CodeMason: Binary-Level Profile-Guided Optimization

David Williams-King
dwk@cs.columbia.edu
Columbia University
New York, NY

Junfeng Yang
junfeng@cs.columbia.edu
Columbia University
New York, NY

ABSTRACT

Optimizing a program for a specific machine or a specific workload is possible with today's compilers, but infrequently used, despite significant performance gains. We implement workload specialization, or Profile-Guided Optimization (PGO), at the binary level. Our system CodeMason runs on x86_64 Linux and is based on a binary rewriting platform called Egalito. CodeMason performs static binary rewriting to obtain program profiles, then adjusts function ordering, alignment, and other binary-level details to achieve faster performance (particularly on the given workload). We obtain 1.98% average performance speedup on SPEC CPU 2006, and 11.8% speedup in the best case. These substantial performance improvements suggest that binary-level PGO may be widely useful when compiler-based PGO is impossible because the source code is inaccessible.

ACM Reference Format:
David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3338502.3359763

1 INTRODUCTION

Optimizing compilers can produce output that is astoundingly faster than its unoptimized equivalent [16]. A compiler can generate even better code if it knows details about the target CPU architecture or target workload, but most distributors do not take advantage of this. The end-user's machine is typically unknown, so unless the user compiles the code themselves (as in Gentoo [2]), the distributor would have to build a large number of separate binaries. Workload-based optimization, otherwise known as Profile-Guided Optimization (PGO), can gain up to about 9% performance improvement across various GCC benchmarks [13], and was one of the original goals of the LLVM project [14]. However, very few developers use PGO, because it necessitates repeated, slow recompilation [6] and is difficult to deploy.

FEAST'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6834-6/19/11…$15.00
https://doi.org/10.1145/3338502.3359763

In this work, we consider the idea of performing per-workload or per-CPU optimizations at the binary level, instead of at the compiler level. The compiler's peephole optimizer has carefully estimated the cycle latency of each instruction to avoid pipeline stalls, and we do not want to undo its work. Hence, we do not perform any instruction substitution. Instead, we aim to make more effective use of a CPU's code caches. Leveraging powerful binary rewriting techniques, we can move code to new addresses to adjust program layout. The scope for improvement is significant: suppose we manage to fit all the hot (frequently used) code a program is executing into L1 code cache. Then code will never need to be fetched from other caches, nor will TLB lookups be spent on code addresses. Of course, the definition of "hot" code depends on the workload, and the L1 cache specifics depend on the user's CPU.

We present our system CodeMason, which adjusts function ordering and alignment and other binary-level details to obtain 1.98% average performance speedup on SPEC CPU (with a potential additive 1.7% average speedup with more aggressive optimizations), and 11.8% speedup in the best case. A user first applies binary-level instrumentation to a program and runs it to gather an execution profile. Then, we perform binary-level PGO to rearrange the binary and obtain one that runs faster on the given workload, and will likely beat the baseline on other workloads too. Hot functions get moved closer together and empty space in the code section can be elided to make the code more compact and cache-friendly. CodeMason is still under development but our substantial performance improvements suggest that binary-level PGO may be useful when compiler-based PGO is inapplicable due to lack of source access or deployment concerns.

2 BACKGROUND/RELATED WORK

Compiler Optimizations. Besides per-module optimizations, some compilers are capable of link-time or whole-program optimizations [8, 10]. Most such optimizations are independent of program input (though the compiler may well specialize a function for particular values). Profile-Guided Optimization is supported in major compilers [6, 13] as well. Our binary-level PGO is similar, but does not require cooperation from the developer to deploy.

Function Permutation. Related work performs function-level permutation, at the compiler- [12] or binary-level [21, 22]. The typical reason to perform function permutation is to increase entropy, for randomization-based defenses. One existing work called Stabilizer [7] measures the performance of

different permutations, but their goal is to statistically average out the differences rather than select the most optimal. Stabilizer is also a source-level technique, while we focus on the binary level.

