2015 International Conference on Intelligent Systems and Knowledge Engineering

Automated Outlier Removal for Mobile Microbenchmarking Datasets

Adam Rehn
James Cook University
Cairns, Australia
adam.rehn@jcu.edu.au

Jason Holdsworth
James Cook University
Cairns, Australia
jason.holdsworth@jcu.edu.au

Ickjai Lee
James Cook University
Cairns, Australia
ickjai.lee@jcu.edu.au

***Abstract*—Microbenchmarking is a useful tool for fine-grained performance analysis, and represents a potentially valuable tool in the development of mobile applications and systems. However, the fine-grained measurements of microbenchmarking are inherently susceptible to noise from the underlying operating system and hardware. This noise includes outliers that must be removed in order to produce meaningful results. Existing microbenchmarking implementations utilise only simple mechanisms for removing outliers. In this paper we propose a heuristic for the automated removal of outliers from mobile microbenchmarking datasets. We then simplify this heuristic for use on mobile devices. Empirical evaluation demonstrates that our outlier removal heuristics are effective across microbenchmarking datasets collected from a range of mobile devices. Our simplified heuristic operates in log-linear time, making it suitable for use on resource-constrained mobile devices. The ability to perform outlier removal on-device without the need for post-processing on desktop or server hardware enhances the utility of mobile microbenchmarking tools. Our results present interesting opportunities for further studies across a broader range of device platforms.**

***Keywords*—outlier removal; mobile; microbenchmarking**

I. Introduction

Microbenchmarking is the performance analysis of individual components or small subsets of an application [1]. Just as application-wide benchmarking is useful in evaluating the real-world performance of entire applications, microbenchmarking can aid developers in measuring the performance of individual application components. Such components could include the responsiveness of user interface interactions, which have been shown to play a significant role in users' perception of mobile applications [2]. As such, microbenchmarking represents a useful tool in the development of mobile platforms and applications.

Nevertheless, fine-grained measurement on small timescales presents additional complexities that must be addressed in order to produce meaningful results. The inherent nature of modern hardware and operating systems introduces noise into microbenchmarking measurements [3]. This noise can include outliers of varying levels of magnitude. Existing microbenchmarking approaches frequently utilise naive approaches to counteract the influence of outliers [1], [4]. Even though there exist many studies in general outlier detection [5], we are aware of no existing work that explores a fully unsupervised method of removing outliers from microbenchmarking data in a mobile context.

In this paper we examine microbenchmarking data collected from numerous mobile devices and identify common characteristics. We propose an automated algorithm for identifying outliers in microbenchmarking datasets. Our heuristic algorithm is based on the observed characteristics of our collected mobile microbenchmarking datasets. We then develop a simplified heuristic suitable for use on mobile devices.

The contributions of this research are as follows:

- We analyse the characteristics of microbenchmarking data collected from a variety of mobile devices. We are aware of no previous study that provides a detailed discussion of microbenchmarking data characteristics across a range of mobile device hardware;
- Our proposed automated outlier detection algorithm is highly amenable to simplification for use on resource-constrained mobile platforms. Evaluation using a number of existing metrics demonstrates the effectiveness of our algorithm;
- The simplified variation of our proposed algorithm runs in $O(n \log n)$ time, making it highly scalable and suitable for use on mobile devices with limited resources. This allows automated outlier removal to be performed on-device, providing meaningful results without the need to transfer collected data to another device for post-processing.

The rest of the paper is organised as follows. First, we provide a background on the cause of noise in microbenchmarking data, and discuss how existing approaches deal with this noise. We motivate our research by describing the unique considerations that must be taken into account on mobile platforms, and examine the characteristics of collected mobile microbenchmarking datasets. We then describe our proposed outlier removal algorithm, and evaluate its effectiveness using existing metrics. Finally, we develop a simplified variation of our outlier removal algorithm, and evaluate both its effectiveness and computational efficiency in comparison to the full algorithm.

II. Related work

Noise in microbenchmarking data arises from the complexities of measuring the wall-clock performance time of

978-1-4673-9323-2/15 $31.00 © 2015 Crown Copyright
DOI 10.1109/ISKE.2015.55

extremely fine-grained tasks. Modern consumer operating systems are designed heavily for multitasking. Processors must constantly switch between running applications, system services, and servicing hardware communication. This results in small perturbations to application performance. The performance variation caused by system activity is referred to as Operating System (OS) noise [6]. OS noise arises from numerous sources, including process scheduling, interrupts, and OS clock ticks [7], and is an inherent side-effect of modern OS design. The magnitude of OS noise is typically on the order of sub-second time-scales [8], [9], and hence of importance primarily when performing extremely fine-grained performance measurements.

