SoftMon: A Tool to Compare Similar Open-source Software from a Performance Perspective

Shubhankar Suman Singh
Computer Science and Engineering, IIT Delhi
shubhankar@cse.iitd.ac.in

Smruti R. Sarangi*
Usha Hasteer Chair Professor, Computer Science, IIT Delhi
srsarangi@cse.iitd.ac.in

ABSTRACT

Over the past two decades, a rich ecosystem of open-source software has evolved. For every type of application, there are a wide variety of alternatives. We observed that even if different applications that perform similar tasks and compiled with the same versions of the compiler and the libraries, they perform very differently while running on the same system. Sadly prior work in this area that compares two code bases for similarities does not help us in finding the reasons for the differences in performance.

In this paper, we develop a tool, *SoftMon*, that can compare the codebases of two separate applications and pinpoint the exact set of functions that are disproportionately responsible for differences in performance. Our tool uses machine learning and NLP techniques to analyze why a given open-source application has a lower performance as compared to its peers, design bespoke applications that can incorporate specific innovations (identified by *SoftMon*) in competing applications, and diagnose performance bugs.

In this paper, we compare a wide variety of large open-source programs such as image editors, audio players, text editors, PDF readers, mail clients and even full-fledged operating systems (OSs). In all cases, our tool was able to pinpoint a set of at the most 10-15 functions that are responsible for the differences within 200 seconds. A subsequent manual analysis assisted by our *graph visualization* engine helps us find the reasons. We were able to validate most of the reasons by correlating them with subsequent observations made by developers or from existing technical literature. The manual phase of our analysis is limited to 30 minutes (tested with human subjects).

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Software comparison, Performance debugging, NLP based matching

ACM Reference Format:
Shubhankar Suman Singh and Smruti R. Sarangi. 2020. SoftMon: A Tool to Compare Similar Open-source Software from a Performance Perspective.

*Joint Appointment with Electrical Engineering

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387444

In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, 12 pages. https://doi.org/10.1145/3379597.3387444

1 INTRODUCTION

In today's complex software ecosystem, we have a wide variety of software for the same class of applications. Starting from mail clients to operating systems, we have a lot of choices with regards to the application, particularly in the open-source space [56]. When the source code is freely available and can be used without significant licensing restrictions it is expected that for the same high-level task, all the competing alternatives will take a roughly similar amount of time. Paradoxically, this is not the case as we show in Table 1. This table compares the time that different open-source software programs take for executing the same high-level task on the same platform. The last column shows the ratio of the time taken for the fastest and the slowest applications. We observe that this ratio varies from **1.1** to **6.2**, which is significant considering the fact that with the rest of the parameters remaining the same namely the hardware, OS version, compiler, binutils, and libraries, the differences purely arise from the code of the application itself. Other researchers have also found similar differences in performance and energy among similar applications such as similar Android apps [56].

Table 1: Popular open-source software categories

Category	Similar open-source software	Task	P
Image processor	GraphicMagick, ImageMagick	Crop an image	6.2
PDF reader	Evince, Okular, Xpdf	Load a pdf	2.1
Text editor	Geany, Gvim, Vim	Open a file	2.6
Music Player	Audacious, VLC, Rhythmbox	Play an audio	4.6
Mail Client	Thunderbird, Balsa, Sylpheed	Compose a mail	1.1
Operating System	Linux, OpenBSD, FreeBSD	Unixbench	1.7

P → Maximum/ Minimum performance

These differences lead to two natural conclusions: ❶ either one of the applications uses suboptimal algorithms [36, 65, 70, 79], ❷ or the applications are optimized for different end goals and the performance differences are a manifestation of that [38, 76]. In this paper, we try to answer this question and find out the *real reasons* for the differences in performance between similar applications that use the same software and hardware environment. We present the design of a tool, *SoftMon*, that can be used to compare similar open-source software that are created by different vendors. It allows us to find the portions of the code that disproportionately account for the difference in performance in large codebases consisting of millions of lines. In this paper, we focus on the largest codebases [20] that are released with open-source licenses, notably operating systems, mail clients, image tools, and multimedia players. From our analyses, we

conclude that the reasons for the differences are a combination of reasons ❶ and ❷: suboptimal design choices and a preference for certain kind of inputs at the expense of others.

To the best of our knowledge, tools like *SoftMon* do not exist (open-source or proprietary), and an effort of this magnitude has not been attempted before. Existing work compares very small pieces of code for either syntactic and semantic similarity; they are not designed to compare million-line code bases from the point of view of performance. There is however a need for such technologies. There are services such as the Technology Evaluation Centers [19], where one can purchase a software comparison report for thousand software solutions in different categories such as ERP, business intelligence and CRM. The results are created and curated by a team of professional analysts and are not based on the analysis of source code. In comparison, we propose a mostly automatic approach that relies on extensively analyzing the source code and function call trees; we extensively use unsupervised machine learning, sophisticated graph algorithms, and natural language processing techniques. In our view, a problem of this scale could not have been solved with traditional algorithms, AI and machine learning techniques were required.

The input to *SoftMon* is a pointer to the root directory of the source codes and the binary. The first step is to generate large function call trees for the binaries that are to be compared. We use existing tools such as binary instrumentation engines for this purpose. Subsequently, we propose a series of steps to parse, prune, and cluster the function call trees. Once this is done, we map similar functions across the binaries using contextual information (position in the call stack and the nature of the function call tree), and textual information such as comments, the name of the function and its arguments, and developers' notes in GitHub. Subsequently, we extend our analysis to also cover functions called within a library. Finally, after several rounds of pruning where we remove nodes that have similar performance across the binaries or whose performance impact is negligible, the final output of the tool is a set of 10-15 functions arranged as a function call graph that shows all caller-callee relationships between them. Most of the time, it is possible to make very quick conclusions by looking at this graph. However, in some cases particularly with operating systems such as FreeBSD and Linux, it is necessary to look at the source code of a few functions (limited to 5). The reasons for the difference are quite visible given the fact that *SoftMon* additionally annotates functions with their detailed performance characteristics such as the cycles taken, cache hit rates, and branching behavior. We tested the usability of our tool with a cohort of human subjects, all of them could pinpoint the reasons within 30 minutes.

❶ Users can use this tool to get a much better understanding of the relative strengths and weaknesses of different open-source software. ❷ Software developers can derive insights from this tool regarding the parts of the code that are suboptimally implemented (*performance bugs*) by comparing their code with similar open-source software. ❸ Even closed source software vendors can use this tool internally to compare different versions of the same software and find the reasons for differences in performance.

The organisation of this paper is as follows. We discuss relevant background and related work in Section 2. Section 3 describes the tool, *SoftMon*, and then we discuss our evaluation and results in Section 4. Finally, we conclude in Section 5. The *SoftMon* tool is available at https://github.com/srsarangi/SoftMon.

2 BACKGROUND AND RELATED WORK

2.1 Taxonomy of Prior Work

The problem of detecting if two pieces of code are the same or not is a classical problem in computer science. In its general form it is undecidable; however, it is possible to solve it using various restrictive assumptions. The solutions find diverse uses in malware detection, vulnerability analysis, plagiarism detection, and in studying the effects of compiler optimizations. There are three broad classes of techniques to find code similarity: *Same, Structural and Functional*. We further classify these proposals into two broad classes: *static* or *dynamic*; the *static* methods [27, 30, 35, 43, 45, 47, 67] only use the source-code or binary to establish similarity but the *dynamic* methods [42, 52, 59, 68, 71, 73, 82, 83] generate traces based on fixed input parameters and use the observed outputs and the behavior (memory state, system calls, etc.) to establish similarity (see the vertical-axis in Figure 1).

