Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization

Mircea Namolaru
IBM Haifa Research Lab
namolaru@il.ibm.com

Albert Cohen
INRIA Saclay and LRI, Paris-Sud 11 University
albert.cohen@inria.fr

Grigori Fursin
INRIA Saclay and University of Versailles
grigori.fursin@inria.fr

Ayal Zaks
IBM Haifa Research Lab
zaks@il.ibm.com

Ari Freund
IBM Haifa Research Lab
arief@il.ibm.com

ABSTRACT

Iterative search combined with machine learning is a promising approach to design optimizing compilers harnessing the complexity of modern computing systems. While traversing a program optimization space, we collect characteristic feature vectors of the program, and use them to discover correlations across programs, target architectures, data sets, and performance. Predictive models can be derived from such correlations, effectively hiding the time-consuming feedback-directed optimization process from the application programmer.

One key task of this approach, naturally assigned to compiler experts, is to design relevant features and implement scalable feature extractors, including statistical models that filter the most relevant information from millions of lines of code. This new task turns out to be a very challenging and tedious one from a compiler construction perspective. So far, only a limited set of ad-hoc, largely syntactical features have been devised. Yet machine learning is only able to discover correlations from information it is fed with: it is critical to select topical program features for a given optimization problem in order for this approach to succeed.

We propose a general method for systematically generating numerical features from a program. This method puts no restrictions on how to logically and algebraically aggregate semantical properties into numerical features. We illustrate our method on the difficult problem of selecting the best possible combination of 88 available optimizations in GCC. We achieve 74% of the potential speedup obtained through iterative compilation on a wide range of benchmarks and four different general-purpose and embedded architectures. Our work is particularly relevant to embedded system designers willing to quickly adapt the optimization heuristics of a mainstream compiler to their custom ISA, microarchitecture, benchmark suite and workload. Our method has been integrated with the publicly released MILEPOST GCC [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-903-9/10/10 ...$10.00.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers

General Terms

Performance, Languages, Algorithms

1. INTRODUCTION AND RELATED WORK

Sophisticated search techniques to optimize programs or improve default compiler heuristics have been proposed to cope with the complexity of modern computing systems [35, 30, 25, 8, 4, 32, 7, 20, 28, 16, 17, 12]. These techniques are already used in industry [10, 1, 18, 9], require little knowledge of underlying hardware and can adapt to new environments. However they are still very restrictive in practice due to an excessively large number of evaluations (recompilations and runs). Machine learning (ML) was introduced to make such search techniques practical and reduce optimization time by enabling optimization knowledge reuse [26, 31, 2, 6]. These studies rely on quantitative characterization of a program to build associations between similar programs and similar optimization spaces. Such a characterization is represented by a vector of floating point numbers, called *numerical features* (for example, the average basic block size may be one such numerical feature). These vectors provide the base for defining different optimizations heuristics, cost-models, and more. It is of critical importance for ML techniques to capture program similarities that effectively correspond to similarities in program optimizations.

Compiler experts have been responsible for identifying the quantitative program characteristics relevant for the problem being addressed. For some extensively investigated optimizations including unrolling, inlining, scheduling and register allocation, several static heuristics were designed based on numerical features. These heuristics involve analytical cost models to provide quantitative estimates of the effects of an optimization [27]. Beyond analytical models, empirical and feedback-directed approaches have also been proposed to guide optimization experts and to help compiler designers [29].

One of the first statistical ML techniques used successfully for solving several compiler optimizations problems is presented in [24]. The information required by the ML component is a vector of numerical features. We note that the optimizations addressed: unrolling, inlining, register allocation, scheduling, etc., all have well-known static heuristics from which these numerical features were drawn by a compiler expert [27].

Some optimization interferences are nearly impossible to predict by a compiler expert. Optimizing performance by tuning optimization flags may be somewhat accessible to an expert for well-understood application characteristics [36], but it quickly becomes intractable when dealing with the fine-tuning of more obscure optimization passes. Besides, this task is entirely dependent on the availability of quantitative features of the program, and on their relevance to the optimization problem. To address this challenge, we experimented with extensive sets of numerical features. This lead us to consider feature extraction as a general translation problem of a given program representation into numerical feature spaces. Unlike ordinary program properties maintained in compiler internals, numerical features must be comparable across different programs and target architectures. Cross-program and cross-target comparability is necessary for the correlations to be statistically representative, hence for ML predictions to be robust.

One important comparability requirement is that the size of the numerical feature vector be constant. As the number of variables, instructions, loops, basic blocks etc. in a program varies, the information about their properties therefore needs to be aggregated. This implies that we might provide inaccurate information for the machine learning component in some cases. To address this problem, we consider more sophisticated, semantically rich properties. For instance, such a property for a given loop may be if the loop is countable, consists of a single basic block, and contains no store instructions. The flexibility required for supporting such complex properties was achieved by an underlying generative mechanism that allows the derivation of complex properties from simpler ones.

A given representation of the program is translated into numerical features in two stages. First we translate the program representation into an intermediate form that contains the basic properties of the program. Then, the second stage performs the derivation of more complex properties, as well as the aggregation needed in order to finally extract the previously established number of numerical features.

The basic properties of a program appear in the compiler's internal representation at compilation time. These properties, extracted from the program in the first stage determine the possible features that can be derived in the second stage. We therefore designed the first stage to extract an exhaustive coverage of the compiler's global data structures representing the program being compiled.

In our approach, the compiler expert is responsible for choosing the basic properties to be extracted from the program and this way defining the space of possible features that can be derived from them. We will demonstrate how to use header files of the compiler to extract basic properties of the program. This approach can be automated, facilitating the complete automation of the feature extraction process.