OS noise only affects the wall-clock performance time of applications, and has no effect on other measurements such as CPU time. CPU time measures the number of CPU cycles spent executing both user-mode and kernel-mode instructions for a process. This measurement excludes time spent executing other processes, thus discarding the influence of OS noise. CPU time is commonly used by profiling tools, whose purpose is to analyse the relative complexity of an application's components with respect to one other, and identify performance bottlenecks [10]. In contrast, the purpose of benchmarking and microbenchmarking is to measure the real-world performance of an application on a specific hardware configuration. Accordingly, microbenchmarking tools utilise wall-clock time, resulting in their susceptibility to the effects of OS noise.

Most existing microbenchmarking tools utilise only extremely simple approaches for addressing the effects of OS noise. The tools *mhz* [4] and *lmbench* [1] both implement robustness against outliers by reporting the median of measured times instead of the mean. In the mobile context, *AndroBench* [11] utilises the three sigma rule to remove outliers. The use of this heuristic implies the assumption of a normal distribution, which is rarely the case for benchmark timing data [1]. The most sophisticated outlier removal we observe is that of the microbenchmarking tool by Jamal and Waheed [12], which utilises k-nearest neighbour clustering to remove outliers. However, this tool is designed purely for traditional desktop platforms, not for a mobile context.

There is a clear lack of microbenchmarking tools for mobile platforms that utilise a comprehensive approach to outlier removal. The development of an outlier removal approach for mobile microbenchmarking must take into account the constraints of mobile platforms. Limited memory and processing power dictate that computational complexity and memory usage be kept to a minimum. This precludes the use of complex algorithms. We propose a heuristic based on the characteristics of the data being processed is best suited to satisfying the constraints of microbenchmarking on mobile devices. In the following section, we analyse microbenchmarking datasets that we collect from real mobile devices, and analyse the common characteristics of the data.

III. Data characteristics

To analyse the characteristics of mobile microbenchmarking data, we collect microbenchmarking datasets from a series of real mobile devices. Our collection harness measures the levels of OS noise present on a device by microbenchmarking the performance of querying the system time. For consistency, we collect data from devices all running the Apple iOS operating system. These devices include the fourth-generation iPod Touch, the iPhone 5, and both the third-generation iPad and fourth-generation iPad Air.

Each dataset contains 5000 samples. After initial testing with various sample sizes, we select this sample size as a trade-off that allows us to collect enough data to perform meaningful analysis, whilst not exceeding the limited memory capacities of the older device models. The collected data possesses a number of characteristics that instruct the choice of an outlier removal approach. These characteristics include the data distribution and clustering structure, and the types of outliers present.

The distribution of the data varies greatly between datasets. We measure the skewness of each dataset using the medcouple, a robust measure of skewness [13]. Medcouple values range from -1 to 1. Negative values indicate a distribution that is skewed to the left, whilst positive values indicate a distribution that is skewed to the right. A medcouple value of zero indicates a symmetric distribution [13]. The medcouple values for the collected datasets span the full range possible, demonstrating varying levels of skewness to both the left and right. This variation precludes the use of distribution-based outlier detection approaches.

Visual inspection of the collected data reveals that all of the datasets contain multiple clusters of varying density. We observe that the majority of the data is typically contained in a small number of extremely dense clusters, which are often nested within a surrounding cluster of a lower density. The innermost cluster almost always contains the median, which is consistent with existing observations of microbenchmarking data [1]. The density of the clusters containing the inliers is largely consistent among mobile devices of the same hardware model, but varies across different hardware models. Figure 1 depicts example datasets for several different mobile device models. The presence of clusters nested within other clusters necessitates the use of a hierarchical clustering algorithm, which can represent these nested relationships.

We observe two distinct types of outliers in the collected data. The first type of outliers are data points that clearly do not belong to any cluster. We refer to these as "standalone" outliers. The second type are data points that form small, low-density clusters. These clusters are extremely small with respect to the sample size, and have a low density when compared to the clusters that contain the majority of the data. We refer to these as "clustered" outliers.

