Intelligent Compilers

John Cavazos

Computer and Information Sciences Department, University of Delaware
103 Smith Hall, Newark, DE, USA
cavazos@cis.udel.edu

***Abstract*—The industry is now in agreement that the future of architecture design lies in multiple cores. As a consequence, all computer systems today, from embedded devices to petascale computing systems, are being developed using multicore processors. Although researchers in industry and academia are exploring many different multicore hardware design choices, most agree that developing portable software that achieves high performance on multicore processors is a major unsolved problem. We now see a plethora of architectural features, with little consensus on how the computation, memory, and communication structures in multicore systems will be organized. The wide disparity in hardware systems available has made it nearly impossible to write code that is portable in functionality while still taking advantage of the performance potential of each system. In this paper, we propose exploring the viability of developing intelligent compilers, focusing on key components that will allow application portability while still achieving high performance.**

I. Introduction

For decades, software developers have enjoyed an increase in application performance from clock speed scaling due to Moore's law. Much of this performance gain resulted from increasing clock speeds. However, due to temperature concerns, this is no longer possible. The industry is now in agreement that the future of architecture designs lies in multicores. As a consequence, all computer systems today, from embedded devices to high-end servers, are being built with multicore processors.

Although there is agreement on having multiple cores per processor, researchers in industry and academia are exploring many different multicore hardware design choices. Because of this diversity in architectural ideas, developing a compiler that fully exploits the available performance of multicore processors is a major unsolved problem.

This paper proposes to fundamentally change the way in which compilers are developed, replacing hand-tuning by self-tuning compilers that adapt automatically to match the characteristics of each target computing system. By automating the process of performance tuning, programmers will have more time to focus on higher level issues, e.g., algorithm design, robustness, and correctness. Moreover, intelligent compilers will reduce the need for specialized expertise across the range of targetted computing systems.

A. Challenges Facing Compiler Writers

In addition to solving challenges in constructing intelligent compilers, compiler writers must address important technical barriers that prevent traditional compiler infrastructures from producing code that achieves even a reasonable fraction of a system's available performance.

First, traditional compilers do not adequately model the computing systems they target. Computer architectures are becoming increasingly complex, and ever more sophisticated compiler optimizations are required to exploit new features of the architecture. Previous architectures were simple enough so that the relevant details that affect performance were easy to discern and model. But today's architectures are intractably complex. And with the advent of multicores, hardware will continue to be difficult to model. Also, the fast-paced development of new processor technology precludes spending a large amount of time hand-crafting.

Second, compiler writers are expected to find effective and efficient heuristics to compute approximate solutions to NP-hard problems, such as instruction scheduling and register allocation. The fact that compiler phases interact with each other further complicates the construction of good heuristics, adversely affecting their effectiveness. This makes it difficult to construct sets of compiler heuristics that interact well with other downstream optimizations.

Third, traditional compilation environments are too rigid, and it is difficult for compiler writers to adapt them quickly to new processors and system configurations. In particular, optimizations and the heuristics that control them are often intertwined. Also, optimizations are usually composed in a fixed order, and other optimization orders are typically not tested, leading to brittle compilers. Furthermore, computer engineers desire the ability to experiment with many different designs of an architecture to measure gains of architectural alternatives over one another. This requires not only quick prototyping of the architectures in question, but also quick prototyping of optimizing compilers for those architectures. Hand-tuning any part of the compiler is ineffective at supporting quick prototyping of optimizing compilers.

Fourth, static analysis has to make conservative assumptions, and many times the analysis does not have the all information to effectively optimize a piece of code. Thus, we believe that intelligent dynamic optimizers will become prevalent due to their potential to improve the optimization of a program using information available at runtime. We believe programs will also incorporate runtime systems to provide runtime characterizations that can be used by the compiler

to optimize subsequent runs of the program.

Our vision is simple: the above technical problems cannot be productively resolved unless there is a radical departure from traditional methods of developing inflexible compilers to new methods for building intelligent compilers.

II. Intelligent Compilation Methodology

There are many benefits to using an automatic technique over the traditional approach to constructing compiler heuristics. A machine learning process substantially reduces the time to construct effective heuristics. Manual construction and tuning of compiler heuristics can take weeks or longer. In contrast, once you have cast an optimization problem as a learning problem and generated a *training set*, machine learning can find effective heuristics in minutes. This methodology produces heuristics that are accurate (i.e., converge to a better solution more often than manual techniques), robust (solve a wide range of problem instances), simple (the heuristics are easy to integrated into optimization code), and are potentially generalizable and retargetable to other problems.

A. Methodology Description

We now describe the steps involved in applying supervised learning to a compiler problem. This process involves six distinct steps: *phrasing the learning problem*, *constructing features*, *generating training instances*, *training*, *integrating the heuristic*, and *evaluating its effectiveness*. Note that, although the following paragraphs are specific to supervised learning, much of the discussion pertains to applying online and/or unsupervised learning techniques to compilation problems as well.