In a machine learning compiler, numerical features are the quantitative links between the properties of the program and the predictive models that complement (or substitute) human-crafted heuristics. Identifying the factors that affect the performance of a given optimization is a time-consuming task. In addition a human can consider only a simplified model of the program, where many characteristics are ignored. Contrary to this, machine learning techniques are able to process huge amounts of data, and may work with a much more detailed and accurate model of a program.

We believe that compiler expertise is still required, but at a different level. For instance, the structure of the control-flow graph (CFG) may affect the output of a given optimization. But instead of requiring the compiler expert to point to the characteristics of the CFG that play a role in the decision, compiler expertise is employed to generate a space of *candidate features* that can be derived from the CFG; the machine learning component is responsible for deriving the most relevant characteristics. Without the machine learning component, the compiler expert would have to resort to a simpler predictive model, with a high probability of missing important correlations.

In our view, the problem of automatically generating numerical features consists of automatically inferring properties of the program and automatically aggregating these properties into features. These two sub-problems are reminiscent of automated theorem proving: starting from a set of basic properties, inference rules can be designed to infer all possible properties. This is precisely the approach we follow. We currently rely on a semi-automatic, logic programming approach to drive the inference towards a bounded set of features; a more futuristic direction would be to further automate the process, synthesizing new features on demand.

In our approach the program is view as a labeled graph, and *Datalog* [33] a first-order logic notion is used for representing this graph. This provides an alternative view of the program as a deductive (an extension of relational) database. The features are provided by evaluating Datalog (or Prolog) queries over this database.

To our best knowledge the only work taking a similar view and generating program features automatically from intermediate representation was introduced recently in [22, 23] - the program is represented by a XML database, and features are provided by evaluating Xquery expressions over this database. We note that only a single major compiler data structure, the IR (the intermediate representation) is processed. The IR used is basically a three-address representation - as a graph this is a tree with a fixed hierarchical structure. Our work addresses several major compiler data structures (beside the IR), represented as graphs with a more complex structures. In addition we provide techniques for translating the program information into a Datalog representation which is then used to generate the features.

It was already shown [34] that *Datalog* representation is suitable for even complex compiler analysis. Inferring new program properties (to be later aggregated into features) requires in fact performing compiler analysis, and the XML representation seems less appropriate for this. Furthermore, by viewing the program as a labeled graph represented by *Datalog* notation we could take advantage of related body of work done in graph (and multi-relational) data mining and ILP (inductive logic programming). We define the space of possible features - this space is huge and an exhaustive exploration is not possible. Similar with [21] we show how this space could be structured and its structure used for effective exploration.

Based on the techniques presented in this paper, we implemented a feature extractor for the GCC compiler, and applied supervised ML techniques for learning optimal settings of the flags. We evaluate our approach on several platforms using combinations of all available compiler optimizations, making it a practical and realistic approach.

Typical machine learning compilers [26, 31, 2, 6, 11] are composed of two main phases, as shown in Figure 1: a *training phase* and a *prediction phase*. In the *training phase* optimization tools gather information about the structure of a training set for different programs, architectures, data sets, etc. The tools extract program features, apply different combinations of optimizations to each program, profile and execute the resulting variants, and record the speedups. A predictive model is then built by correlating program features, optimizations and speedups. In the *prediction phase*, features of a new program are extracted and fed into the predictive model that suggests a "good" combination of optimizations, with the goal of reducing execution time or other optimization objectives such as code size and power consumption. Such techniques show

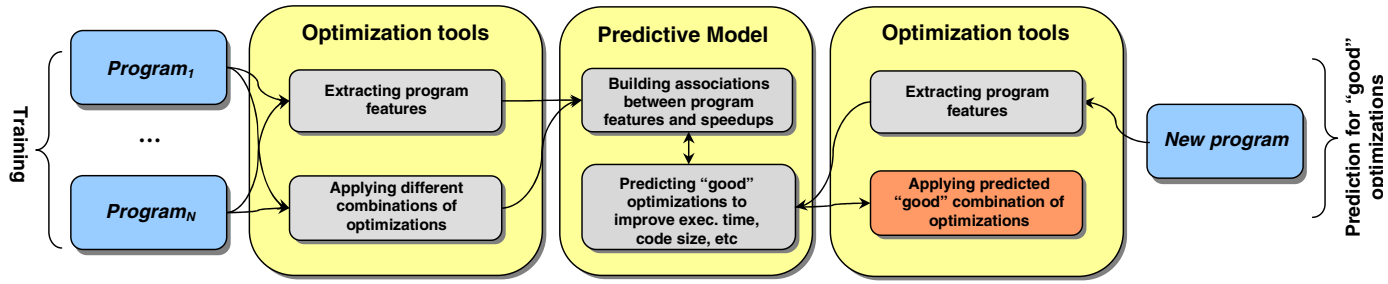

Figure 1: Typical machine learning scenario to predict "good" optimizations for programs. During a training phase (from left to right) a predictive model is built to correlate complex dependencies between program structure and candidate optimizations. In the prediction phase (from right to left), features of a new program are passed to the learned model and used to predict combinations of optimizations.

great potential but require large number of compilations and executions as training examples. Moreover, although program features are one of the key components of any machine learning approach, little attention has been devoted so far to ways of extracting them from program semantics.

2. FEATURE EXTRACTION

We may consider a program as being characterized by a number of entities. Some of these entities are a direct mapping of similar entities defined by the specific programming language, while others are generated during compilation. These entities include:

- functions;
- instructions and operands;
- variables;
- types;
- constants;
- basic blocks;
- loops;
- compiler-generated temporaries.

2.1 Relational View of a Program

A *relation* over one or more sets of entities is a subset of their Cartesian product. Relations can be used to express statements about tuples of entities, i.e., they define predicates. For example, we can define a relation $opcode = \{(i_k, op_l) \mid \text{instruction } i_k \text{ has opcode } op_l\} \subseteq I \times OPS$, where I is the set of program instructions and OPS is the set of all opcodes. Then, the statement $opcode(i_k, op_l)$ is the claim that instruction i_k has opcode op_l. This statement is true or false depending on the set of pairs constituting the relation $opcode$. As another example the relation $in \subseteq I \times B$, $in = \{(i_k, b_l) \mid \text{instruction } i_k \text{ is in basic block } b_l\}$, where B is the set of basic blocks, expresses the membership of instructions in basic blocks.

