Predicting Function Merging Profitability with Deep Learning

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

Pavlos Petoumenos
University of Manchester, UK
pavlos.petoumenos@manchester.ac.uk

Zheng Wang
University of Leeds, UK
z.wang5@leeds.ac.uk

Murray Cole
University of Edinburgh, UK
mic@inf.ed.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Abstract

CCS Concepts: • **Software and its engineering** → **Compilers**.

Keywords: Code Size Reduction, Function Merging, LTO.

ACM Reference Format:
Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Predicting Function Merging Profitability with Deep Learning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3385412.3386030

1 Introduction

In recent years, the market for mobile and embedded systems has been rapidly growing. These systems must often run on inexpensive and resource-constrained devices, with limited memory, storage, CPU caches. Applications for these systems are designed and built with different goals compared to traditional computing systems since their binaries must fit in a budget memory size. Hence, compilation techniques must be primarily focused on optimising for binary size.

One important optimisation capable of reducing code size is function merging. In its simplest form, function merging reduces replicated code by combining multiple identical functions into a single one [2, 10]. More advanced approaches can identify similar, but not necessarily identical, functions and replace them with a single function that combines the functionality of the original functions while eliminating redundant code. At a high level, the way this works is that code specific to only one input function is added to the merged function but made conditional to a function identifier, while code found in both input functions is added only once and executed regardless of the function identifier.

A recent work has generalized function merging to work on arbitrary pair of functions. The state-of-the-art [14, 15] technique works in three major phases: First, it represents functions as nothing more than linear sequences of instructions. Then it applies a sequence alignment algorithm, developed for bioinformatics, to discover the optimal way to create pairs of mergeable instructions from the two input sequences. Finally, it performs code generation, producing the merged function where aligned pairs of matching instructions are merged to a single instruction, while non-matching instructions are simply copied into the merged function.

The state-of-the-art optimization also includes a search strategy based on ranking the function candidates with higher similarity. However, because this strategy is unable to decide which one of those pairs are actually worth merging, the optimizer uses the compiler's built-in cost model in its profitability analysis. This analysis is responsible to decide which merged functions should be kept, replacing the original functions. The profitability is estimated based on a target-specific cost model that assigns weights to instructions.

In this paper, we expose innaccuracies in this profitability analysis, showing there are still opportunities for further improvement. We propose a new approach for the profitability analysis based on partial recompilation. With this technique, the estimated reduction in code size more closely resembles the actual reduction observed in the final object file.

Moreover, we show that most merged functions are actually unprofitable. Even if we consider only the top-ranked candidate functions, about 82% of the candidate

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386030

functions are still unprofitably merged. Since the function merging operation is computationally expensive, we should avoid wasting time with unprofitable merged functions, freeing the compiler to focus more its efforts optimizing code that more likely to be profitable. In order to address this issue, we also describe our heuristic model based on deep-learning that predicts whether or not a pair of functions can be profitably merged, avoiding wasteful merge operations.

2 Background

3 Motivation

In this section, we discuss two weaknesses in the state-of-the-art function merging optimization [15]. First, we show that there is a significant opportunity in reduction that can be gained by having a better profitability analysis. Second, we show that most function merging attempts are trown away as they cause code bloat, according to the profitability analysis.

3.1 Inaccuracies in the Profitability Analysis

Although the proposed technique is able to merge any two functions, it is not always profitable to do so. A pair of functions can be profitably merged when replacing them by the merged function results in an overall smaller code. As it is only profitable to merge functions that are sufficiently similar, for most pairs of functions, merging them increases code size. Since the profitability analysis is critical for the optimisation strategy, we must be able to effectively decide which pair of functions can be profitably merged.

In order to estimate the code-size benefit, we first estimate the size of all three functions, i.e., the two input functions and the merged one. The size of each function is estimated by summing up the estimated binary size of all instruction in the function, in its IR form. The binary size of each IR instruction is estimated by querying the compiler's built-in target-specific cost model. These cost models provide target-dependent cost estimations approximating the code-size cost of an IR instruction when lowered to machine instructions.

As pointed out by many prior work [14**?** , 15], even though cost models offer a good trade-off between compilation time and accuracy, they are expected to contain inaccuracies. Because we are trying to estimate the binary size of the final object code, inaccuracies arise from the fact that we are operating on the IR level and one IR instruction does not necessarily translate to one machine instruction. Because we are operating on the IR level, We cannot know exactly how each IR instruction will be lowered without actually running the compiler's backend. Moreover, several number of optimisations and code transformations will still run prior to machine code generation.

