F3M: Fast Focused Function Merging

Sean Stirling[*], Rodrigo C. O. Rocha[†], Kim Hazelwood[‡], Hugh Leather[‡], Michael O'Boyle[†] and Pavlos Petoumenos[§]
[*]*Codeplay, United Kingdom*, sean.stirling@codeplay.com
[†]*University of Edinburgh, United Kingdom*, {rrocha,mob}@ed.ac.uk
[‡]*Facebook AI Research, United States*, {kimhazelwood,hleather}@fb.com
[§]*University of Manchester, United Kingdom*, pavlos.petoumenos@manchester.ac.uk

***Abstract*—From IoT devices to datacenters, code size is important, motivating ongoing research in binary reduction. A key technique is the merging of similar functions to reduce code redundancy. Success, however, depends on accurately identifying functions that can be profitably merged. Attempting to merge all function pairs is prohibitively expensive. Current approaches, therefore, employ summaries to estimate similarity. However these summaries often give little information about how well two programs will merge. To make things worse, they rely on exhaustive search across all summaries; impractical for real-world programs.**

In this work, we propose a new technique for matching similar functions. We use a hash-based approach that better captures code similarity and, at the same time, significantly reduces the search space by focusing on the most promising candidates. Experimental results show that our similarity metric has a better correlation with merging profitability. This improves the average code size reduction by 6 percentage points, while it reduces the overhead of function merging by 1.8x on average and by as much as 597x for large applications. Faster merging and reduced code size to compile at later stages mean that our approach introduces little to no compile time overhead, while in many cases it makes compilation faster by up to 30%.

***Index Terms*—Code-Size Reduction, Function Merging, LLVM, Compiler Optimization**

I. Introduction

Code size is a first order limitation for computing devices with constrained resources. An application binary package of hundreds of megabytes might be easy to accommodate on a server but will put extra strain on an embedded device or a smartphone, especially if that package needs to be downloaded first. Application delivery platforms might impose additional restrictions on executable size. For example, Google Play does not generally allow APKs larger than 100MB [1], while the Apple App Store does not allow executables larger than 500MB [2]. Under such conditions, larger binary sizes translate directly to higher hardware costs and/or lower adoption.

One promising approach for reducing binary size is function merging. At its most straightforward [3], [4], the idea is to identify identical functions, redirect all their calls to only one of them, and remove all the redundant copies of these identical functions. More sophisticated implementations are able to merge non-identical functions as long as they are similar in some way, for example having isomorphic control flow graphs [5] or overlapping instruction subsequences [6]–[8]. The latter approach, known as function merging by sequence alignment, is very flexible as it can merge even functions with little structural or algorithmic similarities. Candidate function pairs are represented as linear sequences of instructions. A sequence alignment algorithm identifies the overlapping parts of the two sequences. A code generator produces a replacement merged function which contains a single copy of each overlapping subsequence. The remaining non-overlapping subsequences are copied in as they are but their execution is guarded by an original function identifier. The new function replicates the functionality of the two original functions while removing the redundant instruction subsequences.

While this crop of sophisticated function merging algorithms can get more code size reduction, finding the right functions to merge is still very expensive and can hinder the practicality of function merging. A simple function hashing scheme is guaranteed to find identical functions to merge. For sequence alignment on the other hand, there is no accurate way to find functions that align well, apart from attempting to align every function with every other function. The state-of-the-art technique uses a heuristic to find somewhat acceptable function pairs [6], [8]. It computes a *fingerprint* for each function, a fixed size vector of instruction opcode frequencies in the function. Choosing function pairs is then equivalent to finding all nearest neighbor pairs in the n-dimensional fingerprint space. While comparing all fingerprints against all other fingerprints is faster than actually aligning all possible function pairs, this is still an expensive quadratic process. When we used the publicly available code on a range of applications, it worked sufficiently well for the small and medium sized programs of traditional benchmark suites but on Google Chrome matching fingerprints took almost two days, a very high overhead for an approach claiming to deliver function merging for free [8]. Unless we overcome this limitation, sequence alignment based function merging is unlikely to leave the confines of academic research.

Our fundamental insight for solving this problem is that function similarity ranking is at its core just a form of Nearest Neighbor Search (NNS) and there are more efficient NNS approaches than exhaustive search. In particular, approximate NNS techniques, which trade-off accuracy for vastly reduced search times, could accelerate function merging by orders of magnitude. But for this to happen, we also need better

This work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/L01503X/1 (CDT in Pervasive Parallelism). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

fingerprints. Approximate search delivers good results only as long as the similarity metric is a good proxy for the real similarity of the nearest neighbors [9]. For our problem, this means that fingerprint similarity needs to correlate well with function similarity. This is not currently the case. Fingerprints do not carry any information about the structure of code, only about instruction type frequencies. Fingerprint similarity is at its best an optimistic metric: what is the highest possible similarity between two functions if they happen to be as structurally similar as possible. At its worst, it is little better than a random number which correlates poorly with function similarity.

In this work, we propose Fast Focused Function Merging (F3M) a new combination of function fingerprint and search strategy that solves these problems, delivering significant speedups and improved code size reduction in the process. Specifically, we use a MinHash [10] representation of the code as the fingerprint and Locality Sensitive Hashing (LSH) [11] to quickly discover close neighbors in the fingerprint space. MinHash reduces the function into a fixed length vector of hashes, each hash being associated with a different subsequence of instructions in the function. Two functions with similar MinHashes are thus likely to have many identical subsequences which makes them a good candidate pair for merging. With LSH, we assign each function into a small number of buckets based on their MinHashes. When searching for a good merging pair for a function, we only calculate MinHash similarities with the functions mapping into the same buckets, a small fraction of the functions of the program.

This new approach results in significant speedups which become more and more pronounced as the size of the application grows. For large ones, like Google Chrome, F3M delivers orders of magnitude faster function merging, reducing its runtime from days down to minutes. Regardless of size, applications benefit from the improved fingerprint which leads to better merging candidate pairs and higher code size reduction. Evaluated over a large selection of benchmarks and real applications, we decrease object file size by 7.6% while also decreasing compilation time 5.6% compared to a baseline without function merging.

In this paper, we make the following three contributions:

- We introduce a new function similarity metric that correlates strongly with how well two functions merge.
- We combine this metric with a hashing scheme that actively adapts to the code, eliminating in the process most of the overhead of finding similar functions.
- We show that our approach works on real large code bases where we accelerate function merging by orders of magnitude while delivering superior code size reduction.

II. Background and Motivation

In this section, we describe the state-of-the-art function merging technique and show its inefficiency when optimizing large programs.

A. Function Merging via Sequence Alignment

In the last few years, there has been a flurry of function merging techniques that rely on Sequence Alignment [6]–[8].

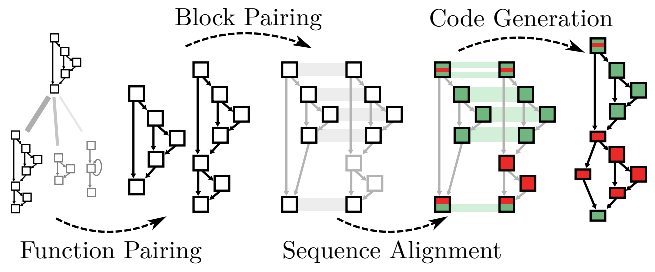


Fig. 1. Overview of HyFM's function merging operation.

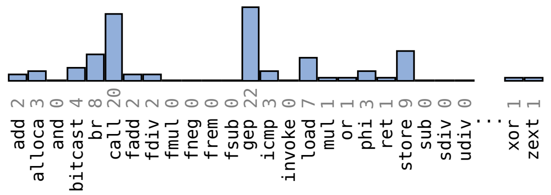


Fig. 2. Example of a fingerprint used by HyFM for pairing functions.