Recompilation/Reassembly. Typical binary analysis frameworks [5, 9, 17, 18] focus on lifting binaries into an intermediate presentation, but do not enable code regeneration after analysis. Some systems provide code execution in a process virtual machine, such as DynamoRIO [4], Pin [15], PSI [23], and Multiverse [3], but this incurs substantial overhead. Multiverse, for example, incurs 60.42% average overhead (288% in worst case) on SPEC CPU. PSI [23] has the potential to be more efficient, as it uses static rewriting, but its shepherding branches still incur 53% overhead on a large set of programs.

In this work, we need to preserve the input assembly as much as possible, since we are making very small changes to the code and hoping to observe small performance improvements. Two promising frameworks are Uroboros [20] and Ramblr [19], which implement binary reassembly. Their aim is to fully solve the disassembly problem, and lift a binary into a `.s` assembly file which can be processed by a standard assembler. We could then modify the code or addresses as necessary. However, these systems are relatively fragile and rely on complex heuristics. We instead chose to use a binary rewriting framework Egalito [1], which calls itself a *binary recompiler* (a more fully-featured version of binary reassembly). For more details, see the next section.

Egalito Framework. In this work, we use the Egalito binary rewriting framework [1], which is currently under submission to an academic conference. Egalito's authors state that it will be released open source once its paper is accepted. The Egalito recompiler leverages metadata present in current stripped x86_64 and ARM64 binaries to generate a complete disassembly, and allows arbitrary modifications that may affect program layout without any constraints from the original binary. Essentially, it fully symbolizes and builds an internal representation of the program using a minimum of heuristics (some heuristics may still be needed to spot jump tables within data-flow analysis structures). The code may be modified, assigned to new addresses, and output into new ELF files. Code modification is even possible at runtime, by injecting some Egalito code and data structures into the program. In this work, we use only static binary rewriting. Importantly, Egalito has very little performance overhead when performing layout transformations (despite every function potentially now living at a new address), which allows us to measure the potentially small performance impacts that we are interested in.

3 DESIGN

CodeMason runs on x86_64 Linux. The Egalito [1] recompilation framework we use supports other platforms including ARM64, with RISC-V and x86_64 Windows support underway, and our techniques should be equally applicable to any platform including these.

In this work, we evaluate the effectiveness of three binary optimizations:

(1) Adjusting the address alignment of the beginning of functions, by inserting padding between functions.
(2) Choosing a different function ordering.
(3) Procedure Linkage Table (PLT) elimination ("collapse"), e.g. turning a dynamically-linked executable into a statically-linked one.

Alignment. On Linux, GCC aligns functions to a 16-byte boundary by default. As a design decision, this makes sense; the Intel Optimization Reference Manual [11] says "All branch targets should be 16-byte aligned." However, this can push functions far enough away that a set of hot functions no longer fit in the code cache. In some cases, it is better to sacrifice performance at misaligned branch targets in exchange for better caching behaviour. One extreme example described in Section 5 is a program that observed a 13.6% performance improvement by using 2-byte function alignment instead of 16. CodeMason can find the best alignment for a given system simply by benchmarking 16-, 8-, 4-, and 2-byte alignments.

Ordering. The biggest potential for performance improvement is to take two functions which call each other frequently and place them adjacent to one another, ideally in the same cache line. That way, even if the program's working set is too large to fit into the code cache, the two functions should be available together whenever one is in the cache. Furthermore, the CPU may automatically prefetch the correct next function when accessing the first one (as in the L2 streamer prefetcher, 3.7.3 of [11]). In this work, we implemented a binary-level equivalent of `gprof` which counts function calls. The user runs the program once (or more) under this profiling, and then we generate a new executable which organizes functions according to how many times they are called.

It is important not to mess too much with the original function ordering. In a compiler toolchain, code from a single source file ends up in one object file, and the linker simply concatenates text sections from each object file. In many codebases, related functions will be grouped together in source files. For example, all the code for a C++ class may be in one object file. If we use a random function ordering, performance is worse than baseline. Hence, we use a stable sort to keep functions in their original order if the profiling code does not wish to reorder them.