During compilation more complex relations among entities are computed, providing supplementary information about the program being compiled. Some of these relations, common to almost every optimizing compiler are:

- call graph;
- control flow graph (CFG);
- loop hierarchy;
- control dependence graph;
- dominator tree;
- data dependence graph;
- liveness information;
- availability information;
- anticipatibility information;
- alias information.

For example, the control flow graph can be viewed as a relation over pairs of basic blocks. New entities and relations relevant to specific optimizations of interest should be considered. For instance if information concerning register pressure is important, new entities and relation such as live range and interference graph, respectively, need to be considered.

Furthermore, the language in which the application is written also gives rise to entities and relations worth considering. As an example, a *class* entity and a class hierarchy graph (*CHG*) relation are relevant for programs written in object-oriented languages such as C++.

We prefer to focus on generic compilation entities and relations (such as the ones enumerated above) over entities and relations that are specific to certain compilers. The features we consider are thus defined in generic compilation terms, ensuring that our work is portable across different optimizing compilers.

We restrict our attention and extract only binary relations from the program. This is not restrictive, as every k-arity relation can be expressed by a set of $k+1$ binary relations. This assumption implies a graphical representation of the relations, a *labeled graph* (i.e. *semantic network*). The labels of the vertices are provided by the entities and the labels of the edges are provided by the relations. For a relation $r \subseteq E_1 \times E_2$, a fact $r(a,b)$ is represented by two nodes with labels E_1 and respective E_2 connected by an edge with label r.

In this program graph, important subgraphs correspond to major compiler data structures such as CFG, def-use chains, IR (the intermediate language) etc. In order to take advantage of their specific properties, we may consider each of these subgraphs separately.

In conclusion, a program may be represented as a collection of (binary) relations over sets of entities, i.e., as a relational database. Our first step is therefore to provide such a representation from the

compiler's data structures. We use the *Datalog* language [33] for this task, as we describe next.

2.2 Datalog

We use the *Datalog* logic-based notation to describe relations. Datalog is a Prolog-like language, but with more restricted semantics, suitable for expressing relations and operations on them [3],[33]. Datalog allows us to provide rules for defining and computing new relations from existing ones.

The elements of Datalog are atoms of the form $p(X_1,...,X_n)$ where p is a predicate and X_1,..., X_n are variables or constants. By convention names beginning with lower case letters are used for constants and predicates, while names beginning with upper case letters are used for variables. A *ground atom* is a predicate with only constants as arguments.

A Datalog database consists of a list of *rules*. Each Datalog rule has the form $H:-B_1,B_2,...B_n$, where $H,B_1,..,B_n$ are atoms. H is called the *head* of the rule, and $B_1,B_2,...,B_n$ form the *body* of the rule. The body of the rule is optional (i.e., $n \geq 0$). Bodyless rules are called *facts*, and can be used to define relations by explicit enumeration. For example, the two facts $x(1,2)$ and $x(3,5)$ define x as the relation $\{(1,2),(3,5)\}$. Rules with bodies serve to infer the head relation from the body relations; meaning that whenever we substitute constants for the variables in the atoms, and this substitution makes all the body predicates true, then the head predicate must also be true.

A Datalog *query* has the form $:-B_1,B_2,...B_n$, where $B_1,..,B_n$ are atoms. An answer to a given query is a set of constants such that when substituted for the variables in the atoms, all predicates of the query become true. A query may result in many answering substitutions.

To obtain a Datalog representation of the program, we enumerate the elements of every entity of interest: variables $V = v_1,v_2,...$, types $T = t_1,t_2,...$, instructions $I = i_1,i_2,i_3...$, basic blocks $B = b_1,b_2,b_3...$, etc. We then extract from the compiler's data structures relations over these entities. For example we specify the relation $in \subseteq I \times B$, $in = \{(i_k,b_l) \mid \text{instruction } i_k \text{ is in basic block } b_l\}$ by a sequence of Datalog ground atoms of the form $in(i_k,b_l)$.

Datalog is able to work with relations and perform operations on them whose results in turn are relations as well. All standard relational algebra operations [33] are expressible, the most useful (for our purposes) being the conjunction (join) of two relations. For instance starting with the relations *store* and *in*, Datalog can compute the relation $st_in \subseteq I \times B$ formed from all pairs (i,b) such that instruction i is a *store* instruction in basic block b. In Datalog this computation is triggered by the rule

$$st_in(I,B):-store(I),in(I,B).$$

2.3 Automatic Inference of New Relations

Given a set of basic relations (such as those listed in Section 2.1), further useful relations can be inferred, including very complex ones. For example, Whaley and Lam [34] were able to perform *interprocedural context sensitive alias analysis* using Datalog inference. Although, as a general rule it is impractical to infer very complex relations automatically, it is still useful to infer new relations easily with Datalog, albeit of limited complexity.

