Machine Learning in Compilers:
Past, Present and Future

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

Chris Cummins
Facebook AI Research
Menlo Park, California
cummins@fb.com

Abstract—Writing optimising compilers is difficult. The range of programs that may be presented to the compiler is huge and the systems on which they run are complex, heterogeneous, non-deterministic, and constantly changing. The space of possible optimisations is also vast, making it very hard for compiler writers to design heuristics that take all of these considerations into account. As a result, many compiler optimisations are out of date or poorly tuned.

Near the turn of the century it was first shown how compilers could be made to automatically search the optimisation space, producing programs far better optimised than previously possible, and without the need for compiler writers to worry about architecture or program specifics. The searches, though, were slow, so in the years that followed, machine learning was developed to learn heuristics from the results of previous searches so that thereafter the search could be avoided and much of the benefit could be gained in a single shot.

In this paper we will give a retrospective of machine learning in compiler optimisation from its earliest inception, through some of the works that set themselves apart, to today's deep learning, finishing with our vision of the field's future.

Index Terms—machine learning, compilers.

I. Introduction

Machine learning in compilers has been around for more than two decades. It is now a burgeoning field. This paper looks back at where the field started, covers some of the stand out works over the years, and then presents our vision for the future. We add to two earlier surveys, [1], [2], offering additional perspectives on the field. First, we begin with the precursor to machine learning in compilers, directly searching the optimisation space with iterative compilation.

II. Iterative Compilation

Developers have known since they first used optimising compilers that the compiler does not always choose the best options. They would try different compiler command lines, either in an ad hoc fashion or more rigorously by searching. Even simple techniques, like exhaustively searching for the best tiling factors for matrix multiplication, could yield considerable benefits. Eventually, this practice would be named iterative compilation in Europe or adaptive compilation or auto-tuning in the US.

The idea is straightforward: define a space of optimisation strategies, such as unrolling factors, tilings, or complete command lines, then use a search technique to find the best one. The search evaluates each strategy by compiling the program and running it with representative inputs enough times to estimate the measure of interest (typically performance). This process is shown in Figure 1. Unlike many compiler heuristics, iterative compilation is blind to the platform, easily adapts to changes, and is based on evidence, not belief. The potential gains are huge, [3] found speedups of up to 2.23× over many data sets.

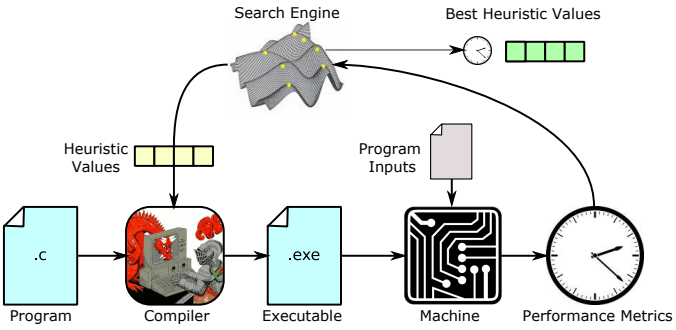

Fig. 1. Iterative Compilation: a search technique explores a space of compilation strategies, continually compiling, executing and profiling to find the best performing strategy.

The paper that first coined the term iterative compilation was [4], in which they show how the search space for one problem is both non-linear and different across architectures. They used a simple grid search, but what they were trying to demonstrate was: first, that any heuristic was going to be difficult for humans to derive; second, that a new heuristic would be needed for each architecture; and third, that the benefits of selecting the right optimisation include considerable performance gain.

There have been many iterative compilation works. Each targets some different heuristic or applies a different search technique. The use of genetic algorithms is common [5]–[7], but random search [8] and greedy approaches [9] feature also. Optimisation targets include phase ordering [7], [10], code size [6], compiler flags [11], and many others. Libraries, such as ATLAS [12] and SPIRAL [13], auto tune on installation. PetaBricks [14] and LIFT [15] are among some of the works that expand the search space to include algorithmic choices.

978-1-7281-8928-4/20/31.00©2020IEEE

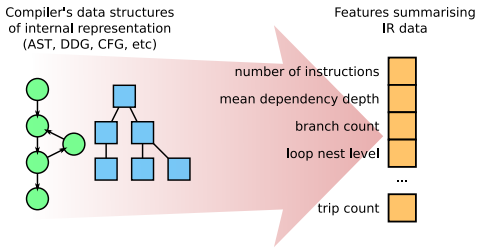

Fig. 2. Feature vectors are designed by compiler experts who decide what information will be most helpful when deciding the best value for the target heuristic.