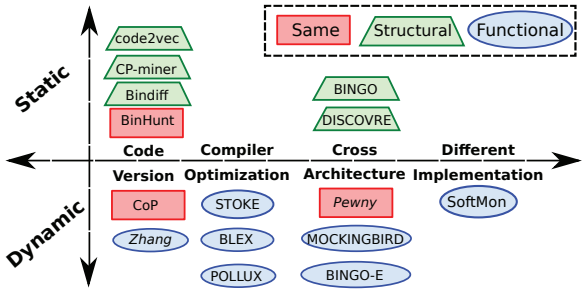

Figure 1: Prior work - classification

We then further classify these works based on the *level* that they operate at (see the horizontal-axis in Figure 1) into different *categories. Code version:* Proposals [27, 45, 47, 67, 68, 83] in this area detect code similarity across multiple versions of the same code. The techniques are used for bug detection and plagiarism detection. The next phase of proposals proposed to detect similarities across different *compiler optimizations* [42, 59, 73]. These techniques were used for binary optimization and malware analysis. The next generation of techniques find similarities irrespective of the *code-version, compiler optimization and instruction set architecture* [30, 35, 43, 52, 71, 82]. These techniques are used for code search and semantic similarity identification.

We add a new category in this taxonomy called *different implementation*, where we compare two codes that represent the same high-level algorithm yet are coded differently, such as bubble sort and quick sort. There are approaches [24, 69] in the literature to find if two such codes are the same; they primarily work by creating a corpus of algorithms and check if a given implementation is similar to any other algorithm in the corpus. This is however not a very scalable technique, hence, we propose a new approximate technique that matches two codes based on the comments, function names, library name, commit logs, Github comments and their context (its callers and callees).

Table 2: Summary of Prior Work

Paper	Venue	Algorithm	Complete?	T_c	Remarks
Bindiff	DIMVA'04	Graph isomorphism on control flow graph of basic blocks (CFG)	Yes	$2^{O((logn)^3)}$	[$n \rightarrow$ # basic blocks]
Zhang	FSE'05	Graph isomorphism on data dependence graph	Yes	$2^{O((logn)^3)}$	[$n \rightarrow$ # instructions]
CP-miner	TSE'06	Frequent sequence mining: code	Yes	$O(n^2)$	[$n \rightarrow$ # lines of code]
Binhunt	ICICS'08	Graph isomorphism on CFG	Yes	$2^{O((logn)^3)}$	[$n \rightarrow$ # basic blocks]
STOKE	ASPLOS'13	Machine state: registers and memory	No	$O(m)$	[$m \rightarrow$ size of the machine state], Does not handle loops.
CoP	FSE'14	Sub-sequence matching on CFG	Yes	$O(n^2)$	[$n \rightarrow$ #basic blocks]
BLEX	USENIX'14	Dynamic features: system calls, memory writes	Yes	$O(d)$	[$d \rightarrow$ # dynamic features]
Pewny	IEEESSP'15	Minhash of input-output	Yes	$O(kn)$	[$k \rightarrow$ # hash operations, $n \rightarrow$ #basic blocks]
POLLUX	FSE'16	Dynamic features (same as BLEX)	Yes	$O(d)$	[$d \rightarrow$ # dynamic features]
discovRE	NDSS'16	Pre-filtering and graph isomorphism on CFG	Yes	$2^{O((logn)^3)}$	[$n \rightarrow$ size of the filtered CFG]
Mockingbird	SANER'16	Longest common sub-sequence on signature	Yes	$O(d^2)$	[$d \rightarrow$ size of the signature]
Bingo-E	TSE'18	Static + Dynamic features	Yes	$O(d)$	[$d \rightarrow$ # dynamic features]
code2vec	POPL'19	Neural network on fixed-length code vector	No	$O(v+n)$ [Prediction]	[$v \rightarrow$ size of the code vector, $n \rightarrow$ size of the code]. Does not handle pointers, and *structs*.
SoftMon	–	Comment Similarity and contextual information (e.g. stack trace)	Yes	$O(c)$	[$c \rightarrow$ size of the comments], Functions are matched based on their comments and other textual information.

T_c: **Time complexity order** $\rightarrow O(c) < O(d) < O(m) < O(n)$

2.2 Techniques to Find Code Similarity

We shall refer to Table 2 extensively in this section; it summarises recent work in chronological order. Note that all the time complexities are normalized to the same baseline: detect if a pair of n-instruction functions are the same (or similar). Given that we are using the *big-O* notation, it does not matter if n refers to the number of lines of code, number of instructions, or the number of basic blocks, because in almost all cases, they are linear functions of each other.

Same: Two codes are said to be the *same* if they contain the same instructions, read the same inputs, and produce the same outputs. The main difference is the order of instructions. The proposals in this domain [47, 68, 71] construct a control flow graph (CFG) representation of the code and apply graph isomorphism or proof techniques to establish similarity. Since graph isomorphism does not have a polynomial-time solution [50], these techniques are not scalable to large codebases. Hence, these techniques generally work well in establishing similarity between different versions of the same set of functions, but fail to scale (see Table 2).

Structural: Several proposals [27, 35, 43, 45, 67] try to identify the similarity between two pieces of code based on their structural similarity. As compared to the previous set of techniques, this is an approximate approach. Structural similarity is useful for plagiarism detection or finding differences across different versions of the same set of functions. Any algorithm in this space needs to reduce a piece of code into an intermediate representation such that the names of variables or the order of instructions are not relevant. The solutions use different intermediate representations such as n-grams, CFGs, DFGs, ASTs, graphlets or tracelets to establish similarity. The advantage of this approach is that the time complexity reduces from exponential to linear or quadratic. For example, CP-miner [67] uses the *frequent sequence mining* algorithm, which is faster than graph isomorphism. However, this technique is limited to identifying copy-paste similarities. Thus, this technique will not work for our problem.

Recent proposals [25–27, 53] apply machine learning to solve this problem. *code2vec* [27] uses a neural network on the abstract syntax tree (AST) of the code snippet to construct a fixed-length code vector. The similarity in the code vector is used to compute the code similarity. We observed that their implementation is not general as it does not handle pointers and *structs*. The time complexity of the prediction phase is linearly dependent on the code size and the code vector. (see Table 2)

Functional: All the works [42, 52, 59, 73, 83] in this domain compare two pieces of code (set of functions) if they produce the same result. Formally, two functions are said to be *functionally the same* if they produce the same output given the same input, and have the same side effects (systems calls, and memory footprint). Most works in this area have also adopted an approximate approach, where they create a signature that encodes the output and the side effects, and then the signatures are compared to determine if the functions are functionally same.

For example, *BLEX* [42] and *POLLUX* [59] use dynamic features such as the sequence of system calls, library calls, and memory read-write values to construct a signature. The difference between these signatures is a measure of the dissimilarity of two functions. *BingoE* [82] uses both static and dynamic features to establish similarity. Hence, it can identify similarities even in the case of a different compiler optimization or a cross-architecture compilation. The time complexity of their algorithm is linearly dependent on the number of dynamic features. However, this technique is only limited to comparing different code versions (example: forked projects). Hence, they will not be able to detect similarities across different implementations.

2.3 Performance Analysis

2.3.1 Profiling Tools. There is a large body of work on performance profiling of software [37, 39, 48, 64, 80]. The profiling can be done by either sampling or instrumenting the given software. *gprof* [48] and *perf* [40] are the most commonly used software profiling tools. Similarly, *ftrace* [6] and *dtrace* [4] are the most commonly used OS profiling tools. These tools rank functions based on their execution time. But, these tools do not provide any mapping between the functions across different applications.

2.3.2 Visualization Tools. A *Flamegraph* [49] visualizes the output of the profiling tools. It visualizes a function call stack and

can be used to identify frequently executed functions. Differential Flamegraphs [31] can be used to understand the differences in the functional call patterns across different versions of the same software. They cannot be used for differential analysis in our case as they do not provide any mapping algorithm for functions across different applications. Our novelty is that we use the function name, textual information and the function context (in the graph) to map functions across different applications.

2.3.3 Performance Bug Detection. Many performance bug detection tools have been proposed [41, 55, 75, 81]. These tools try to find a specific type of hidden performance bug or diagnose the bugs detected by the end users. They employ different rule based techniques or statistical approaches. The idea is to compare the behavior of the same software for different inputs. However, our problem is different: compare different software with the same inputs.

