A Cost/Benefit Approach to Performance Analysis

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

Most performance engineering approaches focus on understanding the use of runtime resources. However such approaches do not quantify the *value* being provided in return for the consumption of these resources. Without such a measure it is not possible to compare the *efficiency* of these components (that is whether the runtime cost is reasonable given the benefit being provided). We have created an empirical approach that measures the value being provided by a code path in terms of the *visible* data it generates for the rest of the application. Combining this with traditional performance cost data, creates an efficiency measure for every code path in the application. We have evaluated our approach using the DaCapo benchmark suite, demonstrating our analysis allows us to quantify the efficiency of the code in each benchmark and find real optimisation opportunities, providing improvements of up to 36% in our case studies.

General Terms

Performance,Measurement

Keywords

efficiency analysis, blended analysis, profiling, runtime bloat

1. INTRODUCTION

Performance is a vital yet elusive attribute for much of the software developed today. Software engineering practices that focus on increasing developer productivity and software reuse have inevitably led to software built upon generalised frameworks and libraries. This often results in software with many layers of abstractions and very complex runtime behaviour. For example an enterprise Java service-oriented application may implement SOAP web services using the Axis web services framework in front of a Hibernate backend accessing a relational database and be deployed in a J2EE application server. This approach means that even simple requests induce a chain of furious activity requiring hundreds, maybe thousands, of method calls and data transformations to complete [13]. The tendency of modern applications to engage excessive activity to complete straightforward tasks has become known as runtime bloat [11, 21].

Most performance engineering approaches focus on understanding an application's *cost* i.e. its use of runtime resources. However understanding cost alone does not necessarily help find optimisation opportunities. One piece of code may take longer than another simply because it is performing more necessary work. For example it would be no surprise that a routine that sorted a list of elements took longer than a routine that returned the number of elements in the list. The fact that the costs of the two routines are different does not help us understand which may represent an optimisation opportunity. However if we had two different routines which both achieved the same result e.g. two different sorting algorithms, then determining which is the more efficient solution becomes a simple cost comparison.

The key is to understand the runtime *value* provided by a piece of code. Without this understanding it is not possible to compare different branches of code to find the superfluous activity that characterises runtime bloat. Existing performance analysis approaches generally do not measure value, traditionally this has been left to the engineer to determine through experience, intuition or guesswork. The challenge of intuitively divining value becomes much more difficult in modern, large-scale applications. Large-scale applications have thousands of methods interacting to implement runtime behaviours with millions of code paths[6]. Establishing the value provided by each method via manual inspection is not practical with such numbers of method calls. If we were able to empirically measure runtime value then we could combine this with traditional runtime cost information to establish the *efficiency* of each code path in the application.

This then is the key idea presented in this paper, we describe an approach to empirically quantify the runtime value of all code paths in an application so we can measure their efficiency. This allows us to find the most inefficient code paths in the application and analyse them for optimisation opportunities.

The main contributions of this paper are:

- We introduce the concept of focussing on the efficiency rather than purely on the cost of runtime behaviour.

ICPE'16, March 12–18, 2016, Delft, Netherlands.

ACM 978-1-4503-4080-9/16/03...$15.00.
DOI: http://dx.doi.org/10.1145/2851553.2851558.

- We describe *blended efficiency analysis*, a blended analysis approach to measuring code efficiency.
- We empirically evaluate our approach over standard benchmarks to characterise typical efficiency attributes.
- We demonstrate the utility of blended efficiency analysis 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. The costs 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 (CCT) [1].

A CCT is a data structure that 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. Therefore the method labels on the path from a node to the root of the tree describe a complete and distinct calling context.

Figure 1 in Section 4 shows an example CCT. Each node records the aggregated execution costs for a specific calling context. The execution cost recorded at a node does not include the cost of any child method calls, therefore it is known as the *exclusive* cost. The *inclusive* cost at a node does include the cost of child method calls and is the sum of the exclusive costs of all the children in the sub-tree rooted at the node.

The difficulty in analysing a CCT profile is that large-scale applications tend to have runtime costs that are thinly distributed across thousands of methods, meaning there are few obvious hotspots to target [21]. Table 1 shows the top ten hot methods from the `h2` DaCapo benchmark and illustrates the typical challenges. Five of the ten hot methods are low level utility methods from the Java system library, methods that are typically very difficult to optimise or avoid. Four of the remaining five methods (numbers 2, 4, 7 & 9) are core routines in the H2 implementation that are again both simple and widely used, e.g. `Row.getValue()` is a basic getter method that occurs in over 3000 locations in the CCT. This leaves only `TPCC.calculateSumDB`, which we discuss in Section 5.1.1, as a realistic optimisation opportunity from the top ten hot methods.

Intuitively then nine of the ten hot methods are hot simply because they are being frequently executed and not because they are inherently inefficient. In our experience this is typical of many large-scale object oriented programs. However within the `h2` benchmark there exist a number of optimisation opportunities not easily discoverable from the traditional performance profile, in fact over 10% of the benchmark's cost is spent in methods that achieve nothing at all.

An example of one of these `no effect` methods is `JdbcResultSet.checkColumnIndex()`, which is called every time a field is retrieved from a result set. It is responsible for raising a suitable exception should the column index not be in the valid range for the result set. However it also checks that both the underlying JDBC statement and connection are still open. In the course of our benchmark run these conditions were always satisfied, so in practice the method never raised an exception and hence its runtime value was zero – it was making no practical contribution to the benchmark processing. Reviewing the implementation we found that the checks on the statement and connection were unnecessary as neither were used during a field access. All field data is loaded into memory when the result set cursor is advanced to the current row via `JdbcResultSet.next()`. Both the statement and connection are confirmed to be open at this point, making the repeated checking on every field access redundant. We were able to refactor the checks performed on every field access and almost completely eliminate their runtime cost. This change reduced the total amount of benchmark code that was having no effect by over 75% and reduced the overall benchmark time by 8%.

Table 1: Top 10 Hot Methods — h2

Method	%Cost
org.dacapo.h2.TPCC.calculateSumDB(String, int)	08.011
org.h2.index.BaseIndex.compareRows(Row, Row)	06.159
java.lang.String.charAt(int)	05.767
org.h2.index.BaseIndex.compareValues(Value, Value, int)	04.063
java.lang.Integer.getChars(int, int, char[])	03.922
java.lang.System.arraycopy(Object, int, Object, int, int)	02.967
org.h2.table.TableFilter.next()	02.954
java.lang.String.length()	02.473
org.h2.result.Row.getValue(int)	02.321
java.lang.AbstractStringBuilder.append(String)	01.964

3. RELATED WORK

There is a large body of work into investigating software performance that we cannot adequately describe here due to space limitations. We discuss the most closely related work below but a full review of relevant empirical performance analysis approaches can be found in our review paper [9].

3.1 Runtime Bloat Analysis

The body of work that most closely matches ours in its motivation is the existing research into runtime bloat [21]. Generally the research has focussed on memory bloat (excessive memory use)(e.g. [5, 2]) or they have taken a data-flow centric approach [12, 13], looking for patterns of inefficiently created or used data structures, collections and objects [17, 19, 22, 18, 20, 23, 14]. Most of these approaches are looking for optimisation opportunities of a similar nature to those we are searching for, but our more generalised notion of understanding the *value* provided by all code paths in an application, rather than focussing primarily on understanding the use of data structures, allows us to discover a variety of different optimisation opportunities. Our previous work [10] on subsuming methods analysis aimed to finding compact repeated patterns of method calls that represented a significant hotspot and optimisation opportunity.