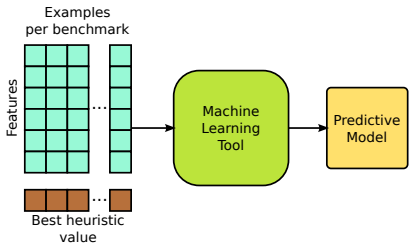

Fig. 3. A supervised machine learning tool creates a predictive model from training examples.

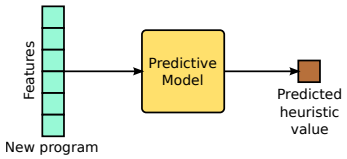

Fig. 4. In production, the predictive model replaces a human constructed heuristic. Features are calculated for a new program and the predicts the best heuristic value.

The high cost of iterative compilation has been addressed by statistical techniques [16]. Frameworks to support iterative compilation exist, such as Collective Tuning [17], OpenTuner [18], and CLTune [19].

III. Machine Learning

Although iterative compilation can improve program performance to a remarkable degree, the search, which must be repeated for every new program and might require thousands of compilations and executions, is so time-consuming as to be impractical for all but a few specialised use cases. Early pioneers began to look to supervised machine learning to get the same benefits as iterative compilation but in a single compilation.

The principle is relatively straightforward. Training programs are iteratively compiled to find the best compilation strategy for each. For example, if the optimisation is loop unrolling, then iterative compilation will find the best unroll factor for each loop in a number of training benchmarks. A compiler writer decides what information summarises the programs in a way that may be useful in deciding which compilation strategy to apply to any particular program. For loop unrolling, this might be a vector of values, such as the loop's trip count, the number of instructions in the loop, the dependency depth, and so forth. These pairs of summary vectors and desired strategies found by iterative compilation become training data for a supervised machine learner. The summary data are called feature vectors. The output of the learner is a model that can be used, given the features of a new, unseen program, to predict the best compilation strategy or heuristic value. The model can then be inserted into the compiler, replacing whatever human-built heuristics existed previously. Moreover, should there be any changes to the architecture, operating system, the rest of the compiler, or the target application domain, then the training data can simply be regenerated in the new environment and machine-learned heuristic retuned on that. These steps are shown in figures 2, 3 and 4.

The earliest example we can find is [20] which used a neural network for branch prediction, both in hardware and in compiler optimisations. Since then, many different compiler optimisations have been targeted, each working at a different granularity. Many learn command line options for whole programs or compilation units [8], [21]. Others consider individual heuristics, such as loop unrolling [22], [23], instruction scheduling [24], inlining [25], data partitioning [26], or thread coarsening [27]. Dozens of different heuristics have been examined. A series of works used genetic programming[1] to learn heuristics [28]–[30].

Perhaps one of the more well known works was MilepostGCC [21]. Milepost brought together the many necessary tools to do complete machine learning in compilers experiments. These tools included the basic iterative compilation, simple static program features, and so forth to enable a modest range of experiments to be undertaken. The work was featured on the extremely popular slashdot website, leading to widespread attention for a while.

An interesting work [8] employed a hybrid approach. They used machine learning, not to directly predict the best optimisation, but rather to predict which part of the search space would be profitable for further iterative compilation. Their model determines which points in the space are likely within 5% of the optimal. Iterative compilation then searched that reduced space, lowering the cost of iterative compilation and reducing the burden on the machine learning to get the prediction correct.

A. Fitting into the ML Mould

The examples previously cited learn the heuristic directly. Often this is predicting some best category, such as loop unroll factor or whether to inline a function. Not all problems fit so neatly, and when it is not clear how to represent the heuristic values as a category then they

[1]Conflating genetic programming with machine learning is a good way to get in a row with machine learning experts, as one author has found out to his cost.

instead require more careful consideration for how to fit them into a machine learning paradigm.

Instruction scheduling, for example, requires permuting instructions to improve their performance. Rather than learning permutations of instructions, [31] and [30] learn priority functions that compare two instructions and determine which should come first. This neatly sidesteps the permutation issue, as the ordering is generated implicitly.