The main operation we use for relation inference is the joining of two relations: given two relations $r \subseteq E_1 \times \cdots \times E_k$ and $p \subseteq F_1 \times \cdots \times F_l$ such that some of the Es are identical to some of the Fs, we select a nonempty subset I of pairs of identical entities and essentially concatenate the two relations with the common entities (in I) appearing only once. The simplest way to explain this is through a Datalog example. Suppose the two relations are $r \subseteq E_1 \times E_2 \times E_3$ and $p \subseteq F_1 \times F_2 \times F_3$ such that $E_2 = F_1$ and $E_3 = F_2$. Then we can join the two relations in the following three ways.

```
rel1(E1, E2, E3, F2, F3) :-
  r(E1, E2, E3), p(E2, F2, F3).
rel2(E1, E2, E3, F1, F3) :-
  r(E1, E2, E3), p(F1, E3, F3).
rel3(E1, E2, E3, F3) :-
  r(E1, E2, E3), p(E2, E3, F3).
```

By repeated joining, starting from a set of basic relations, we can obtain new relations of increasing complexity. As the example shows, this is straightforward to automate. In a practical setting, though, the number of relations and their complexity must be kept to a limit. For example, we may limit the number of joinings that lead to a relation, the number of times any relation may appear in such a sequence, the arity of the resulting relation, and more.

2.4 Extracting Relations from Programs

During compilation a compiler maintains an internal representation of the program being compiled using several data structures. We use the definitions of these data structures to extract and identify basic entities and relations. The data types express the entities: in C such data types are typically of type `struct T`, having a number of fields[1]. Each such field may define a relation between the entity represented by the parent `struct` and the entity represented by the type of the field. For example, the data structure for an edge of a control-flow graph can be a `struct edge` containing two fields `src` and `trg` (among others) that are pointers to `struct basic_block`, as in the case of GCC. The data types `struct edge` and `struct basic_block` introduce two entities E and B, and the fields `src` and `trg` introduce two relations over $E \times B$: `edge_src` and `edge_trg`.

The above mechanical method provides compiler specific entities and relations, which we then map to generic entities and relations. This mapping may be straightforward as in the example above, or may require some additional processing and semantic understanding. For example, GCC uses `struct tree` to represent different generic entities such as variables and types, with a selector field in the `struct` identifying the intended semantics. Other fields of this data structure are overloaded, and their meaning depends on the entity the tree represents. For example, one of the fields in a `struct tree` that represents a variable contains a pointer to another `struct tree` that represents the variable's type. Knowing this allows us to deduce a relation on (*variable*,*variable type*) pairs.

2.5 Extracting Features from Relations

A machine learning tool requires a quantitative measurement of the program, provided by a vector of numerical features. In this section we present several techniques for deriving numerical features from a relational representation of the program.

We consider first the case of entities having numerical values. These values may need to be aggregated into their sum, average, variance, max, min, etc., and in this way produce numerical features for the relation. For example, given relation

$$count = \{(b,n) \mid b \text{ is a basic block whose estimated number of executions is } n\},$$

we may want to compute numerical features such as the maximal number of estimated executions of a basic block, or the average number of estimated executions of a basic block.

[1]We focus on C because our work is implemented in the context of GCC, which is written in C.

We focus now on the case of entities having categorical values (i.e., symbols). Most of the entities important for the compilation process belong to this class. Typically, numerical features describing relations over such entities provide information on basic structural aspects of the relation such as the number of tuples in the relation, the maximum out-degree of nodes in a tree relation, etc. We show how to extract several typical types of numerical features by applying the standard selection and projection operations, together with the *num* operator, defined as returning the number of tuples in a relation.

First we note that applying *num* to a relation already provides a numerical feature which is often of interest. This is particularly so in the case of unary relations (e.g., number of basic blocks) but may also be the case for higher arity relations (e.g., number of edges in the control flow graph). Also, applying *num* to the projection of relation r on dimension i—yielding the unary relation $r_i = \{e \mid \exists t \in r$ such that t has e at position $i\}$—often provides an interesting numerical feature. For example, consider the relation

$$st_in_block = \{(i,b) \mid i \text{ is a store instruction in basic block } b\}.$$

Then $num(st_in_block_1)$ is the number of stores in all basic blocks, while $num(st_in_block_2)$ is the number of basic blocks containing *store* instructions.

We consider now the case of a binary relation $r \subseteq E_1 \times E_2$. For every element $e \in E_i$, $1 \leq i \leq 2$, we consider the selection induced by this element, i.e., the relation $r^i(e)$ defined as the set of pairs in r that contain e at position i. By associating with e the value of $num(r^i(e))$ we define a new relation in $E_i \times \mathbb{N}$. For this relation, numerical features can be derived by aggregating the numerical values in the second position.

For example, consider again the relation *st_in_block*. For a given basic block b, the value $num(st_in_block^2(b))$ is the number of *store* instructions in basic block b. Thus the relation consisting of all pairs $(b, num(st_in_block^2(b)))$ associates each block with the number of *store* instructions it contains. By aggregating these counts we may obtain numerical features such as the average number of stores in a basic block.

For the general case of a k-arity relation r where $k > 2$, we may derive a number of binary relations by considering the projection of r on any two dimensions i, j, $i \neq j$. For each such binary relation we derive new features by the above technique. Furthermore, for a relation $r \subseteq E_1 \times ...E_k$ we can also consider any two disjunct subsets I and J of the index set $\{1, \ldots k\}$. The projection of r on the dimensions in I and J may be seen as a binary relation over the sets $S_1 = E_{i_1} \times \cdots \times E_{i_p}$ and $S_2 = E_{j_1} \times \cdots \times E_{j_q}$, where $I = \{i_1, \ldots, i_p\}$ and $J = \{j_1, \ldots, j_q\}$. Again, for this binary relation new numerical features may be derived.

The techniques described above for derivations of numerical features from relations can be automated. We implemented the extraction of numerical features from the Datalog-derived representation of the program in Prolog, as the required aggregation operations are not supported in Datalog.

2.6 Structural Code Patterns

In the previous section we examined some basic structural properties of a graph as number of edges, average number of neighbors for a vertex etc. These properties represent poorly the graph structure for labeled graphs with a small number of labels for vertices and edges (e.g., CFG, DDG, dominator tree, etc.). We try to characterize such graphs by a number of (subgraph) patterns - the numerical features are provided by the number of occurrences of such patterns in the graph.