3.2 Blended Analysis

Blended analysis (a term first coined by Dufour et al in their work on blended escape analysis [7, 8]) is a combination of dynamic and static analysis, where the dynamic analysis determines the program region over which the static analysis is performed. With the notable exception of the work by Du-

four et al, very little existing performance analysis research uses either static or blended analysis to aid in the gathering or interpretation of performance data [9]. In particular the majority of the runtime bloat research instead uses pure dynamic analysis in the form of specialised profiling implemented by modifying an open-source research JVM, often incurring very high runtime overheads.

We have implemented our efficiency analysis as a blended analysis, leveraging both dynamic profiling information and an intra-procedural forward data flow static analysis. Using blended analysis allows us to leverage the benefits of both static and dynamic analysis. To quote from Dufour et al [8]:

> Blended analysis offers many advantages compared to a purely static or dynamic analysis. First, blended analysis limits the scope of the static analysis to methods in the program that actually were executed, thus dramatically reducing the cost of a very precise static analysis by reducing its focus, allowing achievement of high precision over an interesting portion of the program. Second, blended analysis only requires a lightweight dynamic analysis, thus limiting the amount of overhead and perturbation during execution.

We gained the same advantages, allowing us to implement an offline analysis that could measure the runtime value of all code paths captured by a lightweight dynamic profile. Where the focus of Dufour et al's work was to understand the use of temporary objects our more general focus was to appreciate the value being provided by the code paths captured in an application profile.

4. BLENDED EFFICIENCY ANALYSIS

We wish to quantify the efficiency of the code in an application. Our approach to achieving this is to calculate a measure of value for every node in a captured calling context tree profile. We can then trivially combine this with the traditional cost measures captured in the profile to calculate the efficiency at every calling context:

$$CC_{eff} = CC_{value}/CC_{cost}$$

We can also similarly calculate the efficiency of any method using the aggregated method value and cost. Our measure of value at each calling context is an inclusive measure for the entire sub-tree rooted at that node. Therefore the efficiency measure is not just for the activity in the method at that node but all the activity in the sub-tree.

Our approach to quantifying value is to measure the volume of data created by a method that becomes visible to the rest of the application i.e. it escapes the context of the method. Our rationale is that the value a method is providing can only be imparted by the data it creates that is subsequently visible outside of the method. Intermediate objects and calculations that are created during processing but then discarded do not contribute to this final value. Intuitively two method calls that produce identical results (given the same arguments) and have no other side effects, are providing the same value, regardless of their internal implementation.

Specifically we track the number of object field updates that ultimately escape their enclosing method. An object field update is any assignment to an object field or array element e.g. `foo.value = 1` or `bar[0] = 1`. For brevity we will refer to such an assignment as a *write*. Writes to static fields are handled as a special case of an object field write that is being applied to a special global object. Assignments to primitive local variables are **not** recorded as writes as they hold only intermediate values that must be copied to an object field in order to escape the current calling context.

To escape the current calling context, a write must be applied to either:

- a globally accessible field or object (globally escaping)
- a method operand (operand or argument escaping)
- an object returned from the method (returned)

To these traditional escape states we also track when an object has been passed to an IO output routine (output escaping) as the data has escaped to an external source.

Our concept of escaping writes is similar to *object escape analysis*[4]. Escape analysis is a method of characterising the effective lifetime of objects, it calculates whether newly created objects become visible outside of the method which created them. Our tracking of writes rather than objects gives us a more fine-grained view of the work being done and our concept of output escaping writes has no parallel in object escape analysis.

Each write that occurs at a calling context we call a *local* write. The *total* writes for a calling context is the sum of the total writes of its child methods plus its local writes. Therefore the sum of the local writes in any sub-tree will equal the total writes for the node at the root of the sub-tree. Each one of the total writes either *escapes* the calling context or we say it is *non-escaping*. Escaping writes are categorised as *global*, *operand*, *returned* or *output* depending on how they escape. An escaping write may satisfy more than one of these classifications simultaneously. For example a write to an object that is both an operand and returned will be counted as both an operand and returned write, but only a single escaping write.

Operand and returned writes may be *captured* by a parent context if, for example, an object created and returned by one method is then discarded by the calling method. Global and output writes however are globally escaping, they can never be captured. To be exact the *captured* writes for a calling context is the number of local and child escaping writes that do not escape the context. This definition means the sum of the captured writes in any sub-tree will equal the non-escaping writes for the node at the root of the sub-tree.

Finally the *value* for each calling context is the number of escaping writes at the node plus, for methods with primitive non-void return types, we add the method invocation count. This ensures they have a value of at least one for each invocation, to reflect the value of the primitive they return.

Example 1 lists some examples from Java library code:

- `Arrays.copyOf()` – has only returned writes as it returns the new array object it creates and populates. There are no operand, global, output or captured writes.
- `String.getChars()` – has only operand writes as it simply populates the buffer passed as its third operand
- `AbstractStringBuilder.expandCapacity()` – has only operand writes, as it only updates the state of the `this` object to create a new larger internal buffer
- `AbstractStringBuilder.append(String)` — has both operand and returned writes as it updates its internal state and then returns a reference to itself

Example 1 Selected Java library methods

```
 1: public class AbstractStringBuilder {
 2:   public AbstractStringBuilder append(String str) {
 3:     if (str == null) str = "null";
 4:     int len = str.length();
 5:     ensureCapacityInternal(this.count + len);
 6:     str.getChars(0, len, this.value, this.count);
 7:     this.count += len;
 8:     return this;
 9:   }

10:   private void ensureCapacityInternal(int minimumCapacity) {
11:     if (minimumCapacity - this.value.length > 0)
12:       expandCapacity(minimumCapacity);
13:   }

14:   void expandCapacity(int minimumCapacity) {
15:     int newCapacity = this.value.length * 2 + 2;
16:     if (newCapacity - minimumCapacity < 0)
17:       newCapacity = minimumCapacity;
18:     if (newCapacity < 0) {
19:       if (minimumCapacity < 0) // overflow
20:         throw new OutOfMemoryError();
21:       newCapacity = Integer.MAX_VALUE;
22:     }
23:     value = Arrays.copyOf(this.value, newCapacity);
24:   }
25: }

26: public class String {
27:  public void getChars(int bgn,int end, char dst[],int pos) {
28:    System.arraycopy(this.value, bgn, dst, pos, end - bgn);
29:  }
30: }

31: public class Arrays {
32:  public static char[] copyOf(char[] orig, int len) {
33:    char[] c = new char[len];
34:    System.arraycopy(orig,0,c,0,Math.min(orig.length,len));
35:    return c;
36:  }
37: }
```

Our blended efficiency analysis consists of three phases:

- the capturing of a dynamic calling context tree profile during the execution of an application
- a pure intra-procedural static analysis for each method captured in the dynamic profile
- a final analysis phase where the results of the static analysis are combined with the dynamic profile to build a complete inter-procedural write analysis

Our implementation is built to work with Java applications, but the concepts are easily transferable to any object-oriented language.

4.1 Profiling

We used the JP2 profiler developed at the University of Lugano [15, 16] to capture our CCT profiles. JP2 is an instrumentation based profiler that captures basic block level profiles in a calling context tree data structure. Each node has an array of invocation counts, one for each basic block in the method. JP2 calculates the runtime cost for each node as the number of bytecode instructions executed. This platform independent metric appealed to us for our experiments because it was portable and reproducible.