In [32], it is similarly not obvious how the problem will be mapped to machine learning. The input problems are parallel streaming task graphs where each node represents some computation that is fed along edges to the next computation. The goal is to allocate the tasks to threads to reduce communication overhead while also improving parallelism and throughput. The approach taken in the paper again avoids having to directly learn the heuristic. Instead, they take an input graph, describe it using some features, and then predict what the features of the ideal, mapped version of that graph should be. This gives them something to aim for and they then do a hill-climbing search, randomly applying merge or split operations to regions of the graph, aiming to approach the predicted ideal. As the features are static, the mapped program need not be run during the search, accelerating the search.

There are several works that predict the performance of a program after applying a transformation [33]–[35]. In these papers, the general idea is that, after learning how the speedup of different parts of the optimisation space, the space can be quickly searched for good optimisations without having to re-execute the program.

B. Feature Design

Machine learning tools need features that describe the data to the heuristic. For example, in loop unrolling the number of times the loop will iterate may be a good feature. The most common feature types have been the frequencies of instruction types in the code to be optimised [24], [27], [36], [37]. When learning branch prediction routines, [20] used features about the type of the branch and the successor instructions, whether the branch is a loop and its direction. [30] learned hyper-block formation policy with features including the maximum dependency height of instructions in the path, the total number of instructions, and whether there are memory hazards. For register allocation, they use the number of calls in the containing basic block, use-def counts and estimates of spill costs and benefits. [24] also included features about the number of garbage collection points in a method and the possibility of causing a thread switch. [37] summarise loops according to the number of memory accesses, histograms of the different instruction types, and iteration count estimates. [22], [31] use counts and lengths of use-def chains, dependency heights, and latencies.

Sensible features are essential to learning good heuristics. Conversely, whether the indentation was four spaces or one tab is probably not a useful feature[2]. This example is facetious but it is quite common for features to be poorly designed. Features that are in essence random relative to the problem at hand can confuse the machine learner. It can find a coincidental correlation in the random data and then attempt to learn that, ruining the generality of the model. A feature may be a function of other features, which is sometimes helpful when the machine learner cannot combine information that way, and sometimes not helpful when the redundant information makes learning slower and less accurate. Features may also be incompatible with the chosen machine learning tool. For example, if features place different classes in nonlinear regions, linear models cannot separate them.

Some solutions to this have been explored. [38] determined the distance between two programs by a graph similarity metric. They begin by comparing the similarity of basic blocks and then expand that outwards to measure the similarity of CFGs. Armed with this distance metric, they use a k nearest-neighbour model to find which heuristic values to use. The downside of the approach is the computational cost. Not only do many training graphs need to be shipped with the compiler but the distance calculation itself is expensive.

Genetic programming was used to search for features in [23]. A grammar described a programming language of features. A genetic program evolved individuals from the feature language, searching to improve the accuracy of a machine learning model for the target heuristic.

A similar technique to searching a feature space was taken by [39] where the program was represented by facts in the logic programming language, Datalog. From these facts, they used a solver to infer rules matching program facts to the desired heuristics.

Static compiler features have always been an issue. It is impossible to know statically how often code will be executed. Consider a function with two possible control paths. Very different optimisation may be necessary, depending on which path is taken. It may be that at run time one path is rarely or never taken, but the static compiler cannot know this. Many machine learning works have used features like 'the number of instruction in a function' and are oblivious even to static control flow, let alone dynamic control flow. Performance counters have been used as features to solve this [40]. The program is run once, collecting counters for hot code. These counters are used directly as features which now depend on the run time behaviour, rather than purely static analyses.

C. Offline vs Online

Nearly all machine learning works for compilers do the learning offline. Different compilation options are applied to example programs in the lab. The reasons that the learning is not done live on users' machines are twofold.

[2]Although clearly, anyone using tabs is psychotic.

Firstly, it is quite common to find that, while looking for the best optimisation strategy, the search touches on appallingly bad strategies that can trash the program's performance. Users would be distinctly nonplussed to find their programs running at half speed, even if it was in the cause of eventually good performance. Secondly, in the laboratory, the inputs are always the same so that for deterministic programs the timings of any two runs are directly comparable and it is easy to see which strategy is superior. In a live system, the inputs are different each time and, as there is no guarantee that two runs will do the same amount of work, the difference in their run times may be due to that, rather than better optimisation. Chen [41], [42] shows how much input data can affect the best optimisation choices.