They have significant differences but they all work following these key stages: a) select pairs of similar functions for merging, b) merge each pair of functions by using sequence alignment to identify equivalent code that can be reused by both functions, and c) if deemed profitable, replace the two original functions with the merged function that combines the functionality of both. The execution of mismatched subsequences in the merged function is guarded by an original function identifier but matching subsequences are not guarded and are executed regardless of the function identifier. Figure 1 illustrates the function merging workflow in HyFM [8], which is the state-of-the-art used throughout this paper. Its code alignment strategy works on the basic block level, with a linear alignment algorithm applied on pairs of similar blocks.

Merging two functions is an expensive operation. Applying it on every possible pair of functions to discover which ones should be merged is infeasible even for medium sized programs. Instead, all existing approaches rely on a ranking-based mechanism for pairing similar functions. Each function is associated with a *fingerprint*, i.e., a vector representing the frequencies of all the instruction opcodes in its function body. Figure 2 shows an example of a fingerprint. When searching for function pairs to merge, called *ranking*, each candidate function is compared against all other functions in terms of fingerprint similarity. The most similar function, the one with the lowest fingerprint distance from the candidate, is then aligned and, perhaps, merged with the candidate.

B. Limitations of the State of the Art

While all these works have significantly improved the performance of the function merging operation, the ranking mechanism used for pairing similar functions still scales poorly to larger codebases. Figure 3 shows the compilation time breakdown of HyFM for three different programs: `400.perlbench`, the Linux kernel, and Google Chrome.

The ranking overhead grows quadratically as the number of functions in the program increases. While for

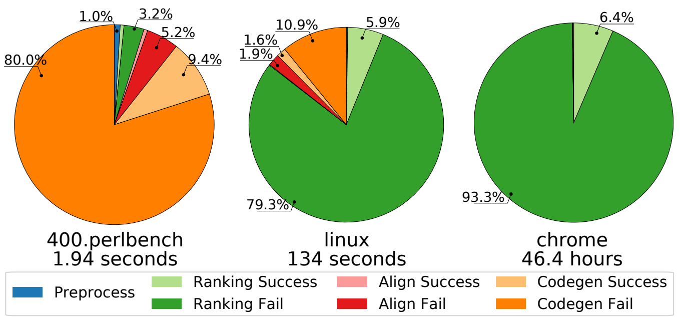


Fig. 3. Breakdown of the relative runtime for the different stages of HyFM [PA]. Ranking takes 0.07 seconds, 1.9 minutes, and 46.3 hours on `400.perlbench`, `Linux`, and `Google Chrome`, respectively. More than 90% of the ranking time is spent on unprofitable function pairs.

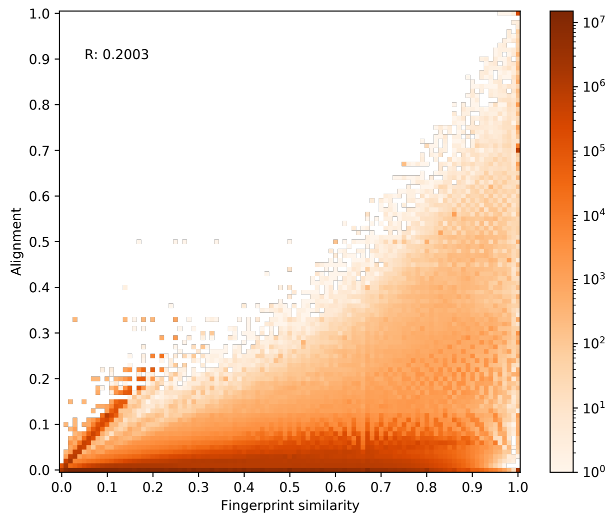
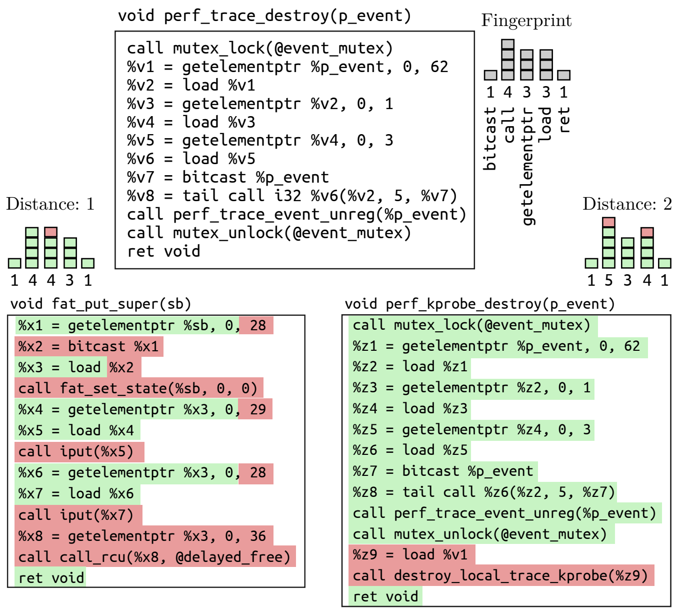


Fig. 4. Heatmap of normalized fingerprint similarity for all 800M possible candidate function pairs in the Linux kernel versus their alignment ratio. The 2D space is discretized into squares of size 0.01x0.01. Each square's color indicates the number of functions pairs with that similarity and alignment ratio. Quality of alignment and fingerprint similarity show low correlation (R=0.20)

`400.perlbench`, ranking its 1837 functions represents only a small (but not negligible) part of the optimization's runtime, for the 45k functions of Linux, ranking takes minutes and is responsible for 80% of HyFM's runtime. For the 1.2m functions of Google Chrome, practically the whole compilation overhead (almost two days!) is due to ranking. Even worse, most of this effort, about 93%, does not contribute anything towards reducing code size, it is wasted pairing functions that will be later deemed unprofitable by the compiler's cost model.

This is a result of the quadratic nature of the function ranking mechanism. The act of comparing fingerprints itself is only moderately expensive: for HyFM it involves calculating the Manhattan distance of two integer vectors of 68 elements. But even medium sized programs, e.g. Linux, need to perform approximately 800M such calculations which is enough to dominate the runtime of function merging.

This wasteful aspect of HyFM is a side-effect of the poor quality of the ranking metric itself. Figure 4 shows the ratio of aligned instructions for all possible function pairs of the Linux


```
void perf_trace_destroy(p_event)
call mutex_lock(@event_mutex)
%v1 = getelementptr %p_event, 0, 62
%v2 = load %v1
%v3 = getelementptr %v2, 0, 1
%v4 = load %v3
%v5 = getelementptr %v4, 0, 3
%v6 = load %v5
%v7 = bitcast %p_event
%v8 = tail call i32 %v6(%v2, 5, %v7)
call perf_trace_event_unreg(%p_event)
call mutex_unlock(@event_mutex)
ret void
```

```
void fat_put_super(sb)
%x1 = getelementptr %sb, 0, 28
%x2 = bitcast %x1
%x3 = load %x2
call fat_set_state(%sb, 0, 0)
%x4 = getelementptr %x3, 0, 29
%x5 = load %x4
call iput(%x5)
%x6 = getelementptr %x3, 0, 28
%x7 = load %x6
call iput(%x7)
%x8 = getelementptr %x3, 0, 36
call call_rcu(%x8, @delayed_free)
ret void
```

```
void perf_kprobe_destroy(p_event)
call mutex_lock(@event_mutex)
%z1 = getelementptr %p_event, 0, 62
%z2 = load %z1
%z3 = getelementptr %z2, 0, 1
%z4 = load %z3
%z5 = getelementptr %z4, 0, 3
%z6 = load %z5
%z7 = bitcast %p_event
%z8 = tail call %z6(%z2, 5, %z7)
call perf_trace_event_unreg(%p_event)
call mutex_unlock(@event_mutex)
%z9 = load %v1
call destroy_local_trace_kprobe(%z9)
ret void
```

Fig. 5. Linux function `perf_trace_destroy` and the merging candidate selected by HyFM `fat_put_super`. Their fingerprints differ only in one value but half their instructions are not matched and another quarter has different operands. HyFM cannot merge the two functions profitably. The ideal candidate would be `perf_kprobe_destroy` which has a less similar fingerprint but is almost identical to `perf_trace_destroy`.