We made one extension to JP2 to support our specific profiling needs. `System.arraycopy()` is a native method used to copy array content from a source array to a destination array and it is passed a length parameter which defines the number of elements copied. We felt it important to be able to record its activity, because it is a very frequently invoked method whose sole purpose is to populate an array i.e. create writes. Because it is a native method we cannot record its activity using normal bytecode instrumentation. Therefore we augmented JP2 to record the length parameter at runtime and aggregate it into a cumulative count kept alongside the basic block counts for each `System.arraycopy()` node in the CCT. This allows us to know precisely how many writes have been performed at each of these nodes in the CCT.

4.2 Static Write Analysis

The static analysis we use is an intra-procedural forward data flow analysis (implemented using the ASM 5.0.4 bytecode manipulation library[1]) that tracks the origin of each object reference utilised within the method. We first find and record the basic blocks within the method before then performing the data flow analysis. Like all forward data flow analyses our analysis simulates all execution paths through the method and maintains a complete stack frame for each instruction representing the possible values of all local variables and method operands after that instruction. The key implementation points of our analysis are:

- Each object reference has a value (we call its *origin value*) indicating its possible origin, being one of:
  - `Operand` – a method operand
  - `Local` – a new locally created object
  - `ReturnValue` – the result of a child method call
  - `Global` – retrieved from a global (static) field
  - `Output` – an operand to an IO output method
  - a composite value made up of a set of the above values
- The initial stack frame is initialised with `Operand` values for the method operands
- New `Local`, `ReturnValue` and `Global` values are created in response to the *new object*, *invoke method* and *get static field* instructions respectively
- The result of referencing an object field (via the get field instruction) is the value of the parent object e.g. the field of a `Local` is a `Local`
- For each *put field* instruction we record a *write* against the current basic block with the origin value of both the updated field and the value being put
- We also record a *write* for each newly created object or array so that new objects, even with no explicit field puts, have a non-zero write count
- Arrays are handled the same way as objects, with the *new array* instruction creating a new `Local` value and getting or setting of an array element being treated the same as an object field get or put
- For each child method call instruction we record the origin values of the method operands used for the call
- For each return instruction we record against the current basic block the origin value of the returned object

At the completion of the analysis we have recorded for each method:

- the child method call information, including the origin value for every operand to those calls
- the basic blocks in the method, and for each basic block:
  - every field write, including the origin value of both the updated and put objects
  - the origin value of any return instruction (there can be at most one in a basic block)

[1] http://asm.ow2.org/

This information gives us the static escape information for each potential write performed by the method. However it is not until we combine it with the runtime profile information that we can determine how many writes were performed and what the escape status of those writes is.

For example the static information for `Arrays.copyOf()` from Example 1 can tell us that:

- There is one new object (an array) created at line 33, which we denote `Local[33]`
- There is one method call made to `System.arraycopy()` where we pass `Operand[0]` as the first parameter and `Local[33]` as the third parameter
- We return `Local[33]`

At this intra-procedural static analysis phase we do not know what, if any, writes `System.arraycopy()` may perform and therefore we do not know the actual number of returned writes in `Local[33]` or operand writes applied to `Operand[0]`.

4.3 Inter-Procedural Write Analysis

The final stage of the blended analysis completes the inter-procedural write analysis using a post-order (i.e. child-first) traversal of the CCT to merge the dynamic profile data with the static write information. The essence of the process is to use the basic block count information from the dynamic profile to determine which statically recorded writes actually occurred. We use this write information to construct an *object update graph* that represents which objects are assigned to the fields of which other objects. This allows us to account for situations such as when a write is applied to a `Local` object which is then put into an `Operand` object. The initial write has been propagated to an operand and needs to be counted as operand escaping.

The actual steps performed for each node in the CCT are:

- find the node's runtime return value
- build the node's object update graph
- use the object update graph to find the escape status for each occurring write and record these writes
- aggregate the write information from the child nodes to complete the write counts at this node

We also need to *resolve* any `ReturnValue` origin values from the static analysis phase. *Resolving* an origin value is the process of determining a concrete runtime origin value for `ReturnValue` types i.e. whether it is in fact a `Local`, `Global` or `Operand` object. We do this by finding the actual child in the CCT associated with the method call from the `ReturnValue` origin value and using the runtime return value from that node. If this runtime return value refers to an `Operand` origin value we then need to use our recorded child call information to determine the value of the operand when the method was called. So consider resolving the call to `StringBuilder.append(String)` at line 4 (we denote this as `ReturnValue[append()@4]`) in the method below:

```
1:  public static String buildString() {
2:    final StringBuilder sb = new StringBuilder();
3:    return sb.append("Hello")
4:             .append("World").toString();
5:  }
```

We know that `StringBuilder.append(String)` returns `this` so the child node in the CCT will have a return value of `Operand[0]`. Looking at the static call information for the `[append()@4]` method call we can see that operand 0 for that call is `ReturnValue[append()@3]` i.e. the result of the append on line 3. So we then resolve the call to `ReturnValue[append()@3]`. The associated child node will again have a return value of `Operand[0]` but this time the static call information for `[append()@3]` tells us that operand 0 is `Local[2]` (the new `StringBuilder` created at line 2 assigned to the `sb` variable). Therefore the final resolved value for `ReturnValue[append()@4]` is `Local[2]`.

The complete algorithm for resolving values is listed in Algorithm 1. Note that when processing the child return value *cv* there is no need to handle `ReturnValue` types as these child return values have themselves already been resolved i.e. they cannot have a `ReturnValue` type.

Algorithm 1 Resolve Value

```
function RESOLVE(OriginValue v, CCTNode node)
  if ISCOMPOSITE(v) then
    OriginValue result ← null
    for all OriginValue next in v.values do
      OriginValue r ← RESOLVE(next, node)
      result ← MERGE(result, r)
    return result
  else if ISRETURNVALUE(v) then
    CCTNode child ← node.GETCHILD(v.call.method)
    OriginValue cv ← child.returnValue
    OriginValue result ← null
    if ISOPERAND(cv) then
      OriginValue op ← v.call.arg[cv.opIndex]
      OriginValue r ← RESOLVE(op, node)
      result ← MERGE(result, r)
    if ISLOCAL(cv) then
      result ← MERGE(result, Local)
    if ISGLOBAL(cv) then
      result ← MERGE(result, Global)
    return result
  else
    return v
```

The algorithm for finding the runtime return value for a node is listed in Algorithm 2. The basic idea is that we merge together the possible return values from the basic blocks with non-zero invocation counts. Any basic block with a zero invocation count has not been executed and therefore we can exclude it from the runtime return value for the current node. Finally we resolve this merged runtime value (using Algorithm 1) to remove any references to `ReturnValue` types giving us a runtime return value purely in terms of `Local`, `Global` and `Operand` origin values.

Algorithm 2 Runtime Return Value

```
function FINDRUNTIMERETURNVALUE(CCTNode node)
  OriginValue result ← null
  int i ← 0
  for all BasicBlock bb in node.method.blocks do
    if node.blockCount[i] > 0 then
      result ← MERGE(result, bb.returnValue)
    i ← i + 1
  return RESOLVE(result, node)
```

An overview of the algorithm used to build the object update graph is listed in Algorithm 3. In the first stage we iterate over every recorded write from a basic block with a non-zero invocation count and add an edge from the putted value to the updated value. In the second stage we iterate over the child nodes in the CCT and merge in the parts of their object update graphs that impact the objects in the current node. That is every edge in the child graph that leaves an operand type or the returned object. When

we merge edges from the child graphs we need to resolve operand types in the child to their values in the parent using the recorded child call information. In the final stage, if the method at the current node is an IO output method, we add an edge from each output operand to the special `Output` origin value type.