Phrasing the Learning Problem: Generally this means that, when the optimizer needs to make a decision, certain factors (the *features*) are inputs to a decision function, whose output selects a choice from two or more possibilities. For example, one way to phrase the optimization phase-ordering problem is: "Given certain optimizations already applied and two possible optimizations to apply next, choose which of the two to perform." This decision function can be used to run a tournament among three or more optimizations, choosing which optimization to apply to the code being optimized. Applying one optimization changes the characteristics of the code being optimized, and therefore the next optimization predicted to be beneficial. One can iterate this process until some fixed number of optimizations have been applied or until the characteristics of the code reaches a state where the learning algorithm predicts that no further optimizations should be applied. The key property is framing the heuristic decision function so that it selects among a small set of classes (in the example, the *first* optimization of the two, or the *second*), often just two classes.

Feature Construction: Construct a set of properties (features) that you deem important to the optimization problem. Finding the "right" set of features and a good representation for this information, is the most challenging, and probably least automatable, part of the process. For example, in instruction scheduling, it is well known that considering the *critical path* of instructions helps, so one might use features related to the critical path. However, for other compiler problems with no prior art, one might have to guess which features might work.

Generating Training Instances: Instrument the compiler to generate a training instance at every point of the optimization algorithm where the heuristic would typically be applied, or at least at a significant, randomly chosen sample of those points. This generally involves arranging things so that you can pursue all decision possibilities at that point. You need a way to evaluate the end result of each choice so that you can label the instance with the best choice. You must also output the values of the features at the decision point. A training instance says: "In this situation (features) you should do X (label)."

An interesting issue arises when, in order to evaluate a decision, you have to take more decisions. For example, in instruction scheduling, to evaluate a given choice, you need to schedule the rest of the block. There are several alternatives to solving this problem. Sometimes you can enumerate all possibilities (exhaustive search) and pick the best. This works for short blocks in instruction scheduling. An alternative is to consider a randomly chosen sample (an approach useful for large blocks, which may have a huge number of possible schedules). Finally, you can run to the end of the problem using one or more heuristics already known to be competent (though perhaps not as good as you are aiming for). For instruction scheduling, you might use a generic critical-path based heuristic.

Another significant issue is *evaluating* each choice. For instruction scheduling, one can use a cycle-level model of the CPU to estimate the number of cycles that each schedule for a block would take when executed. For evaluating a short piece of optimized code one could also use a cycle-accurrate timer. Note that these estimators only need to be good in a *relative* sense, leading to good decisions about which choice is better. Their *absolute* estimates do not matter as much.

Training the Learning Component: Here, the compiler writer can present the training instances to one of a variety of supervised learning components in order to construct a tuned heuristic. This is automatic and easy to do, given training instances in the appropriate form. In addition to the learned heuristic, machine learning tools generally provide statistics concerning their effectiveness (classification accuracy) on the training set and on one or more test sets. We recommend using a process called *leave-one-out cross-validation* to test the accuracy of your heuristics. That is, given a suite of N benchmarks, one can generate training instances from $N-1$ of the benchmarks and test the accuracy of the generated heuristic on the benchmark that was left out. This gives the learning algorithm a broad collection of instances from which to learn, but insures evaluation on instances *not* used in the learning process. This checks the generality of the heuristic, insuring that it is not overly specialized to particular benchmarks.

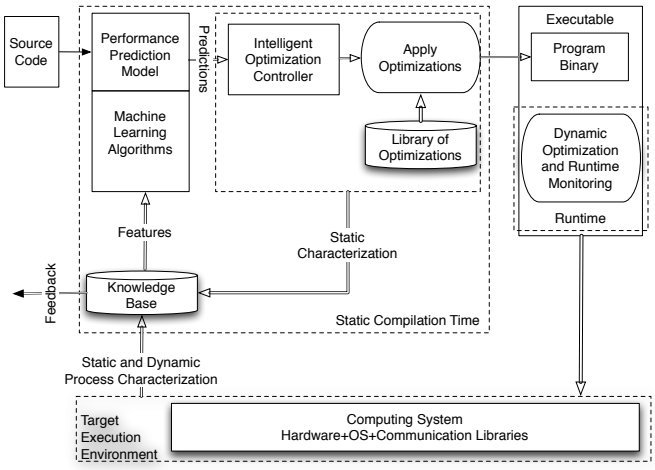

Fig. 1. Overview of an Intelligent Compiler.

Integration of the Induced Heuristic: The output of the supervised learning component can now be converted into code and integrate it into the compiler. Many supervised learning components already have options to produce code as output, making this particularly easy.

Evaluation of Induced Heuristic: Finally, the performance of the tuned heuristic can be empirically evaluated. While classification accuracy is usually available from the learning tool and gives a good sense of how well the features predict the labels, there is usually some degree of approximation or modeling in the generation of training instances, so one must test the heuristic's effectiveness in practice. If the effectiveness is not good enough, one may need to adjust the set of features or the process for generating instances.