For instance, the control flow graph (CFG) may be considered as a relation over $B \times B$, where B is the set of basic blocks. New relations over $B \times B$ may be induced from this relation by taking into account the way in which two basic blocks are connected. For example, we may consider blocks connected via an `if-then` or an `if-then-else` pattern in CFG. The following Datalog rules provide possible definitions for these two relations. (In this example the relation *bb_edge* specifies whether two basic blocks are connected by an edge in the CFG.)

```
bb_ifthen(B1,B3) :-
  bb_edge(B1,B3), bb_edge(B1,B2), bb_edge(B2,B3).

bb_ifthen_else(B1,B4) :-
  bb_edge(B1,B2), bb_edge(B1,B3),
  bb_edge(B2,B4), bb_edge(B3,B4).
```

These new relations may in turn induce new relations over basic blocks connected via nested `if-then` or `if-then-else` patterns. The following Datalog rule provides a possible definitions for a relation having as elements pairs of basic blocks connected via a direct edge and a nested `if-then` pattern (an `if-then` pattern in which the `then` alternative is itself an `if-then` pattern).

```
bb_ifthen_n(B1,B4) :-
  bb_edge(B1,B4), bb_edge(B1,B2),
  bb_ifthen(B2,B3), bb_edge(B3,B4).
```

In a similar way we may derive relations describing patterns in any graph structure computed during compilation. These patterns can be described easily by Datalog rules. The semantics of the graph structure being analyzed provide guidance in selecting the patterns to consider. Additional knowledge about the code may help further trim the pattern space. For instance, knowing that for C programs without `switch` statements every node has at most two successors in the CFG could limit the number of possible patterns we look for.

Other patterns in graphs such as cycles may be considered as well. For the CFG, the loop structure may be extracted either from relevant data structures of the compiler if available, or by computing simple patterns directly from the CFG, such as single basic block loops or innermost loops with a simple structure (e.g., containing a single `if-then` pattern inside the loop body).

Finally we note that every binary relation $r \subseteq E \times F$ can be viewed as a bipartite graph in which the partite sets correspond to E and F. For example, the *def-use* relation over operand pairs induces a bipartite graph in which one of the partite sets consists of the *def*s and the other consists of the *use*s. This allows us to apply the techniques presented in this section to any binary relation. For instance, let r denote the *def-use* relation. then the *web* relation below defines a *web* pattern in the bipartite graph corresponding to the *def-use* relation.

```
web(E1,E2,F1,F2) :-
  r(E1,F1), r(E2,F1), r(E1,F2), r(E2,F2).
```

As can be seen, a large number of structural patterns can be easily expressed and tested using our feature extraction framework. Techniques for exploring the space of structural patterns are further discussed in the next subsection.

2.7 Exploring the Structural Pattern Space

In our framework, Datalog queries are used to represent subgraph patterns. For a query q, its $frequency(q,r)$ is defined as the number of substitutions for which the query is true with respect to

a Datalog database r. The *frequency* provides a metric for a pattern that maps the pattern to a feature. Given a set of patterns, the features vector is provided by their frequencies. Thus, the features space is determined by Datalog patterns (i.e. queries) space.

We use a pattern growth approach, in which more complex patterns are successively derived from a set of initial patterns. We refine the scheme of inference of new relations presented previously by imposing constraints (chosen by a compiler expert) on the variables. In this way only potentially important patterns are generated, significantly reducing the space of patterns to be considered.

We exemplify our extension techniques for the case of the CFG, represented by the relation $bb_edge \subseteq B \times B$. The possible queries are sequences of *bb_edge* predicates of arbitrary length

$$:-bb_edge(X_1,X_2), bb_edge(X_1,X_3), bb_edge(X_3,X_4), ...$$

For each variable $X_1, X_2, \ldots$ in the sequence, some constraints control the sharing of variables between the *bb_edge* predicates. The constraints are of the form (m,n) where m is the maximal number of occurrences of the variable as the first argument, and n is the maximal number of occurrences of the variable as the second argument. Intuitively these constraints limit the number of predecessors and successors for the vertices substituted to the variable and are chosen on basis of domain expert knowledge - for CFG the constraints chosen are $(1,2),(2,1),(1,1),(2,2)$.

A query is extended by adding at each step a *bb_edge* predicate. If a new variable is introduced, the possible four constraints mentioned above should be attached to it - in fact there are four new resulting relations. If no variable is introduced, the addition of the new added predicate (that uses two existent variables) should conform with the constraints imposed on the variables. As an example we consider the query below, the constrains associated with the variables, and a possible legal extension:

```
constraint(B1) = (2,2)
constrains(B2) = (1,1)
constrains(B3) = (2,1)

Before extension
:- bb_edge(B1,B2), bb_edge(B1,B3).

After extension
:- bb_edge(B1,B2), bb_edge(B1,B3), bb_edge(B2,B3).
```

We note that after the extension variable $B2$ could not be further shared with any newly added predicate as this would violate its constraints. Similarly $B1$ and $B3$ could not appear as the first and second arguments of a newly added predicate, respectively.

We note that our techniques could be extended to any labeled graph. As mentioned before the compiler expert should define the constraints to be used based on the specific properties of the graph.

The pattern growth approach previously described, introduces a partial order $\prec$ over the set of Datalog queries, where $q_1 \prec q_2$ means that query q_2 is an extension of query q_1. The pattern space is the lattice spanned by the partial order $\prec$; the inference of the patterns may be seen as a search problem in this space.

For a collection S of Datalog databases, we define the support of a query q with respect to S as

$$support(q,S) = |\{r \in S \,|\, frequency(q,r) > 0\}|$$

— intuitively the number of databases r where the pattern g occurs. A pattern q is called *frequent* [21] if $support(q,S)$ is greater or equal to a threshold specified by the user.