3 THE DESIGN OF *SOFTMON*

The *SoftMon* tool comprises the following components. ❶ A Trace collector to generate the sequence of function calls invoked in the execution of a program and to construct a function call tree from the trace. ❷ A Classification and Clustering step to classify the call trees into different high-level tasks and then cluster the different call trees into fewer groups. ❸ A *Graph* engine to compress and filter the function call trees, ❹ an *Annotation* step to annotate the call trees with their relevant comments. ❺ A *Map* engine to find the mappings between the nodes across the call trees of different applications and ❻ a *Graph Visualization* engine to render the mapped trees to simplify human analyses (Figure 2).

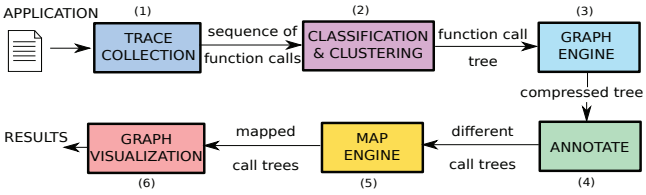

Figure 2: Flow of actions

3.1 Trace Collector

For a set of applications, we need to first define the high-level tasks whose performance we wish to compare. Once we have defined a set of high-level tasks, we need to generate a trace of functions (with appropriate annotations) for each application. Each trace is a sequence of 3-tuples <*name of the function, name of the caller, number of execution cycles*>.

The Trace collector is a thin wrapper on the built-in tools, *ftrace* [6] in Linux and *dtrace* [4] in FreeBSD/OpenBSD to collect OS traces. We use the *PIN tool* [16] to collect the application traces. We subsequently post-process this trace to generate a function call tree (FCT), where each node is a function, and there is an arrow from node A to node B, if node A is the caller and node B is the callee. Note that for recursive function calls, or when we call the same function over and over again, we generate different nodes. The main advantage of this method is that it is very easy to find the frequently executed paths (also used by [45, 47, 56, 83]).

It is possible that a high-level task may call functions billions of times. To reduce the size of the generated FCT we propose a *compression technique*. For every sub-trace of 200,000 nodes (average number of functions in a system call), we generate a *signature*. Whenever a signature matches any of the previous signatures in the collected trace we discard the sub-trace. We add the number of execution cycles corresponding to each function in the discarded sub-trace to the corresponding function nodes in the matched trace. The signature is generated as follows. It is the *Jaccard distance* [54] (also used in [59]) between the set of unique functions present in both the sub-traces. The *Jaccard distance* ($J(A,B)$) measures the similarity between two finite sets, A and B, and is defined as: $J(A,B) = |A \cap B|/|A \cup B|$

Two signatures are said to *match* if their *Jaccard distance* is more than a threshold T_t.

3.2 Classification and Clustering

For the same high-level task, we can obtain thousands of different FCTs due to the following reasons: different arguments, conditional statements, wakeup/sleep statements, interrupts and thread switches done by the scheduler. Analyzing and comparing thousands of such trees is hard. We also found that a single tree is not sufficiently informative to represent the behavior of a task. Hence, we cluster the different trees based on the set of functions that they contain, and choose a representative tree from each cluster for doing further analyses. The clustering is done by using the *Jaccard distance* as the distance metric.

We proceed to cluster the call trees based on this metric using hierarchical clustering [60]. Initially, all the clusters are initialized containing a single tree. The clusters are then merged based on the similarity of the clusters calculated using the *Jaccard distance*. Note that if we consider a minimum *Jaccard distance* of κ for the sake of creating clusters, then it is guaranteed that the set of unique functions will at least have a $(100 \times \kappa)$ percentage overlap between two call trees in the same cluster. We limited the total number of clusters to C_n. We select a representative tree for each cluster that has the following properties. Its total execution cycles should be closest to the average value for that cluster. Let us call this the *m-tree* (output of the classification and clustering step).

3.3 Graph Engine

We annotate each node of an *m-tree* with additional information: the number of instructions executed by all the functions that are a part of it, and the number of cycles taken by the sub-tree with the node as the root. In our workloads, the size of the *m-tree* is very large. Understanding, analyzing, and comparing such large trees is computationally prohibitive; hence, we used graph reduction techniques to reduce the size of the trees. This will help to simplify our analyses (see Figure 3).

3.3.1 **Reduce**. We observed that the number of nodes in an *m-tree* is significantly larger than the set of unique functions in it. This implies that many functions are called multiple times. We also found many repeating patterns in the *m-tree*, because of iterative structures in the code. Hence, the next logical step is to reduce the size of the *m-tree* by folding repeating sub-trees into a single node. We mapped this problem of finding the largest repeating sub-tree

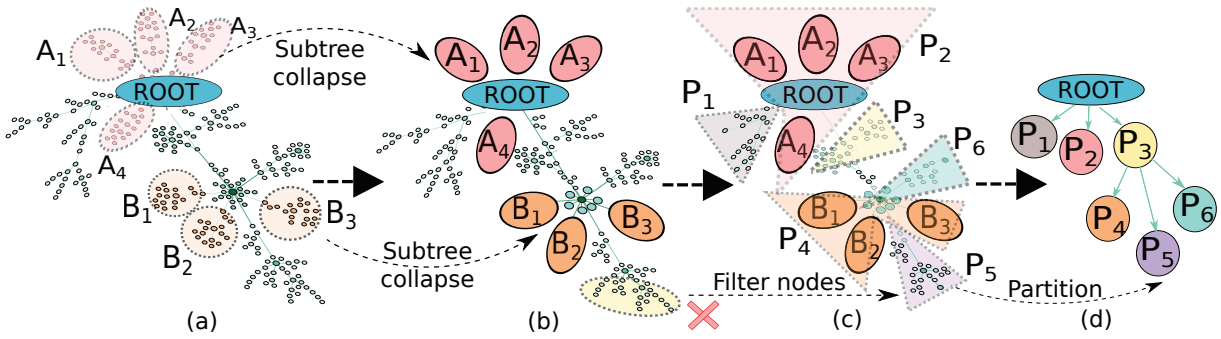

Figure 3: Graph Engine: (a) Initial (b) Reduce (c) Filter (d) Partition (*getdents* system call for *Find* in Linux)

with finding the number of occurrences of all the sub-strings in a given string.

We first do a pre-order traversal of the tree. This lists all the nodes in the order that they are seen first. Now, let us consider the output of the pre-order traversal – list of nodes – as a string. Every sub-tree is a contiguous string in this representation. Let us consider all repeating sets of sub-strings, where there is a constraint on the minimum size of the sub-string, and the minimum number of repetitions. Let us represent the length of the substring S_{ij} (between indices i and j) as $s = j - i + 1$, and the number of occurrences as o. We then *reduce* the tree by replacing each occurrence of S_{ij} with a new single node (see Figure 3(b)). Thus, we replace $s \times o$ nodes with o nodes, reducing the size of the tree by $s \times o - o$ nodes.

We use a dynamic programming algorithm [33, 72] that provides the number of occurrences of each substring, whose length is greater than a threshold. For each substring, we compute its weight, i.e., the number of nodes that will be reduced if it is replaced by a single node ($= s \times o - o$), and order them in the descending order of weights. We then run an iterative algorithm, where in each iteration we replace the most frequent substring in our list of substrings with a single node. It is possible that other substrings might have an overlap with it; they are removed from this list.

For each of these nodes that represents a sub-tree, we store the sub-tree in a separate data structure, which is basically a forest. Each sub-tree in the forest, has been replaced by a single node in the new reduced tree (*rm-tree*). For the added node, the value of the number of instructions and cycles are the sum of the corresponding numbers for each node in the corresponding sub-tree.

3.3.2 **Filter**. Subsequently, we applied a filter function on the *rm-tree* to filter out nodes that take very few cycles to execute (less than a threshold, C_t). Our goal is to understand the difference in performance across similar applications. Hence, the nodes that do not have a significant contribution to the total number of execution cycles can be discarded.