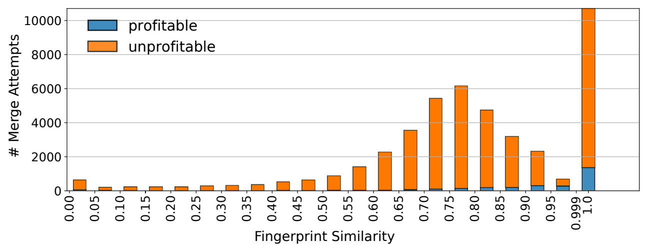


Fig. 6. Normalized fingerprint similarity histogram for function pairs selected by HyFM, indicating whether the pairs are profitable or not. These pairs are by definition nearest neighbors in the fingerprint space but they might still be distant from each other. Profitable pairs are more common for similar fingerprints but almost 10% of the profitable pairs have low similarity (< 0.5).

kernel versus the normalized similarity of their fingerprints. The correlation between the two is low, around 21%. Fingerprints only carry information about the frequency of different instruction opcodes in the functions, ignoring important aspects, such as the order these instructions appear in the function, their data types, or their inputs. As a result, dissimilar functions often have similar or even identical fingerprints. Conversely, functions with enough similarity may have vastly different fingerprints. Figure 5 shows such an example from the Linux kernel. HyFM matches functions `perf_trace_destroy` and `fat_put_super` because their fingerprint distance is only one. The two functions have similar instruction frequencies but their structure is so different that they cannot be merged profitably. At the same time, `perf_trace_destroy` aligns almost perfectly with function `perf_kprobe_destroy` but they are never selected because the two extra instructions in the second function lead to a fingerprint distance of two.

This does not just mean that HyFM often selects candidate pairs that are unprofitable, it also means that speeding up ranking with an approximate nearest neighbor approach will have a further detrimental effect on the effectiveness of function merging [9]. This is clear in Figure 6, which shows the distribution of fingerprint similarities for the Linux function pairs selected by HyFM through a nearest neighbor search. While the majority of selected pairs have similar fingerprints, selected pairs can be in very different areas of the fingerprint space: 8% of them have a similarity less than 0.5, 25% a similarity less than 0.7. These are not low quality merging candidates. 8% to 10% of them are profitable, roughly the same success rate as for similar (but not identical) fingerprints. A faster approximate search that focuses in neighboring areas of the fingerprint space would miss such distant profitable pairs leading to code size increase.

Making function merging feasible for large codebases requires us to solve these two problems together: reduce the computational complexity of ranking *and* improve the quality of the ranking metric. The rest of the paper examines how.

III. Our Approach

In this section, we propose F3M (Fast Focused Function Merging), an application of the MinHash and Locality Sensitive Hashing techniques that can effectively pair functions at just-above-linear speeds. Our solution has a three-fold effect: 1) A better metric for estimating the similarity of two functions leads to more profitable function merging; 2) Sufficiently dissimilar functions are not even considered for merging; 3) Merging candidate lookup times are reduced enough to be of little concern anymore. We also discuss several design choices that can be tuned to better suit the needs of the application.

A. Overview

In a pre-computational stage, we generate a MinHash fingerprint for each function. The MinHash fingerprint is a fixed-size integer vector containing multiple hashes computed across all instructions of the function. Each hash is directly produced by one instruction subsequence in the function's body, so identical hashes indicate identical instruction subsequences.

Based on the hashes in their fingerprints, we assign each function to a number of buckets in a hashmap structure. Functions in the same bucket have overlapping fingerprints, so they are more likely to be similar to one another. We exploit this by limiting function ranking (and full fingerprint comparison) to candidate functions that share buckets. This reduces the number of candidates that each function must be compared against down to only a handful or even none.

The rest of the function merging process continues after that in the same way it does in previous works [8]: we align the two selected functions, we generate the code of the merged function, and if it is profitable, we replace the selected functions with the merged one. Function merging as a whole is applied after all source files have been optimized for size (-Os) and their IRs have been linked together.

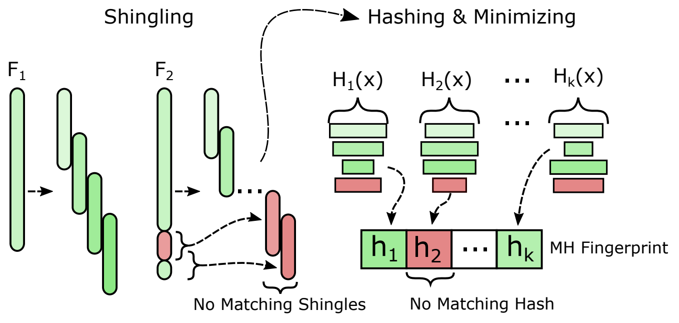


Fig. 7. The MinHash algorithm: Textual documents are broken up into overlapping subsequences; Each subsequence is hashed with k different hash functions; For each function, the smallest hash is saved creating a fingerprint of k entries. In this example, two functions differ in only a couple extra instructions inside F2. For F2's fingerprint, this creates shingles and hashes with no matches in F1, representing the slight difference between them.

B. MinHash Fingerprint

In Section II, we exposed the low correlation between the existing similarity metric used for pairing functions and the actual result of aligning the paired functions. A better metric should be able to identify when two functions have instruction subsequences that can be aligned. If we represent functions as sets of such instruction subsequences, then the *Jaccard index* perfectly captures this similarity. Formally, the Jaccard index of two sets is the fraction between the number of intersecting elements and the magnitude of their union:

$$J(A,B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid} \tag{1}$$

In the case of function similarity, the Jaccard index represents the likelihood of matching instruction subsequences. Unfortunately, calculating the Jaccard index is linear on the size of the sets so it cannot work as a practical function similarity metric for moderately sized programs. Instead, we can rely on *MinHash* [10], a popular technique for estimating the Jaccard index in constant time. Figure 7 illustrates the generation of the MinHash fingerprint. The central idea is to split the text into a set of *shingles*, which are fixed-length overlapping subsequences, and then to hash each shingle multiple times using a different hash function each time. For each hash function, we save the smallest value generated. Given a set of k hash functions, this gives us a fingerprint of size k.

Each of the hashes in the fingerprint corresponds (roughly) to a randomly selected shingle. When two fingerprints contain the same hash value, this indicates a very high probability that the corresponding texts include the same shingle. When two hash values are different, this indicates a lower probability of matching shingles. Over the whole length of the fingerprints, the likelihood of identical hash values (the fingerprints' Jaccard index) is an approximation of the likelihood of identical shingles (the texts' Jaccard index) within an error of $\mathcal{O}(1/\sqrt{k})$.

Applying MinHash directly on the function's textual representation can work but delivers suboptimal results. MinHash, as described above, would produce identical hashes only for 100% identical instruction sequences. But for the purposes of function merging, two shingles can be merged even if they are not textually identical, for example if the instruction opcodes are the same but their arguments differ. We need MinHash to help us identify such shingle pairs which means producing identical hashes for them despite their textual differences.

A straightforward way to achieve this is by applying MinHash on a different representation of the function, ignoring aspects of the function that are insignificant to the alignment strategy. For this representation we translate each instruction into a 32-bit integer that encodes the four most important properties with regards to merging: opcode, result type, number of operands, and operand types. Operand count is already an integer and LLVM already associates an integer with each opcode, so we can store these two numbers directly in specific bits of the encoded instruction. Types are a bit more complicated. We assign a unique number to each type, for example, we can use the addresses of the unique pointers used internally by LLVM for each type. For the combined type of all the operands, we multiply all the numerical representations of the operand types. We store this number into the remaining bits of the encoded instruction. Despite its simplicity and certain ad-hoc aspects of it, this scheme generally succeeds in producing identical integer numbers for mergeable instructions and different numbers for instructions that cannot merge.

Having transformed the sequence of instructions into a sequence of integers, we split it into shingles of length $K = 2$. We empirically found that this produces the best results: $K > 2$ leads to fewer hash matches and higher cost for generating the hashes, while $K = 1$ works on individual instructions and does not capture the function's structure.