The distinction between the two types of outliers is based on visual inspection. To confirm this distinction as being meaningful, we validate it using existing metrics. We compute

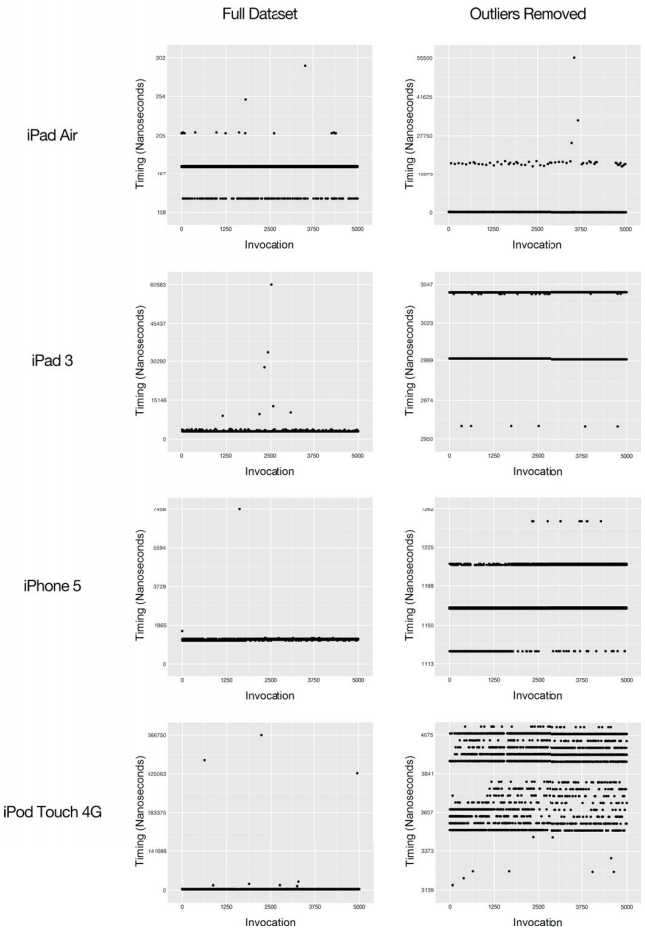

Figure 1. Example datasets for each of the mobile device hardware models. The top row depicts each dataset in full, whilst the bottom row depicts each dataset once outliers have been removed.

the Local Outlier Factor (LOF) [5] for the points in all of the collected datasets. LOF measures the extent to which each data point is isolated from its local neighbourhood, and assigns it a value to represent the degree to which it is an outlier [5]. The local nature of LOF makes it ideally suited to identifying noise points that are distant from the nearest cluster. LOF has one parameter, *MinPts*, which determines the number of surrounding points that constitute a point's local neighbourhood. When computing the LOF values for the collected datasets, we use a *MinPts* value of 10, which is the lowest value recommended [5]. We find that the points identified as standalone outliers are consistently assigned LOF values greater than 1.0, whilst clustered outliers are consistently assigned LOF values <= 1.0. We also observe that there almost always exists at least one border point in each outlier cluster that is identified as a standalone outlier, based on its LOF value.

Formally, we define inliers and the two types of outliers as such:

Definition 1: For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, a **standalone outlier** is a point $p_i \in P$ not belonging to any cluster, having an LOF value > 1.0.

Definition 2: For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, a **clustered outlier** is a point $p_i \in P$ belonging to a cluster with a size < 1% of the dataset size (|P|), having an LOF value $\leq$ 1.0.

Definition 3: For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, an **inlier** is a point $p_i \in P$ belonging to a cluster with a size > 1% of the dataset size (|P|), having an LOF value $\leq$ 1.0.

Examples of inliers and the two types of outliers are illustrated in Figure 2.

In a small number of instances, we observe data points that resemble standalone outliers in the left tail of the dataset. It is important to note that these points should not be considered outliers. For any given task, the minimum possible time in which it can be performed is determined by the processor speed. OS noise can only increase the amount of time taken to perform processing above this minimum. Accordingly, although there is no theoretical upper bound to the performance timing of a task, there is always a hard lower bound [1]. The only exception to this rule is when the system or device is not in a steady-state [14]. However, our data collection methodology ensures all of our datasets are collected during steady-state operation.

IV. Outlier removal algorithm

Due to the presence of nested cluster relationships within the collected data, a hierarchical clustering algorithm is necessary to capture the clustering structure [15]. We select agglomerative hierarchical clustering using the complete linkage metric. Complete linkage is selected because it is less sensitive to the presence of outliers than other metrics such as single-linkage [16], and due to the availability of an efficient implementation [17]. We utilise Minkowski distance as the dissimilarity metric, which is equivalent to Manhattan distance in this context, as our data is univariate [18].