The *filter* operation is done by an in-order traversal of the *rm-tree* and deleting the nodes that take less than C_t cycles. The complete sub-tree corresponding to such a node N is deleted. Note that we store the number of cycles taken by the sub-tree with the given node(N) as the root, within N itself. (see Figure 3(c)).

3.3.3 **Partition:** Finally, we partition the filtered *rm-tree* into few sub-trees (10-15) based on the number of cycles and the structure of the *rm-tree*.

We observed that the call trees corresponding to two different implementations do not match at the structural level. For example, one implementation may have more functions as compared to the other one. However, we observed that both the implementations do similar high-level tasks. Thus, the call-trees can be easily matched at the task level. Hence, we partition the *rm-tree* into few sub-trees that represent high-level tasks.

The *partition* operation is done by iteratively reducing the depth of the tree by coalescing leaf nodes with their parent node (see Figure 3(d)). In each round we consider all the internal nodes with the maximum depth, and coalesce (merge their leaves with the parent) all of them; we repeat this process until the number of total nodes that are left is less than a threshold, P_t.

3.4 Annotation

We use the *cscope tool* [21] to find the location of the functions' definitions in the application's source code. Then, we extract the *comment* at that location and store it in a hashtable. Some comments even describe the function arguments, which are not necessary for our annotation. Hence, we filter out the definitions of the function arguments (using a script). We also found a lot of abbreviations or undefined terms in the comments such as 'buf' for buffer, 'getblk' for get block, 'async' for asynchronous, etc. We automatically corrected them by using the online kernel glossary [13].

The *comment* and the *function-name* are then used to annotate the nodes in the partitioned *rm-tree*. In case the comments are not available, we also mine the GitHub commit logs. We use an automated script that uses the *git diff* command to check whether a given function was edited in a particular commit. If there is an edit, we use the commit message to annotate the node. A particular partition is annotated with the comments, commit message and the function-names corresponding to all the functions/nodes in the partition. The final annotation is the concatenation of all the corresponding comments. Henceforth, we shall use the term *comment* to the entire annotation.

3.5 Map Engine

The input to the Map Engine is a set of partitioned *rm-trees* of different applications, which are annotated with comments. Now, our goal is to map the partitions across the applications with similar functionality so that we can look for differences. For example, the *pagecache_get_page* function in Linux and the *getblk* function in

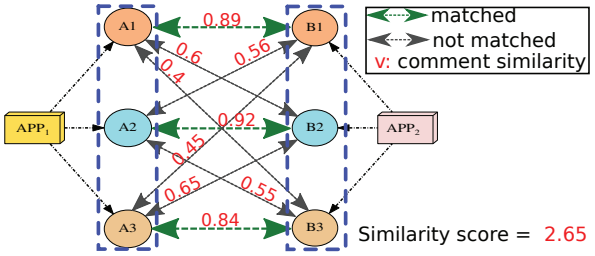

Figure 4: Mapping Engine pairs: {(A1, A2); (A2, B2); (A3, B3)}

FreeBSD have similar functionality. We automate this process by using the Map Engine.

3.5.1 **Comments and Name Similarity:** We use *spaCy* [22] – an NLP tool – to calculate the similarity index between a pair of comments/function-names. The tool converts a text into a token vector of fixed length. The Euclidean distance between the two vectors is used as the objective function to determine the similarity (value between 0 and 1) between two text inputs. An output of 1 means a perfect match. We used the *en_core_web_md* model provided by the tool for computing the similarity index. It is trained over web-blogs, news and comments and consists of 685,000 keys and 20,000 unique vectors. The average accuracy of the tool is 92.6% [51]. We aim to detect the functional similarity of two partitions based on their comment+name similarity.

3.5.2 **Bipartite Matching:** The mapping of two partitioned *rm-trees* (rm_1 and rm_2) is done as follows. We construct a complete bipartite graph using the partitions of the two *rm-trees*. The edge weight is the comment (+name) similarity index. The set of edge weights represented as a 2-D matrix is provided to the optimal Hungarian algorithm [63]. The optimization function is to maximize the sum of the weights of the matched edges (also used by [77]). Finally, we obtain a one-to-one (not necessarily a bijection) mapping between the partitions in the two trees. An example of a mapping is shown in Figure 4. We run the *Map Engine* for all the pairs of *rm-trees* of the same task across different applications.

Once, we have the mapping, we remove all the pairs of partitions that have similar (within 15% i.e. mean - 3-sigma values for the deviation in execution time) performance. Then, we run the *Map Engine* again (on-demand) on the partition-pairs that are left to find a function-wise mapping. This helps us ensure that the context (parent-child relation) of the nodes is maintained across mappings. To summarize, we follow a top-down approach. We first match at the level of large partitions and then focus our attention on similar partitions that have differing performance. We run the map engine again on these partitions to match function groups (typically *reduced* nodes) and then prune out those groups that have similar performance.

The presented steps suffice for code such as operating systems and image tools, where the entire source code is available, and third-party libraries account for a miniscule fraction of the execution. However, if a large part of the execution is accounted for by code in third-party libraries, then an additional pass is required, where we analyze the stack trace (function call trace) of invoked functions.

3.6 Library Engine: Analysis of Stack Traces

The execution of an application in general consists of two components: application source code and library calls. We can analyze the functions in the application source code by mapping their *rm-trees* based on their source code comments. But, this will not be applicable in the case of the library functions since we do not have the source code. There are two possibilities when comparing two such applications; both of them use the same libraries (*libc*, *libz*, etc.) or they use different libraries to perform the same operation (example: *libQT* vs *libglib*). From the trace, we have a list of functions called and their cycle counts. We observed that there exist many library functions (example: *malloc*, *hash_table_lookups*) that account for a disproportionate fraction of the execution, and are responsible for the differences in performance. Hence, to understand why these library functions were called so frequently, we need to create the corresponding stack traces (function call trace from the main function to the current function). We observed that there are many such paths to a particular library function that are pruned and clustered as follows. Note that we analyze the stack traces of only those library functions, which are called frequently (number of calls is beyond a certain threshold).

Step 1) We first collate all the paths to a particular library function. We find all the nodes corresponding to the library function by applying a depth first search on the function call tree. Next, for each such node, we follow the parent pointer and construct a path to the root. Each such path is annotated with the cycle count of the leaf node (library function). This is known as *the cost of the path*.

Step 2) We then construct a graph using these paths and we remove the rest of the nodes. The cost of each node is equal to the sum of the cost of the paths that intersect it.

Step 3) We then filter the graph by removing the nodes whose cost is less than a threshold.

Step 4) We then run the mapping engine on the stack trace graph for the same library function across different software. Here, we consider all the textual information that we can find: name of the function, and any GitHub comments.

The benefit of this step is that even if the libraries themselves are responsible for differences in performance, we can pinpoint the relevant functions within the libraries (or the functions in the application code that invoke them) and after processing their stack traces we end up with a small graph (10-15 nodes), which is given to the visualization engine.

3.7 Graph Visualization

The visualization engine produces two kinds of graphs: one for the source code, and one for the source code along with library code.

Source code browsing: We used the *graphviz* [46] tool to visualize the matched functions of the *rm-trees* (see an example in Figure 4). We display the output of the Mapping Engine in the form of a function call graph. In the function call graph, the matched nodes are in the same color. Each node is further annotated with the function name, instruction count, source code comments, number of calls and link to the source code (file name and line number).