Algorithm 3 Building the Object Update Graph

```
function BUILDUPDATEGRAPH(CCTNode node)
  Graph result ← new Graph
  int i ← 0
  for all BasicBlock bb in node.method.blocks do
    if node.blockCount[i] > 0 then
      for all Write w in bb.writes do
        OriginValue from ←RESOLVE(w.putted, node)
        OriginValue to ←RESOLVE(w.updated, node)
        graph.ADDEDGE(from, to)
    i ← i + 1
  for all CCTNode child in node.children do
    for all OriginValue from in child.graph do
      if ISOPERAND(from) then
        Call call ← node.method.GETCALL(child.method)
        OriginValue op ← call.arg[from.opIndex]
        OriginValue value ← RESOLVE(op, node)
        for all OriginValue to in child.graph do
          if child.graph.HASPATH(from,to) then
            if ISOPERAND(to) then
              to ← call.arg[to.opIndex]
              to ← RESOLVE(to, node)
              graph.ADDEDGE(value, to)
            if ISGLOBAL(to) then
              graph.ADDEDGE(value, Global)
            if ISOUTPUT(to) then
              graph.ADDEDGE(value, Output)
            if child.ISRETURNED(to) then
              graph.ADDEDGE(value, Local)
  if ISOUTPUT(node.method) then
    for all int op in node.method.operands do
      if ISOUTPUTARGUMENT(op) then
        graph.ADDEDGE(Operand[op], Output)
  return graph
```

Once the object update graph has been built it is straightforward to count the object writes performed at the current node. We iterate again over the writes from the basic blocks with non-zero counts. For each write we use the object update graph to determine the escape status of the write and then account for the write accordingly.

The final stage of the process is to aggregate the child write information so that we have at each node the complete aggregate write information for the sub-tree. Total, global, output and non-escaping writes can be aggregated directly from the child nodes as for each of these types of writes the status is preserved from the child to the parent e.g. a global write in the child will always be a global write in the parent. The difficulty comes in determining the status of the operand and returned writes from the child, whether they escape the parent or are captured.

To accurately aggregate the child escaping write information we replay the operand and returned writes recorded by the child node, after again resolving the operands to their appropriate origin values in the current node using the recorded child call information. This allows us to evaluate the escape status for these writes in the current node and account for them accordingly. When we account for a write at a given node we keep a record of that accounting so that the parent can later use it to replay the operand and returned writes. These *write records* include the relevant writes from all child nodes, therefore a node only needs to check the write records of its direct children and not all the nodes further down the sub-tree.

The final task is to calculate a value for each CCT node. This is simply the number of escaping writes at the node plus an adjustment for methods with primitive non-void return types. For these methods we added the method invocation count to the value, effectively giving them a value of at least one for each invocation, to reflect the primitive data value they return.

4.4 Example - Time Formatting

Consider the example code listed in Example 2. The CCT with the calculated write information for this code is shown in Figure 1 and the final write statistics aggregated by method are shown in Table 2.

Example 2 Time formatting

```
 1:  public static void outputTime(OutputStream out) {
 2:    final StringBuilder sb = new StringBuilder();
 3:    formatTime(sb, 123456789, false);
 4:    out.write(sb.toString().getBytes());
 5:  }

 6:  public static void formatTime(StringBuilder sb, long time,
 7:                                boolean appendMillis) {
 8:    long seconds = time / 1000;
 9:    long minutes = seconds / 60;
10:    long hours = minutes / 60;
11:
12:    sb.append(zeroPad(4, hours)).append(" hours ");
13:    sb.append(zeroPad(2, minutes % 60)).append(" minutes ");
14:    sb.append(zeroPad(2, seconds % 60));
15:    if (appendMillis) {
16:      sb.append(".").append(zeroPad(3, time % 1000));
17:    }
18:    sb.append(" seconds");
19:  }

20:  public static String zeroPad(int l, long v) {
21:    String result = "" + v;
22:    while (result.length() < l) {
23:      result = "0" + result;
24:    }
25:    return result;
26:  }
```

This code outputs a formatted time string to the given `OutputStream`. The code as written with the hard-coded value of 123456789 outputs the string `"0034 hours 17 minutes 36 seconds"` (32 characters). We begin our analysis with the `zeroPad` method. This method constructs a zero-padded string representation of a `long` that is at least a given minimum length. On every invocation the statement on line 21 creates a new `StringBuilder` and calls `append(long)` and then `toString()` on it. When the result needs to be zero-padded, the statement on line 23 creates an additional `StringBuilder` (this time via the `<init>(String)` constructor) and more calls to `append(String)` and `toString()`. In our example `zeroPad` is invoked 3 times, and on one of these invocations it twice iterates around the zero-pad loop, so in total we have 5 calls to `toString()` which cumulatively have 33 returned writes. The return value for `zeroPad` is the result of these `toString()` calls so the number of returned writes for `zeroPad` is 33. These are the only escaping writes for the method, all writes generated by the use of the `StringBuilder` objects are captured by the method.

The `formatTime` method populates a `StringBuilder` object (passed as operand 0) with a formatted time string for a given `long` value. It calls `append(String)` 6 times, 3 times