B. Benefits of a Machine Learning Methodology

This methodology offers several benefits over manual methods:

1) Saves programmer-hours compared to time-consuming manual techniques; it can induce competent heuristics in minutes, versus weeks or months; fast turn-around may support prototyping compilers to assist with new chip development.
2) Induces heuristics that are based on sound machine learning principles; manual techniques are often not as sound or principled.
3) Constructs compiler heuristics systematically, not by trial and error.
4) Can assist in developing heuristics, where no hand-crafted ones exist yet.

We now turn to describing the separate components of an intelligent compiler.

III. Overview of an Intelligent Compiler

Figure 1 depicts, from left to right, the high-level flow of compilation from source code to execution on a computing system (including section numbers where important components are discussed). At static compilation time, an intelligent compiler will analyze source code with *performance prediction models* that predict the optimizations likely to be beneficial. These models, developed using *machine learning algorithms*, will output a probability distribution of the benefit of optimizations, given the specific code being compiled. The *intelligent optimization controller* will apply optimizations based on these probability distributions pertaining to the benefit of improving a particular metric (e.g., performance, power, or code size). This framework can also integrate a performance prediction model into the binary that predicts when to apply certain *dynamic optimizations*. Information used by the prediction models comes from three potential sources: the *static and dynamic process characterization*, the results optimizing similar programs, and the analysis of the program being optimized. All this information is stored in a *knowledge base* for future compilations.

A. Intelligent Optimization Controller

The **intelligent optimization controller** will choose optimization sequences to apply from the areas of the optimization space delineated by the performance prediction model. These areas are predicted with high probability to contain beneficial optimization sequences. The intelligent optimization controller can either generate a program executable in one trial, or it can be used in an iterative manner. The framework may determine that it is possible to fine-tune the selection of optimizations by getting information from the execution of the application on the target system. The process can iterate until the selection of optimizations converges. The generated executable can be linked with a **dynamic optimization module** if the models predict that further optimizations at runtime will be beneficial (described in Section III-D).

1) An Example: In previous work, we developed an intelligent optimization controller that used static code features to correlate the programs to be optimized with previous knowledge in order to predict the probability that certain optimizations would be beneficial [1]. Consider the diagram in Figure 2(a), which shows the behavior of the `adpcm` program on the Texas Instrument C6713 processor. This diagram is an attempt at plotting all points within 5% of the optimum in the space of all optimization sequences of length 5 selected from a set of 13 optimizations.[1] It is difficult to represent a large 5 dimensional space graphically, so each optimization sequence $(t_1 t_2 t_3 t_4 t_5)$ is plotted at position $(t_1 t_2)$ on the x-axis, which denotes prefixes of length 2, and position $(t_3 t_4 t_5)$ on the y axis, which denotes suffixes of length 3. The most striking feature is that minima are scattered throughout the space, which means that finding the very best optimization sequence is a difficult task. Prior knowledge about where good points are likely to be could *focus search*, allowing the minimal point to be found faster. Alternatively, given a fixed number of evaluations, improved performance can be expected if good areas of the space to search are known. We

[1]Unrolling factors were counted as individual optimizations, and since unrolling was only allowed to appear once in any optimization sequence, the total number of optimization sequences exhaustively evaluated for this benchmark was 88000.

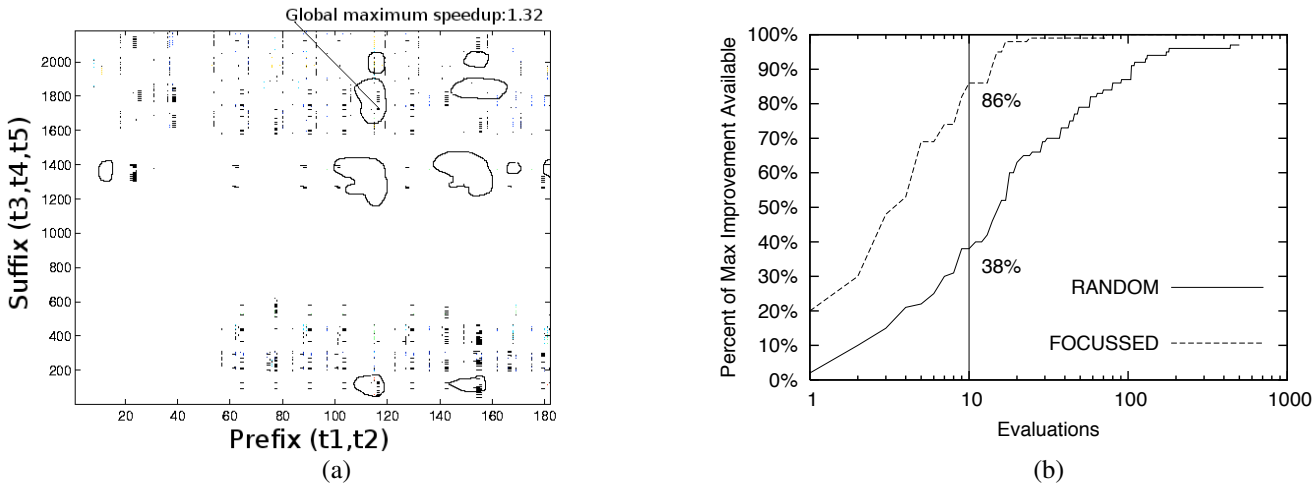

Fig. 2. (a) The points correspond to optimization sequences whose performance is within 5% of the optimum for adpcm on the TI C6713. The contours are the areas predicted to contain good optimizations.(b) How close to the best performance random and focussed search achieve for each program evaluation. The random algorithm achieves 38% of the maximum improvement in 10 evaluations; the focussed search 86%.

developed a technique that uses machine learning to construct a predictive model from a knowledge base of search data from other programs. The predictive model used code features to define good regions of the space to search. In Figure 2(a), the contour lines enclose those areas where the model predicted good points. Note that the model predicts the area which contains the optimal optimization sequence.