The specialization operator is anti-monotonic w.r.t. to the *support* relation for a set S of Datalog databases, i.e. if $q_1 \prec q_2$ then $support(q_1,S) \geq support(q_2,S)$. The anti-monotonicity property allows us to effectively prune the extension of a query — if a query is not frequent, then none of its extensions are frequent.

3. METHODOLOGY

In our work we attempt to overcome two major methodological flaws that limit the dissemination of current and past research on iterative compilation and machine learning compilation, namely:

- the use of proprietary, unreleased or outdated transformation, compilation and feature extraction tools;
- the very limited set of optimizations and features, making it difficult or even impossible to replicate and improve upon previous results.

In an attempt to curb these tendencies, we decided to implement our feature extractor inside the popular, free software, production-quality GCC compiler [13]. Recent versions of GCC achieve performance levels competitive with the best commercial and research compilers. GCC also supports a large number of platforms, a large and fast-growing number of optimizations, and modern intermediate representations facilitating the extraction of semantically rich properties and features. It is a unique tool available for research purposes in compilation of real-world applications.

Based on the techniques described in the previous section, we implemented our feature extractor as additional passes in GCC 4.2-4.4 versions [14]. It is invoked on demand after the compiler generates the data needed for producing features. The feature extractor works in two stages:

- extracting a relational representation of the program;
- computing a feature vector based on this representation.

4. EVALUATION

The technique presented in this paper shows how to automate and generalize feature extraction for use in predicting good optimizations. To make its benefits more concrete, we propose a complete and realistic scenario about how a compiler expert may incrementally enhance a machine learning compiler. We assume the compiler expert is working in an embedded system design group, targeting an ARC 725D 700MHz embedded processor – *ARC*. As is common in such a context, the design group is very small and does not have the resources to tune the heuristics, optimization pass selection and compilation flags for this particular platform.

We use the popular, freely-available MiBench [15] benchmark suite that comes with a variety of embedded and general-purpose desktop applications.

1. The expert first constructs a search space where significant speedups can be obtained using traditional iterative compilation.
2. She uses this space to build a machine learning model.
3. She trains this model over multiple (desktop and server) platforms: *AMD* – Athlon 64 3700+, *IA32* – Intel Xeon 2.8GHz, *IA64* – Itanium2 1.3GHz.
4. The expert aims to use this knowledge base to predict how to select the best optimizations, when running the same benchmarks but on the embedded *ARC* target.

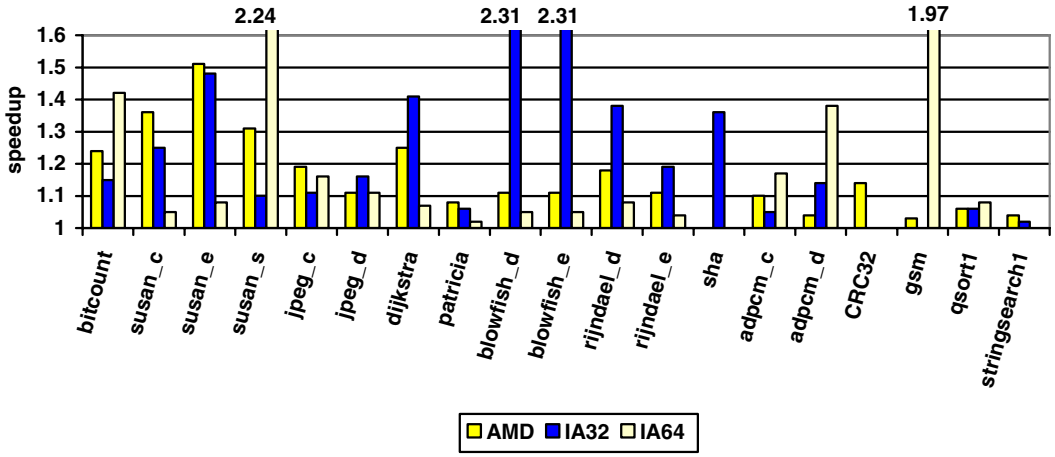

Figure 2: Speedups obtained using iterative search on 3 platforms (500 random combinations of optimizations with 50% probability to select each optimization)

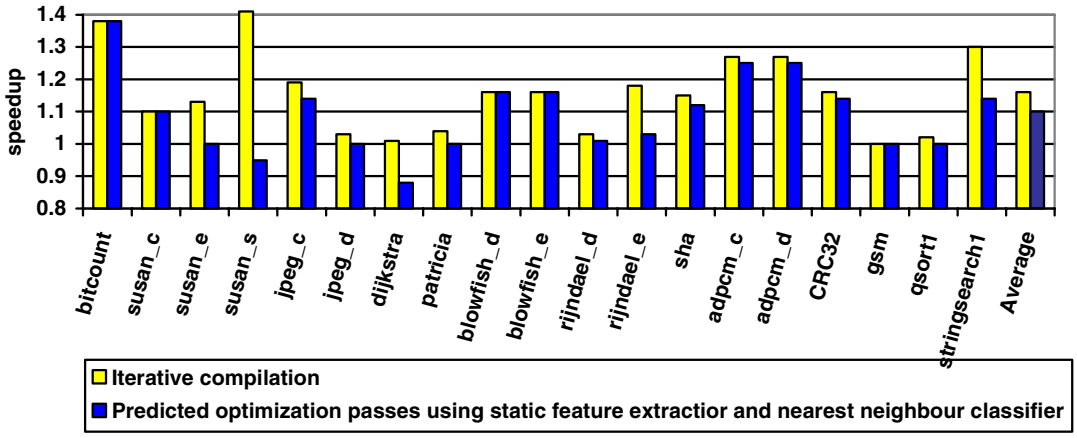

Figure 3: Speedups when predicting best optimizations based on program features in comparison with the achievable speedups after iterative compilation based on 500 runs per benchmark (ARC processor)

5. In the process, her first experiments are disappointing: the predictions achieved by the model only reach a fraction of the performance of the best combination of optimizations available in the search space.