Figure 1 presents the code size reduction that can be achieved with an oracle. This oracle measures code size by compiling the whole program down to its final object file, which provides perfect information for the cost model. It shows the potential for improvement there exists from having a better profitability analysis, almost doubling the reduction. By compiling the whole program down to its binary form, we are able to precisely capture the impact on other optimizations and machine code generation, as well as the overheads that result from replacing the callsites to the merged function or keeping a *thunk*.

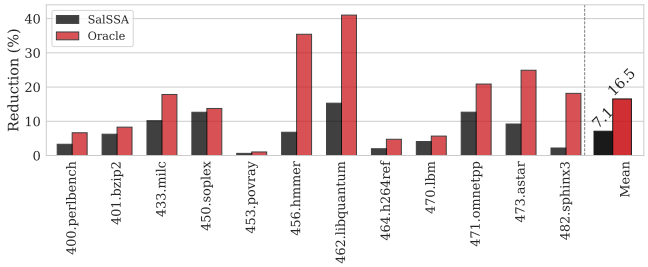


Figure 1. .

However, the cost of compiling the whole program for every merging attempt is prohibitive. The re-compilation overhead can be severely aggravated for larger programs with multiple functions, where not only each compilation takes longer but the whole program is also re-compiled many times.

3.2 Wasteful Merge Operations

The fingerprint-based ranking strategy helps the function merging optimization to pair functions that are more similar. However, the current strategy is unable to decide which one of those pairs are actually worth merging. Figure 2 shows that about 82% of the top ranked candidate functions are actually unprofitably merged. As a result, a considerable amount of compilation time is wasted producing merged functions that will be thrown away, keeping the original pair of functions.

Since most of the merged functions are thrown away for being unprofitable, it is expected that most of the compilation time is also spent producing those merged functions. This impact is also aggravated when several of the unprofitable merged functions are much larger than the profitable ones. Therefore, it is of utmost importance that we avoid merging unprofitable functions. If we could eliminate all the time wasted on unprofitable merge operations, we would free compilation time for more useful computation.

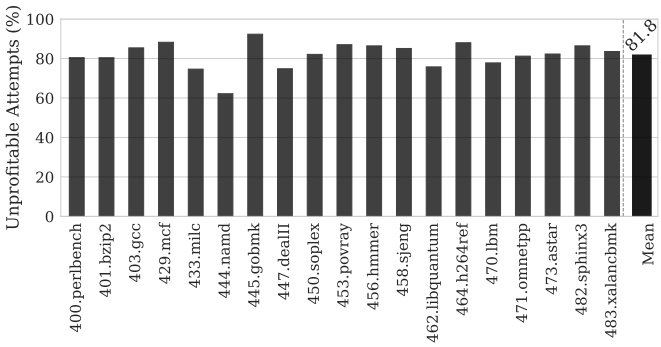


Figure 2. An average of about 82% of merging attempts are unprofitable.

3.3 Summary

In this paper, our goal is to develop a solution capable of identifying whether or not a given pair of functions can be profitably merged, allowing us to use a more expensive profitability analysis based on partial re-compilation. If we could predict which pairs of function are more likely to cause code bloat, we could avoid wasting time merging them in the first place and having to estimate their binary sizes. Bailing out early frees time to be spent on more profitable merge operations.

4 Our Novel Profitability Analysis

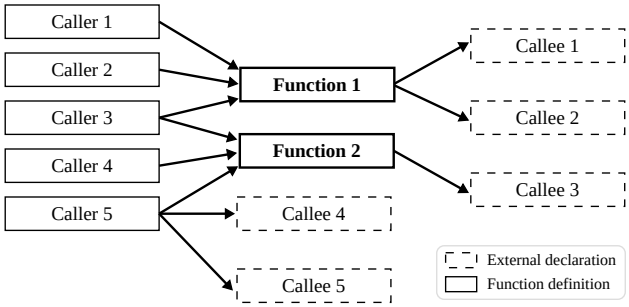
4.1 Realistic Code-Size Cost

In Section **??**, we describe the limitations of code size estimation using existing compiler's cost models. In our motivation, we use an oracle cost model, with perfect information, that can be obtained by recompiling the whole program in order to decide which version produces a smaller binary. However, this solution is infeasible due to compilation time overheads. Large programs with thousands of functions can take days to optimise due to the excessive number of long recompilations, sometimes taking up to a couple days.