Collapsing PLTs. Procedure Linkage Tables—PLTs—are indirect jumps from one executable (or library) to another shared library. They involve one additional memory dereference, and an additional code basic block, compared with an in-image direct call. But the alternative to PLTs is to statically link a program, bringing all the code needed into the same executable, which prevents code-sharing between processes. Actually, multiple instances of the same process can share code, but e.g. two different programs using `glibc` will need two separate physical memory pages. It is quite conceivable that a user may be willing to pay this cost for a

performance improvement, so one of CodeMason's optimizations is to collapse PLTs into direct calls.

4 IMPLEMENTATION

The heavy lifting of binary rewriting is done by the Egalito framework [1]. We implemented or used the following tools.

Function Alignment. Egalito is a recompiler. It resolves all references in the input program, so that an arbitrary new layout can be used for the output. Hence, it was trivial for us to implement different function alignments (Egalito places `nop`s as needed to fill the empty gaps). There are no trampoline jumps in the output, nor any copy of the original code; it is as similar as possible to the output of a (compiler + linker) toolchain told from the beginning to use a specific function alignment.

Function Profiling. We implemented our own binary-level profiling, similar to `gprof`. Our instrumentation transforms every function to include a counter increment in the function prologue (the counter will only be incremented once per function call, even in the presence of loops). We allocate a new counter global variable for each function, and give them symbols so that gdb can examine function counts. We inject a small amount of code at program exit which appends the current array of counter variables to a "profile.data" file, analogous to `gprof`'s "gmon.out", and provide an additional tool (like the `gprof` binary) which dumps the accumulated counts in text form.

Function Permutation. By default, Egalito uses the same function order that was present in the original binary. When libraries are also being transformed, code from different ELFs is simply concatenated, much as the linker does for object files. We implemented support for selecting a random function permutation, and also for specifying a function permutation in an input file. Specifically, the function ordering file can give a rank to each name; functions with highest rank are given first (lowest) addresses. However, within the same rank, the original ordering prevails. In other words, we perform a stable sort based on function ranks.

In our experiments, we use two ways of calculating function ranks:

(1) Set the rank to the number of times N the function was invoked on a representative test run.
(2) Set the rank to $log_2(N)$, the log base 2 of the number of function invocations, unless $N < 3$ in which case set the rank to zero (treat small counts as the same rank).

For more information, see Section 5.

Collapsing PLTs. Egalito supports parsing an ELF file and all its shared library dependencies, then outputting a single merged ELF ("union ELF mode"). All calls through PLT entries are pre-resolved and turned into direct calls. (Some special cases are handled differently: IFUNC PLTs are statically resolved based on an executed instance of `glibc`, while .plt.got PLTs are still generated.) In this mode, all functions are placed in a single merged `.text` section, so code locality is slightly improved. The relative order of functions within each ELF is not modified, by default, though new addresses are assigned.

5 EVALUATION

Evaluation was performed on the following three machines:

- **M1**: Debian buster with an Intel Core i7-4770 (4 cores, 8 threads) and 32GB of RAM. Compiler: GCC 7.2.0.
- **M2**: Debian stable 9.6 with dual socket Intel Xeon X5550 (8 cores, 16 threads total) and 24GB of RAM. Compiler: GCC 6.3.0.
- **M3** (graphs not shown): Debian stable 10.0 with an AMD Ryzen 7 1800X (8 cores, 16 threads) and 32GB of RAM. Compiler: GCC 8.3.0.

The compiler used for evaluation was each machine's default. More details on each CPU can be found in Figure 1. Throughout our evaluation, the default system is M1 unless otherwise specified. Whenever we mention an average time, it is the geometric mean by default.

Motivating Example. When experimenting with Egalito on binaries installed on our Debian system, we transformed `/usr/bin/perl`. We used the following simple micro-benchmark:

```
perl -e 'for(1..10000000){$x+=$_}print $x'
```

Simply by transforming the binary with Egalito (default 16-byte function alignment), we saw a 2.22% speedup (over a multi-second execution). But we observed even faster performance with smaller alignment, the fastest being a reproducible *13.6% speedup* at 2-byte alignment. We believe that the smaller alignments coincidentally moved functions close enough together that the code used in this loop would all fit into the CPU's code cache.