Using this model we were able to reduce the amount of search to find the best sequence - rapidly reducing the cost of iterative search. This can be seen in Figure 2(b), which compares random search[2] (**RANDOM**) with a predictive model (**FOCUSSED**). The x-axis (logarithmic scale) denotes the number of evaluations performed by the search. The y-axis denotes the best performance achieved so far by the search: 0% represents the original code performance, 100% the maximum performance achievable. It is immediately apparent that the predictive model rapidly speeds up the search. For instance, after 10 evaluations, random searching achieves 38% of the potential improvement available, while the focused search achieves 86%. As can be seen from Figure 2(b), such a large improvement would require over 80 evaluations using random search, justifying further investigation into predictive modelling.

B. Application and Architecture Characterization

Intelligent compilers will have to collect both static and dynamic characteristics of applications being optimized as well as architectures being targeted. It is important to characterize all resources that significantly affect application performance that can be effectively used with predictive modelling techniques to drive compiler optimizations. A proper characterization of the application and the target computing system is critical to the prediction of beneficial optimization choices and to the identification of good areas of an optimization search space to traverse.

[2]This search was an average of 20 trials to be statistically meaningful.

It is essential that representations be created which enrich the **knowledge base** with information that is relevant to any decision point and range of choices that affect performance. The knowledge base in an intelligent compiler will be a coherent aggregate of static and dynamic characterization both of the application and target system. For applications, this characterization can be extracted from compiler analysis phases or through instrumenting the code to retrieve dynamic information, such as performance counter information. For the targeted computing system, this characterization can be retrieved from a configuration file and/or can be efficiently characterized with microbenchmarks (as in previous work [2]).

1) An Example: In previous work, we successfully used performance counters to characterize applications and to predict optimizations that were beneficial[3]. This section looks at just one program to illustrate how performance counters can be used to select beneficial compiler optimizations. As the performance counter values are related to actual program performance, they can be used by a modelling technique to select good optimization settings (described in the next section). We developed models that take as input performance counter values of a new program being optimized. By using prior knowledge from previously examined programs, the models predict the optimizations most likely to result in a speedup and improved performance counter values.

Figure 3 shows the performance counter values for the `181.mcf` benchmark from the SPEC benchmark suite on the AMD processor, compiled with the commercially available PathScale optimizing compiler. What is immediately apparent is that `181.mcf` is an unusual program. It has a much greater number of memory access per instruction than average - up to 38 times more in the case of L2 store misses (L2_STM). A learned model should identify this and predict optimizations that reduce the impact of cache accesses.

Figure 4 shows the performance counter values after applying two optimization schemes, -Ofast (FAST), the highest

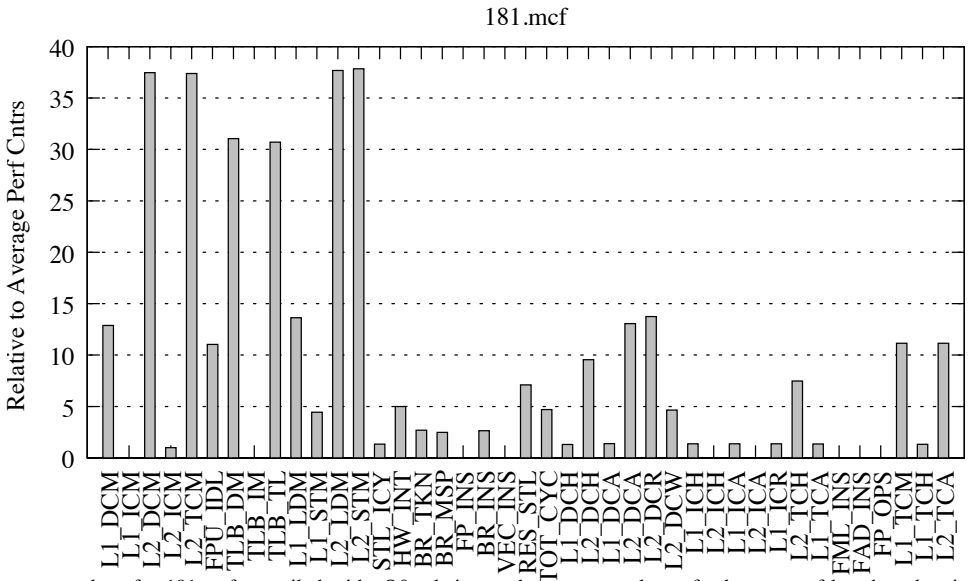

Fig. 3. Performance counter values for 181.mcf compiled with -O0 relative to the average values of a large set of benchmark suites (SPECFP, SPECINT, MiBENCH, Polyhedron).