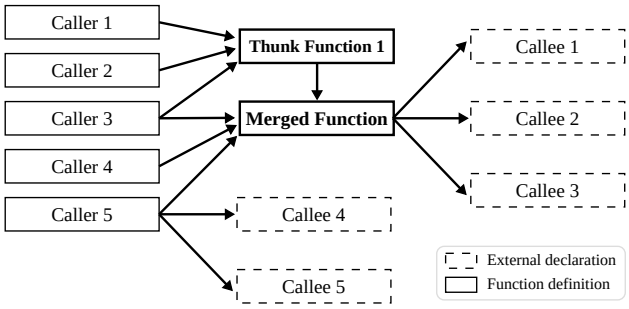
In this section, we propose a novel approach based on partial recompilation. This approach is capable of significantly reducing the compilation time required by the oracle cost model, while still providing equivalent benefits.

Our goal is to extract to a separate module only the code that can be potentially affected by merging a given pair of functions. Beyond the difference in size of the merged function itself, code size can also be affected by the need for a thunk that calls the merged function while preserving the original interface. Call-sites updated to call the merged function require extra arguments which might affect code-size directly or indirectly. Finally, inter-procedural analyses and optimisations can have their decision making altered by the merged function.

In order to capture all the possible ways that function merging can influence code size, we extract to a different module the functions being merged as well as their user functions, such as callers or functions that take their address. We also need to extract all global definitions and variables referenced by any of these functions, which include the declaration or signature of functions called by any of them. Figure 3a illustrates an example of code extracted by our partial recompilation approach.



(a) Extracted code without function merging.



(b) Extracted code with function merging.

Figure 3. Example of the code extracted to compare the binary size before and after function merging, in order to decide whether or not it is profitable to merge a given pair of functions.

After compiling and measuring the size of the extracted code, we then apply the changes caused by merging the given pair of functions, as illustrated in Figure 3b. These changes represent the real impact function merging would cause in the real program. Once the function merging has been applied, the code is recompiled. If function merging produces a smaller binary, then these changes are also applied to the real program.

4.2 Learning a Profitability Model

Figure 4 provides an overview of the prediction mechanism. Our mechanism follows a similar approach to previous deep-learning techniques for tuning compilers [6, 11].

The same linearised functions used for the merge operations, as described in Chapter **??**, are used as input to the prediction model. First, we use a language model

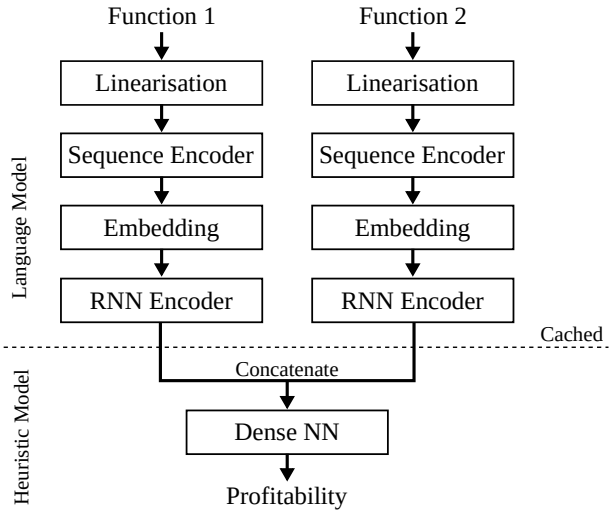


Figure 4. The proposed deep-learning model architecture that predicts which pairs of functions will be profitably merged. Code properties are extracted from each function into *context vectors* by the language model. These context vectors are cached to be later fed to the heuristic model to produce the final profitability prediction.

based on recurrent neural networks to encode the input functions into context vectors of fixed size. These vector encodings can be computed only once per function an cached. Finally, the context vectors of two input functions are concatenated and fed to a feed-forward neural network that classifies whether or not those functions are profitably merged.