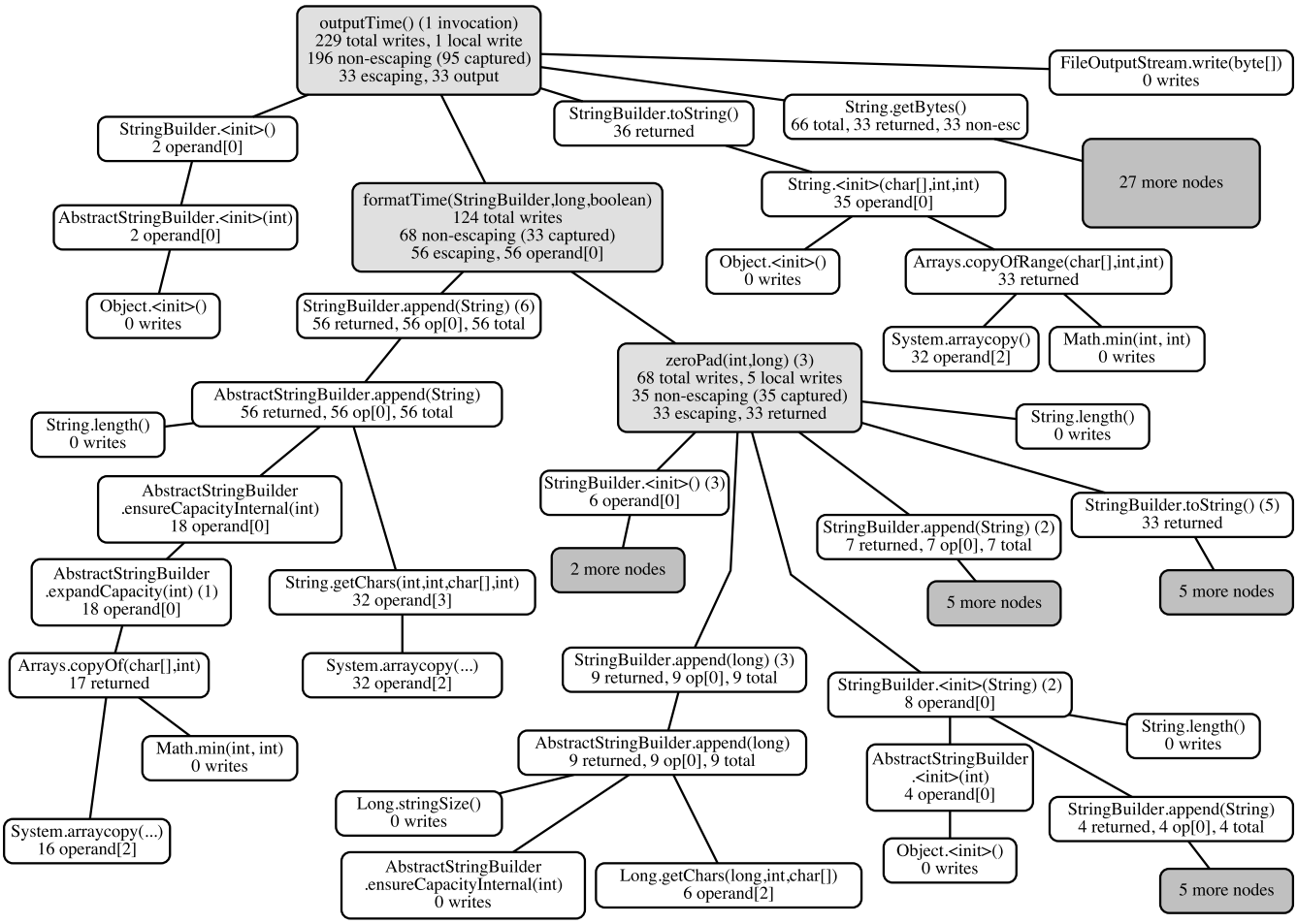

Figure 1: CCT annotated with write information for Example 2

The shaded blocks represent sub-trees that have been truncated for simplicity, most of these are shown in full elsewhere in the tree. The number in parentheses is the node's invocation count, where it is not shown the invocation count is the same as for the parent node. All non-zero write counts for each node are also shown, where a count for a particular type of write is not given it can be taken as 0.

Table 2: Aggregated Method Write Counts - Example 2

Method	Inv.	Cost	Value	Efficiency	Total	Esc	Non-Esc	Cap	Out	Ret	Op	Local
outputTime(OutputStream)	1	3563	33	00.926	**229**	33	**196**	95	33	0	0	1
System.arraycopy(...)	18	924	132	14.285	132	132	0	0	0	0	132	132
formatTime(StringBuilder,long,boolean)	1	2100	56	02.666	124	56	68	33	0	0	56	0
StringBuilder.toString()	6	675	75	11.111	69	69	0	0	0	69	0	6
zeroPad(IJ)	3	1291	36	02.788	68	33	35	35	0	33	0	5
StringBuilder.append(String)	10	1069	67	06.267	67	67	0	0	0	67	67	0
AbstractStringBuilder.append(String)	10	1009	67	06.640	67	67	0	0	0	67	67	10
String.getBytes()	1	1121	33	02.943	66	33	33	0	0	33	0	0
StringEncoder.encode(char[],int,int)	1	906	33	03.642	66	33	33	33	0	33	0	1
String.<init>(char[],int,int)	6	621	63	10.144	63	63	0	0	0	0	63	12
Arrays.copyOfRange(char[],int,int)	6	477	51	10.691	51	51	0	0	0	51	0	6
String.getChars(int,int,char[],int)	10	473	39	08.245	39	39	0	0	0	0	39	0
Arrays.copyOf(byte[],int)	1	243	33	13.580	33	33	0	0	0	33	0	1
UTF_8$Encoder.encode(char[],int,int,byte[])	1	603	33	05.472	32	32	0	0	0	0	32	32
AbstractStringBuilder.expandCapacity(int)	1	153	18	11.764	18	18	0	0	0	0	18	1
AbstractStringBuilder.ensureCapacityInternal(int)	13	247	18	07.287	18	18	0	0	0	0	18	0
Arrays.copyOf(char[],int)	1	132	17	12.878	17	17	0	0	0	17	0	1
AbstractStringBuilder.<init>(int)	6	42	12	28.571	12	12	0	0	0	0	12	12
AbstractStringBuilder.append(long)	3	414	9	02.173	9	9	0	0	0	9	9	3
StringBuilder.append(long)	3	432	9	02.083	9	9	0	0	0	9	9	0
StringBuilder.<init>(String)	2	186	8	04.301	8	8	0	0	0	0	8	0
StringBuilder.<init>()	4	44	8	18.181	8	8	0	0	0	0	8	0
Long.getChars(long,int,char[])	3	228	6	02.631	6	6	0	0	0	0	6	6
String.length()	17	68	17	25.000	0	0	0	0	0	0	0	0
Math.min(int,int)	9	47	9	19.148	0	0	0	0	0	0	0	0
Long.stringSize(long)	3	78	3	03.846	0	0	0	0	0	0	0	0
Totals								**196**				**229**

with the result of a call to `zeroPad` and 3 times with string literal values. These 6 calls induce 56 operand escaping writes to the `StringBuilder` made up of:

- 32 individual character writes
- 6 writes (one per call) to the builder's count variable
- 18 further writes that occur when the builder needs to expand the capacity of its internal buffer - this expansion creates a new buffer (1 write), copies the existing 16 characters into it and assigns it (1 more write) to its buffer variable

The 33 returned writes from `zeroPad` are all captured by `formatTime` as the strings returned from `zeroPad` do not escape `formatTime`. Therefore we have 68 non-escaping writes (the 33 captured by `formatTime` plus the 35 captured by `zeroPad`), 56 escaping writes and 124 writes in total.

The `outputTime` method creates a `StringBuilder`, passes it to `formatTime`, then calls `toString()` and finally `getBytes()` on the result to create a byte array it can pass to the given `OutputStream`. The writes associated with the builder (3 from its construction and 56 operand writes from `formatTime`) plus the 36 writes returned from `toString()` make up the 95 captured writes as neither of these objects escape the method. The 33 returned writes from `getBytes()` are output escaping because the returned byte array is passed to our IO output method. There are 33 returned writes for our 32 length byte array because of the one write allocated for each newly constructed object, in this case the byte array itself. The `getBytes` call also has 33 non-escaping writes because the string encoder does not know the exact length of the byte array to allocate ahead of time. Therefore it needs to encode into an oversized byte array and copy the array when it is finished to return an array of the exact size.

Even from this simple example there are some points of note in the method write statistics in Table 2.

- The methods with the most obvious inefficiencies (such as `zeroPad` and `formatTime`) had high numbers of captured writes and low efficiency values.
- The difference in efficiency between the builder's `append(String)` and `append(long)` methods shows how much more work has to be done to convert a two's complement form value into a string when compared to simply copying existing character data.
- `System.arraycopy` was the root source of more than half of all the writes in our example. Whilst this is a contrived example this was a trend we also noticed in our later experiments.

5. EVALUATION

In order to evaluate our blended efficiency analysis we have conducted experiments with the 14 benchmark applications in the DaCapo-9.12-bach suite [3]. Using the results of those experiments we have undertaken:

- A study of the characteristics of application efficiency
- Three detailed case studies that describe real optimisations we made in the benchmark applications based upon our efficiency analysis results

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 Oracle's Java SE Runtime Environment (build 1.7.0_71-b14) with the HotSpot 64-Bit Server VM (build 24.71-b01, mixed mode).