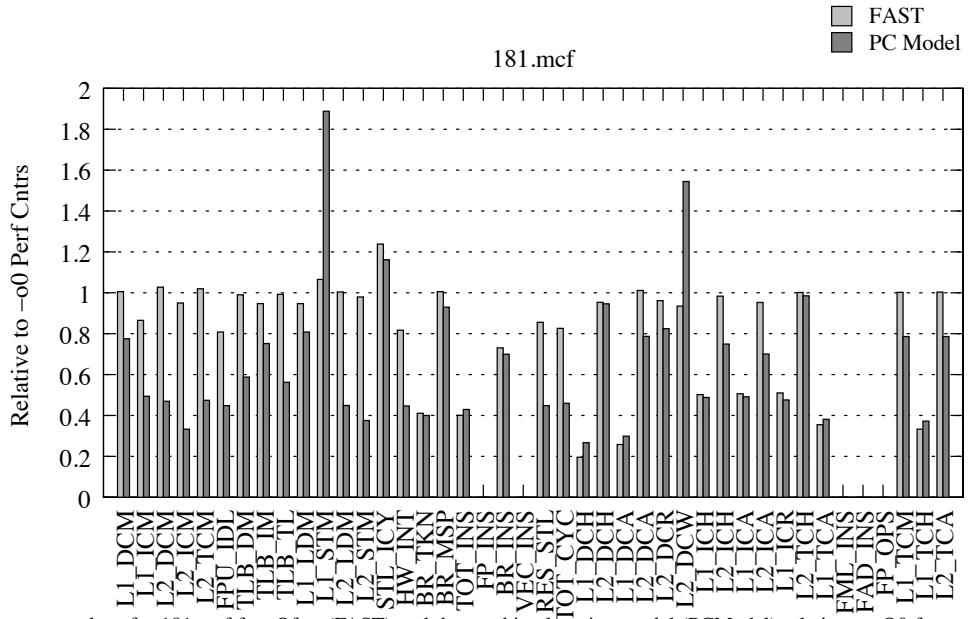

Fig. 4. Performance counters values for 181.mcf for -Ofast (FAST) and the machine learning model (PCModel) relative to -O0 for each performance counter.

optimization setting available with PathScale, and the optimization setting found by the performance counter model (*PCModel*). PCModel applies optimizations that are able to significantly improve the use of the L1 and L2 cache. This is shown in the rightmost three bars of Figure 4 in the columns labelled L1_TCM (L1 total cache miss), L1_TCA (L1 total cache accesses), and L2_TCA (L2 total cache accesses). For instance, the model is able to reduce the number of L1 cache misses by 20%, which has the effect of reducing the number of L2 accesses by 20%. -Ofast, on the other hand, has no effect on these values. In fact, -Ofast is able to achieve a 1.24 speedup over -O0, while PCModel gives a speedup of 2.33, i.e., a speedup of 1.88 over -Ofast.

If the optimizations selected by PCModel are examined, it becomes evident that the model has learned that data cache misses and branch instructions (via the performance counter data) are the critical characteristics of this program. Moreover, it suggested that the compiler convert pointers from 64-bit to 32-bit, because 64-bit pointers are reducing the effective cache capacity and memory bandwidth. This demonstrates the strength of automatic model construction. It has no prior human bias about what are important program characteristics or optimizations – it learns solely based on empirical evidence.

C. Performance Prediction Models

Performance prediction models in an intelligent compiler can be constructed by searching for the best set of optimizations for a program and learning from this search. When a new program is encountered, the compiler can automatically select a good sequence of optimizations without the need for many online trials. This is based on machine learning methods

that take as input a characterization of a problem (e.g., an application being optimized and an architecture being targeted) and predict the best set of optimizations to use. During a significant training period, predictive modelling techniques integrated in the compiler learn how programs, optimizations, and architectures interact and thus form the core of the intelligent compiler.

The output of previous runs of pure search can be used to train the performance prediction models. These models will allow the compiler to be portable and to tune its optimizations automatically to any new program or processor configuration. We are exploring a variety of different machine learning methods to induce these models automatically (described in Section III-F).

D. Dynamic Optimization and Runtime Monitoring Techniques

Traditional compilers typically output one optimized version of the program executable in the hope that this "one-size-fits-all" version of the code will run efficiently in the face of changing runtime contexts, application phases, and dynamic architecture environments. However, we realize that, most likely, there is not one optimized version of the code that is best for all environments.

Thus, we propose that intelligent compilers will integrate into each binary a dynamic compiler along with a runtime monitoring component that drives dynamic compilation and generates a characterization of the program at running time. This dynamic compilation environment will be able to perform specific optimizations at runtime that are predicted to be beneficial, but that cannot be performed due to lack of information at static compile time (e.g., function parameters or loop bounds). The runtime monitoring component will also provide an accurate dynamic characterization of the application and architecture. This information will be stored in the knowledge base to be used by the prediction models to drive future compilation.