The tool also highlights the nodes that are responsible for the maximum performance difference. A user can then further inspect the node by expanding the sub-tree corresponding to the node. We

Table 3: Software Categories, Properties and Description of Benchmarks (*loc* → lines of code)

	Software	# files	# functions	# loc	List of benchmarks	
Image	ImageMagick-6,7, GraphicMagick-1.3.33	450-800	700-1400	500K	*crop*, *cnvt*: convert from png to jpg and *flop* an 1024x512 pixel image	(file size: 24 KB)
Pdf	Xpdf-4, Evince-3.27, Okular-1.9.7	200-500	9K-15K	150K	*open*, load a *page* (page number), *full*: open in presentation mode	(file size: 2 MB)
Text	Vim-8.1, Gvim-8.1, Geany-1.27	230-300	2K-9K	120K-450K	*open*: load a text file	(file size: 1 MB)
Music	Vlc-3.0.8, Rhythmbox-3.4.3	400-1800	12K-14K	200K-700K	*open*: play an audio file for 30 seconds	(file size: 1 MB)
Mail	Balsa-2.5.9, Sylpheed-3.7	335-348	10K	150K	*compose*, *attach*: a mail with/without an attachment file	(file size: 4 KB)
OS	OpenBSD-5.7, Linux-4.2, FreeBSD-10.2 $K \rightarrow 1000$, $M \rightarrow 1000000$	5K-40K	20K-90K	3M-18M	**File copy**, **Pipe**: communication, **Process creation**, **System call**, **FileIO**: read-writes, **OLTP**: database server, **Find**: search a file, **Iscp/Oscp**: secure copy [2, 62]	

can also re-run the matching algorithm on the expanded sub-tree to do a deeper analysis (on demand).

Library code browsing In this case the engine produces a small graph using the *graphviz* toolkit that typically fits in a single screen. It is possible to visually identify the different execution paths within the library, and it is also possible to identify which paths contributed to the differences in performance.

Summary: This is the only part of the tool that requires human intervention. Note that to get here we had to prune and cluster thousands of functions; however, at this stage a human is required to identify the observed patterns and decide the reasons for the difference in performance. On the basis of user studies, we shall argue in the evaluation section that given the small size of graphs, this process can be done easily within 10-30 minutes.

4 EVALUATION

4.1 Implementation Details

All the benchmarks were compiled using the same version of the compiler *(gcc-7.5)* and libraries *(glibc 2.27)*. The applications and the benchmarks are described in Table 3. The number of files in these applications lie in the range 200 (in smaller apps) to 40,000 for Operating systems. We also observe that the number of lines of code is 10x higher for OSs (million lines) as compared to other applications. The OS benchmarks were run on all three OSs with the same input parameters that were provided with the Unixbench suite. These benchmarks have also been used in other works in the OS evaluation space [28, 29, 44, 57, 58, 61, 74]. We discarded a few benchmarks as they did not make any system calls and did not have any OS contribution. They had the same performance on all the three OSs. We conducted all our experiments on a Dell R610 server with 2 Intel Xeon X5650 processors with 12 cores each (2.67 GHz frequency) and 32 GB of memory running Ubuntu 18.04.

All the components of the *SoftMon* tool including the *clustering*, *graph* and *map engines*, and the *graph visualization tool* are implemented in Python 3. The comments are extracted with the help of the *cscope* tool [21] that has support for C, C++, Java and other languages.

4.2 Execution of the *Graph Engine*

We started with the complete function call trace for all the benchmarks that correspond to 27K (for image tools) to 200M (for OS) function calls. Then, the trace was converted into a call-tree after applying classification and clustering. Finally, we used the *graph engine* to compress the call-trees. The *m-trees* contained up to 10,000 nodes. The thresholds C_t and T_t were set to 500 cycles (mean - 3-sigma values for the deviation in execution cycles of functions) and 0.9 respectively. We observed a **2-4X** and **4-8X** reduction in the number of nodes after applying the *Reduce* and the *Filter* functions respectively. The *Partition* function further reduced the tree to a few partitions **(10-15)** (P_t). The total reduction in the size of the *m-tree* is thus between **600-1000X** across the different stages of our algorithm.

Table 4: Comment Similarity (from *spaCy*)

Software	Function	Comment in the source code	Similarity
ImageMagick	CropImage	extracts a region of the image starting at the offset defined by geometry.	
GraphicMagick	CropImage	extract a region of the image starting at the offset defined by geometry.	100%
Linux	pagecache_get_page	Find and get a page reference. Looks up the page cache slot.	
FreeBSD	getblk	Get a block given a specified block and offset into a file.	89%
Linux	__radix_tree_lookup	Lookup and return an item at position index in the radix tree	
FreeBSD	pctrie_lookup	Returns the value stored at the index.	86%

Table 5: *SoftMon*: runtime of different steps

Steps	Input size	Time taken (s)
Classification	Trace of 200 M functions	60 s
Clustering	Call tree with 25 K nodes	6 s
Reduce	*m-tree* with 10 K nodes	112 s
Filter & Partition	*rm-tree* with 4K nodes	15 s
Map Engine	125 tree pairs of 15 nodes	5 s
Total time per benchmark		**198 s**

4.3 Execution of the *Map Engine*

We automatically mapped functions among different applications using the textual annotations. On an average, we found comments for **50%** of the functions across all the applications. Table 4 shows a few examples of the comment similarity values calculated using the *spaCy* tool. Most of these comments described the behavior and the working of the corresponding functions. Recall that we aggregate the functions into *reduced* nodes, and also concatenate all the textual information (comments, function names, etc.). Hence, we were able to map functions from the different applications with a comment semantic equivalence (NLP similarity score) of more than 80%. We manually verified that the mapping provided by comment similarity actually translates to similar functionality of these functions.

The engine mapped 40 pairs of partitioned *rm-trees* across the 18 benchmarks, which resulted in the mapping of 400 pairs of functions. The matches were validated manually by examining the source code of the respective functions using a team of 10 participants. Each participant was given a set of 120 function pairs. She had to look at the comments and the code of the function and assign a 0/1 score if the functions matched or not. Each function pair was analyzed

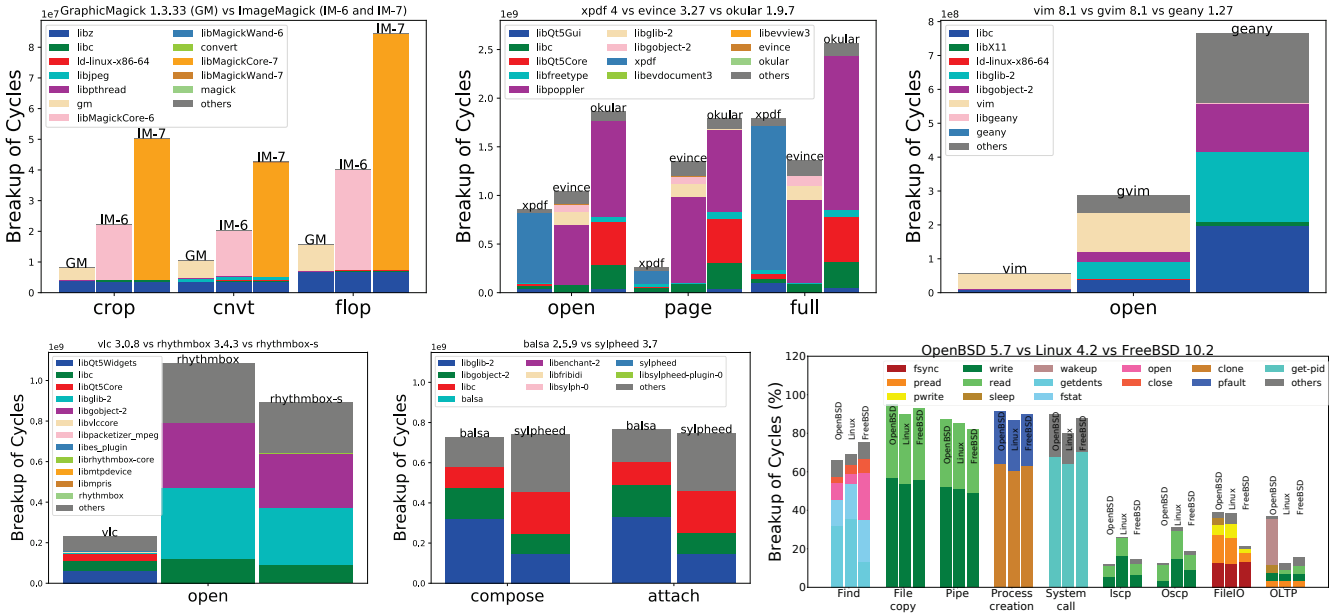

Figure 5: Breakup of Cycles (smaller value is better): (a) Image (b) Pdf (c) Text (d) Audio (e) Mail (f) Operating system

for 5 minutes on average. We evaluated each function pair by three different participants. Finally, we selected the majority value.