5.1 SPEC CPU Performance Evaluation

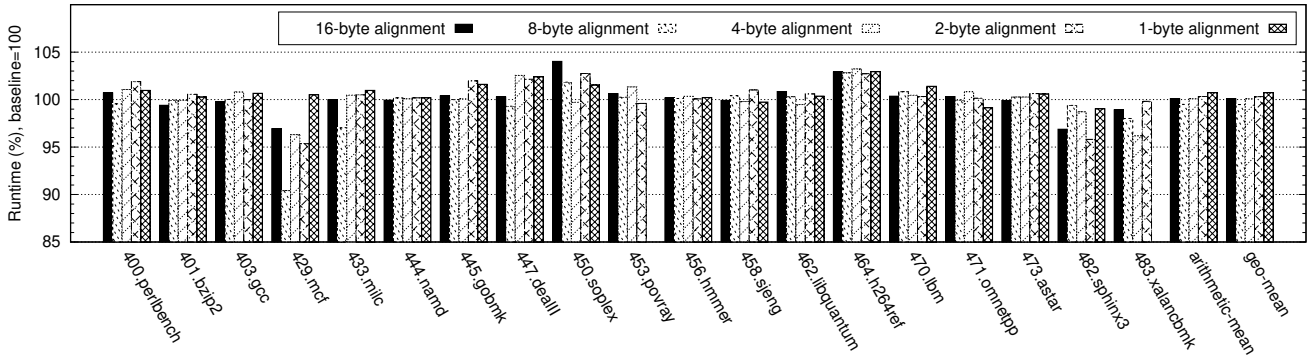
Here, we evaluate CodeMason's performance on SPEC CPU 2006. Unless otherwise noted, we transformed only the main executables and not shared libraries. Since Egalito does not yet support C++ exceptions, we modified SPEC CPU by replacing exceptions with conventional control flow in omnetpp (20-line change) and povray (15 lines). We also fixed a compile error in soplex (1 line) in recent GCC versions.

Function Alignment. The default function alignment used by our GCC is 16-byte alignment. We transformed all programs at the binary level, starting from 16-byte aligned code, and outputting 16-, 8-, 4-, 2-, and 1-byte aligned code. Results are shown in Figure 2a and Figure 2b.

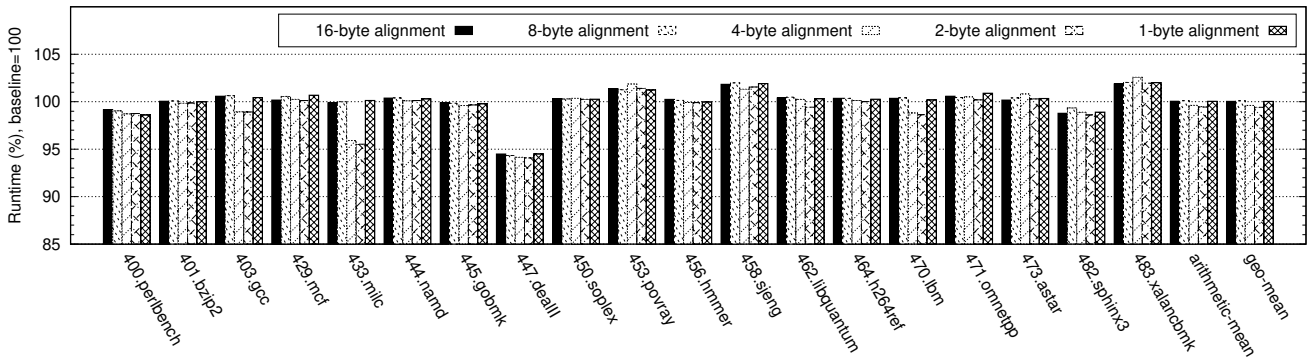
On M1 (Figure 2a), the winning parameter value is 8-byte alignment. Note that 1-byte alignment fails in two cases (povray and xalancbmk): `libstdc++` exception-handling code uses the least-significant bit of a function address to indicate that an exception has been thrown, and so odd addresses cause phantom exceptions in try-catch blocks. Meanwhile, on M2 (Figure 2b), the best alignment is 2-byte alignment. With a different (older) version of `libstdc++`, 1-byte alignment runs in all cases. We ran the same experiment on M3, but