[43] identified stable phases in a program execution during which performance comparisons could be made. They then used multi-versioning to select different compiled versions to improve performance. Mars [44] proposes using IPC for online adaptation. During the learning phase, competing versions of a hot function are executed, each for the same amount of time. The one with the highest number of retired instructions is selected for use. [45], [46] have used user inputs to perform adaptation in distributed data centers. Each compilation worker receives a subset of the input data set on which to evaluate a small set of optimization settings. The best such setting from each round is used for subsequent executions of the same code. The best-found compilation strategy is then refined over time, by testing new settings and re-evaluating old ones on new data sets. This approach only works well with MapReduce-like workloads, since it relies on the framework for repeating the same computation multiple times without causing side-effects. [47] solves some of the online problems for iterative compilation only. They capture a memory snapshot as a hot function is being executed live. The snapshot is recreated offline and compilation strategies are searched. This enables tuning specifically for each user while ensuring the user does not suffer slow performance during the search.

IV. Deep Learning

The advent of deep learning has begun to pervade every discipline and compilers are no exception. Deep learning means using very large neural networks with many, or 'deep', layers. Previously, these large networks were infeasible to train, but processing power is now up to the task. What has changed by these advances is that large, deep neural nets can scale up to much larger training data sizes and produce much more detailed models that can learn functions with an ease that might previously have seemed magical. But, the game-changer is that whereas before the choice of features was so crucial, it is now possible to feed in raw data and the deep learner will make sense of it.

The first such work was [48]. They parsed input programs as source token streams and then used a deep neural network to directly predict from that what the right heuristic value should be for some optimisations in OpenCL. They made use of an existing technology that had had great success in natural language processing called long short term memory networks (LSTM). These nets are able to process streams of inputs and remember events from arbitrarily far in the past. This enables them to somewhat understand the structure of the program, such as whether a variable has been declared in the past. The results improved over prior, hand-built features.

The authors of [49] went further. They realised that while a token representation is well suited to ambiguous natural language, a graph-based representation would suit programming languages better. They represent the instructions of a program as edges in a graph describing the relationships between variables. They then learn vectors to represent each instruction given its context in the graph. A program can then be processed by LSTMs as a sequence of these vectors. [50], [51] extends this idea so that the graph structure is used not just to decide the vectors used to represent instructions, but also how the learner processes them. They use message passing neural networks where each node has a state. That state is sent along edges to each neighbour who merges that into its own state with a learned function. After some rounds of message passing a learned aggregation function gives the heuristic value.

Compared with machine learning of the past, deep learners are hungry for data sets far larger than is typically seen in compiler research. While in some cases this can be mitigated by generating data synthetically [52], the pace of innovation will increase considerably with an increase in the availability of large labelled data sets.

V. Reinforcement Learning

Recently, reinforcement learning techniques have begun to make inroads in compiler optimization. Reinforcement learning concerns the process of iterative decision making of an agent within an environment. The environment provides a state, a set of actions, and a reward signal. The goal of an agent is to select the sequence of actions, one at a time, that will maximise cumulative reward.

A recent work [53] casts loop vectorization as a reinforcement learning problem. An environment represents a program containing a single loop of interest, observations are provided by summarizing paths through the program's AST, and reward is calculated using the runtime of the program after applying a given vectorization choice. This approach works well, but frames the problem in such a way that an agent only makes a single decision per problem. One of the key strengths of reinforcement learning is the ability to decompose large problems into a sequence of smaller discrete choices.

Many compiler optimization problems can be broken down into a sequence of smaller decisions to fit the

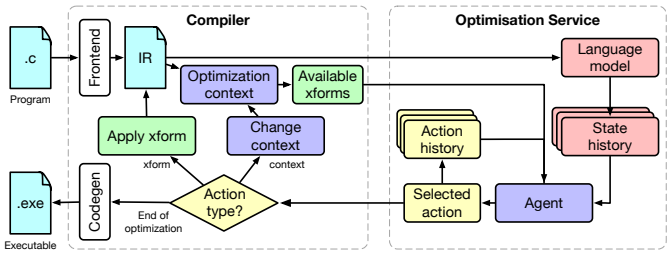

Fig. 5. Our vision: A reinforcement learning system to control all aspects of the compiler.

reinforcement learning mould. For example, in [54], the full optimization pipeline of LLVM is presented as an environment in which a partially optimized program provides the state, and an action is to select a single optimization pass to run. The selected pass is then run, producing a new state. Reward is provided by compiling the partially optimized program and estimating the execution cycle count. In this manner, the sequence of transformations that produces the best performing code can be found through incremental improvements.