The accuracy (fraction of correct matches determined subjectively (similar to [78])) of our tool was **90%**. If we would not have partitioned the *rm-trees*, then the accuracy would have reduced to 68% owing to the large number of false-positives (thus context information captured by partitioning is important). One such example of a false positive by our tool is (*GetCacheNexus, SetCacheNexus* in ImageMagick) which was assigned a score of 0.77 but they are different functions. We do not have false negatives as we consider all relevant matches. Next, we report the run times of the individual steps in Table 5 (maximum times across benchmarks reported). The maximum cumulative execution time of all the stages is limited to **200 seconds**.

4.4 Differences in Performance

We show the cycle count of different applications for each category in Figure 5 (lower value is better). The performance difference varies from **1.1x** to **6.2x**. The two mail clients have a similar performance since they use the same set of libraries, whereas *GraphicMagick* is 6.2x faster as compared to *ImageMagick*. Next, we will discuss the cycle breakup of different applications into application code, libraries and OS system calls.

The cycle count of the libraries (libz and libc) is similar in the case of *ImageMagick* and *GraphicMagick*. The major difference is due the application code. We also observed that *ImageMagick-7* performs poorly as compared to *ImageMagick-6*. In other applications, the cycle count is dominated by the library code. There are two observations, ❶ the same set of libraries are used but they differ in the cycle count (*Evince* vs *Okular*), ❷ different libraries are used to perform the same function (libglib vs libQT5 for GUI rendering). Similarly, for OSs, we observed that the total cycle count is dominated by different system calls. OpenBSD is better suited for executing search and network based applications (*Find, Iscp* and *Oscp*), while FreeBSD is better suited for executing I/O heavy applications (*FileIO*, *Pipe* and *OLTP*); Linux provides better performance in *File copy*, *Process creation* and *System call* benchmarks.

4.5 Reasons for the Differences in Performance

We shall first present three representative case studies (all the reasons cannot be presented because of a lack of space). Hence, we shall briefly summarize all the reasons at the end in Section 4.6 and also provide pointers to discussions that allude to the same reasons.

4.5.1 **Case Study 1***: Image Tools.* The performance of *GraphicMagick* is better than *ImageMagick* for all the benchmarks (see Figure 5(a)). We now show the output the Mapping Engine in Figure 6(a). We found the following reasons by comparing the matched nodes (after extensive pruning and filtering). In the *ReadImage* function, *ImageMagick* makes a copy of the *pixel* data-structure and then passes it to the next function. In contrast, *GraphicMagick* directly passes the pointer to the *pixel* data-structure. Hence, this function is faster in *GraphicMagick* as compared to *ImageMagick*. The *CropImage* function is a nested *for* loop with three levels in *ImageMagick*. In contrast, *GraphicMagick* uses a single *for* loop that iterates over the image's rows and uses *memcpy* to copy the cropped image – this is a faster implementation. In *ImageMagick*, a *PixelCache* is maintained, which is not used in *GraphicMagick*. It creates a copy of the full image from the source cache to the destination cache. It adds an additional 14M cycles (28%) to the overall execution.

Comparison with *gprof* and *PIN*: The *gprof* and *PIN* tools rank functions based on their execution time. The ranks of the functions *ReadImage*, *CropImage* and *PixelCache* were 5, 7 and 12 respectively in the output of *gprof*. These functions pairs were ranked **1**, **2** and **3** respectively in the output of *SoftMon*. Also, these tools do not provide any function mapping across different applications.

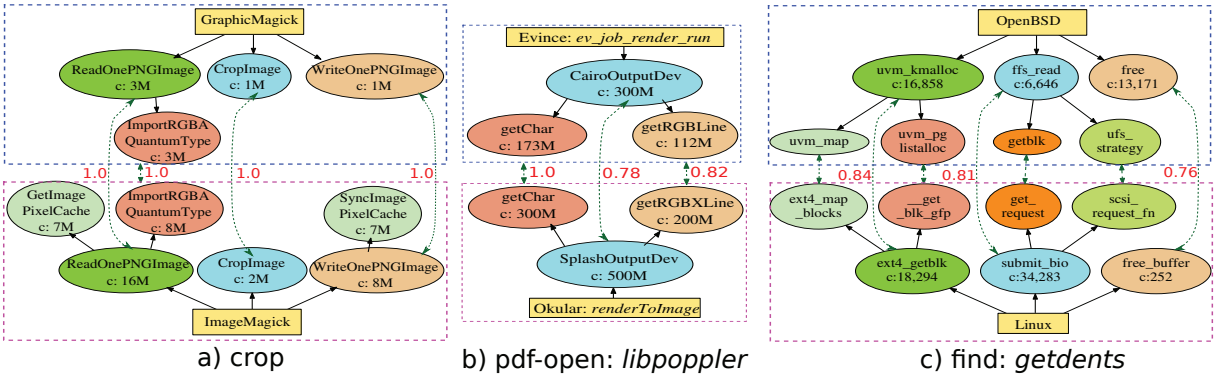

Figure 6: Case Studies: Output of *SoftMon* (a) Image Tools (b) Pdf readers (c) Operating Systems

4.5.2 **Case Study 2***: Pdf readers (Library: libpoppler).* We observe that the library *libpoppler* has the highest cycle footprint for both *Evince* and *Okular*. The *getChars* and *getRGBLine* functions inside libpoppler are the most frequent functions. Because of the presence of libraries, we need to construct and analyze the stack trace (see Figure 6(b)) of these functions for both *Evince* and *Okular* (as described in Section 3.6). The *OutputDev* function is called multiple times in *Okular* as compared to *Evince*. This leads to 20% more cycles in Okular. The function *getRGBXLine* in *Okular* has an extra instruction (for padding) in the *for* loop as compared to that of the similar function in *Evince*. This leads to 10% more execution cycles in *Okular*.

Comparison with *gprof* and *PIN*: The ranks of the matching functions *ev_job_render_run* and *renderToImage* were **3,905** and **3,813** respectively in the *gprof* output of *Evince* and *Okular*. The ranks of the matching functions *CairoOutputDev* and *SplashOutputDev* were **92** and **31** respectively. These functions pairs were ranked **1** and **2** respectively in the output of *SoftMon*.

4.5.3 **Case Study 3***: OS (Find).* In all three OSs the *getdents* – get the directory entries – system call has the maximum cycle footprint (up to 40%). This is followed by the *fstat* – get file status – system call (10-20%). Let us thus compare the execution of the *getdents* system call for OpenBSD and Linux and find out why OpenBSD is 26% faster.

Classifier Engine: Linux is dominated by only one cluster that accounts for 97% of the cycles. Whereas, in OpenBSD, we see two different clusters that contribute 66% and 32% respectively to the cycle count. The second cluster is an example of a fast-path system call invocation. Here, the *getdents* system call exits in the case when the I/O device is busy instead of waiting (like Linux). One of the reasons for Linux being slower is that it does not exploit this fast path.

Map Engine: We feed the partitioned *rm-trees* of both the OSs to the Map Engine. The output of the mapping is shown in Figure 6(c). There are three major partitions: a) allocate memory blocks, b) read the directory structure into the memory blocks, c) free memory blocks. The respective partitions match with a comment similarity index of 84%, 81% and 76% respectively. The *memory allocation* operation takes almost an equal number of cycles for both the OSs. But, the *read* operation of OpenBSD takes only 6,646 cycles as compared to 34,283 cycles in Linux.

Reasons (found after visualization):

Data Structure in the map function: Linux uses a *red-black tree* to store the directory structure whereas OpenBSD uses a *Trie* data-structure. The search operation in a *red-black tree* takes $O(log\ n)$ time, where n is the total number of elements in the tree. Whereas, the search operation's complexity in a *Trie* is $O(k)$, where k is the maximum length of all the strings in the set. The *red-black tree* is always a balanced tree, whereas in the case of a *Trie* we can have severely unbalanced trees in worst cases, leading to slower lookups. Hence, the choice of the data structure thus depends on the size of the file-system. Validation: [32] (Section 9.3 in the book).