Agglomerative hierarchical clustering forms clusters from data points by iteratively increasing the neighbourhood distance at which objects are considered to be part of the same cluster [17]. This process halts once a neighbourhood distance has been reached that results in the entire dataset being placed in a single cluster. The output of hierarchical clustering is a data structure known as a dendrogram. Dendrograms are tree structures, wherein the leaf nodes represent the individual data points, and each interior node represents a cluster containing all of the leaf nodes that are its descendants [15]. Each interior node is marked with a "height", which is the neighbourhood distance at which the cluster that the node represents was formed. The root node of the tree represents a single cluster containing the entire dataset [18].

To extract clusters from a dendrogram, the tree is typically "cut" at a particular height. Cutting a dendrogram at a given height produces the clustering corresponding to the neighbourhood distance represented by the height [15]. Since a dendrogram represents the entire hierarchical clustering structure of a dataset, the choice of cut height significantly impacts the number of extracted clusters. Cuts closer to the

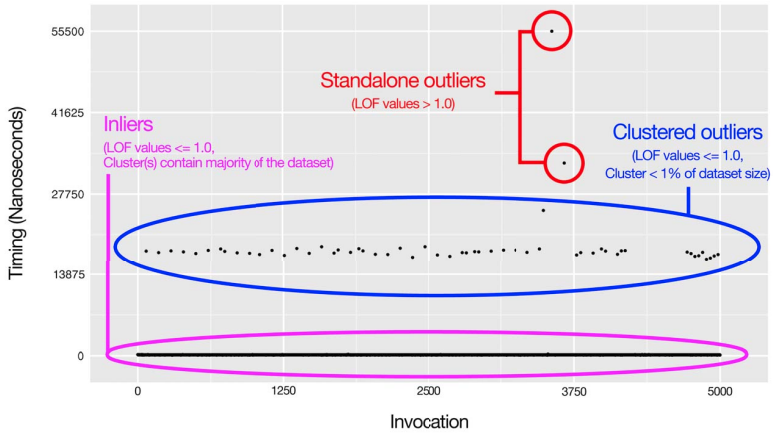

Figure 2. Example dataset highlighting both inliers and the two types of encountered outliers.

root of the tree will result in fewer clusters, whilst cuts closer to the leaves will result in a greater number of clusters. The choice of cut height is typically specific to the data being clustered, and is often selected manually [18]. Automating the choice of cut height allows the clustering process to become fully unsupervised. To automate the choice of cut height for mobile microbenchmarking datasets, we develop a heuristic that takes into account the observed characteristics of the data.

We aim to satisfy the following goals in developing our outlier detection heuristic:

- Outlier removal should be fully automated and require no manual intervention;
- Both standalone and clustered outliers should be removed; and
- Any complex analysis performed on the data should be able to be precomputed, in order to achieve simplified complexity when performing outlier removal on mobile devices.

The last goal is of particular importance, as it restricts the types of processing that can be performed. It precludes the use of complex algorithms whose output cannot be used to discover relationships between the underlying data and the optimal choice of cut height. A number of existing approaches for unsupervised extraction of clusters from dendrograms have been proposed [19], [20], [21], but these algorithms focus on characteristics of the dendrogram alone, and not on relationships between the dendrogram and the underlying dataset. Ensuring that any discovered relationships can be simplified into a cheaply computed representation facilitates performing outlier removal on actual mobile devices. The ability to perform outlier removal on-device greatly increases its utility to mobile microbenchmarking applications.

To ensure the removal of both standalone and clustered outliers, we utilise different methods for detecting each type. Standalone outliers are easily identified by computing the LOF values for a dataset. Based on our observations, those points with LOF values greater than 1.0 are standalone outliers. To remove clustered outliers, we utilise a threshold for the minimum size of a cluster. The points in any cluster that is smaller than the threshold value are marked as outliers. This threshold also removes standalone outliers when they are clustered alone or with neighbouring outlier clusters. To address the fact that outliers can only exist in the right-hand tail of the data, we only mark clusters as outliers when the minimum value in the cluster is greater than the median of the dataset. Based on analysis of the data, we set the minimum cluster size threshold at 1% of the size of the dataset.

To automate the selection of a dendrogram cut height, we exploit the relationship between standalone and clustered outliers. For almost all collected datasets, we observe that each outlier cluster contains at least one border point that is identified as a standalone outlier by its LOF value. Early in the agglomerative clustering process, these border points are clustered with their neighbouring outlier cluster. As the clustering process progresses, the outlier cluster itself is eventually clustered with other clusters, until the point at which it is clustered with inliers. By examining how many points with LOF values greater than 1.0 are marked as outliers, we can identify the stage at which outliers start to be clustered with inliers. The neighbourhood distance immediately preceding this stage represents the highest level of clustering that partitions outliers and inliers separately. We refer to this neighbourhood distance as the "ideal" cut height. It is this cut height that we wish to extract.