Finally, we hash the shingles, get the minimum for each hash function, and produce the fingerprint. In our implementation, we use for that purpose the FNV-1a variant of the Fowler–Noll–Vo hash function [12]. We chose FNV-1a for its robustness to permutations, computational efficiency, widespread use in practice, and simple implementation but any well behaving hash function should produce similar results. To reduce the computational cost, instead of using k different variations of the FNV-1a hash, we use a single function whose output for each shingle is xor-ed with k different random values to produce k different hashes. This has a very small effect on the quality of the MinHash while making its generation many times faster.

Larger values for k give a better estimation of the Jaccard index of the two functions but deliver diminishing returns while increasing the effort required to compare fingerprints. In practice, a value of $k = 200$ (the default for the rest of the paper) achieves a good compromise as we see in Section IV.

C. *Locality Sensitive Hashing*

Comparing MinHash fingerprints requires calculating the Jaccard Index of the two fingerprints (or its opposite, their Jaccard distance). While this is usually faster than directly calculating the Jaccard Index of two functions or even aligning them, it is still an expensive operation if performed exhaustively on all function pairs.

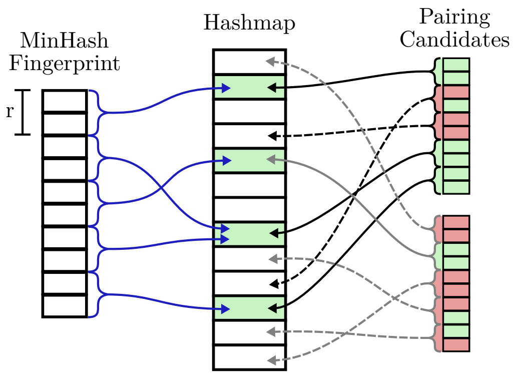


Fig. 8. LSH - Similar sequences hashing to the same buckets. Three fingerprints with 10 hashes each, setting b to 5 and r to 2. References to each item are placed in each of their respective buckets. Since they share at least one of their *bands* they will be compared during candidate lookup.

Luckily, MinHash works well with an approximate nearest neighbor search strategy known as *Locality Sensitive Hashing* (LSH). The underlying idea is to group together fingerprints with overlapping hashes and limit the search to within these groups. We associate each fingerprint with b groups by breaking up the vector of integers representing the fingerprint into b non-overlapping r-sized sub-vectors, where $k = b{*}r$ is the size of the fingerprint. We then hash each sub-vector using FNV-1a yielding b values. Similar fingerprints are likely to match on at least one of these b values, known as *bands*. Conversely, fingerprints with no matching bands are unlikely to contain meaningful similarities. Hence, when searching for similar items, only the ones which share at least one of these bands needs to be compared against, eliminating the vast majority of pairwise computations.

In practice, we implement this around a *hashmap*. For each band, we insert the function into the corresponding hashmap bucket. When we start merging, we iterate serially over all functions to find a similar function for each. We do this by getting that function's bands and retrieving the other functions placed in the same buckets. This principle can be seen in Figure 8. We then calculate the Jaccard distances of the function with all its candidates, selecting the nearest one.

The values of b and r have a direct effect on the probability that two items will share at least one of their bands. For the same fingerprint size, increasing r and decreasing b lowers the probability of matching bands, whereas decreasing r and increasing b will make matching on at least one of their *bands* much more likely. It only takes sharing one *band* for two items to be fully compared. The probability that two items will share at least one of those bands can be calculated as follows:

$$p = 1 - (1 - s^r)^b \qquad (2)$$

where s is the Jaccard similarity of the MinHashes of the two items. As two fingerprints become more similar, the chances of them being placed into the same bucket become larger. Depending on the number of functions and the minimum pairwise similarity that we aim for, b and r can be adjusted appropriately to favor more bucket matches of less similar

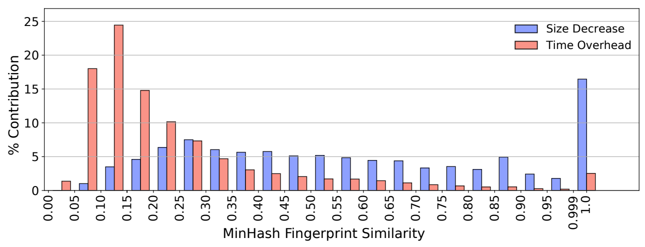


Fig. 9. Contribution of each function pair selected by F3M on Linux in terms of merging time overhead and estimated size reduction. Results are grouped by fingerprint similarity. Less similar pairs are responsible for most of the time overhead but contribute little to code size reduction.

fingerprints at the cost of more fingerprint comparisons or, conversely, fewer comparisons and faster ranking at the cost of potentially missing a few highly similar fingerprints.

While most buckets have few or no functions, there are outliers with a large number of functions. This is usually due to very common instruction subsequences, causing the same hash values to be generated for multiple functions, which lead to the same bucket being selected time and time again. This is a common occurrence, especially for larger programs and smaller r values, though it only affects a very small number of buckets. Still it may limit the efficiency of our approach, forcing us to calculate the Jaccard similarity of each fingerprint with thousands or tens of thousands of other fingerprints mapping into the same bucket. For that reason, we limit the number of fingerprint comparisons per bucket to 100. As we examine in Section IV, this affects only a few buckets, allowing highly similar functions to match through other, less crowded, buckets.

D. Adaptive Construction

Previous implementations of sequence alignment suffer from having to align every selected pair of functions regardless of how dissimilar the two functions may be. This is a direct consequence of the inexact similarity metric: functions with different fingerprints are still likely to be profitably merged (see Figure 6) so we need to try merging them. In contrast, Figure 9 shows the relative contributions in terms of code size reduction and merging overhead of all the function pairs selected by F3M when applied to Linux. These contributions are accumulated by fingerprint similarity. This Figure indicates both that less similar pairs of MinHash fingerprints are less likely to lead to significant code size reduction *and* that these pairs are responsible for most of the overhead of merging. Merging them is often not worth the effort.

Despite that, we discovered (see Section IV-D) that there is no global ideal cutoff point for rejecting dissimilar pairs. The same cutoff point can cause us to reject many profitable pairs for some applications while not rejecting enough unprofitable pairs for others. While we cannot predict this ahead of time, we can predict how each outcome would affect us. For smaller programs, function merging is already fast so a few extra unsuccessful merges will have little negative impact on the overall compilation, while a few missed successful merges will have a disproportionate effect on code size. Conversely, for larger programs accelerating function merging is critical, while skipping a few successful merges would have little effect given the large number of merged functions.

We empirically found that programs with fewer than 5000 functions do not benefit from aggressive similarity thresholds. A very conservative threshold of 0.05 is enough to catch the few very dissimilar pairs that may be selected but almost definitely will not be profitably merged. LSH by design already filters out most of these pairs, so this low threshold has little effect. At the other end, we found that functions with similarity over 0.4 are very likely to be profitable and are almost always worth our effort, even when dealing with very large programs. Between those two extremes, we need the threshold to rise gradually.

Based on these observations, we define our threshold t as:

$$t = \begin{cases} 0.05 & x <= 10^{3.5} \\ \frac{log_{10}(x)-3.0}{10} & 10^{3.5} < x < 10^{7} \\ 0.4 & x >= 10^{7} \end{cases} \tag{3}$$

where x is the number of functions in the program.

As discussed earlier, the b and r sizes have a strong effect on the efficiency of our searching strategy. According to Equation 2, more bands means higher probability of two similar fingerprints sharing a bucket (and thus a higher probability of calculating their fingerprint similarity) but it also means more buckets to examine when looking for matches. As we show in Section IV-D, this creates a delicate trade-off between the effectiveness of function merging at reducing code size and the overhead it introduces.