Greedy page allocation: OpenBSD's developers have written a function *get1page* (not there in Linux), which quickly allocates a page from a pool of pages. If multiple pages need to be allocated, then all the pages after the first page are read in conventional fashion. This optimization is inefficient if we are running a lot of memory intensive workloads because the pool's size will become very large.

I/O batching (submit_bio): We found two different clusters in the case of OpenBSD. The *memory read* operation happens only in one of the clusters. The fast-path (second cluster) returns from the system call when the device is busy. Whereas in Linux, there are no fast paths. It calls a *__const_udelay* function when the device is not available. We validated this finding by looking through the code of the latest Linux kernel. We found that the *__const_udelay* function is not being used for this action in the later Linux kernel (4.3). We found that this *performance bug* was first reported in 2012 [14] but was ultimately diagnosed and fixed in 2018.

Comparison with *ftrace* and *dtrace*: The functions *uvm_map* and *ext4_map_blocks* were ranked **20** and **32** respectively. The functions *getblk* and *get_request* were ranked **15** and **26** respectively. These functions pairs were ranked **1** and **2** respectively by our tool.

4.6 Summary of Differences

Tables 6 and 7 shows a summary of the reasons found by *SoftMon* and subsequent manual analysis. In each row, we display the reason for the difference in performance for a particular benchmark using *SoftMon* and a reference to the online validation. We classify the reasons into two sets: performance bug (B) and feature (F). We found performance bugs that used inefficient data structures or algorithms, called unnecessary functions, or repeated previously done tasks.

Table 6: Summary of differences in Applications (B/F → Performance Bug/Feature)

	Function	Reasons	B/F	Validation
Image	*ReadImage*	*ImageMagick* makes a copy of the pixel data-structure but *GraphicMagick* passes a pointer.	B	*GraphicMagick* Blog [12]
Image	*CropImage*	*IM-7* uses a nested three level for loop to update the image but *GM* uses a single for loop and *memcpy*.	B	Verified from the source code [11]
Image	*ClonePixel*	In *IM-7*, *ClonePixel* function creates a copy of the image from the source cache to the destination cache.	B	*ImageMagick* Documentation [10]
PDF	*getChars*	This function is called multiple times in *Okular* as *Splash* is an inefficient library as compared to *Cairo*.	B	*Cairo* vs *Splash* [3]
PDF	*getRGBLine*	This function has an extra instruction in the *for* loop as compared to that of the function in *Evince*.	B	Verified from the source code [9]
PDF	*GUI lib*	*Evince* uses *libglib* for creating the GUI, whereas *Okular* used *libQT* for the same.	F	Verified from the source code
PDF	*full*	In *Xpdf*-full the file is loaded twice, first in a window and then in the presentation mode.	B	Verified from the source code
Text	*malloc*	*Geany* loads the file and runs multiple passes over it to determine encodings and highlighting.	F	*Geany* GitHub comments [8]
Text	*plugins*	*Geany* loads multiple plugins (example: spell check) by default. It can be turned off.	F	*Geany* Documentation [7]
Audio	*GUI lib*	*Rhythmbox* uses *libglib* and *libgobject* for creating the GUI, whereas *VLC* uses *libQT* for the same.	F	*VLC* vs *Rhythmbox* [23]
Audio	*Library*	*Rhythmbox* saves the audio file into a play-list by default. It can be turned off.	F	*Rhythmbox* Documentation [17]
Audio	*plugins*	*Rhythmbox* loads multiple plugins (example: python support) by default. It can be turned off.	F	*Rhythmbox* man-pages [18]
Mail	*GUI lib*	*Balsa* uses a *css* based library to render a general GUI. *Sylpheed* uses a fixed GUI format.	F	*Balsa* GTK library [1]

Table 7: Summary of differences in OSs (✔ : Optimization available, ✖ : Optimization not available, – : Not relevant)

	Reasons	Linux	OpenBSD	FreeBSD	B/F	Validation
Find	**Fast Path vs Slow Path**: function exits when device unavailable	✖	✔	✔	B	Linux change logs: *native_read_tsc*
Find	**Batching-Type1**: OpenBSD fetches more pages	✖	✔	✔	F	Verified from the source code comments [66]
Find	**Greedy Page Allocation**: *get1page* function	✖	✔	✔	F	OpenBSD user manual: *getblk*
Find	**Data Structure for storing directory**	Red-Black Tree	Trie	–	F	Book: Understanding the Linux Kernel (9.3)
Iscp	**Batching-Type2**: function *uiomove* to write data in one go	✖	✔	✔	F	OpenBSD user manual: *uiomove*
Copy	**Architecture Optimization**: *rep* is used to optimize copy	✔	✖	✖	B	Verified from the source code
Copy	**Prefetching (Sequential access)**: Linux uses prefetching	✔	✖	✖	B	FreeBSD user manual: *prefetch*
Copy	**Prefetching (Locks)**: type of lock algorithm used	seqlock	–	RW-lock	B	Linux user manual: *seqlock*
Fork	**Process Creation**: Copy on demand optimization	✔	✖	✖	B	FreeBSD: kernel limits [5]
OLTP	**Multi-core**: scheduling algorithm for equal work division	✔	✖	✔	B	OpenBSD: *Slashdot* article [15]
FileIO	**No Prefetching for non-sequential access**	✖	✔	✔	B	Linux: kernel documentation *read-ahead*
FileIO	**Data Structure for page-cache**	Radix Tree	–	Queue	F	Aggressive read-ahead in Linux [34]

We also discovered some specific features such as loading of plugins or multiple passes over the data that increase the loading time of the application. We also discovered features such as prefetching and caching in operating systems that benefit the application.

4.7 User Study

To validate the effectiveness of our tool, we performed a user study. The study had 10 participants: 1 faculty, 1 PhD student and 8 undergrad. students from the Computer Science (CS) department at IIT Delhi, India. All the participants were proficient in C/C++. We floated a non-graded (pass/fail) course that was open to all, and anybody who wished to participate joined (selected without bias). The benchmarks were divided among the students and the reasons for analyzing the performance were not conveyed to them before the end of the grading. We constructed a golden data set of the reasons using available online content such as source code comments, commit logs, Github user questions, code documentation, news-group discussions, and developer manuals (Tables-6, 7). The success criteria was that the students needed to analyze the source code and find reasons that are mentioned in our list of collected reasons.

We first held a session in which we demonstrated the different software categories and the benchmarks. We provided them with the output of the *PIN*, *gprof* and *ftrace* tools for the relevant benchmarks. We gave them the task of identifying reasons for the differences in performance for three benchmark pairs. Each participant spent a week (minimum 20 hours) to analyze these applications. We held another session after this task and checked their progress. Only one participant was able to find the relevant reasons for Image tools: *ReadImage* and *CropImage* functions. The image tools are one of the simplest and smallest (in terms of the number of lines of code) programs as compared to other applications. Other participants faced difficulty as there were a large number of functions, function mappings were not provided and the function call graph consisted of thousands of nodes.

We held another session and demonstrated the working of the *SoftMon* tool and provided them with the output (function mapping, pruned and annotated function call graphs) to all the participants. All the participants were able to find the reasons described in Table 6 and Table 7 for a benchmark-pair within **30 minutes**.

Summary: *SoftMon* saved **500** man hours of analysis, and found **25** reasons for 6 categories of large open-source programs.

5 CONCLUSION

We solved a very ambitious problem in this paper, which is to compare some of the largest open-source programs and explain the reasons for their performance difference. We were able to find a diverse set of reasons that explain most of the differences, and we were able to validate them against various sources. Out of the reasons, many were performance bugs, and the rest were application-specific features. *SoftMon* takes just about 200s to complete the analyses for the largest code bases (operating systems) and to reduce the search space from roughly 50-100k functions to 10-15 functions.