Definition 4: The **ideal cut height** for a dataset is the greatest cut height which results in a clustering that partitions outliers and inliers separately.

Once hierarchical clustering has been performed on a dataset, the algorithm to identify the ideal cut height is relatively straightforward. First, we extract the set of unique cut heights from the dendrogram. We refer to this set as the "cut candidates" for the dendrogram.

Definition 5: The **cut candidates** for a dendrogram are the set of unique cut heights that exist in that dendrogram.

We then sort the list of cut candidates. Sorting the list effectively assigns each cut candidate a ranking value equal

Algorithm 1 Full outlier removal heuristic.

Input: threshold - the minimum cluster size threshold
Input: dataset - the dataset that cas been clustered
Input: D - the dendrogram, $D = (V, E)$
Output: the height at which to cut the dendrogram

function HEURISTICFULL(threshold, dataset, D)
  $Candidates \leftarrow \emptyset$
  for v **in** V **do**
    if children[v] $\neq \emptyset$ **and** height[$node$] $\notin Candidates$ **then**
      $clusters$ = CUTDENDROGRAM(D, height[$node$])
      $sumLOF \leftarrow 0$
      $numSamples \leftarrow 0$
      for $cluster$ **in** $clusters$ **do**
        if $|cluster| \geq threshold$ **or** min(cluster) $\leq$ median(dataset) **then** ▷ If cluster members are inliers
          $sumLOF \leftarrow sumLOF +$ LOF($cluster$)
          $numSamples \leftarrow numSamples + |cluster|$
        end if
      end for
      $cleanedLOF \leftarrow sumLOF \div numSamples$
      $Candidates \leftarrow Candidates \cup \{\text{height}[v], cleanedLOF\}$
    end if
  end for
  $minLOF \leftarrow \min(l \mid \{h, l\} \in Candidates)$
  $Candidates' \leftarrow \{h, l\} \in Candidates \mid l = minLOF$
  find $\{h, l\} \in Candidates' \mid h = \max(x \mid \{x, y\} \in Candidates')$
  return h
end function

to its position in the sorted list. The cut candidate with the best rank represents the ideal cut height. The pseudocode for the heuristic is listed in Algorithm 1.

V. Simplified heuristic for mobile devices

In the section that follows, we analyse the generated ranked lists and produce a simplified representation of the identified relationships for use on mobile devices. The use of a ranking-based approach was selected because the ranked list of cut candidates represents an output that can be further analysed to identify relationships within the collected data.

To determine if our outlier removal heuristic can be simplified for use on mobile devices, we analyse the ranked lists of cut candidates for the collected datasets. For each cut candidate, the number of outliers identified is recorded. We observe that ranges of neighbouring cut heights frequently identify the same number of outliers as one another. This suggests that in each dendrogram there exists a number of contiguous ranges of cuts whose clusterings can be treated as equivalent in regard to their outlier detection result. Most importantly, for most of the collected datasets we observe that there exists a range of cut heights that all identify the same number of outliers as the ideal cut height. The relationships between these ranges represent a potential simplification of our heuristic.

Before it is possible to identify any inter-dataset relationships, it is necessary to normalise the cut heights to produce values that facilitate comparison. Cut heights represent neighbourhood distances in the coordinate space of the distance

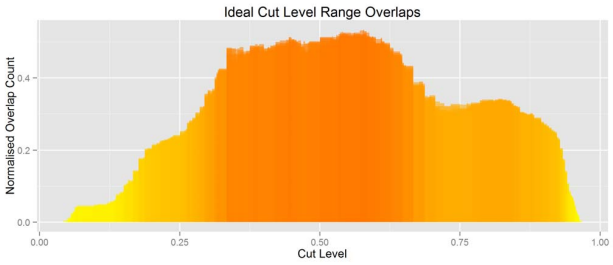

Figure 3. The unique intersections between the ideal ranges of the collected datasets. The y-value and colour saturation of each bar represents the normalised overlap count for that intersection.

metric used to compare points in the underlying dataset [15]. The simplest approach is to normalise cut heights with respect to the maximum cut height present in their respective dendrogram. However, since this maximum is equal to the depth of the entire dataset, the resulting normalisation is affected by the scale of the underlying data. To alleviate this, we instead normalise cut heights based on their position in the list of unique cut heights extracted from their respective dendrogram. We refer to this normalised value as the "cut level". The use of cut levels instead of cut heights effectively rescales dendrogram space to eliminate any influence from the scale of the underlying data.