Given that we already reject pairs with similarity below the threshold, there is no benefit from discovering such pairs to begin with, so we need values for k, r, and b that reflect that. We use Equation 2 to connect the three values to the similarity threshold under the following (relatively ad-hoc) restrictions: a) r always equals 2, and b) we should have at least 90% probability of discovering pairs with similarities slightly above the threshold. This gives us the following equation:

$$b = \lceil \frac{log(0.1)}{log(1.0 - (t + 0.1)^2)} \rceil \tag{4}$$

Since this value is close to 100 for small programs with less than 5k functions, we set it to exactly 100 for them. For larger programs the number of bands decreases fast: 57 for programs with 10k functions, 25 for 100k functions, 14 for 1m functions. By reducing the number of bands (and with them the fingerprint size) so aggressively, we are able to cut the ranking overhead drastically while having a minimal impact on the ability of F3M to discover profitable function pairs.

E. Code Generation

F3M's core aim is to improve the selection of functions to merge. This is mostly orthogonal to *how* they will be merged, so in this work we reuse the code merging part of HyFM.

In the process of evaluating F3M, we found that the binaries produced for some programs were broken. We traced the source of the problem back to how HyFM (and SalSSA [6] before it) resolves cases where the merged code violates the dominance property of the SSA form. While most such violations are resolved by inserting new phi-nodes, a small number of them is resolved by breaking the use-def chains of variables via the stack memory: HyFM stores them in memory directly after definition and loads them back right before they are used.

In two cases where HyFM cannot store a variable right after definition, this is handled incorrectly. In the first one, the definition is a phi-node followed by other phi-nodes. HyFM incorrectly saved the variable at the end of the block, while still updating all its uses in that block to load the value from memory. We fixed this by inserting the store in the first legal point after the definition. In the second one, the definition is an invoke instruction and its use is a phi-node in a succeeding block.Since the invoke is a terminator, the first place we can insert a store is in the succeeding block, after the phi-nodes and the value's use. Similarly, a load instruction can only be placed in the block prior to the phi-node, before the terminator instruction and before the value's definition. Therefore, there is no valid placing for the store-load pair. This is easily resolved by just not inserting any instructions. The dominance property was never violated in the first place as the invoke is guaranteed to be defined as an incoming value for the phi-node.

While these bugs were seemingly minor, they caused undefined behavior in the affected basic blocks. Some of them were then mistakenly optimized away. HyFM attributed this to function merging, causing it to erroneously report higher code size reduction. After backporting the bug fixes to HyFM, we measured their effect on the set of benchmarks from Section IV. In total, 13 out of 40 benchmarks were visibly affected. The worst cases were for `510.parest_r` (20% lower), `447.dealII` (18% lower), and `444.namd` (8% lower). On average, this reduced the code size savings from 8.5% to 7.2%. The rest of this paper uses the bug free version of HyFM.

IV. Evaluation

A. Experimental Setup

Our experimental system is a dedicated server with a quad-core Intel Xeon CPU E5-2650 and 64GB of RAM. The OS is Ubuntu 18.04. The compiler for all experiments is LLVM v14. Timings were collected while the system was idle to reduce noise. To increase our confidence in our results, we repeat each experiment 10 times or for three hours, whichever is shorter. We report the average and 95% confidence interval.

We evaluate our novel technique, F3M, against the state-of-the-art function merging approach, HyFM [8], modified with the bug fixes of Section III-E. Since the main focus of this work is on reducing the cost of merging, both HyFM and F3M use fast pairwise alignment. We evaluate two variants of F3M. The static one uses the default $k = 200$, $r = 2$, $b = 100$, $t = 0.0$, while the adaptive one chooses these parameters according to the equations in Section III-D. Additionally, we examine the effect of various design parameters: fingerprint size, number of LSH rows, similarity threshold, and bucket size cap.

TABLE I
Benchmarks used in the evaluation

Application	NumFunctions	Notes
SPEC CPU2006	19 to 19359	19 benchmarks
SPEC CPU2017	20 to 45682	16 benchmarks
GNU gcc cc1plus	35756	Based on version 11.2.0
Google Chrome	1239140	Cloned from official repo on 2021-08-10
Libreoffice	286390	Version 7.2.0. Combined most of the ∼250 dynamic libraries implementing its functionality into a monolithic one
Linux kernel	45250	Sami Tolvanen's LTO branch v5.11
LLVM clang	104204	v14 RelMinSize target

Table I lists the workloads used in our evaluation. They include not only C/C++ benchmarks from SPEC CPU2006 and SPEC CPU2017 but also code from real large applications. For all of them, we compiled and linked all their source files to a monolithic LLVM bitcode file. We can then apply function merging and optimize the monolithic LLVM bitcode file in a LTO fashion. In cases where low level code and/or name conflicts made it impossible to link all bitcode files together, we linked as many as possible, leaving the remaining code to be compiled and linked separately.

For the experiments, we report the time needed for function merging alone as well as the time needed to compile the optimized bitcode down to the target object file, including all further optimizations, object code generation, and linking. We also report code size reduction for the generated object file. Benchmarks, when appearing on the x-axis, are ordered by number of functions. All baselines are built with LLVM v14 in full LTO mode without any function merging.

B. Similarity Metric and Code Size

Our approach aims to overcome the limitations of fingerprints based on opcode frequency that are used in the existing function merging techniques. The most fundamental limitation is the low correlation between the similarity score computed from the fingerprints versus the actual functions. In this section, we examine how well MinHash fairs and how this translates into more code size reduction.

Figure 10 shows the relationship between fingerprint similarity and the alignment quality for the same 800m function pairs from the Linux kernel as in Figure 4. We see that the two variables are much more closely correlated, 61.6% vs 20%, 3x higher. The relationship is not perfect though. There are 832 pairs with identical fingerprints and no alignment, as well as 142k pairs with no fingerprint overlap but perfect alignment. The former might cause us to expend effort on the wrong pairs but it is uncommon. The latter group could represent missed merging opportunities but in closer inspection they cause no problem. All of these functions contain only a single return of different data types. Since we encode the data type into our hashing scheme, our fingerprint similarity metric differentiates these return instructions, while the sequence alignment treats them as mergeable regardless of type. Given the overhead needed to handle the different types, merging these function pairs turns out to be unprofitable anyway.

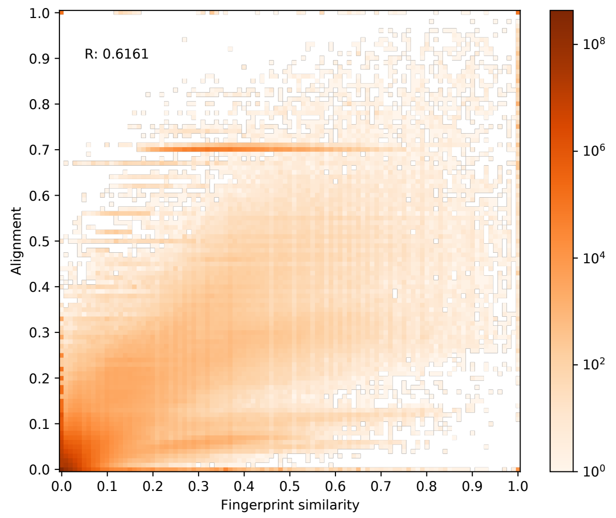


Fig. 10. Heatmap of MinHash similarity for all 800M possible candidate function pairs in Linux versus their alignment ratio. Quality of alignment and MinHash similarity show high correlation (R = 0.62).

For most pairs, however, our predicted similarities lead to better merging decisions. On average, this means 20% more successful merges and 19% fewer unsuccessful ones. For all benchmarks, except `429.mcf` which is very small, F3M pairs functions that are more likely to be profitably merged.

Figure 11 shows how this translates into code size reduction. Although F3M attempts to merge fewer pairs of functions by filtering out dissimilar fingerprint pairs, its results are typically better or similar to HyFM. The two povray-based benchmarks, two of the gcc-based benchmarks, and libquantum are the only ones with a statistically significant size increase for F3M relative to HyFM, typically 0.5% to 1.0%. All other benchmarks end up smaller by up to 3.8% (`638.imagick_s`), 0.4% on average. This represents a clear improvement on code size reduction from 7.2% to 7.6%.

F3M-adapt achieves similar results in terms of code size reduction. In most cases, the difference from F3M is not statistically significant. On average, the adaptive approach achieves almost the same code size reduction despite aggressively reducing the function merging overhead as we see below.