E. Knowledge Base

An important issue in an intelligent compiler is the collection and proper storage of information from different sources into a knowledge base. Thus, it is important to build a standardized database to store learning data in order to facilitate the communication between machine learning components, optimization algorithms, compiler and instrumentation tools, compiler writers, as well as application developers.

A knowledge base can consist of data from previous optimization experiments, e.g., static and dynamic characterizations of code previously optimized as well as architectures previously targeted. The database should be populated with the results of optimization experiments and with extensive architecture characterization experiments performed using micro-benchmarks.

We propose that standard formats be developed for representing this information, which will enable the communication between the different tools in an intelligent compiler. A standard format for this information will serve two additional purposes. First, compiler writers will be able to collect and use information from outside sources if data is generated by using the same standard format, ultimately allowing the database of training data to quickly grow. Second, by documenting this standard format, compiler and application developers will be able to exploit the database for their own purposes, e.g., in developing new optimization algorithms or as feedback to developers.

An important part in developing this schema includes defining the features to be investigated. After an initial set of features is defined, compiler writers can create the necessary mechanisms to extract static program features during compilation and dynamic program features at runtime. Compiler writers should modify their frameworks to extract all desired static program features during compiler analysis (e.g., average size of basic block, whether a function is a leaf/non-leaf) and must instrument the executable to produce dynamic program features (e.g., performance counter information). Defining good features is an iterative process; therefore work should proceed in parallel with the development of feature extraction tools and the evaluation of machine learning methodologies. Standard statistical techniques, such as mutual information, can be useful to evaluate the usefulness of different features.

F. Machine Learning Methods

There is a large breadth of different learning techniques that can be applied to compiler problems, ranging from simple techniques, such as logistic regression and nearest neighbor classification, to advanced probabilistic approaches, such as Bayesian and reinforcement learning approaches. Simple learning algorithms are beneficial to compiler writers who will develop and integrate these algorithms into their compiler infrastructures. However, simple techniques may not solve all learning problems encountered, and thus compilers writers will have to judiciously use more advanced techniques where needed.

Application and architecture characterization and optimization experiments can generate enormous amounts of learning data; therefore we contend that scalable learning algorithms are important for solving compilation problems.

G. Multicore Compiler Optimizations

Multicore processors bring new performance optimization challenges, and machine learning techniques must be adapted to address these. From a parallelism perspective, the granularity of parallelism, the number of cores to use for a particular computation, and the computation partitioning across cores all impact performance significantly. Appropriate predictive modelling techniques to handle these problems will be important.

Multicore-based architectures will also require tuning parallelism, the number of threads per core, locality optimizations, thread affinity and scheduling, as well as data placement and movement. All of these will be interesting problems to tackle with machine learning.

IV. Related Work

A complete review of the literature is impractical for this paper. Instead, existing work is highlighted that is most closely related.

Autotuning

An area that is closely related to intelligent compilers is the study of automatic code generation and optimization for different computer architectures, which has been explored in many existing studies touching many different applications. There is a number of automatic library generators that automatically generate high-performance kernel routines, including FFT [4], [5], [6], BLAS [7], [8], [9], [10], [11], Sparse Numerical Computation [12], [13], [14], [15], [16], and domain specific routines [17], [18], [19]. Recent research efforts expand automatic code generation to routines whose performance depends not only on architectural features, but also on input characteristics [20], [21], [22]. These systems are a significant step toward automatically optimizing code for different computer architectures. However, these works do not explore the benefit of learning from a knowledge base of previously explored applications and architectures or of adapting to a changing runtime context.

Machine learning applied to Compilation

Machine learning and search techniques applied to compilation has been studied in many recent projects [23], [24], [25], [26], [27], [28], [29], [8], [30], [31]. These previous studies have developed machine learning based algorithms to efficiently search for the optimal selection of optimizing transformations, the best values for the transformation parameters, or the optimal sequences of compiler optimizations. Generally, these studies customize optimizations for each program or local code segments, some based on code characteristics. Intelligent compilers will not only use program characteristics, but will use architecture features to adapt to new computing systems, and will be aware of different runtime contexts of a program.

Several researchers have looked at searching for the best set or sequence of optimizations for a particular program [32], [31], [26], [27]. Cooper *et al.* [33] used genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding "good" compiler optimization sequences that reduced code size. Their technique was successful at reducing code size by as much as 40%. Unfortunately, their technique was application-specific, i.e., a genetic algorithm had to be retrained to find the best optimization sequence for each new program. Kulkarni *et al.* [34] exhaustively enumerated all distinct function instances for a set of programs that would be produced from different phase-orderings of 15 optimizations. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between different optimization passes in general and not specific to any program. In contrast, intelligent compilers will characterize programs being optimized; therefore these compilers will learn which optimizations are beneficial to apply to "unseen" programs with similar characteristics.