Definition 6: A **cut level** is a cut height normalised with respect to its position in the ordered set of cut candidates.

Algorithm 2 The simplified version of the outlier removal heuristic.

Input: D - the dendrogram, $D = (V, E)$
Input: rankcurve - the curve with which to rank cut levels
Output: h - the height at which to cut the dendrogram

function HEURISTICSIMPLIFIED(D)
  $Heights \leftarrow \emptyset$
  $RankedHeights \leftarrow \emptyset$
  for v **in** V **do**
    if children[v] $\neq \emptyset$ **and** height[v] $\notin Heights$ **then**
      $Heights \leftarrow Heights \cup$ height[v]
    end if
  end for
  for $height$ **in** $Heights$ **do**
    $cutLevel \leftarrow$ index[$height$] / $|Heights|$
    $RankedHeights \leftarrow RankedHeights \cup \{height,$RANKCURVE($cutLevel$)$\}$
  end for
  find $\{h, r\} \in RankedHeights$ **such that** $r = \max(y \mid \{x, y\} \in RankedHeights)$ ▷ Find the height with max rank
  return h
end function

For each dataset, we identify the upper and lower bounds of the range of cut levels that identify the same number of outliers as the ideal cut height. We refer to this as the "ideal range" for that dataset. We then compute the set of unique intersections of the ideal ranges for all of the collected datasets. For each intersection, we count the number of datasets whose ideal range is a superset of the intersection. We refer to this value as the "overlap count". The overlap count for an intersection represents how frequently the range of cuts represented by the intersection yields the same number of outliers as the ideal cut height. Figure 3 depicts the intersections' ranges plotted against their overlap counts, which have been normalised with respect to the number of collected datasets.

The visualisation of the overlap counts reveals a distribution that is largely symmetric around the centre, indicating that cut candidates whose cut level is between 0.3 and 0.6 most frequently result in the ideal number of outliers being detected. The presence of this relationship facilitates the simplification of our heuristic to a simple curve that ranks a list of cut candidates according to their cut level. Sorting can be replaced with simple evaluation of a curve value for each cut candidate. The pseudocode for this simplified, curve-based heuristic is listed in Algorithm 2.

In order to utilise the simplified heuristic, a curve must first be selected. We fit a sixth-degree polynomial to the silhouette of the overlap count plot, which results in a curve that follows the silhouette very closely. However, we intuit that the overall maxima of the curve is more important than the rest of its shape, as the peak indicates the cut levels that will be ranked the highest. To validate this intuition, we use the standard bell curve of the normal distribution. This curve is simple to model, and its peak aligns closely with the silhouette of the overlap count plot. The two curves are shown in Figure 4.

To determine which curve best represents the trends represented by the overlap count plot, we compare the outlier

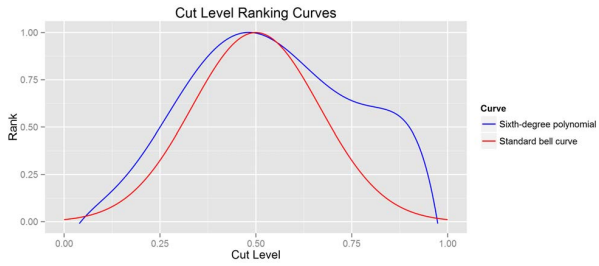

Figure 4. The two cut level ranking curves we evaluate for use with the simplified heuristic.

Table I
NORMALISED MEAN OUTLIER DETECTION ERROR VALUES FOR THE TWO CURVES.

Curve	Normalised mean outlier detection error
Sixth-degree polynomial	0.004356
Standard bell curve	0.004354

detection results of the two curves with respect to the full heuristic. We rank the lists of cut candidates for all collected datasets using each curve in turn. The number of outliers detected by the highest ranked cut candidate is compared in each case to the number of outliers detected by the ideal cut candidate for that dataset. We then normalise the outlier detection error values with respect to the maximum possible outlier error, which is equal to the size of the dataset. The mean error values for both of the curves are listed in Table I.

Interestingly, the mean error value for the standard bell curve is slightly lower than the mean error value for the polynomial curve. This result suggests that matching the exact silhouette of the overlap counts plot is less important than matching the overall distribution. This result also validates

our intuition that the position and shape of the maxima of the curve are the most important features. The low mean outlier detection error value indicates that the curve is indeed a fairly close approximation of the behaviour of the full heuristic.