C. Function Merging Overhead

In this section, we demonstrate how F3M significantly reduces compilation time for large programs, from days to hours, while still producing smaller object files. Figure 12 shows the normalized compilation time across the whole benchmark suite. It includes the time for the back-end compilation pipeline, starting from the function merging pass on the fully linked bitcode and ending with the final object file. We measure the impact on the whole compilation pipeline because reducing code size has knock-on effects: reducing the number of functions tends to reduce the amount of work for subsequent compilation passes, but only as long as the merged functions are not too complex to process. So, we cannot fully understand the trade-off between merging overhead and size reduction without examining the compilation process as a whole.

It is immediately apparent how HyFM becomes impractical for large real-world programs. While it has a negligible compile-time overhead for most small programs, it takes over 46 hours to compile the Chrome web browser, 27.6x slower than the default LTO build without any function merging. However, compiling Chrome with F3M takes only 124 minutes, a speedup of 23x over HyFM, representing an overhead of 23% over the baseline. Our adaptive variant improves this even further. By reducing the number of bands to just 13 and raising the similarity threshold to 0.31, it makes compilation *as fast* as no function merging.

We get similar results for all other large programs. The cutoff point seems to be `620.omnetpp_s`. Larger programs ($> 9k$ functions) are always compiled faster with F3M than with HyFM, on average 26% faster with the static version and 30% faster with the adaptive version. For smaller programs, HyFM and F3M achieve similar compilation times, with F3M 1.8% faster and F3M-adapt 2.2% faster. The only significant outlier is `508.namd_r` for which F3M is 18-21% slower.

To help us better understand why, Figure 13 shows how the processing time of the function merging pass is distributed across its various stages. All bars are normalized to the total time of the HyFM pass for each benchmark. For smaller programs, the short compilation time is dominated by code generation. By using a MinHash fingerprint, we increase the preprocessing cost, as well as the cost of calculating fingerprint similarity during ranking. Using an LSH-based ranking mechanism instead of an exhaustive search reduces that cost but given the small number of functions, this has only a limited effect. In most cases, the small differences in function merging time are dwarfed by the longer time required by the rest of the compilation pipeline. The outlier from Figure 12, `508.namd_r`, shows a different behavior. Ranking and preprocessing times are similar for all approaches but using MinHash allows us to merge successfully 14% more functions, which translates into 14% more code size reduction and 42% higher code generation time.

For larger programs, Figure 13 shows that the function merging overhead is mostly ranking. By switching to an LSH-based search, we are able to eliminate increasingly more and more of the cost of function merging. At its most extreme, for Chrome, we reduce the function merging overhead by 98.9% making merging 94x faster. With the adaptive version, we further increase the speedup to a remarkable 597x.

A secondary effect on the compilation time comes from reducing the time spent generating code. This is a result of F3M filtering out merging attempts where two functions have almost no similarity. Unlike HyFM which will pair all functions with another function candidate, F3M pairing depends on the hashing scheme and their predicted similarity. The adaptive similarity threshold has a direct effect on that but, even without it, dissimilar pairs might be missed by the LSH-based search, reducing the code generation workload. We observe this on `458.sjeng` where code generation time is reduced by 25% for the static variant, with the code generation time spent

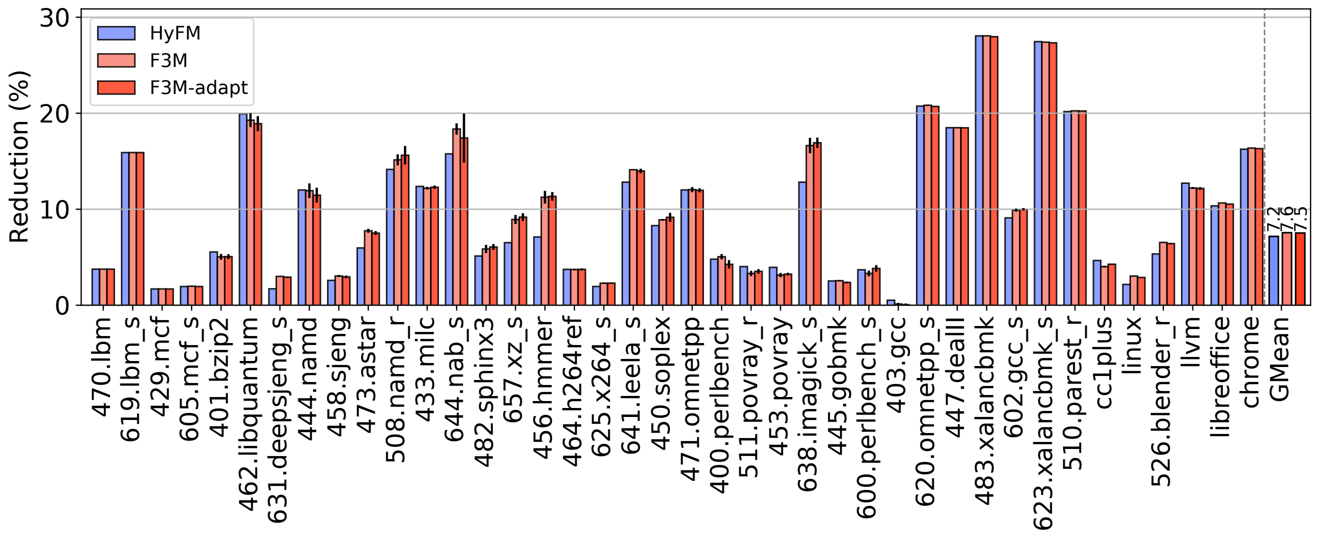


Fig. 11. Linked object file size reduction for HyFM and F3M. Benchmarks are sorted by number of functions. F3M achieves significant code size reduction even while attempting to merge fewer functions.

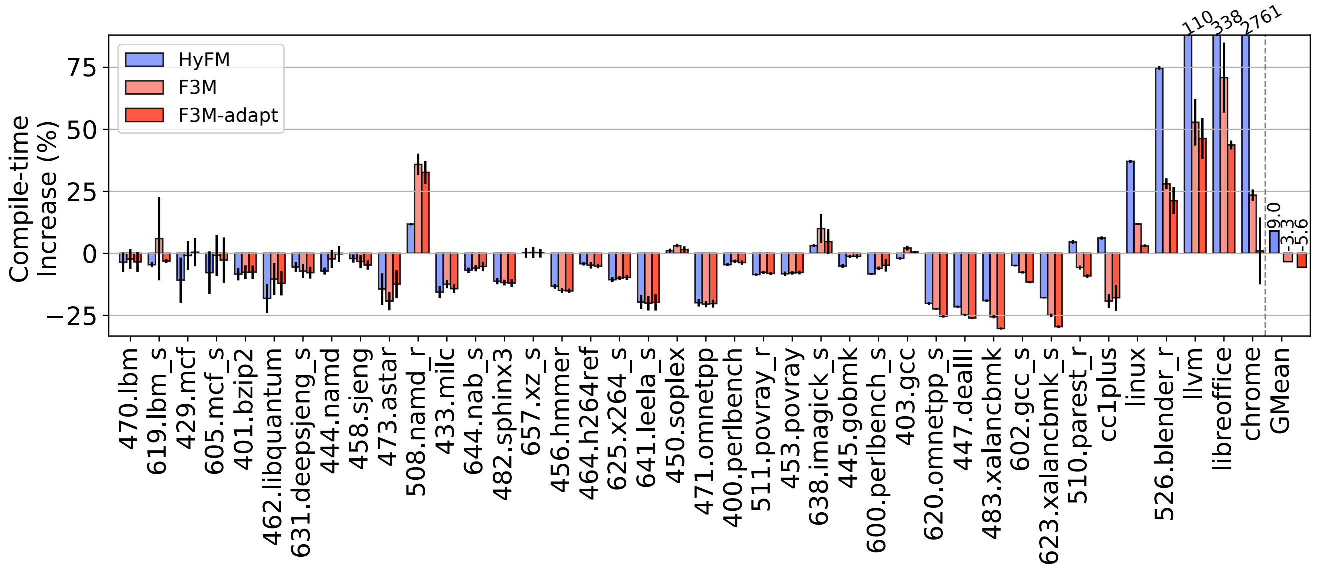


Fig. 12. Compile time overhead for the overall compilation process. F3M behaves similarly to HyFM for smaller programs. It occasionally increases compilation time due to the increased preprocessing costs and numbers of successful merge operations. For larger programs, compilation under F3M becomes much faster.

on unprofitable pairs going from 235ms to 168ms (while also increasing the time spent on profitable pairs from 5ms to 12ms). The adaptive approach brings that further down to 142ms.