ID	CPU	Year	Microarch	L1 code	L1 data	L2	L3
M1	Intel Core i7-4770	2013	Haswell (22nm)	4x32KB 8-way	4x32KB 8-way	4x256KB 8-way	1x8MB 16-way
M2	Intel Xeon X5550 (x2)	2009	Nehalem (45nm)	4x32KB 4-way*	4x32KB 8-way*	4x256KB 8-way*	1x8MB 16-way*
M3	AMD Ryzen 7 1800X	2017	Zen (14nm)	8x64KB 4-way	8x32KB 8-way	8x512KB 8-way	2x8MB 16-way

Figure 1: Details of the CPUs in each machine used for CodeMason testing and evaluation. *=per socket.



(a) SPEC CPU overhead for different function alignments, measured on machine M1.



(b) SPEC CPU overhead for different function alignments, measured on machine M2.

Figure 2: CodeMason SPEC CPU overhead with different function alignments (`nop` padding). For readability, these graphs show overhead relative to baseline=100 instead of relative to baseline=0.

results were noisy clearly due to thermal throttling, and hence are not shown; however, 2-byte alignment appeared to be best on that machine as well.

Finally, Figure 3 selects the best alignment for each SPEC case on M1, showing that we obtain 1.2% performance speedup simply by modifying function alignment, while preserving original function order. On M2, 2-byte alignment is a 0.59% speedup, and selecting the best alignment for each case is only 0.63% speedup. So on M2, a good strategy would be to always use 2-byte alignment. We believe this is because the instruction cache on M2 is only 4-way, whereas M1's 8-way instruction cache gives the processor more leeway to keep code around for longer, benefitting more from the degree of freedom given by adjusting function alignment.

Function Permutation. First, we show that the default function permutation present in SPEC executables is better than a random permutation. Figure 5 shows performance overhead with collapsed PLTs changing from 1.7% speedup to 0.53% slowdown (on M1). This makes sense because the

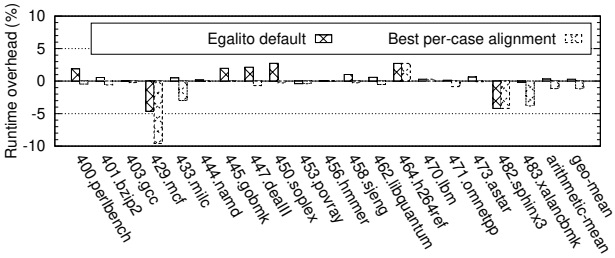


Figure 3: SPEC CPU overhead with the best function alignment from Figure 2a selected for each case.

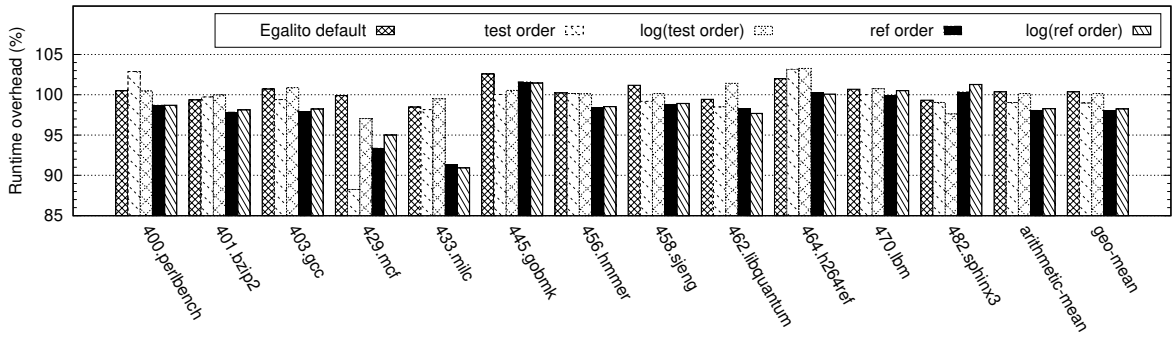


Figure 4: Performance obtained by ordering functions according to a captured execution profile (PGO). Run on M1 on SPEC CPU C programs only due to time constraints.