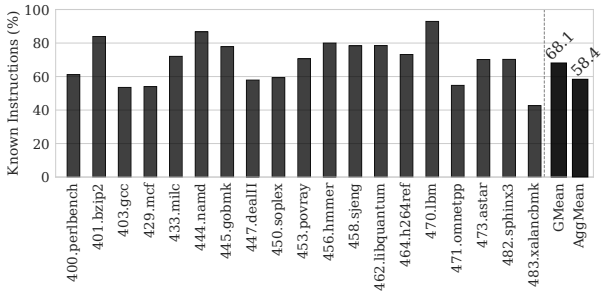


Figure 5. Percentage of instructions in the pre-trained *inst2vec* language model.

4.2.1 *inst2vec* Language Model.

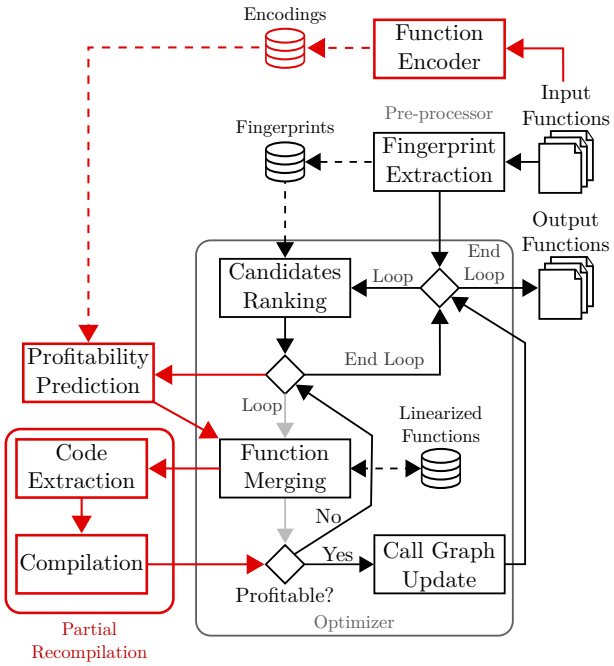


Figure 6. Overview of our extended exploration framework..

4.3 Search Strategy

5 Evaluation

6 Related Work

6.1 Function Merging

Link-time code optimizers like [1, 8, 16] merge text-identical functions at the bit level. However, such solutions are target-specific and need to be adapted for each object code format and hardware architecture.

GCC and LLVM [2, 10] also provide an optimization for merging identical functions at the IR level and hence is agnostic to the target hardware. Unfortunately, they can only merge fully identical functions with at most type mismatches that can be losslessly cast to the same format. The work presented by von Koch et al. [7] advanced this simple merging strategy by exploiting the CFG isomorphism of two functions. However, it requires two mergeable functions to have identical CFGs and function types, where the two functions can only differ between correspoding instructions, specifically, in their opcodes or the number and types of the input operands. The state-of-the-art technique, FMSA [14], lifts most of the restrictions imposed by prior techniques [2, 7, 10]. Although achieving impressive results, it does not directly handle *phi-nodes* which are fundamental to the SSA form. In order to simplify their code generator, Rocha et al. [14] replaced all phi-nodes with memory operations by first applying register demotion. This tends