[55] uses reinforcement learning to tackle a graph partitioning problem, similar to [32]. In [55], the goal is to find the optimal device placement for nodes in large computation graphs so as to minimize runtime by most efficiently exploiting the available hardware and minimizing communication costs. An LSTM model is used to produce a representation of a particular mapping by feeding through a description of each operation's type, shape, and graph adjacencies. A second LSTM model decodes this representation to provide a sequence of device placements. A key challenge here is that these sequence-to-sequence techniques struggle with long sequences, limiting the scalability over large problems. This was addressed in [56] using a hierarchical model to decompose large graphs.

VI. The Future

We have been in this field for the last fifteen years[3]. In that time we have seen it move from a niche academic discipline, struggling for acceptance, to one which industry is now scrambling to adopt.

So, where is this field going? What do we need to do in the coming years? Optimisation is hard and it is only going to get harder still. We need to remove humans from all the heuristics in the compiler, and this will require a coordinated, inter-disciplinary effort. We need compiler writers to build modular compilers that support iterative compilation and machine learning from the ground up. We need machine learning researchers to invent models that are suited to the recurrent, flow-based nature of programs. And we need efficient learning strategies that can cope with the huge number of choices, distant rewards, and slow evaluations that apply to compiler optimisation.

[3]Well, the one of us with gray hair has, anyway.

A. Machine Learning Enabled Compilers

Modern compilers are multi-million line pieces of software that can take years to master. Exposing the compiler as a playground for experimentation will lower the barrier to entry in compiler research, having a democratizing effect. The first step is to enable every optimization choice to be exposed through discoverable APIs with which iterative search and machine learning can interact. For example, when a loop may be unrolled, a search or machine learning tool should be able to determine the range of acceptable factors, make queries about the code and force an unroll factor. Notice that this has to be dynamic, rather than determined by some static list of unroll factors per loop, since earlier choices change the loops that will be considered. The compiler becomes a transformation and query engine, capable of making decisions but not needing to do so. There are a lot of choices made in compilers. In compilers with extensible representations, like MLIR, the challenge of enabling these APIs is greater than in more locked down compilers. The software engineering effort to make a truly machine learning-enabled compiler should not be underestimated.

B. Deep Language Modelling at Scale

Before the advent of deep learning moving into compilers the features that summarised programs were quite basic. The token-based approach of [48], the embedding of inst2vec [49], and the new graph representation of [50] make great strides. These are not enough, however. The more that the machine learning can understand the program, the better. We need better program representations and compiler-specific DNN architectures.

For instance, the most advanced representation [50] does not represent variables, types, operand order, etc. It is, therefore, incapable of replicating the common data-flow analyses that litter any modern compiler – nor can the others. Data-flow is fundamental to practically every optimisation in the compiler. We need models that can reason about the programs in at least as complex a fashion. This will require better representations and graph RNNs that match the data-flow pattern. We may never know that we have the best formulation, but if what we have cannot at least learn all the standard data-flow analyses, then we do know it is not yet enough.

C. Reinforcement Learning Everything

With all of the compiler's choices exposed and suitable representation available, we can begin to replace every heuristic in the compiler with a learned one. We see reinforcement learning as the most promising approach. Reinforcement learning seeks to choose actions that move through a state space so that the final state has the greatest associated reward. For compilers, the states are the IR

of a program, an action is a transformation of some part of the IR, and the reward is the speedup when the program is fully compiled and run on representative inputs. Such a system would continually apply transformations until no further speedup could be squeezed out.