6. The expert identifies the source of the problem using standard statistical metrics [19]. It may come from a model overfit due to a limited number of features, or to lack of effective correlations between these features and the semantical properties that actually impact performance on the *ARC* platform.

7. The expert designs and implements new program feature extractors, *leveraging* her understanding of the optimization process and of the performance anomalies involved.

8. She incrementally adds these features into the training set, until the predictive model shows relevant results.

9. To finalize the tuning, and improve compilation and training time, she performs principal component analysis (PCA) to narrow down the set of features that really make an impact on her platform of interest.

As outlined in the use case scenario, the training of the machine learning model has been performed on all benchmarks and all platforms, except ARC which we used as a test platform for optimization predictions.

To illustrate this scenario in practice, we applied 500 random combinations of 88 compiler optimizations that are known to influence performance, with 50% probability of being selected, and run each program variant 5 times. To make the adaptive optimization fully transparent, we directly invoke optimization passes inside a modified GCC pass manager. Figure 2 shows speedups over the best GCC optimization level *-O3* for all programs and all architectures. It confirms the previous findings about iterative compilation [10, 1, 28, 17] — that it is possible to considerably improve performance over default compiler settings, which are tuned to perform well on average across all programs and platforms. In order to help end-users and researchers reproduce results and optimize their programs, we made experimental data publicly available in the Collective Optimization Database at [9]. Note that the same combination of optimizations found for one benchmark, for example, *susan_corners* on *AMD*, does not improve execution time of

Feature #	Description:
ft1	Number of basic blocks in the method
ft2	Number of basic blocks with a single successor
ft3	Number of basic blocks with two successors
ft4	Number of basic blocks with more than two successors
ft5	Number of basic blocks with a single predecessor
ft6	Number of basic blocks with two predecessors
ft7	Number of basic blocks with more than two predecessors
ft8	Number of basic blocks with a single predecessor and a single successor
ft9	Number of basic blocks with a single predecessor and two successors
ft10	Number of basic blocks with a two predecessors and one successor
ft11	Number of basic blocks with two successors and two predecessors
ft12	Number of basic blocks with more than two successors and more than two predecessors
ft13	Number of basic blocks with number of instructions less than 15
ft14	Number of basic blocks with number of instructions in the interval [15, 500]
ft15	Number of basic blocks with number of instructions greater than 500
ft16	Number of edges in the control flow graph
ft17	Number of critical edges in the control flow graph
ft18	Number of abnormal edges in the control flow graph
ft19	Number of direct calls in the method
ft20	Number of conditional branches in the method
ft21	Number of assignment instructions in the method
ft22	Number of unconditional branches in the method
ft23	Number of binary integer operations in the method
ft24	Number of binary floating point operations in the method
ft25	Number of instructions in the method
ft26	Average of number of instructions in basic blocks
ft27	Average of number of phi-nodes at the beginning of a basic block
ft28	Average of arguments for a phi-node
ft29	Number of basic blocks with no phi nodes
ft30	Number of basic blocks with phi nodes in the interval [0, 3]
ft31	Number of basic blocks with more than 3 phi nodes
ft32	Number of basic block where total number of arguments for all phi-nodes is in greater than 5
ft33	Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
ft34	Number of switch instructions in the method
ft35	Number of unary operations in the method
ft36	Number of instruction that do pointer arithmetic in the method
ft37	Number of indirect references via pointers ("*" in C)
ft38	Number of times the address of a variables is taken ("&" in C)
ft39	Number of times the address of a function is taken ("&" in C)
ft40	Number of indirect calls (i.e. done via pointers) in the method
ft41	Number of assignment instructions with the left operand an integer constant in the method
ft42	Number of binary operations with one of the operands an integer constant in the method
ft43	Number of calls with pointers as arguments
ft44	Number of calls with the number of arguments is greater than 4
ft45	Number of calls that return a pointer
ft46	Number of calls that return an integer
ft47	Number of occurrences of integer constant zero
ft48	Number of occurrences of 32-bit integer constants
ft49	Number of occurrences of integer constant one
ft50	Number of occurrences of 64-bit integer constants
ft51	Number of references of local variables in the method
ft52	Number of references (def/use) of static/extern variables in the method
ft53	Number of local variables referred in the method
ft54	Number of static/extern variables referred in the method
ft55	Number of local variables that are pointers in the method
ft56	Number of static/extern variables that are pointers in the method

Table 1: List of program features produced using our technique to be able to predict good optimizations

the *bitcount* benchmark, and even degrades the execution time of *jpeg_c* by 10% on the same architecture. It is of course a clear signal that program features are key to the success of any machine learning compiler. This of course does not diminish the importance of architecture features and data-set features.

Though obtaining strong speedups, the iterative compilation process is very time-consuming and impractical in production. We use predictive modeling techniques similar to [26, 31, 2, 6] to be able to characterize similarities between programs and optimizations, and to predict good optimizations for a yet unseen program based on this knowledge. To validate our results, we decided to use a state-of-the-art predictive model [2]. This model predicts optimizations for a given program based on a nearest-neighbor static feature classifier, suggesting optimizations according to the similarity of programs. We use a different training set on the embedded system platform *ARC*, and the traditional *leave-one-out* validation where the evaluated benchmark is removed from the training set, to avoid strong biasing of the same optimizations from the same benchmark. When a new program is compiled, features are first extracted using our tool, then they are compared with all similar features of other programs using a nearest-neighbor classifier, as described in [5]. The program is recompiled again with the combi-

nation of optimizations for the most similar program encountered so far.