D. LSH parameters

As discussed in Section III-D, similarity thresholds work by filtering out candidates with low similarity scores to avoid wasteful merging attempts. Figure 14 shows the average reduction of compilation time and the average increase of object file size for a range of thresholds, across all benchmarks except for the three largest ones. The percentages are relative to $t = 0.0$ for each benchmark. We also show the average of an oracle policy that tailors the threshold to each benchmark individually. If we want to minimize compilation time while limiting the average missed code size reduction below 0.1%, the ideal threshold is $t = 0.1$ which reduces compilation time by an average of about 1.5%. But if we choose the ideal threshold for each benchmark individually, the picture is different: ten benchmarks require $t = 0.4$, $t = 0.3$ for four, $t = 0.2$ for three, $t = 0.1$ for eight. For the remaining 12 any threshold higher or equal to 0.1 will have a negative impact on code size. An oracle policy would be able to aggressively reduce the compilation time for 15 benchmark, raising the reduction of compilation time to 2.3%. This indicates how an adaptive approach could significantly reduce compilation time with a negligible impact on code size reduction.

A similar trade-off affects the fingerprint size k, the number of rows r, and the number of bands b. Our default parameters of $k = 200$, $r = 2$, and $b = 100$ favor code size reduction. Figure 15 presents a detailed exploration of this trade-off space, demonstrating how varying fingerprint sizes and number of

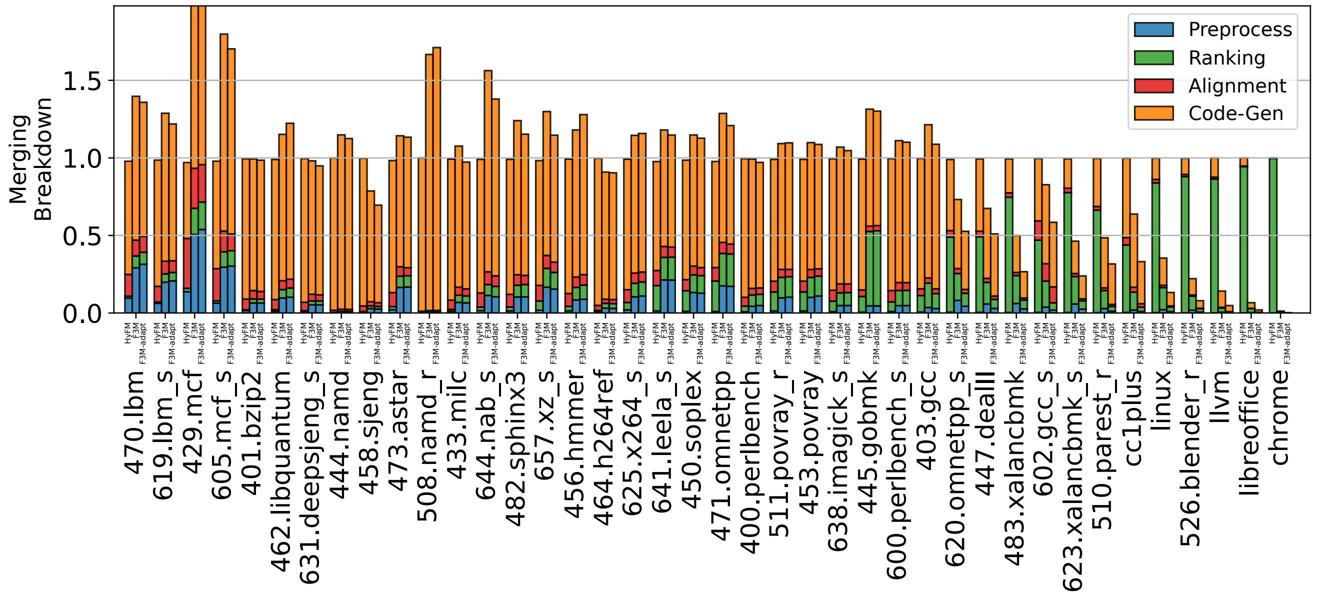


Fig. 13. Breakdown of the function merging cost for HyFM, F3M, and F3M-adapt.

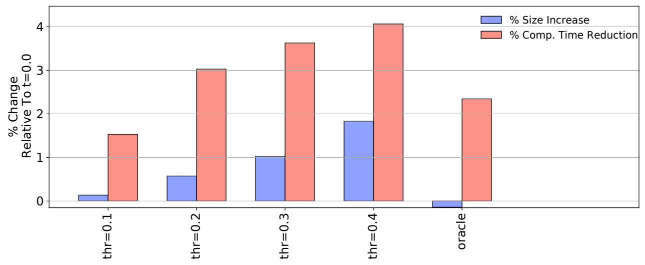
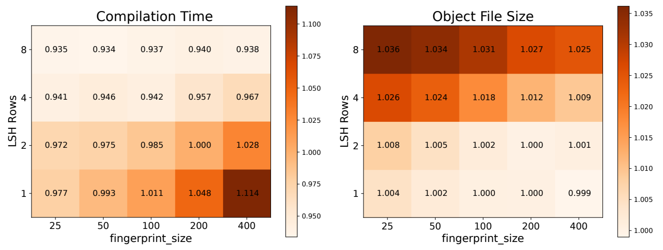


Fig. 14. Average compilation time and object file size vs similarity threshold over all benchmarks except llvm, libreoffice, and chrome. Values are normalized to the default configuration of $t = 0.0$. The oracle selects the threshold for each benchmark individually with the aim of keeping the loss of code size reduction below 0.1%



Compilation Time

LSH Rows	25	50	100	200	400
8	0.935	0.934	0.937	0.940	0.938
4	0.941	0.946	0.942	0.957	0.967
2	0.972	0.975	0.985	1.000	1.028
1	0.977	0.993	1.011	1.048	1.114

Object File Size

LSH Rows	25	50	100	200	400
8	1.036	1.034	1.031	1.027	1.025
4	1.026	1.024	1.018	1.012	1.009
2	1.008	1.005	1.002	1.000	1.001
1	1.004	1.002	1.000	1.000	0.999

Fig. 15. Average compilation time and object file size vs fingerprint size and number of LSH rows over all benchmarks except llvm, libreoffice, and chrome. Values are normalized to the default configuration of $k = 200$ and $r = 2$. Ranking quality is not affected heavily by fingerprint size. In contrast, increasing the number of rows above 2 has a strong negative effect.

LSH rows impact on both compilation time and code size. In general, as the number of rows increases, compilation time drops rapidly but at the cost of increasing code size. For $r = 8$ much of the code size reduction of F3M is lost. But decreasing the rows to 1 provides only limited size reduction. Decreasing the fingerprint size has a similar effect but more gradual. While both parameters can be used to control the trade-off between compilation overhead and size, the fingerprint offers more fine-grained control and reducing its size also reduces the memory requirements of F3M as a side-effect. This is why the adaptive policy always uses $r = 2$ and controls k and b.

Again, this trade-off is not the same for all programs. As the number of functions increases, the contribution of F3M ranking increases in a superlinear way, so it becomes increasingly beneficial to reduce the fingerprint size. Going from $k = 200$ to $k = 25$, reduces the average compilation time by 1.2% for small benchmarks (100-1k functions), by 4.3% for medium sized benchmarks (1k-10k functions), and by 6.2% for large benchmarks (10k-100k functions). The effect on object file size is similar for all of them, 0.2% to 2%. Similarly to the threshold discussion, this is why an adaptive policy that changes the LSH parameters based on the number of functions outperforms any static configuration.