Many researchers have also looked at using machine learning to construct heuristics that control single compiler optimizations. Stephenson *et al.* [23] used genetic programming to tune heuristic priority functions for three compiler optimizations within the Trimaran's IMPACT compiler. For one of the optimizations, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic. Monsifrot *et al.* [30] used a classifier based on decision tree learning to determine which loops to unroll. They showed an improvement of 3% over the hand-tuned heuristic and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively. The results in these papers highlight the diminishing results obtained when only controlling a single optimization. In contrast, we believe that intelligent compilers must control all optimizations available to it.

Recently, researchers in Europe (as part of the MILEPOST project) have modified GCC to allow phase reordering of optimizations and have integrated machine learning techniques to control these optimizations [35]. They show good results on three different architectures.

Dynamic Compilation Environments

Fursin *et al.* [36] investigate online iterative search of optimizations for scientific applications. Prior to the program's execution, a set of optimized versions of code segments are created to be explored during execution. Their system uses a simple phase detection to identify periods of stable, repeatable behavior. During these stable phases, each optimized version is run and timed once, and the best performing version is chosen. Another solution is to evaluate the different statically compiled versions at runtime. For example, Lau *et al.* [37] present an online framework, called *performance auditing*, that allows the evaluation of the effectiveness of optimization decisions. The framework allows for online empirical optimization, which improves the ability of a dynamic compiler to increase the performance of optimizations while preventing performance degradations. Instead of using models to predict an optimization's performance, both of these approaches compile different versions of the same method with different optimization settings chosen randomly and then they run each of these different versions evaluating their performance empirically on the real machine. These approaches worked well for single optimizations or for a very small set of optimization sequences to be tested, but will be impractical for finding good sequences of optimizations from a large set of optimizations. Also, predictive modelling is not used to decide the optimized versions of the code to try.

Furthermore, dynamic compilation has been one of the central research topics in virtual machines. The key question that is explored is how to find the hot spot, that is, the code segment most frequently executed, and when to re-compile hot spots with more aggressive optimizations [38], [39], [40], [41], [42]. However, the changing runtime environment is generally not considered in the dynamic compiler's decision

of which code segment to optimize and which optimizations to apply. In general, previous work is inadequate in developing a systematic solution for optimizing programs to different runtime contexts, but will provide a basis to build on.

Dynamic systems provide a solution to directly modify the behavior of the program at runtime, and there have been several projects in the past pertaining to lightweight dynamic optimization schemes [38], [43], [44], [45], [46]. However, these systems have traditionally been very architecture and/or optimization specific. For example, Adore [44] is an Itanium-based system that only implements a prefetch optimization scheme. Adore is a very specialized dynamic optimization scheme that would be difficult to generalize for other architectures. Also, Dynamo [38] and Deli [47] have high overheads that can outweigh the benefit of their optimizations. For example, the system proposed by Zhao *et al.* [46] is built using DynamoRIO [48] and has a 14% overhead, 13% of which is due to DynamoRIO. Furthermore, these systems are not linked closely to the static compiler and thus do not use static information at runtime nor do they feed information back to the static compiler.

V. Conclusion

A critical component in intelligent compilers is the machine learning algorithms. Machine learning algorithms induce heuristic functions automatically from training data. In previous work, we found that machine learning was excellent at solving the problems we tackled [49], [50], [51], [52], [53], [54], [1], [55], [56]. In this work, we induced heuristics automatically using machine learning whose performance was comparable to hand-tuned heuristics on well-studied problems. We also have found that a variety of learning algorithms all had low classification error rates and thus performed equally well. We conclude that typical compiler problems do not present difficult learning problems, if proper features are extracted and the problem is correctly phrased.

Acknowledgements: We thank Xiaoming Li and Guang Gao for fruitful discussions of earlier versions of Figure 1.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. 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, 2006, pp. 295–305.

[2] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2005, pp. 181–192.

[3] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 185–197.

[4] M. Frigo, "A Fast Fourier Transform Compiler," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1999, pp. 169–180.

[5] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 232–275, February 2005.

[6] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua, "SPL: A Language and a Compiler for DSP Algorithms," in *Proc. of the International Conference on Programming Language Design and Implementation*, 2001, pp. 298–308.

[7] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[8] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-driven Optimization," in *Proc. of Programing Language Design and Implementation*, June 2003, pp. 63–76.

[9] C. Chen, J. Chame, and M. Hall, "Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 111–122.

[10] K. Goto and R. A. van de Geijn, "On Reducing TLB Misses in Matrix Multiplication," in *Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note 9.*, 2002.

[11] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and Bandwidth Aware Matrix Multiplication on the GPU," University of Illinois, Tech. Rep. UIUCDCS-R-2003-2328, Apr. 2003, at ftp.cs.uiuc.edu in /pub/dept/tech_reports/2003/ as UIUCDCS-R-2003-2328.ps.gz.

[12] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.

[13] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels," *Journal of Physics Conference Series*, vol. 16, pp. 521–530, Jan. 2005.

[14] J. Mellor-Crummey and J. Garvin, "Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 2, pp. 225–236, 2004.