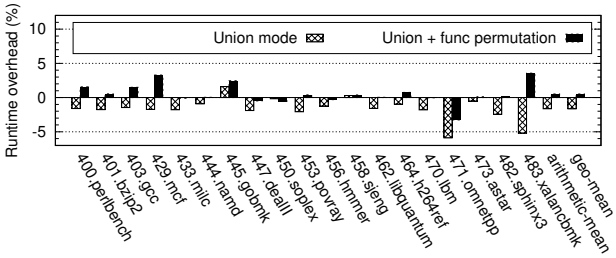


Figure 5: Choosing a random function permutation incurs overhead (2-byte alignment, collapsed PLTs).

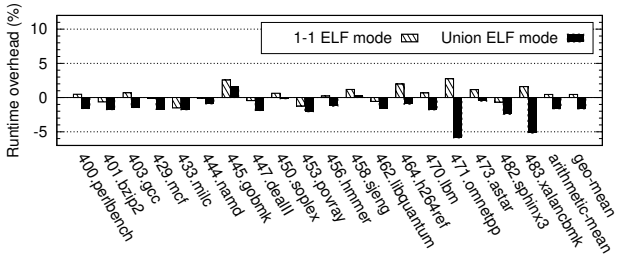


Figure 6: Performance of typical rewriting vs collapsing PLTs; the latter is 1.7% absolute speedup.

standard permutation will place functions from the same object file contiguously, which in most codebases will keep related code together.

Next, we performed several profile-guided optimization experiments based on hot functions. These experiments were only performed on SPEC C programs, due to lack of time. Some C++ programs (especially xalancbmk, with its many tiny virtual functions) will likely benefit even more from function permutation. The experiments, shown in Figure 4, were:

(1) Instrument SPEC with our profiling code, count function calls when running SPEC's "test" input size. Order functions according to number of calls. Then run and benchmark the "ref" input.
(2) Count calls on "test"; order according to $floor(log_2(N))$ of the number of calls N. If $N < 3$, place into bucket zero. Then benchmark on "ref".
(3) Count calls on "ref"; order functions according to exact number of calls. Then run the "ref" input.
(4) Count calls on "ref"; order functions according to the log of the number of calls. Then run the "ref" input.

(The overhead of our function-call instrumentation was only 0.75% on "ref".) The final speedup in the first case is 1.03%. This shows that profiling on one input (the "test" inputs are really very small) yields a useful speed improvement for another input. The second case shows 0.13% slowdown, so either the buckets were too coarse-grained or the "test" input size did not generate enough calls. When training on "ref" in the third case, we observe 1.98% speedup; using buckets slows performance slightly, to 1.76% speedup. Hence, we observe 1-2% speedup overall, with increased performance when the exact input is known in advance. Precise function ordering works better than bucketing by the log of the counts.

Note that the last two cases, trained on "ref" counts, were repeated three times each for reproducibility. Variance is fairly low, averaging 1.93%, 2.39%, 1.98% in the former, and 2.42%, 1.76%, 1.55% in the latter (all numbers are speedups). The graph shows the results from the mid-performing run.

Collapsing PLTs. Using 2-byte function alignment, we measured on M1 the performance of collapsing all PLT entries. Results are in Figure 6. We went from a 0.46% slowdown to 1.7% speedup by collapsing PLTs. Clearly there is a significant performance improvement when collapsing PLTs.

As to the memory overhead: the total mapped memory of union ELFs is between 79%-95% (average 88.4%) that of the original SPEC programs, when measured at the program entry point. The total size of the code sections is 540KB-2.56MB larger (1.7MB average) in the outputs due to the addition of library code, though the total file-backed resident memory use is 152KB-1.25MB less (598KB average), measured near program exit. Thus, union ELFs need less total memory due to the omission of PLTs, but for SPEC-sized executables use up to 2.5MB more per process due to static linking of library code.

Summary. We showed that 8-byte function alignment is best on M1, while 2-byte alignment is best on M2 and M3 (which use different processor architectures). We clustered functions close together with 2-byte alignment in our hot function experiments, ordering code according to program profiles. We measured significant performance improvement from collapsing PLTs. We expect the performance improvement from function alignment and collapsing PLTs to be additive, but we did not actually have time to test the combination. Hence, Figure 4 ("ref" order) shows the best performance we obtained on SPEC CPU.

