, rather than height, as a measure of the range of behaviour a method induces. We also plan to investigate using static analysis to identify subsuming methods in other ways, such as identifying classes that implement key interfaces or hold key controlling roles in an application.

One of the weaknesses of our current approach is that methods such as `java.lang.Method.invoke`, whose child method calls are determined by their parameter values and that are used in a large number of different calling contexts, may get highlighted as subsuming methods. In actual fact it is some ancestor method higher up the call chain that is 'responsible' for the cost of the child method calls as it constructed the parameter values to pass to the method. It is a straightforward problem to work around for specific cases but we are interested in developing a general heuristic or approach for identifying these methods.

The evaluation we have presented in this paper is based upon empirical results and case studies from the DaCapo benchmark suite. We chose to use the benchmark suite as it provides a good range of different runtime behaviours from an independent source. The benchmarks in the suite are not ideal as examples of large-scale applications but our results have been encouraging and indicate that our approach should scale up well to handle the software found in industry that motivated our work. We have recently completed an industrial case study in which we applied our approach to a real-world large-scale application with promising results.

7. CONCLUSION

Experienced software engineers know that there are repeated patterns of method calls within a profiled application, induced by the design patterns and coding idioms used by the software, which represent significant optimisation opportunities. In this paper we have presented an approach to assist in automatically detecting these repeated patterns. By identifying the key *subsuming* methods within the calling context tree profile we are able to discover new optimisation opportunities not readily apparent from the original profile data. Our approach is implemented as an efficient offline analysis that can be applied to previously collected data from existing tools and environments. This makes it practical to apply, even for large-scale live production systems, and a useful additional tool for a wide range of performance investigations.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments and suggestions.

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

9. REFERENCES

[1] A. Adamoli and M. Hauswirth. Trevis: A Context Tree Visualization & Analysis Framework and its Use for Classifying Performance Failure Reports. *Proc. of the 5th Int'l Symp. on Software Visualization - SOFTVIS '10*, pages 73–82, 2010.

[2] L. Adhianto, J. Mellor-Crummey, and N. R. Tallent. Effectively presenting call path profiles of application performance. *39th Int'l Conf. on Parallel Processing Workshops*, pages 179–188, 2010.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation - PLDI '97*, pages 85–96, 1997.

[4] M. Arnold and P. Sweeney. Approximating the calling context tree via sampling. *IBM TJ Watson Research Center*, 2000.

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

[6] W. Binder. A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting. *Lecture Notes in Computer Science*, 3780:178–194, 2005.

[7] W. Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.

[8] 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 SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications - OOPSLA '06*, pages 169–190, 2006.

[9] 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*, 6813:383–407, 2011.

[10] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation - PLDI '11*, pages 516–527, 2011.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. *Proc. of the 2007 Int'l Symp. on Software Testing and Analysis - ISSTA '07*, pages 118–128, 2007.

[12] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. *Proc. of the 16th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering - SIGSOFT '08/FSE-16*, pages 59–70, 2008.

[13] J. Larus. Spending Moore's Dividend. *Communications of the ACM*, 52(5):62–69, 2009.

[14] S. Lin, F. Taʼıani, T. C. Ormerod, and L. J. Ball. Towards Anomaly Comprehension: Using Structural Compression to Navigate Profiling Call-Trees. *Proc. of the 5th Int'l Symp. on Software Visualization - SOFTVIS '10*, pages 103–112, 2010.

[15] N. Mitchell, E. Schonberg, and G. Sevitsky. Four Trends Leading to Java Runtime Bloat. *IEEE Software*, 27(1):56–63, 2010.

[16] 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*, page 85, 2005.

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

[18] P. Moret, W. Binder, D. Ansaloni, and A. Villazón. Visualizing Calling Context Profiles with Ring Charts. *5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis - VISSOFT '09*, pages 33–36, 2009.

[19] 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*, pages 268–278, 2013.

[20] 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*, 79:146–157, 2014.

[21] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine. *Proc. of the 9th Int'l Conf. on Principles and Practice of Programming in Java - PPPJ '11*, page 11, 2011.

[22] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation - PLDI '09*, pages 408–418, 2009.

[23] 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 SIGSOFT Int'l Symp. on Foundations of Software Engineering - ESEC/FSE-13*, pages 136–145, 2005.

[24] G. Xu. Finding reusable data structures. *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications - OOPSLA '12*, page 1017, 2012.

[25] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation - PLDI '10*, pages 174–186, 2010.

[26] 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*, pages 421–425, 2010.

[27] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation - PLDI '10*, pages 160–173, 2010.

[28] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. *34th Int'l Conf. on Software Engineering (ICSE)*, pages 134–144, 2012.