We profiled the benchmarks in the same manner that we conducted our experiments in our previous paper [10]. That is we executed the majority of the benchmarks in the fashion outlined in the most recent JP2 paper [15], but for the client/server benchmarks (`tomcat`, `tradebeans` and `tradesoap`) and the benchmarks with background worker threads (`eclipse` and `xalan`) we used our own wrapper which activated profiling for the entire run of the benchmark. We also had to disable intrinsic methods (a feature of the JIT compiler) in order to obtain complete profiles as otherwise our instrumentation was by-passed for these methods.

Once we had obtained our captured profiles we ran our offline blended analysis over the profiles to analyse the runtime behaviour and efficiency of the benchmarks. Our results are summarised in Table 3. The table shows for each benchmark:

- the size of the captured CCT
- the number of unique methods in the CCT
- the cost – the number of bytecode instructions executed
- the number of IO output writes
- the efficiency rating, this is $output/cost$

The next three columns show the percentage of the total benchmark activity that had certain properties:

- had no side effects – no output, operand or global writes
- had no effect – no side effects and no return value
- had low efficiency – when $value/cost < 0.1$

We were also interested in the distribution of inefficient behaviour within the benchmarks. Therefore we analysed the percentage of the captured writes across each benchmark that were attributed to different locations, specifically:

- the top single capturing node in the CCT
- the top capturing method
- the ten top capturing methods

To analyse the importance of `System.arraycopy` we calculated the percentage of writes for which it is responsible. Finally we list the time taken for the analysis to complete.

The results show that across the benchmarks, using median values:

- 42.2% of all activity was side effect free
- 2.0% of all activity **had no effect**
- 25.3% of all writes were caused by `System.arraycopy`
- 78.4% of all captured writes came from the top ten capturing methods

5.1 Case Studies

To evaluate the full potential of our blended efficiency analysis we investigated several of the benchmarks in more detail. For each we used the results of our analysis to identify several optimisation opportunities. We implemented improvements for these opportunities and re-ran the benchmarks to confirm the improvement. Here we present the results of the first case study in some detail, before summarising the other case studies[2].

5.1.1 Case Study: h2

The `h2` benchmark runs a series of SQL load tests using the H2 pure Java relational database implementation. We ran our blended efficiency analysis and reviewed the top ten:

- side effect free methods (Table 4)

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

Table 3: Results for DaCapo benchmarks

Benchmark	CCT Nodes	Method Count	Cost *millions*	Output *thousands*	Effncy	Side Effect Free	Effect Free	Low Effncy	Captured Writes 1 Node	Captured Writes 1 Meth	Captured Writes 10 Meth	System arraycopy	Analysis Time (s)
lusearch	127,386	2,592	9,131	421,942	04.621	22.07%	00.86%	05.85%	46.77%	46.77%	98.34%	24.75%	2.8
avrora	271,610	3,448	8,434	600	00.007	26.22%	05.07%	06.34%	00.67%	00.69%	01.75%	00.88%	12.9
luindex	279,484	3,435	2,869	92,555	03.226	28.37%	03.01%	10.93%	42.31%	44.57%	97.18%	12.31%	6.2
h2	412,539	4,580	13,934	11,108	00.079	75.83%	10.25%	72.49%	44.80%	54.42%	96.42%	24.01%	7.9
sunflow	432,017	4,473	50,430	17	00.000	21.15%	00.80%	18.17%	14.24%	79.98%	99.27%	00.14%	9.4
xalan	506,441	4,545	9,218	237,653	02.578	36.24%	01.59%	14.40%	02.39%	28.62%	82.36%	21.78%	21.4
batik	782,742	7,645	2,549	18,339	00.719	14.33%	00.48%	05.19%	27.93%	27.93%	73.94%	16.27%	26.8
fop	814,713	7,572	1,015	16,642	01.639	46.60%	02.69%	21.82%	09.85%	14.81%	76.50%	40.67%	365.0
tomcat	3,360,093	13,802	5,488	92,593	01.687	41.24%	01.40%	17.85%	13.63%	17.00%	67.73%	57.46%	361.5
pmd	5,314,934	5,425	2,630	65	00.002	43.24%	03.34%	31.17%	01.69%	36.82%	85.96%	25.88%	129.9
tradebeans	9,859,665	29,764	28,187	85,040	00.301	63.53%	04.26%	31.48%	02.88%	27.05%	78.76%	47.54%	575.4
tradesoap	10,558,423	30,387	31,993	284,125	00.888	50.16%	02.01%	21.10%	03.41%	16.24%	71.72%	50.22%	569.9
eclipse	22,720,671	17,124	99,268	2,271,788	02.288	63.25%	01.15%	31.82%	05.04%	20.23%	77.96%	47.97%	1,579.6
jython	26,501,991	9,082	15,879	863	00.005	70.36%	01.97%	36.33%	05.72%	12.03%	64.74%	56.55%	1,264.1
Minimum						14.33%	00.48%	05.19%	00.67%	00.69%	01.75%	00.14%	
Median						**42.24%**	**01.99%**	**19.63%**	**07.79%**	**27.49%**	**78.36%**	**25.32%**	
Maximum						75.83%	10.25%	72.49%	46.77%	79.98%	99.27%	57.46%	

- effect free methods (Table 5)
- low efficiency methods (Table 6), and
- capturing methods (Table 7)

From these methods we targeted the following opportunities:

`JdbcResultSet.checkColumnIndex()` — The top four no effect methods are all closely related in that `checkColumnIndex` calls `checkClosed` on the result set which in turn calls `checkClosed` on the connection. We discussed the improvement we were able to make to `checkColumnIndex` in our motivating example in Section 2. We were able to almost completely eliminate the cost of `checkColumnIndex`.

Much of the remaining effect free cost is due to `TraceObject.debugCodeCall()`, which uses debug logging to record the value of every API method call. This method is effect free when debug level logging is disabled. This cost could be trivially eliminated from a production build if trace logging was never to be used in production.

`TPCC.calculateSumDB()` — This method was the top low efficiency method and the second highest capturing method in the benchmark. This is part of the DaCapo benchmark harness that generates the SQL load against the database. The method calculates a checksum for the database state so that it can be verified the database has been reset between each benchmark iteration. It uses a very inefficient procedure for creating the checksum, converting every field retrieved from the database into a string before processing the strings into a digested checksum. We were able to change the implementation to avoid this conversion into strings for number and timestamp fields, reducing the cost of `calculateSumDB` by 62%.