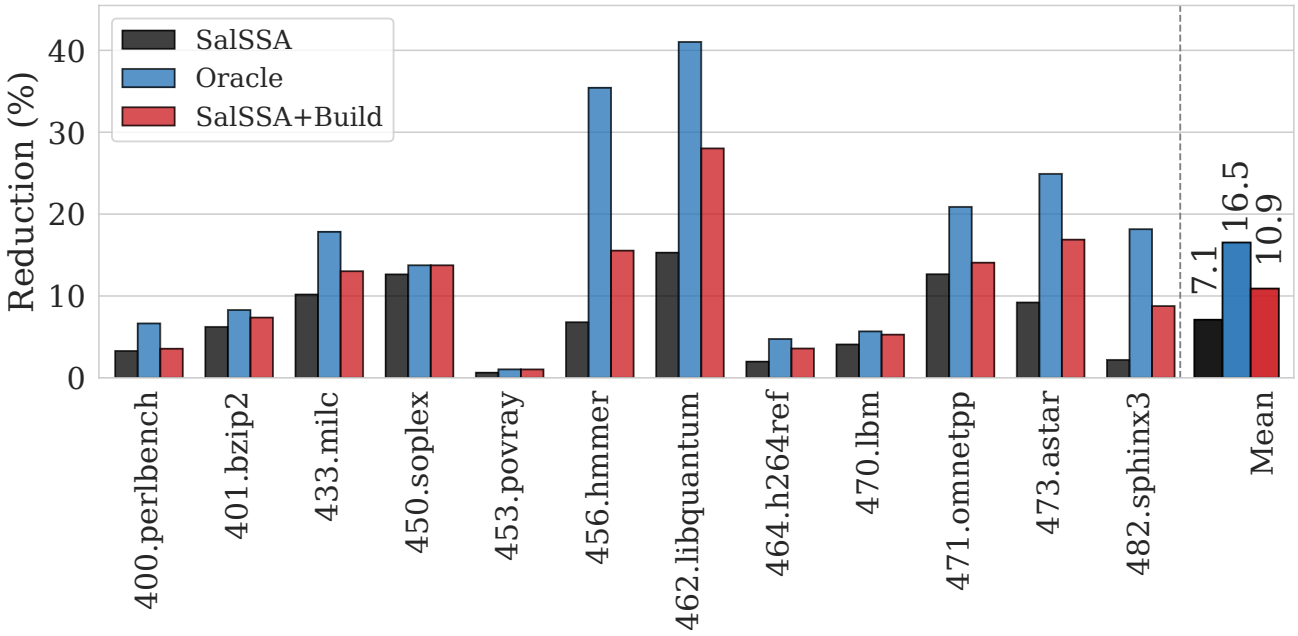


Figure 7. .

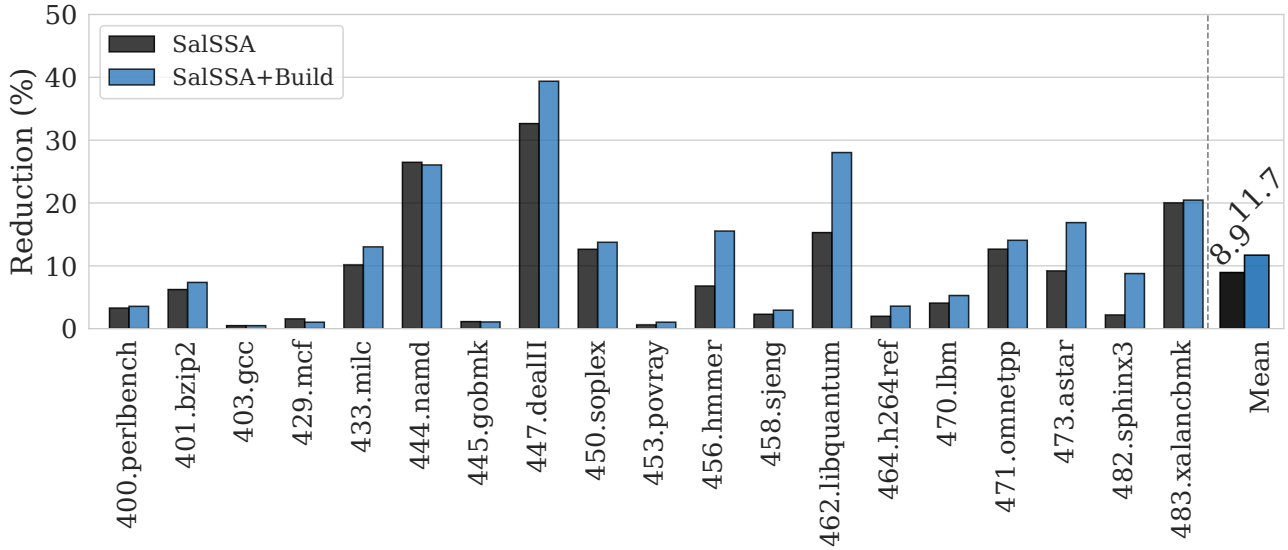


Figure 8. .

to almost double the function size, increasing compilation time and hindering function merging, as it expects that applying register promotion to the merged function will reverse the negative effect of the earlier register demotion. This is often not possible because function merging can add complexity to the memory operations, resulting in unnecessarily larger merged functions.

In order to avoid register demotion, Rocha et al. [15] describes a novel approach, called SalSSA, for merging functions which is capable of effectively handling the SSA form. Their approach achieves this with a new code generator that, instead of translating the aligned sequences directly into a merged function, generates code from the input control-flow graphs, using the alignment only to specify pairs of matching labels and instructions. SalSSA then produces code top-down, starting with the control flow graph of the merged function, then populating with instructions, arguments and labels, and finally with phi-nodes which maintain the correct flow of data.