Our proposed architecture is shown in Figure 5. The compiler front-end processes the program source code as normal, constructing an intermediate representation. It chooses an initial optimisation context on which to focus (most likely the main function). The context can be later changed by the RL system, to look at optimisations on other functions, or different granularities, such as loops, basic blocks and individual instructions. For each context the compiler can determine a set of applicable and valid transformations which it passes to an RL agent to make its choice. The IR is consumed by a language model which compresses the IR into a finite state vector. The agent chooses the next action based on the current state of the program and a history of actions and states it has seen. An action is either a transformation to apply to the current focused context or a change of focus to another context. The RL system will take actions that increase the likely future reward which will be the speedup found by applying the action sequence to the code. When the predicted future reward is zero, then no further speedup can be gained by additional actions and the process can stop, delivering the final executable to the user.

This problem is larger than those to which reinforcement learning is typically applied. The state space is huge – programs come from a space of unbounded dimension. The action space is also huge – hundreds or thousands of transformations are possible, many can be parametrised, and there are many places in the code to apply them. Evaluating the reward is slow – the program must be compiled to a binary and executed with representative inputs enough time to give statistically sound timings. All these challenges will need careful thought and vast computing power to solve, but in the end, we will have compilers that far exceed the quality of today's.

VII. Conclusion

Machine learning is making a significant impact on compiler optimisation and will continue to in coming years.

References

[1] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," ACM Comput. Surv., vol. 51, no. 5, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3197978

[2] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," Proceedings of the IEEE, vol. 106, no. 11, pp. 1879–1901, 2018.

[3] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 448–459. [Online]. Available: https://doi.org/10.1145/1806596.1806647

[4] F. Bodin, T. Kisuki, P. Knijnenburg, M. Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," Workshop on Profile and Feedback-Directed Compilation, 03 2000.

[5] S. J. Beaty, "Genetic algorithms and instruction scheduling," in Proceedings of the 24th Annual International Symposium on Microarchitecture, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 206–211. [Online]. Available: https://doi.org/10.1145/123465.123507

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," SIGPLAN Not., vol. 34, no. 7, p. 1–9, May 1999. [Online]. Available: https://doi.org/10.1145/315253.314414

[7] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," in LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems. New York, NY, USA: ACM, 2004, pp. 231–239.

[8] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams, "Using machine learning to focus iterative optimization," in CGO '06: Proceedings of the International Symposium on Code Generation and Optimization. Washington, DC, USA: IEEE Computer Society, 03 2006, pp. 295–305. [Online]. Available: http://www.anc.ed.ac.uk/machine-learning/colo/cgo06.pdf

[9] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in CGO '06: Proceedings of the International Symposium on Code Generation and Optimization. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332.

[10] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 171–182. [Online]. Available: https://doi.org/10.1145/996841.996863

[11] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff, "Automatic selection of compiler options using non-parametric inferential statistics," in PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–132.

[12] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in Conference on High Performance Networking and Computing. IEEE Computer Society, 1998, pp. 1–27.

[13] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "Spiral: Code generation for dsp transforms," Proceedings of the IEEE, vol. 93, no. 2, pp. 232–273, feb 2005.

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," SIGPLAN Not., vol. 44, no. 6, p. 38–49, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542481

[15] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance gpu code generation," in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017, pp. 74–85.

[16] H. Leather, M. O'Boyle, and B. Worton, "Raced profiles: Efficient selection of competing compiler optimizations," in LCTES '09: Proceedings of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems, June 2009.

[17] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, A. D. Malony, Z. Chamski, D. Novillo, and D. Del Vento, "Collective mind: Towards practical and collaborative auto-tuning," Sci. Program., vol. 22, no. 4, p. 309–329, Oct. 2014. [Online]. Available: https://doi.org/10.1155/2014/797348

[18] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in

Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 303–316. [Online]. Available: https://doi.org/10.1145/2628071.2628092

[19] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels." in MCSoC. IEEE Computer Society, 2015, pp. 195–202.

[20] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," ACM Transactions on Programming Languages and Systems, vol. 19, 1996.

[21] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle, "Milepost gcc: machine learning based research compiler," in Proceedings of the GCC Developers' Summit, June 2008.

[22] M. Stephenson and S. Amarasinghe, "Predicting Unroll Factors Using Supervised Classification," in International Symposium on Code Generation and Optimization (CGO). IEEE, 2005.

[23] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in CGO '09: Proceedings of the International Symposium on Code Generation and Optimization, March 2009.

[24] J. Cavazos and E. Moss, "Inducing heuristics to decide whether to schedule," vol. 39, 06 2004, pp. 183–194.