`ConditionAndOr.getValue()` — The third and fourth top low efficiency methods are part of the database engine that evaluates conditional matching clauses in SQL statements. The profile for these methods shows a very large proportion of their cost was being spent doing string comparisons. We changed the string comparison code to only perform a full comparison if the strings hashcode values matched, reducing the cost of `ConditionAndOr.getValue()` by 19%. This was effective because:

- the vast majority of comparisons are between strings that do not match, meaning the hashcode check can be used to cheaply avoid most full comparisons
- H2 internally caches and reuses string values, so the cost of calculating the hashcode value is amortised over the number of times the string is compared

`JdbcResultSet.getColumnIndex()` – Internally H2 stores the result set data in indexed arrays, so `getColumnIndex()` is used to convert a column or alias name into an index to retrieve the appropriate data. To do this efficiently the implementation builds a map from the column and alias names to column index values the first time it is called. However, because H2 uses case-insensitive column names, it blindly converts all column and alias names to upper case before storing them in the map. It therefore needs to convert the passed column name to upper case on every subsequent method call so it can check the map for the correct index. We found it much more efficient to instead use the map as a cache of index values for previously seen column names, and search for the column index by brute force when it is not found in the map. This avoids the need for the converting of strings to

Table 4: Top 10 No Side Effect Methods — h2

Method	%Cost
org.h2.index.BaseIndex.compareRows(SeachRow,SeachRow)	18.057
org.h2.value.ValueTimestamp.getString()	14.745
java.sql.Timestamp.toString()	14.585
org.h2.index.BaseIndex.compareValues(Value,Value,int)	10.068
org.h2.expression.Expression.getBooleanValue(Session)	09.535
org.h2.expression.ConditionAndOr.getValue(Session)	08.396
org.h2.expression.Comparison.getValue(Session)	08.133
org.h2.index.TreeIndex.findFirstNode(SearchRow,boolean)	08.006
org.h2.jdbc.JdbcResultSet.checkClosed()	06.183
org.h2.jdbc.JdbcResultSet.checkColumnIndex(int)	06.142

Table 5: Top 10 No Effect Methods — h2

Method	%Cost
org.h2.jdbc.JdbcResultSet.checkClosed()	06.183
org.h2.jdbc.JdbcResultSet.checkColumnIndex(int)	06.142
org.h2.jdbc.JdbcConnection.checkClosed(boolean)	03.547
org.h2.jdbc.JdbcConnection.checkClosed()	01.936
org.h2.message.TraceObject.debugCodeCall(String,long)	01.166
org.h2.message.TraceObject.debugCodeCall(String)	00.121
org.h2.command.Prepared.checkParameters()	00.075
org.h2.command.CommandContainer.recompileIfRequired()	00.017
org.h2.table.Column.updateSequenceIfRequired(Session,Val)	00.015
org.h2.table.TableData.checkRowCount(Session,Index,int)	00.015

Table 6: Top 10 Low Efficiency Methods — h2

Method	%Cost
org.dacapo.h2.TPCC.calculateSumDB(String,int)	48.969
org.dacapo.h2.TPCC.calculateSumDB()	48.969
org.h2.expression.ConditionAndOr.getValue(Session)	08.396
org.h2.expression.Comparison.getValue(Session)	08.133
org.h2.index.TreeIndex.findFirstNode(SearchRow,boolean)	08.006
org.h2.jdbc.JdbcResultSet.checkClosed()	06.183
org.h2.jdbc.JdbcResultSet.checkColumnIndex(int)	06.142
org.h2.expression.ExpressionColumn.getValue(Session)	04.998
org.h2.jdbc.JdbcConnection.checkClosed()	03.547
org.h2.result.Row.getValue(int)	02.321

Table 7: Top 10 Capturing Methods — h2

Method	Captured
java.sql.Timestamp.toString()	85,455,479
org.dacapo.h2.TPCC.calculateSumDB(String,int)	54,946,291
org.h2.jdbc.JdbcResultSet.getColumnIndex(Strg)	4,939,507
java.math.BigInteger.toString(int)	1,047,292
org.h2.jdbc.JdbcSQLException.buildMessage()	993,060
java.text.MessageFormat.applyPattern(String)	935,691
java.math.BigDecimal.divideAndRound(...)	814,314
org.apache.derbyTesting .system.oe.direct.Standard.payment(...)V	813,290
java.math.BigInteger.pow(I)	776,589
java.text.Format.format(Object)	698,562

upper case and the up front caching of every column name, when only a few columns may be requested. This change reduced the cost of `getColumnIndex` by 80%.

`JdbcSQLException.buildMessage()` — This method on the `JdbcSQLException` class is used to generate the message string returned from its `getMessage()` method. However `buildMessage` was often called multiple times during the construction of an exception as different pieces of information about the exception were added to the exception object, and then often the expensively constructed message was never used by the exception handler that ultimately received the exception. We changed the implementation to instead lazily construct the message if and when it was first requested. This change completely eliminated all calls to `buildMessage()` in the benchmark.

With these combined changes we were able to reduce the total runtime cost of the benchmark by 36%.

5.1.2 Case Study: fop

The `fop` benchmark uses the Apache FOP library to apply an XSL-FO stylesheet to an XML document to create a PDF document.

`AbstractLayoutManager.addChildLM` — The top capturing write method in the benchmark is responsible for adding a child layout manager to an existing parent. However the method constructs a string for a trace level logging call that was not guarded by a check that trace level logging was enabled, so the string was expensively constructed on every call even though in practice it was never used. We guarded the construction of the string with a `log.isTraceEnabled()` call and reduced the cost of this method by 97%.

`Glyphs.charToGlyphName` — The top six low efficiency methods recorded in the benchmark were all related to this one method that is used during the postscript header generation. During the encoding of font information in the postscript header several common character sets are encoded using information loaded from text files. This encoding information was loaded into a number of large arrays which were then searched in a brute force manner for the correct encoding information, resulting in many thousands of string comparisons. We loaded the encoding information into maps instead, resulting in a much more efficient lookup process, reducing the cost of this method by 99%.

`PSRenderer.renderText` – The second top capturing write location is responsible for generating the correct postscript command for generating a specific piece of text. This method takes an input string and constructs another (usually very similar) string from it, applying font mapping and postscript character escaping. For many values the input text was unchanged. We added code to handle this common case much more efficiently, falling back to the original behaviour when necessary, reducing the cost of method by 89%.

With these three changes we were able to reduce the overall runtime cost of the benchmark by 28%.

5.1.3 Case Study: luindex

The `luindex` benchmark uses the Apache Lucene library to index a set of text documents.

`Token.initTermBuffer` — The top effect free method in the benchmark is responsible for checking and maintaining a consistent internal state in the `Token` data structure. This data structure is used to represent a string and some additional meta data about the context in which the string occurred. The string value can be initialised as either a Java `String` or `char []` and the data structure has different fields that store these values respectively. The `initTermBuffer` method is used internally to convert the Java `String` value (if it exists) to the equivalent `char []` value as necessary. The majority of the methods on `Token` wish to process the string value as a `char []` and therefore `initTermBuffer` is called very frequently to ensure the `char []` value is available. Given that the value is almost always a `char []` already, often this work is unnecessary. We changed the implementation to convert the Java `String` value to a `char []` when the value was initialised so that we then never had to check a later point if the `char []` value existed. This removed the cost of this method entirely from the benchmark and reduced the overall cost of the benchmark by 4%.

6. DISCUSSION

The overall results show that there is a significant proportion of the activity in the benchmarks that is side effect free, effect free or has low efficiency. This is important because each of these represent potential optimisation opportunities. With side effect free activity the implementation can be treated as a black box that can have its results cached or be completely replaced. Effect free methods represent behaviour that could potentially be eliminated completely as it is having no practical benefit. Low efficiency methods have a poor cost / benefit ratio, meaning they too represent potential optimisation opportunities, albeit ones that are more difficult to realise than with either side effect free or effect free methods.

Interpreting the overall application efficiency measure for the benchmarks needs to be done with caution. Whilst we feel that our chosen measure has value for many real-world applications it is not appropriate for some of the benchmarks (e.g. `avrora`, `pmd` and `jython`) that are not focussed on producing output. In the other benchmarks it perhaps gives some insight into the nature of the benchmark rather

than being a fair absolute measure of efficiency. For example the low efficiency number for sunflow is understandable given the computational nature of graphics rendering. The overall efficiency measure is useful when comparing similar applications, or different releases of the same application.