E. Bucket Search Cap

As mentioned in Section III-C, some buckets can become over-populated and for that reason we cap the number of fingerprint comparisons per bucket to 100. This happens very infrequently. Even for a large program like Linux, with more than 45k functions, our results show that over-populated buckets ($population >= 128$) are extremely uncommon, less than 0.03% of the total and their contribution to the search process is relatively small. But because the cost of searching within each bucket is quadratic on the number of bucket entries, large buckets dominate the search process' runtime: approximately 75% of the fingerprint comparisons happen in that tiny group of overpopulated buckets.

Figure 16 shows the effect of applying a cap on the number of fingerprint comparisons per bucket when using F3M on Linux. It confirms that a cap of 100 has no statistically significant effect on the output of F3M while at the same time reducing the overall compilation time by 4%. It also shows that this is a conservative choice. Even a cap of two would have no

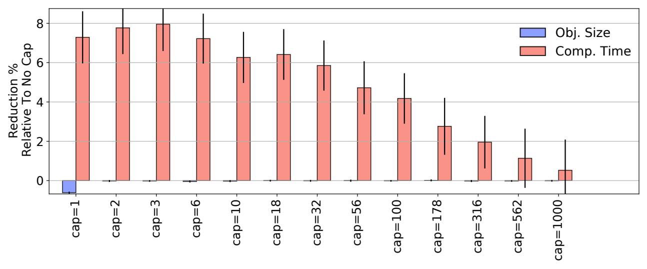


Fig. 16. Effect on compilation time and object file size of varying the bucket search cap on Linux. Capping the maximum number of fingerprint comparisons per bucket to 100 has no negative impact on the effectiveness of function merging. For Linux, anything above $cap = 1$ is acceptable.

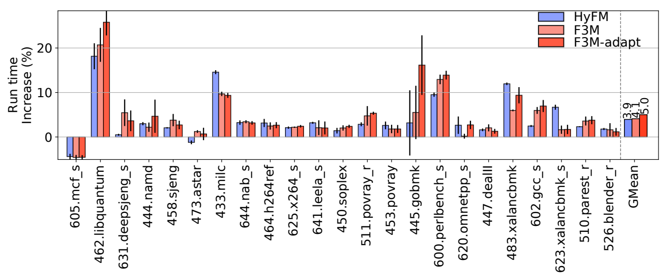


Fig. 17. Effect of function merging on the execution time of SPEC benchmarks. Benchmarks for which no technique is statistically likely to affect runtime relative to the baseline (Student t-test with $p > 5\%$) are excluded.

negative effect. This is connected to the distribution shown in Figure 9. Most matches (and definitely the ones that contribute the most) are highly similar. Because of that they are likely to map together into multiple buckets. Even if the cap filters out some of those shared buckets, they should still have enough overlapping buckets for the search process to match them. So, applying even an aggressively low cap should have little effect on function merging effectiveness.

F. Impact on Program Performance

As previous work has also shown [6], [7], function merging can have a negative impact on program performance. Merging two functions, especially two with significant differences, requires extra code and extra control flow. If this merged function is frequently used at runtime, the code overhead will manifest as performance overhead.

Figure 17 shows this effect on 23 of the SPEC benchmarks. For the remaining 12, no function merging technique affected performance. For this set of benchmarks, the average slowdown ranges from 3.9% to 5%. For most benchmarks, the slowdown is relatively small, below 5%, though, for five it can reach over 10%. The amount of slowdown is rather random. Benchmarks where F3M achieves more code size reduction than HyFM, show little impact on runtime performance. For 462.libquantum, where F3M causes the most significant slowdown relative to HyFM, F3M actually merges fewer functions. This is because whether the merged code causes slowdown depends on whether that code is executed. Neither function merging technique takes this into account. They may merge a function with a frequently used function, even if another similarly good and rarely used candidate exists. A more performance-aware implementation of function merging would use profiling information to influence candidate selection towards infrequently used functions. This would eliminate all or almost all performance overhead.

V. Related Work

Compilers have tackled code size by replacing a code segment with another smaller but semantically-equivalent code segment [13], [14], deleting unnecessary code [15], [16], combining redundant code within a function [17], [18]. Merging equivalent code for code size reduction has been explored in several different forms [5]–[8], [19], [20]. Linkers merge text-identical functions at the bit level [21], [22]. However, this optimization is platform-specific and need to be adapted for each object code format and hardware architecture.

Established compilers, such as GCC and LLVM, also provide a target-independent optimization for merging identical functions at the IR level [3], [4]. Merging only identical functions allows for an efficient exploration based on a hashing strategy, since identical functions have identical hashes.

More recently, we have seen a progress towards merging non-identical functions [5], [6]. Rocha et al. [6] proposed a technique capable of merging arbitrary pairs of functions in order to reduce code size. Their technique employs a sequence alignment algorithm to find equivalent code segments. Functions with enough similarity are merged in a way that their equivalent code is reused between them. Mismatching code segments of code are also added to the merged function but have their code guarded by a function identifier. They later propose a better code generator that enables the function merging operation to effectively support the SSA form [7]. More recently, they have proposed HyFM, where they avoid the quadratic aspect of the alignment operation in two folds: first, HyFM works on the basic block level, reducing the granularity of the inputs for the alignment algorithm in practice; second, it employs a simpler linear alignment strategy.

VI. Conclusion

In this paper, we have proposed a hash-based fingerprint to summarize functions. The hash-based representation allows us to compare the similarity score only among functions that share some of their hash values, immediately excluding most of the unprofitable candidates. This avoids the quadratic aspect of the search strategy employed by the state-of-the-art, resulting in significant speedups. For large real-world programs, such as Google Chrome, we reduce compilation time from almost two days to around two hours, a massive 23x speedup without sacrificing code size reduction. Our adaptive approach increases the speedup to 27x making compilation as fast as without function merging.

For future work, we envisage further improvements that can be achieved by integrating function merging to a summary-based link-time optimization framework, such as ThinLTO in LLVM. We also plan to analyze the interaction between function merging and other optimizations such as inlining, outlining, and code splitting.

References

[1] Google, "Create and set up your app - play console help," https://perma.cc/RC9M-APDD, 2021, accessed: 2021-08-10.

[2] Apple, "Maximum build file sizes - app store connect help," https://perma.cc/3T75-CFG5, 2021, accessed: 2021-08-10.

[3] "The LLVM Compiler Infrastructure. MergeFunctions pass, how it works," http://llvm.org/docs/MergeFunctions.html, 2020.

[4] M. Liška, "Optimizing large applications," *arXiv preprint arXiv:1403.6997*, 2014.

[5] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 85–94.

[6] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 149–163.

[7] ——, "Effective function merging in the ssa form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 854–868.

[8] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "Hyfm: Function merging for free," in *22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems: Co-located with PLDI 2021*, 2021.

[9] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, p. 117–122, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327494

[10] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253–262.

[12] L. C. Noll, "Fowler / noll / vo (fnv) hash," http://www.isthe.com/chongo/tech/comp/fnv/index.html, 2001, accessed: 2021-08-24.

[13] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 122–126.

[14] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982.

[15] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '99. New York, NY, USA: ACM, 1999, pp. 1–9.

[16] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 147–158.

[17] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 20–24.

[18] W. K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Static Analysis*, R. Cousot, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 401–417.

[19] G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes, "Code factoring in GCC," in *Proceedings of the 2004 GCC Developers' Summit*, 2004, pp. 79–84.

[20] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production iOS mobile applications," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. US: IEEE Press, 2021, pp. 1–12.

[21] S. Tallam, C. Coutant, I. L. Taylor, X. D. Li, and C. Demetriou, "Safe ICF: Pointer safe and unwinding aware identical code folding in gold," in *GCC Developers Summit*, 2010.

[22] D. Kwan, J. Yu, and B. Janakiraman, "Google's C/C++ toolchain for smart handheld devices," in *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*, April 2012, pp. 1–4.