[25] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," SIGPLAN Not., vol. 41, no. 10, pp. 229–240, 2006. [Online]. Available: http://homepages.inf.ed.ac.uk/jcavazos/oopsla-2006.pdf

[26] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions." vol. 26, 07 1991, pp. 213–223.

[27] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 455–466. [Online]. Available: https://doi.org/10.1145/2628071.2628087

[28] N. Paterson, M. Livesey, and K. Ss, "Evolving caching algorithms in c by genetic programming," in In Genetic Programming. MIT Press, 1997, pp. 262–267.

[29] M. O'Neill and C. Ryan, "Automatic generation of caching algorithms," in Evolutionary Algorithms in Engineering and Computer Science. John Wiley and Sons, 1999, pp. 127–134.

[30] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ser. PLDI '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 77–90.

[31] E. Moss, P. Utgoff, J. Cavazos, C. Brodley, D. Scheeff, D. Precup, and D. Stefanović, "Learning to schedule straight-line code," in Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10, ser. NIPS '97. Cambridge, MA, USA: MIT Press, 1998, p. 929–935.

[32] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 307–318. [Online]. Available: https://doi.org/10.1145/1854273.1854313

[33] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 45–55. [Online]. Available: https://doi.org/10.1145/1669112.1669121

[34] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," in Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, ser. CGO '11. USA: IEEE Computer Society, 2011, p. 119–129.

[35] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," SIGARCH Comput. Archit. News, vol. 34, no. 5, p. 185–194, Oct. 2006. [Online]. Available: https://doi.org/10.1145/1168919.1168881

[36] Z. Wang, D. Grewe, and M. F. P. O'boyle, "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems," ACM Trans. Archit. Code Optim., vol. 11, no. 4, Dec. 2014. [Online]. Available: https://doi.org/10.1145/2677036

[37] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, ser. AIMSA '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 41–50.

[38] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in Proceedings of the Tenth International Symposium on Code Generation and Optimization, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 196–206. [Online]. Available: https://doi.org/10.1145/2259016.2259042

[39] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in Proceedings of the 2010 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2010, Scottsdale, AZ, USA, October 24-29, 2010, V. Kathail, R. Tatge, and R. Barua, Eds. ACM, 2010, pp. 197–206. [Online]. Available: https://doi.org/10.1145/1878921.1878951

[40] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam, "Automatic performance model construction for the fast software exploration of new hardware designs," in CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. New York, NY, USA: ACM Press, 2006, pp. 24–34.

[41] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 448–459.

[42] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," ACM Trans. Archit. Code Optim., vol. 9, no. 3, pp. 21:1–21:30, Oct. 2012.

[43] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, "A practical method for quickly evaluating program optimizations," in International conference on high-performance embedded architectures and compilers, 11 2005, pp. 29–46.

[44] J. Mars and R. Hundt, "Scenario based optimization: A framework for statically enabling online optimizations," in CGO'09.

[45] Y. Chen, S. Fang, L. Eeckhout, O. Temam, and C. Wu, "Iterative optimization for the data center," in Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 49–60.

[46] S. Fang, W. Xu, Y. Chen, L. Eeckhout, O. Temam, Y. Chen, C. Wu, and X. Feng, "Practical iterative optimization for the data center," ACM Trans. Archit. Code Optim., vol. 12, no. 2, pp. 15:1–15:26, May 2015.

[47] P. Mpeis, P. Petoumenos, and H. Leather, "Iterative compilation on mobile devices," in Proceedings of the 6th International Workshop on Adaptive Self-tuning Computing Systems (ADAPT), January 2016.

[48] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2017), September 2017.

[49] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code

semantics," in Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 3588–3600. [Online]. Available: http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf

[50] A. Brauckmann, S. Ertel, A. Goens, and J. Castrillon, "Compiler-Based Graph Representations for Deep Learning Models of Code," in CC, 2020.

[51] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," arXiv preprint arXiv:2003.10536, 2020.

[52] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2017, pp. 86–99.

[53] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning," in CGO, 2020.

[54] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "Autophase: Compiler Phase-ordering for HLS with Deep Reinforcement Learning," in FCCM, 2019.

[55] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in ICML, 2017.

[56] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A Hierarchical Model for Device Placement," in ICLR, 2018.