As was evident in the case studies, measuring and comparing efficiency at the individual method level within an application is more valid. Activity within a single application is more homogeneous, making efficiency outliers more unusual and interesting.

In our case studies we highlighted a selection of the optimisation opportunities we found and how we were able to effectively address them. These were not the only optimisation opportunities that we discovered, there were numerous others, and in analysing these we noted a number of trends.

String processing is often inefficient – Many of the optimisation opportunities we found involved the processing and manipulation of strings. There seemed to be two root causes for many of these problems:

1) Transcoding between the JVM's UTF-16 based char representation and the most common 8-bit character sets (e.g. ISO-8859-1 or UTF-8). Frequently input data would be received as UTF-8, be decoded to a Java string, passed around and manipulated before being encoded back to UTF-8 on output. If the underlying representation of a string in Java had been based on UTF-8 rather than UTF-16 then much of this transcoding would be unnecessary.

2) APIs that use Java strings instead of mutable `StringBuilder` objects. There are many routines that build up a large text results out of smaller values but many of these accept and return the immutable Java `String` object as values. Consequently when the results of these methods are composed the character content is copied into a new string object each time. If more of these methods accepted `StringBuilder` or `CharBuffer` objects then much of this repeated copying could be avoided.

No effect methods are common – We found a great number of methods that were having no practical effect. In almost all cases the method would check some condition to determine whether or not to take an action, and in practice the condition never held true. The most common examples were assertion checkers such as `ArrayList.rangeCheck()`, responsible for raising an exception if some condition did not hold, and debug logging methods. Often these can be eliminated with the trade-off of less flexible logging or less precise exception reporting.

Captured write locations are often quick wins – Through our analysis we were able to find different types of optimisation opportunities but often the top captured write methods were the easiest to address. Locations with a high number of captured writes naturally contained code that performed work that was then discarded. Often it was possible to quickly understand what work was being wasted and could be somehow avoided or reused.

Inefficient activity is very localised – Our results show that the top ten capturing methods accounted for a median value of 78.4% of all captured writes in each benchmark. This means that an engineer often has only a very few, usually high value, locations to inspect to attempt to substantially improve efficiency.

System.arraycopy is the source of many writes – Across our benchmarks `System.arraycopy` accounted for a median value of just over 25% of all writes. For the benchmarks most similar to many real world applications (`tomcat`, `tradebeans`, `tradesoap`, and `eclipse`) this was even higher, accounting for at least 47% of all writes. This is important because it shows that `arraycopy` is a significant hotspot for many applications and yet in our experience most Java profilers fail to show this, probably because it is a native method. In our experiments we had no evidence that `arraycopy` was such an important source of activity until we specifically customised the JP2 profiler to record its writes.

6.1 Threats to Validity

The most obvious threat to validity for our analysis is that our measure of value may not accurately reflect a true measure of value. In our case studies we have found the concept of escaping writes to be intuitive and useful. It seems to naturally quantify the amount of useful work a method is producing and also works well to explain (via captured writes) how work done by some methods is then wasted by a calling method. Our value measure however does not reflect the complexity of the data escaping a method. For example a method which compresses a large byte array before returning it would be valued less than one that did not, even though intuitively the results are equivalent.

There are also some sources of imprecision in our analysis which can lead us to overstate the value being provided by a particular execution path.

Our analysis does not account for situations where a single field is being repeatedly overwritten with new values. In this case the actual amount of new data escaping a method may be less than the number of writes the method is performing. Ideally we would like to classify the writes that are overwritten as being captured.

The approach we take to handling object field references again means that we can overstate the value provided by many methods. A future improvement would be to more precisely model the use of field references so we can track the escape status of writes to individual fields. Presently all writes to any field of an object are regarded as escaping a method if any other field in the object escapes the method.

We do not distinguish between different types of output. For example debug logging being sent to a file on disk is categorised in a similar manner to an HTTP response being sent over the network. It is likely that these two types of output would be valued differently by an engineer investigating the performance of their application.

Despite these sources of imprecision causing us to overvalue some execution paths we were still able to find real optimisation opportunities. Our analysis is deliberately conservative so that none of these problems cause us to understate the value of a method. Therefore we can take our current measures for the percentage of activity that is side effect free, effect free and low efficiency as lower bounds on their real values. Future work to help improve the accuracy of our analysis should make it even more effective.

Finally we would like to evaluate our efficiency analysis on real world large scale object oriented software. We are encouraged by the results we have achieved with the DaCapo benchmarks, which we believe are representative of many Java applications and therefore we are confident that our results are applicable for many real world applications. However we would like to validate our approach with a real world case study.

7. CONCLUSION

Existing performance analysis tools and approaches focus on understanding the distribution of runtime costs in an application. However without understanding the value being provided in return for this expense it is difficult to establish where effort is being wasted and where optimisation opportunities might truly exist. In this paper we present a blended analysis approach to quantifying the value provided by all execution paths in an application, thereby enabling an analysis of their efficiency and a practical cost/benefit approach to performance analysis. This allows the discovery of new optimisation opportunities not readily apparent from the original profile data. The results of our experiments and the performance improvements we made in our case studies demonstrate that efficiency analysis is an effective technique that can be used to complement existing performance engineering approaches.

8. ACKNOWLEDGMENTS

David Maplesden is supported by a University of Auckland Doctoral Scholarship.

9. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *Proc. of the Conf. on Prog. Language Design and Impl.*, pages 85–96, 1997.

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

[3] 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 Conf. on Object Oriented Prog. Systems Languages and App.*, pages 169–190, 2006.

[4] B. Blanchet. Escape analysis for object-oriented languages: application to java. *Proc. of the Conf. on Object Oriented Prog. Systems Languages and App.*, pages 20–34, 1999.

[5] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O. Sullivan, T. Parsons, and J. Murphy. Patterns of Memory Inefficiency. *Lecture Notes in Comp. Sci.*, 6813:383–407, 2011.

[6] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. *Proc. of the Conf. on Prog. Language Design and Impl.*, pages 516–527, 2011.

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

[8] 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 Int'l Symp. on Foundations of Soft. Eng.*, pages 59–70, 2008.

[9] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy. Performance Analysis for Object-Oriented Software: A Systematic Mapping. *IEEE Transactions on Soft. Eng.*, 41(7):691–710, 2015.

[10] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy. Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software. *6th ACM/SPEC Int'l Conf. on Performance Engineering*, pages 175–186, 2015.

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

[12] 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*, page 85, 2005.

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

[14] K. Nguyen and G. Xu. Cachetor: detecting cacheable data to remove bloat. *Proc. of the 9th Joint Meeting on Foundations of Soft. Eng.*, pages 268–278, 2013.

[15] 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.

[16] 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 Int'l Conf. on Principles and Practice of Prog. in Java*, page 11, 2011.

[17] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. *Proc. of the Conf. on Prog. Language Design and Impl.*, pages 408–418, 2009.

[18] G. Xu. Finding reusable data structures. *Proc. of the Conf. on Object Oriented Prog. Systems Languages and App.*, page 1017, 2012.

[19] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. *Proc. of the Conf. on Prog. Language Design and Impl.*, pages 174–186, 2010.

[20] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Transactions Soft. Eng. Methodology*, 23(3):23:1–23:50, 2014.

[21] 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 Soft. Eng. Research*, pages 421–425, 2010.

[22] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. *Proc. of the Conf. on Prog. Language Design and Impl.*, pages 160–173, 2010.

[23] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. *Int'l Conf. on Soft. Eng.*, pages 134–144, 2012.