ACKNOWLEDGMENTS

We would like to thank Prathmesh Kallurkar who helped build the initial prototype. A big thank you to everyone who participated in our user studies and helped us with evaluating the tool. This research is supported by NetApp, Bangalore. We express our thanks to the anonymous reviewers of the manuscript.

REFERENCES

[1] 2019. Balsa GTK library. https://developer.gnome.org/gtk3/stable/GtkCssProvider.html#gtk-css-provider-load-from-path
[2] 2019. Byte-UnixBench. https://github.com/kdlucas/byte-unixbench.
[3] 2019. Cairo vs Splash. https://gitlab.freedesktop.org/poppler/poppler/issues/285.
[4] 2019. Dtrace for BSDs. http://dtrace.org/blogs/
[5] 2019. FreeBSD: kernel limits . https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/configtuning-kernel-limits.html.
[6] 2019. Ftrace for Linux. https://elinux.org/Ftrace.
[7] 2019. Geany Manual. https://www.geany.org/documentation/manual/.
[8] 2019. Geany: slow loading. https://github.com/geany/geany/issues/1969.
[9] 2019. getRGBXLine vs getRGBLine. https://gitlab.freedesktop.org/poppler/poppler/commit/e9350899e77c28452c48b56349ad7758b3fd47ba?view=inline.
[10] 2019. ImageMagick PixelCache. https://imagemagick.org/script/architecture.php#cache.
[11] 2019. ImageMagick: transform.c, line:676. https://github.com/ImageMagick/ImageMagick/blob/master/MagickCore/transform.c.
[12] 2019. ImageMagick vs GraphicMagick. http://www.graphicsmagick.org/benchmarks.html.
[13] 2019. Kernel Glossary. https://kernelnewbies.org/KernelGlossary.
[14] 2019. Linux: performance bug. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/998204.
[15] 2019. OpenBSD multi-core performance. https://bsd.slashdot.org/story/18/06/19/2327212/.
[16] 2019. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[17] 2019. Rhythmbox library. https://help.gnome.org/users/rhythmbox/stable/library.html.en.
[18] 2019. Rhythmbox man-pages. https://linux.die.net/man/1/rhythmbox.
[19] 2019. Technology Evaluation Centers. https://www3.technologyevaluation.com/.
[20] 2019. The biggest codebases in history. https://www.freecodecamp.org/news/the-biggest-codebases-in-history-a128bb3eea73/.
[21] 2019. The cscope Tool. http://cscope.sourceforge.net/.
[22] 2019. The spaCy Tool. https://spacy.io/.
[23] 2019. VLC vs Rhythmbox. https://ubuntu-mate.community/t/rhythmbox-should-not-be-the-default-music-player-for-ubuntu-mate/18093/4.
[24] Christophe Alias and Denis Barthou. 2003. Algorithm Recognition based on Demand-Driven Data-flow Analysis.. In *WCRE*, Vol. 3. 296–305.
[25] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.
[26] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 404–419.
[27] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.
[28] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. 2012. SLICC: Self-assembly of instruction cache collectives for oltp workloads. In *MICRO*.
[29] Islam Atta, Pinar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. 2013. STREX: boosting instruction cache reuse in OLTP workloads through stratified transaction execution. In *ACM SIGARCH*.
[30] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to Recognize Functions in Binary Code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 845–860.
[31] Cor-Paul Bezemer, Johan Pouwelse, and Brendan Gregg. 2015. Understanding software performance regressions using differential flame graphs. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 535–539.
[32] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".
[33] Robert S Boyer and J Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10 (1977), 762–772.
[34] Cao. 2018. Aggressive read-ahead in Linux. https://lwn.net/Articles/372384/.
[35] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.
[36] J Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael D Smith. 1996. The measured performance of personal computer operating systems. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 3–40.
[37] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 89–98.
[38] Yan Cui, Yingxin Wang, Yu Chen, and Yuanchun Shi. 2011. Experience on comparison of operating systems scalability on the multi-core architecture. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 205–215.
[39] Charlie Curtsinger and Emery D Berger. 2015. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 184–197.
[40] Arnaldo Carvalho De Melo. 2010. The new linux perf tools. In *Slides from Linux Kongress*, Vol. 18.
[41] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky. 2008. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 59–70.
[42] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 303–317.
[43] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*.
[44] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *MICRO*.
[45] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*. Citeseer, 161–174.
[46] Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.
[47] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.
[48] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.
[49] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.
[50] Harald Andrés Helfgott, Jitendra Bajpai, and Daniele Dona. 2017. Graph isomorphisms in quasi-polynomial time. *arXiv preprint arXiv:1710.04574* (2017).
[51] Matthew Honnibal and Mark Johnson. 2015. An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 1373–1378.
[52] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 57–67.
[53] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.
[54] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.
[55] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.
[56] Abhilash Jindal and Y. Charlie Hu. 2018. Differential energy profiling: energy optimization via diffing similar apps. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. USENIX Association.
[57] Prathmesh Kallurkar and Smruti R Sarangi. 2016. pTask: A smart prefetching scheme for OS intensive applications. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.
[58] Prathmesh Kallurkar and Smruti R Sarangi. 2017. Schedtask: a hardware-assisted task scheduler. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 612–624.
[59] Sukrit Kalra, Ayush Goel, Dhriti Khanna, Mohan Dhawan, Subodh Sharma, and Rahul Purandare. 2016. POLLUX: safely upgrading dependent application libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 290–300.
[60] George Karypis, Eui-Hong Sam Han, and Vipin Kumar. 1999. Chameleon: Hierarchical clustering using dynamic modeling. *Computer* 8 (1999), 68–75.
[61] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 166–177.
[62] Alexey Kopytov. 2004. Sysbench: A System Performance Benchmark. https://github.com/akopytov/sysbench.
[63] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2, 1-2 (1955), 83–97.
[64] Tilman Küstner, Josef Weidendorfer, and Tobias Weinzierl. 2009. Argument controlled profiling. In *European Conference on Parallel Processing*. Springer, 177–184.
[65] Kevin Lai and Mary Baker. 1996. A Performance Comparison of UNIX Operating Systems on the Pentium.. In *USENIX Annual Technical Conference*. 265–278.
[66] Michael Larabel. 2016. File system: BSDs are better than Linux. https://www.phoronix.com/scan.php?page=article&item=3bsd-10linux&num=2.

[67] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[68] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.

[69] Robert Metzger and Zhaofang Wen. 2000. *Automatic algorithm recognition and replacement: a new approach to program optimization.* MIT Press.

[70] Sang-Og Na and Dong-Soo Han. 1999. Microbenchmarks of operating system performance on the multiprocessor platform. In *TENCON 99. Proceedings of the IEEE Region 10 Conference*, Vol. 1. IEEE, 435–438.

[71] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 709–724.

[72] Mireille Régnier. 1989. Knuth-Morris-Pratt algorithm: an analysis. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 431–444.

[73] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems*, Vol. 48. ACM, 305–316.

[74] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 33–46.

[75] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 561–578.

[76] Diomidis Spinellis. 2008. A tale of four kernels. In *Proceedings of the 30th international conference on Software engineering*. ACM, 381–390.

[77] Balaji Vasan Srinivasan, Tanya Goyal, Varun Syal, Shubhankar Suman Singh, and Vineet Sharma. 2016. Environment Specific Content Rendering & Transformation. In *Companion Publication of the 21st International Conference on Intelligent User Interfaces*. ACM, 18–22.

[78] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[79] Nisha Talagala, Satoshi Asami, and David A Patterson. 1998. *A Comparison of PC Operating Systems for Storage and Support.* University of California, Berkeley, Computer Science Division.

[80] Nathan R Tallent and John M Mellor-Crummey. 2009. Effective performance measurement and analysis of multithreaded applications. In *ACM Sigplan Notices*, Vol. 44. ACM, 229–240.

[81] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 90–100.

[82] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* (2018).

[83] Xiangyu Zhang and Rajiv Gupta. 2005. Matching execution histories of program versions. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 197–206.