6.2 Tuning Compilers with Deep Learning

There is been many work using machine learning as a heuristic for tuning runtime systems [3, 4, 12, 13, 17] and compilers [5, 6, 9, 11, 18].

Cavazos and O'Boyle [5] propose the use of genetic algorithm to tune the heuristics of function inlining. They use genetic algorithm to optimise the values for different features that control the inlining heuristic. These features are chosen by the compiler writer and they define the maximum size allowed by the inlining transformation for the callee and the caller functions, the maximum size for callee functions that are hot or that should always be inlined, etc. The optimised features are then used to define the rules of the inlining heuristic, describing which call sites should be allowed for inlining. The fitness function of the genetic algorithm involves the actual runtime of the compiled program, rendering the feature optimisation process very costly. However, their approach is able to achieve significant speedups over the baseline.

The quality of these features is critical to the improvements resulting from machine learning solutions. Leather et al. [9] propose the use of genetic programming in order to also automate the selection of these features. The feature space is described by a grammar and is then searched with genetic programming and predictive modelling, to avoid recompilation of the program for each step in searching the optimization space. The genetic programming technique is used to generate features that are fed to a decision tree. This machine learning solution form the decision-making heuristics for the loop-unrolling optimisation. They show that the automated selection of features outperform hand-coded features, for the same machine learning procedure based on decision trees.

Cummins et al. [6] propose DeepTune, which uses deep neural networks to learn optimization heuristics directly on raw code, unifying the search for features and decision-making heuristics into a single learning model. Since the program, in its textual form, can be seen as a sequence of tokens of variable length, using a recurrent neural networks becomes a natural choice. DeepTune has an LSTM-based language model that processes raw code, producing a fixed-size encoding which is then fed to a heuristic model based on a feed-forward neural network.

Mendis et al. [11] propose Ithemal, a tool which uses deep neural networks to predict the throughput of a set of machine instructions. Ithemal can be used as a cost model for compiler optimisations and code generation, aiding the decision of whether a transformation would result in faster code. Similar to DeepTune, Ithemal also processes raw machine instructions using an LSTM-based language model. However, Ithemal has an architecture with two LSTM stages. The first LSTM processes the tokens that compose one instruction. The second LSTM processes the encoded instructions that are produced by the first LSTM. The output of the second LSTM is aggregated into the final throughput prediction.

7 Conclusion

Acknowledgment

This work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/P003915/1 (SUMMER) and EP/M01567X/1 (SANDeRs). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] 2020. Microsoft Visual Studio. Identical COMDAT Folding. https://msdn.microsoft.com/en-us/library/bxwfs976.aspx.

[2] 2020. The LLVM Compiler Infrastructure. MergeFunctions pass, how it works. http://llvm.org/docs/MergeFunctions.html.

[3] Eva Andreasson, Frank Hoffmann, and Olof Lindholm. 2002. To Collect or Not to Collect? Machine Learning for Memory Management. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*. USENIX Association, USA, 27–39.

[4] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. Méhaut. 2011. A machine learning-based approach for thread mapping on transactional memory applications. In *2011 18th International Conference on High Performance Computing*. 1–10.

[5] J. Cavazos and M. F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 14–14.

[6] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232.

[7] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting Function Similarity for Code Size Reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*. ACM, New York, NY, USA, 85–94.

[8] Doug Kwan, Jing Yu, and B. Janakiraman. 2012. Google's C/C++ toolchain for smart handheld devices. In *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*. 1–4.

[9] H. Leather, E. Bonilla, and M. O'Boyle. 2009. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *2009 International Symposium on Code Generation and Optimization*. 81–91.

[10] Martin Liška. 2014. Optimizing large applications. *arXiv preprint arXiv:1403.6997* (2014).

[11] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning*

Research), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 4505–4515.

[12] Alyson Pereira, Rodrigo C. O. Rocha, Luiz Ramos, Marcio Castro, and Luis F. W. Goes. 2017. Automatic Partitioning of Stencil Computations on Heterogeneous Systems. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 43–48.

[13] Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos, and Luís F. W. Góes. 2017. TOAST: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience* (2017). https://doi.org/10.1002/cpe.4053

[14] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 149–163.

[15] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868.

[16] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit*.

[17] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, NC, USA) *(PPoPP '09)*. Association for Computing Machinery, New York, NY, USA, 75–84.

[18] Z. Wang and M. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.