As outlined in the use case scenario, we iterated on this baseline method while gradually adding more and more features. We eventually reached 11% average performance improvements across all benchmarks, out of 15% when picking the optimal points in the search space (i.e., factors 1.11 and 1.15 in Figure 3. Adding more features did not bring us more performance on average across the benchmarks. The list of the 56 most important features identified in this iterative process that are able to capture complex dependencies between program structure and a combination of multiple optimizations is presented in Table 1. Though we did not reach the best performance achieved with iterative compilation, we showed that our technique for automatic feature extraction can already be used effectively for machine learning, to enable optimization knowledge reuse and automatically improve program execution time. The simplicity and expressiveness of the feature extractor is one key contribution of our approach: a few lines of Prolog code for each new feature, building on a finite set of pretty-printers from GCC's internal data structures into Datalog entities.

Our results pave the way for a more systematic study of the quality and importance of individual program features, a necessary step towards automatic feature selection and the construction of robust predictive models for compiler optimizations.

Our main contribution is to construct program features by aggregation and filtering of a large amount of semantical properties. But comparison with other predictive techniques is a relevant question in itself, related to the selection of the features and machine learning classifier or predictor. Our work is intended to ease such comparisons, transforming the work of others into a common machine learning optimization platform.

5. CONCLUSION

Though the combination of iterative compilation and machine learning has been studied for more than a decade and showed great potential for program optimizations, there are surprisingly few research results on the problem of selecting good quality program features. This problem is relevant for effective optimization knowledge reuse, to speedup the search for good optimizations, to build predictive models for compilation heuristics, to select optimization passes and ordering, to build and tune analytical performance models, and more.

Up to now, compiler experts had to manually construct and implement feature extractors that best suit their purpose. Without a systematic way to construct features and evaluate their merits, this task remains a tedious trial and error process relying on what the experts *believe* they understand about the impact of optimization passes. In a modern compiler like GCC, more than 200 passes compete in a dreadful interplay of tradeoffs and assumptions about the program and the target architecture (itself very complex and rather unpredictable). The global impact of these heuristics can be very far from optimal, even on a major compiler target such as the x86 ISA and its most popular microarchitectural instances. But what about embedded targets which attract less attention from expert developers and cannot afford large in-house compiler groups? What about design-space exploration of the ISA, microarchitecture and compiler?

So far, a limited set of largely syntactical features have been devised to prove that optimization knowledge can be reused and derived automatically from feedback-directed optimization. However, machine learning is only able to recover correlations (hence optimization knowledge) from the information it is fed with: it is critical to select topical program features for a given optimization problem. To our knowledge, this is the first attempt to propose a practical and general method for systematically generating numerical features from a program, and to implement it in a production compiler. This method does not put any restriction on how to logically and algebraically aggregate semantical properties into numerical features, offering a virtually exhaustive coverage of statistically relevant information that can be derived from a program.

This method has been implemented in GCC and applied to a number of general-purpose and embedded benchmarks. We illustrate our method on the difficult problem of selecting the optimal setting of compiler optimizations for improving the performance of an application, and demonstrate its practicality achieving 74% of the available speedup obtained through iterative compilation on a wide range of benchmarks and 4 different general-purpose and embedded architectures. We believe this work is an important step towards generalizing machine learning techniques to tackle the complexity of present and future computing systems. Feature extractor presented in this paper is now available for download within MILEPOST GCC at [14] while experimental data is available at [9] to help researchers reproduce and extend this work.

6. ACKNOWLEDGMENTS

This work was partly supported by the European Commission through the FP6 project MILEPOST id. 035307 and by the HiPEAC Network of Excellence.

7. REFERENCES

[1] ACOVEA: Using Natural Selection to Investigate Software Complexities. `http://www.coyotegulch.com/products/acovea`.

[2] 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 *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[3] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition, 2007.

[4] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.

[5] Edwin V. Bonilla, Christopher K. I. Williams, Felix V. Agakov, John Cavazos, John Thomson, and Michael F. P. O'Boyle. Predictive search distributions. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine learning*, pages 121–128, New York, NY, USA, 2006. ACM.

[6] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2007.

[7] K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[8] K.D. Cooper, P.J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[9] Collective Tuning Infrastructure: automating and accelerating development and optimization of computing systems. http://cTuning.org.

[10] ESTO: Expert System for Tuning Optimizations. http://www.haifa.ibm.com/projects/systems/cot/esto.

[11] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[12] Grigori Fursin and Olivier Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.

[13] GCC: GNU Compiler Collection. http://gcc.gnu.org.

[14] MILEPOST GCC: Collaborative development website. http://cTuning.org/milepost-gcc.

[15] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[16] K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.

[17] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2008.

[18] Shih-Hao Hung, Chia-Heng Tu, Huang-Sen Lin, and Chi-Meng Chen. An automatic compiler optimizations selection framework for embedded applications. In *Intl. Conf. on Embedded Software and Systems (ICESS'09)*, pages 381–387, 2009.

[19] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[20] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.

[21] L.Dehaspe and H.Toivonen. Discovery of frequent datalog patterns. In *Data Mining and Knowledge Discovery*, pages 7–36, 1999.

[22] H. Leather, E. Yom-Tov, M. Namolaru, and A. Freund. Automatic feature generation for setting compilers heuristics. In *2nd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'08), colocated with HiPEAC'08 conference*, 2008.

[23] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.

[24] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, Berlin, 1971.

[25] F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[26] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.

[27] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.

[29] David Parello, Olivier Temam, Albert Cohen, and Jean-Marie Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM/IEEE Conf. on Supercomputing (SC'04)*, page 15, Washington, DC, 2004.

[30] B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.

[31] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.

[32] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

[33] J. D. Ullman. *Principles of Database and Knowledge Systems*, volume 1. Computer Science Press, 1988.

[34] J. Whaley and M.S. Lam. Cloning based context sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[35] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.

[36] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *ACM Symp. on Principles & practice of parallel programming (PPoPP'90)*, pages 137–146, Seattle, Washington, United States, 1990.