[15] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Trans. on Graphics*, vol. 22, no. 3, July 2003, (Proc. SIGGRAPH 2003).

[16] X. S. Li and J. W. Demmel, "SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, 2003.

[17] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of High-performance Parallel Programs for a Class of ab initio Quantum Chemistry Models," *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 276–292, February 2005.

[18] R. Choy and A. Edelman, "Parallel MATLAB: Doing It Right," *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 331–341, February 2005.

[19] C. Lin and S. Z. Guyer, "Broadway: a Compiler for Exploiting the Domain-specific Semantics of Software Libraries," *In Proc. of the IEEE, special issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 342–357, February 2005.

[20] X. Li, M. J. Garzarán, and D. Padua, "A Dynamically Tuned Sorting Library," in *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 111–124.

[21] ——, "Optimizing Sorting with Genetic Algorithms," in *In Proc. of the International Symposium on Code Generation and Optimization (CGO)*, March 2005, pp. 99–110.

[22] S.-C. Han, F. Franchetti, and M. Püschel, "Program Generation for the All-pairs Shortest Path Problem," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM Press, 2006, pp. 222–232.

[23] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta Optimization: Improving Compiler Heuristics with Machine Learning," in *Proc. of Programing Language Design and Implementation*, June 2003.

[24] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast Searches for Effective Optimization Phase Sequences," in *PLDI*

'04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. New York, NY, USA: ACM Press, 2004, pp. 171–182.

[25] M. Stephenson and S. Amarasinghe, "Predicting Unroll Factors Using Supervised Classification," in *CGO '05: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–134.

[26] K. D. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, , and T. Waterman, "Searching for compilation sequences," Rice University, Tech. Report, 2005.

[27] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "ACME: Adaptive Compilation Made Efficient," in *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM Press, 2005, pp. 69–77.

[28] ——, "Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms," *J. Supercomputing*, vol. 36, no. 2, pp. 135–151, 2006.

[29] K. Yotov, K. Pingali, and P. Stodghill, "Think Globally, Search Locally," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 141–150.

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

[31] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin, "Probabilistic source-level optimisation of embedded programs," *SIGPLAN Not.*, vol. 40, no. 7, pp. 78–86, 2005.

[32] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 204–215.

[33] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Workshop on Languages, Compilers, and Tools for Embedded Systems*. Atlanta, Georgia: ACM Press, July 1999, pp. 1–9. [Online]. Available: citeseer.nj.nec.com/cooper99optimizing.html

[34] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Exhaustive optimization phase order space exploration," in *Fourth Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, New York City, NY, March 2006, pp. 306–318.

[35] 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.

[36] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, "A practical method for quickly evaluating program optimizations," in *Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, ser. LNCS, no. 3793. Springer Verlag, November 2005, pp. 29–46.

[37] J. Lau, M. Arnold, M. Hind, and B. Calder, "Online performance auditing: using hot optimizations without getting burned," *SIGPLAN Not.*, vol. 41, no. 6, pp. 239–251, 2006.

[38] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2000, pp. 1–12.

[39] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Dynamic Optimizing Compiler for Java," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*. New York, NY, USA: ACM Press, 1999, pp. 129–141.

[40] E. Duesterwald and V. Bala, "Software Profiling for Hot Path Prediction: Less is More," in *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2000, pp. 202–211.

[41] M. Arnold, M. Hind, and B. G. Ryder, "Online feedback-directed optimization of java," in *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 2002, pp. 111–129.

[42] T. Kistler and M. Franz, "Continuous Program Optimization: A Case Study," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[43] E. Duesterwald, "Design and engineering of a dynamic binary optimizer," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[44] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[45] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Chicago, IL, USA: ACM Press, 2005, pp. 190–200.

[46] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "Ubiquitous memory introspection," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. San Jose, California: IEEE Computer Society, March 2007, pp. 299–311.

[47] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher, "Deli: a new run-time control point," in *35th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2002, pp. 257–268.

[48] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004, supervisor-Saman Amarasinghe.

[49] J. Cavazos, "Automatically constructing compiler optimization heuristics using supervised learning," Ph.D. dissertation, University of Massachusetts Amherst, September 2004.

[50] J. Cavazos and J. E. B. Moss, "Inducing heuristics to decide whether to schedule," in *PLDI '04: Proceedings of the 2004 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2004, pp. 183–194.

[51] J. Cavazos and M. O'Boyle, "Automatic tuning of inlining heuristics," in *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2005, p. 13.

[52] J. Cavazos, J. E. B. Moss, and M. F. O'Boyle, "Hybrid optimizations: Which optimization algorithm to use?" in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, ser. LNCS. Vienna, Austria: Springer-Verlag, Mar. 2006, pp. 185–201.

[53] J. Cavazos and M. F. P. O'Boyle, "Method-specific Dynamic Compilation Using Logistic Regression," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 2006, pp. 229–240.

[54] 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, 2006, pp. 24–34.

[55] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using predictive modeling for cross-program design space exploration in multicore systems," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 327–338.

[56] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: part ii, multidimensional time," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 90–100.