A. Evaluation

1) Outlier removal effectiveness: In this section we validate the effectiveness of our outlier removal approach. For each collected dataset, we examine the original polluted dataset, the version of dataset after outlier removal using the full heuristic, and the version after outlier removal using the simplified heuristic. We treat each version of the dataset as a single cluster. This allows us to utilise existing measures of cluster compactnes to quantify the effect of outlier removal. We compute values for the following measures:

- Depth (convex hull);
- Radius;
- Total Within-Group Distance (TWGD); and
- The absolute difference between the cluster's centroid and its medoid.

The values for these metrics are listed in Table II. All four metrics demonstrate a large difference between the polluted versions of the collected datasets and the cleaned versions. Very little difference exists between the results for the full heuristic and the simplified heuristic. The results of these metrics confirm the effectiveness of our outlier removal approach, and also demonstrate that the simplified heuristic performs just as well as the full heuristic, when an appropriate cut ranking curve is utilised.

Table II
RESULTS FOR THE FOUR COMPACTNESS METRICS: TWGD, DEPTH, RADIUS, AND THE DIFFERENCE BETWEEN THE CENTROID AND THE MEDOID.

Metric	Polluted Dataset	
	Mean	Std. Dev
TWGD	2325035669.78	2860970117.61
Depth	41017.59	247786.01
Radius	1034.84	3719.54
Diff Centroid Medoid	58.55	113.78
	Cleaned Dataset (Full Heuristic)	
	Mean	Std. Dev
TWGD	598681922.73	526122255.17
Depth	381.37	1085.84
Radius	47.97	42.42
Diff Centroid Medoid	14.77	11.64
	Cleaned Dataset (Simplified Heuristic)	
	Mean	Std. Dev
TWGD	594372693.58	563944711.27
Depth	237.15	198.79
Radius	46.16	41.67
Diff Centroid Medoid	15.03	12.59

2) Algorithm efficiency: We determine the time complexity for both the full and simplified heuristic by performing theoretical complexity analysis on the algorithms' pseudocode. The time complexity for both algorithms is listed in Table III. Note that the complexity listed excludes the construction of the dendrogram itself, which will vary in complexity depending on the clustering algorithm used [17].

Table III
TIME COMPLEXITY FOR THE FULL AND SIMPLIFIED OUTLIER DETECTION HEURISTICS, EXCLUDING DENDROGRAM CONSTRUCTION.

Heuristic	Worst-case time complexity
Full	$O(n^2)$
Simplified	$O(n \log n)$

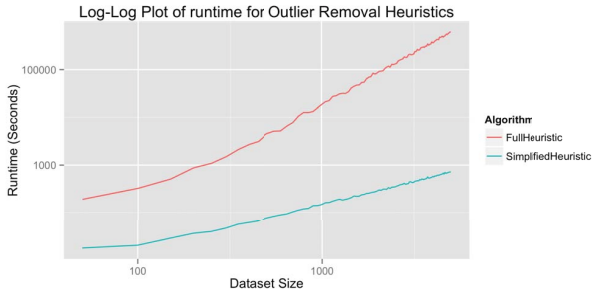

Figure 5. Log-log plot of the average runtime for the full heuristic and the simplified heuristic, excluding dendrogram construction.

The full heuristic operates in $O(n^2)$ time in the worst case. This is due to the fact that the dendrogram must be cut once for each unique cut height. Cutting the dendrogram has $O(n)$ complexity in all of the implementations we have tested. In the worst case scenario the number of unique cut heights in the dendrogram will be equal to the size of the dataset. It is worth noting, however, that in all of our collected datasets, the number of unique cut heights was in fact quite small with respect to the size of the dataset. The worst case scenario can only occur when the set of pairwise distances between all points in a dataset contains at least as many unique distances as there are data points. This scenario is not typical of the observed characteristics of microbenchmarking results, and is unlikely to be encountered in real usage.

The simplified heuristic operates in $O(n \log n)$ time in the worst case. This is due to the fact that for each interior node, it is necessary to search the existing set of unique heights to determine whether or not to add the current node's height to the set. We assume an implementation that can perform this search in $O(\log n)$ time. This search is also present in the full heuristic, but is dominated by the complexity of cutting the dendrogram for each cut height.

In addition to performing theoretical complexity analysis, we also measure the average runtime of the two heuristics for our collected datasets. The results of these measurements are depicted in Figure 5. The trends demonstrated by the average measured runtimes conform to the time complexities determined by our theoretical analysis. Note that, as is the case in the theoretical complexity analysis, the measured runtime values do not include dendrogram construction.