6 DISCUSSION AND FUTURE WORK

The traditional trade off between dynamically-linked and statically-linked executables is that the former enables code sharing, while the latter avoids the expense of indirect calls. We showed that statically-linked code can perform significantly better (1.7% on SPEC CPU), if the user is willing to duplicate shared code used by different programs.[1] However, this speedup likely comes from a very small set of PLT calls. We could instrument and count PLT invocations, and only "inline" the functions which are used frequently. For example, maybe a program needs only `memcpy` to be statically linked in order to get significant performance improvements, which will not substantially increase the memory footprint of the program. We see this as a promising direction for future work, the creation of partially-statically-linked executables with increased performance and minimal cost.

Egalito supports function-level debloating. It can statically analyze a program to determine functions that can never be used, and exclude them from the output. We believe that this may combine well with the techniques in our work: if certain functions can be removed, then our function ordering is more likely to be able to place invoked functions within the same cache lines. Furthermore, the call-graphs computed by debloating can be used to inform the order in which we place functions. Instead of merely placing hot functions together, we could first check if one function calls the other.

Finally, although we performed static binary rewriting in this work, the Egalito framework is actually designed for dynamic recompilation. Leveraging this capability, we may be able to rearrange a program's code dynamically as it enters different phases of its execution. The main challenge will be to make our monitoring code as lightweight as possible; our function-counting mechanism is a good fit, but we may need other metrics. We believe the goal of a binary that self-optimizes based on its own runtime and input is a realistic possibility, and we are excited to explore this in future work.

7 CONCLUSION

We presented our system CodeMason, based on the Egalito binary rewriting framework, for binary-level profile-guided optimization (PGO). We discussed three binary optimizations: function alignment, function reordering, and collapsing PLTs (converting dynamically-linked to statically-linked code). Though this is a small set of operations, like compiler-level PGO we sometimes achieve very significant performance improvements—up to 11.8%. In future, we hope to develop a partial PLT collapse mechanism to trade off between memory overhead and runtime performance.

[1] Multiple copies of the same statically-linked program can still use code page sharing, especially for position-independent executables; the duplication only occurs when a diverse set of programs is used.

ACKNOWLEDGMENTS

Thanks to Graham Patterson for assisting with development that led to this project, and the reviewers for their feedback and experiment suggestions. This work is supported in part by ONR grants N00014-17-1-2788 and N00014-16-1-2263, and NSF grant CNS-1564055.

REFERENCES

[1] 2019. Egalito: Layout-Agnostic Binary Recompilation. (under submission).
[2] 2019. Gentoo Linux. https://www.gentoo.org/.
[3] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, Ahmad M Mustafa, Gbadebo Ayoade, Khaled Al-Naami, Latifur Khan, Kevin W Hamlen, Bhavani M Thuraisingham, Frederico Araujo, et al. [n. d.]. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Network and Distributed Systems Security Symposium (NDSS)*, Vol. 12. Springer, 40–47.
[4] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *CGO*.
[5] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.
[6] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 42–52.
[7] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proc. of ACM SIGARCH*. 219–228.
[8] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. 2004. Link-time optimization of ARM binaries. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 211–220.
[9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev. Ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 131–141.
[10] Honza Hubička. 2014. Linktime optimization in GCC, part 1 - brief history. http://hubicka.blogspot.com/2014/04/linktime-optimization-in-gcc-1-brief.html.
[11] Intel. 2011. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual.
[12] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *22nd Annual Computer Security Applications Conference (ACSAC)*.
[13] Michael Larabel. 2018. A Fresh Look At The PGO Performance With GCC 8. https://www.phoronix.com/scan.php?page=article&item=gcc-82-pgo&num=1.
[14] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.
[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.

[16] Phoronix. 2012. The Performance Between GCC Optimization Levels. https://www.phoronix.com/scan.php?page=article&item=gcc_47_optimizations&num=1.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.

[18] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[19] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. (2017).

[20] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 236–247.

[21] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of ACM CCS*. 157–168.

[22] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*. 367–382.

[23] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. *ACM SIGPLAN Notices* 49, 7 (2014), 129–140.