VI. Conclusion

Microbenchmarking is a useful tool for the performance analysis of individual application components. In particular, microbenchmarking represents a potential tool for use in the development of mobile systems and applications. However, microbenchmarking features inherent complexities that must be addressed in order to produce meaningful results. The fine-grained nature of microbenchmarking measurements reveals noise generated by the underlying hardware and operating system. This noise includes outliers that must either be removed or otherwise accounted for when processing collected data.

Existing microbenchmarking implementations utilise only simple approaches to mitigating the effects of outliers. These approaches rarely utilise data mining techniques. This reduces the quality of the information produced by these tools. There is a lack of mobile microbenchmarking tools that provide a comprehensive approach to outlier removal.

In this paper, we proposed a heuristic for the automated removal of outliers from mobile microbenchmarking datasets. The design of this heuristic was informed by the observed characteristics of microbenchmarking datasets collected from a range of mobile devices. We then developed a simplified version of the outlier removal heuristic for use on mobile devices. Finally, we performed empirical evaluation of both the effectiveness and efficiency of the proposed heuristics.

The results of our evaluation demonstrate that both our full and simplified heuristic provide acceptable outlier removal effectiveness. Our simplified heuristic operates in log-linear time, making it suitable for use on resource-constrained mobile devices. This makes it feasible to perform outlier removal on-device prior to exporting the collected data. The inclusion of automated outlier removal without the need for post-processing on another device represents an enhancement to the usefulness of mobile microbenchmarking tools.

The effectiveness of the simplified heuristic demonstrates that the relationship between the characteristics of the underlying input data and the ideal cut height can indeed be captured and expressed in an easily computable form. This represents an intriguing avenue for further research. In future, we intend to collect microbenchmarking datasets from a wider variety of both mobile and desktop hardware. We are interested to explore alternative forms of simplification than the curve-fitting approach presented here, and determine if an overarching framework can be developed that facilitates the automated removal of outliers from microbenchmarking data across all platforms.

References

[1] C. Staelin, "lmbench: an extensible micro-benchmark suite," *Software: Practice and Experience*, vol. 35, no. 11, pp. 1079–1105, 2005.

[2] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *Computer*, vol. 39, no. 3, pp. 46–52, 2006.

[3] S. Wang, S. Kodase, K. G. Shin, and D. L. Kiskis, "Measurement of OS services and its application to performance modeling and analysis of integrated embedded software," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, ser. RTAS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 113–.

[4] C. Staelin and L. McVoy, "Mhz: Anatomy of a micro-benchmark," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 13–13.

[5] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 93–104.

[6] H. Akkan, M. Lang, and L. M. Liebrock, "Stepping towards noiseless linux environment," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:7.

[7] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 303–312.

[8] A. Morari, R. Gioiosa, R. Wisniewski, F. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 852–863.

[9] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007.

[10] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982.

[11] J.-M. Kim and J.-S. Kim, "Androbench: Benchmarking the storage performance of android-based mobile devices," in *Frontiers in Computer Education*, ser. Advances in Intelligent and Soft Computing, S. Sambath and E. Zhu, Eds. Springer Berlin Heidelberg, 2012, vol. 133, pp. 667–674.

[12] M. H. Jamal and A. Waheed, "Precise measurement of execution time of concurrent, symmetric, and short tasks," in *34th International Computer Measurement Group Conference, December 7-12, 2008, Las Vegas, Nevada, USA, Proceedings*. Computer Measurement Group, 2008, pp. 149–160.

[13] G. Brys, M. Hubert, and A. Struyf, "A robust measure of skewness," *Journal of Computational and Graphical Statistics*, vol. 13, no. 4, 2004.

[14] J. Y. Gil, K. Lenz, and Y. Shimron, "A microbenchmark case study and lessons learned," in *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011, pp. 297–308.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, Sep. 1999.

[16] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[17] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms," *The Computer Journal*, vol. 26, no. 4, pp. 354–359, 1983.

[18] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. New York, USA: John Wiley & Sons, 1990.

[19] J. Almeida, L. Barbosa, A. Pais, and S. Formosinho, "Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering," *Chemometrics and Intelligent Laboratory Systems*, vol. 87, no. 2, pp. 208 – 217, 2007.

[20] P. Langfelder, B. Zhang, and S. Horvath, "Defining clusters from a hierarchical cluster tree: the dynamic tree cut package for R," *Bioinformatics*, vol. 24, no. 5, pp. 719–720, 2008.

[21] J. Sander, X. Qin, Z. Lu, N. Niu, and A. Kovarsky, "Automatic extraction of clusters from hierarchical clustering representations," in *Proceedings of the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, ser. PAKDD '03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 75–87.