*Reprinted from the*

# Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

# MILEPOST GCC: machine learning based research compiler

Grigori Fursin, Cupertino Miranda, Olivier Temam
*INRIA Saclay, France*

Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson
*IBM Haifa, Israel*

Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle
*University of Edinburgh, UK*

Phil Barnard, Elton Ashton
*ARC International, UK*

Eric Courtois, Francois Bodin
*CAPS Enterprise, France*

*Contact: grigori.fursin@inria.fr*

## Abstract

Tuning hardwired compiler optimizations for rapidly evolving hardware makes porting an optimizing compiler for each new platform extremely challenging. Our radical approach is to develop a modular, extensible, self-optimizing compiler that automatically learns the best optimization heuristics based on the behavior of the platform. In this paper we describe MILEPOST[1] GCC, a machine-learning-based compiler that automatically adjusts its optimization heuristics to improve the execution time, code size, or compilation time of specific programs on different architectures. Our preliminary experimental results show that it is possible to considerably reduce execution time of the MiBench benchmark suite on a range of platforms entirely automatically.

## 1 Introduction

Current architectures and compilers continue to evolve, bringing higher performance, lower power, and smaller size while attempting to keep time to market as short as possible. Typical systems may now have multiple heterogeneous reconfigurable cores and a great number of compiler optimizations available, making manual compiler tuning increasingly infeasible. Furthermore, static compilers often fail to produce high-quality code due to a simplistic model of the underlying hardware.

[1]MILEPOST—*M*ach*I*ne *L*earning for *E*mbedded *P*r*O*gram*S* op*T*imization [4].

The difficulty of achieving portable compiler performance has led to iterative compilation [10, 15, 14, 25, 32, 18, 29, 23, 24, 17, 19, 21] being proposed as a means of overcoming the fundamental problem of static modeling. The compiler's static model is replaced by a search of the space of compilation strategies to find the one which, when executed, best improves the program. Little or no knowledge of the current platform is needed so programs can be adapted to different architectures. It is currently used in library generators and by some existing adaptive tools [35, 27, 30, 8, 1, 3]. However, it is largely limited to searching for combinations of global compiler optimization flags and tweaking a few fine-grain transformations within relatively narrow search spaces. The main barrier to its wider use is the currently excessive compilation and execution time needed in order to optimize each program. This prevents its wider adoption in general-purpose compilers.

Our approach is to use machine learning, which has the potential to reuse knowledge across iterative compilation runs, gaining the benefits of iterative compilation while reducing the number of executions needed.

The MILEPOST project's [4] objective is to develop compiler technology that can automatically learn how to best optimize programs for configurable heterogeneous embedded processors using machine learning. It aims to dramatically reduce the time to market of configurable systems. Rather than developing a specialised compiler by hand for each configuration, MILEPOST aims to produce optimizing compilers automatically.

A key goal of the project is to make machine-learning-based compilation a realistic technology for general-purpose compilation. Current approaches [28, 31, 9, 13] are highly preliminary, limited to global compiler flags or simple transformations considered in isolation. GCC was selected as the compiler infrastructure for MILEPOST as it is currently the most stable and robust open-source compiler. It supports multiple architectures and has multiple aggressive optimizations, making it a natural vehicle for our research. In addition, each new version usually features new transformations demonstrating the need for a system to automatically re-tune its optimization heuristics.

In this paper we present early experimental results showing that it is possible to improve the performance of the well-known MiBench [22] benchmark suite on a range of platforms including x86 and IA64. We ported our tools to the new ARC GCC 4.2.1 that targets ARC International's configurable core family. Using MILEPOST GCC, after a few weeks of training, we were able to learn a model that automatically improves the execution time of MiBench benchmark by 11%, demonstrating the use of our machine-learning-based compiler.

This paper is organized as follows: the next section describes the overall MILEPOST framework and is itself followed by a section detailing our implementation of the Interactive Compilation Interface for GCC that enables dynamic manipulation of optimization passes. Section 4 describes machine learning techniques used to predict good optimization passes for programs using static program features and optimization knowledge reuse. Section 5 provides experimental results and is followed by concluding remarks.

## 2 MILEPOST Framework

The MILEPOST project uses a number of components, at the heart of which is the machine-learning-enabled MILEPOST GCC, shown in Figure 1. MILEPOST GCC currently proceeds in two distinct phases, in accordance with typical machine learning practice: training and deployment.

**Training** During the training phase we need to gather information about the structure of programs and record how they behave when compiled under different optimization settings. Such information allows machine learning tools to correlate aspects of program structure, or *features*, with optimizations, building a strategy that predicts a good combination of optimizations.

In order to learn a good strategy, machine learning tools need a large number of compilations and executions as training examples. These training examples are generated by a tool, the Continuous Collective Compilation Framework[2] (CCC), which evaluates different compilation optimizations, storing execution time, code size, and other metrics in a database. The features of the program are extracted from MILEPOST GCC via a plugin and are also stored in the database. Plugins allow fine-grain control and examination of the compiler, driven externally through shared libraries.

**Deployment** Once sufficient training data is gathered, a model is created using machine-learning modeling. The model is able to predict good optimization strategies for a given set of program features and is built as a plugin so that it can be re-inserted into MILEPOST GCC. On encountering a new program, the plugin determines the program's features, passing them to the model, which determines the optimizations to be applied.

**Framework** In this paper we use a new version of the Interactive Compilation Interface (ICI) for GCC which controls the internal optimization decisions and their parameters using external plugins. It now allows the complete substitution of default internal optimization heuristics as well as the order of transformations.

We use the Continuous Collective Compilation Framework [2] to produce a training set for machine learning models to learn how to optimize programs for the best performance, code size, power consumption, and any other objective function needed by the end user. This framework allows knowledge of the optimization space to be reused among different programs, architectures, and data sets.

Together with additional routines needed for machine learning, such as program feature extraction, this forms the MILEPOST GCC. MILEPOST GCC transforms the compiler suite into a powerful research tool suitable for adaptive computing.

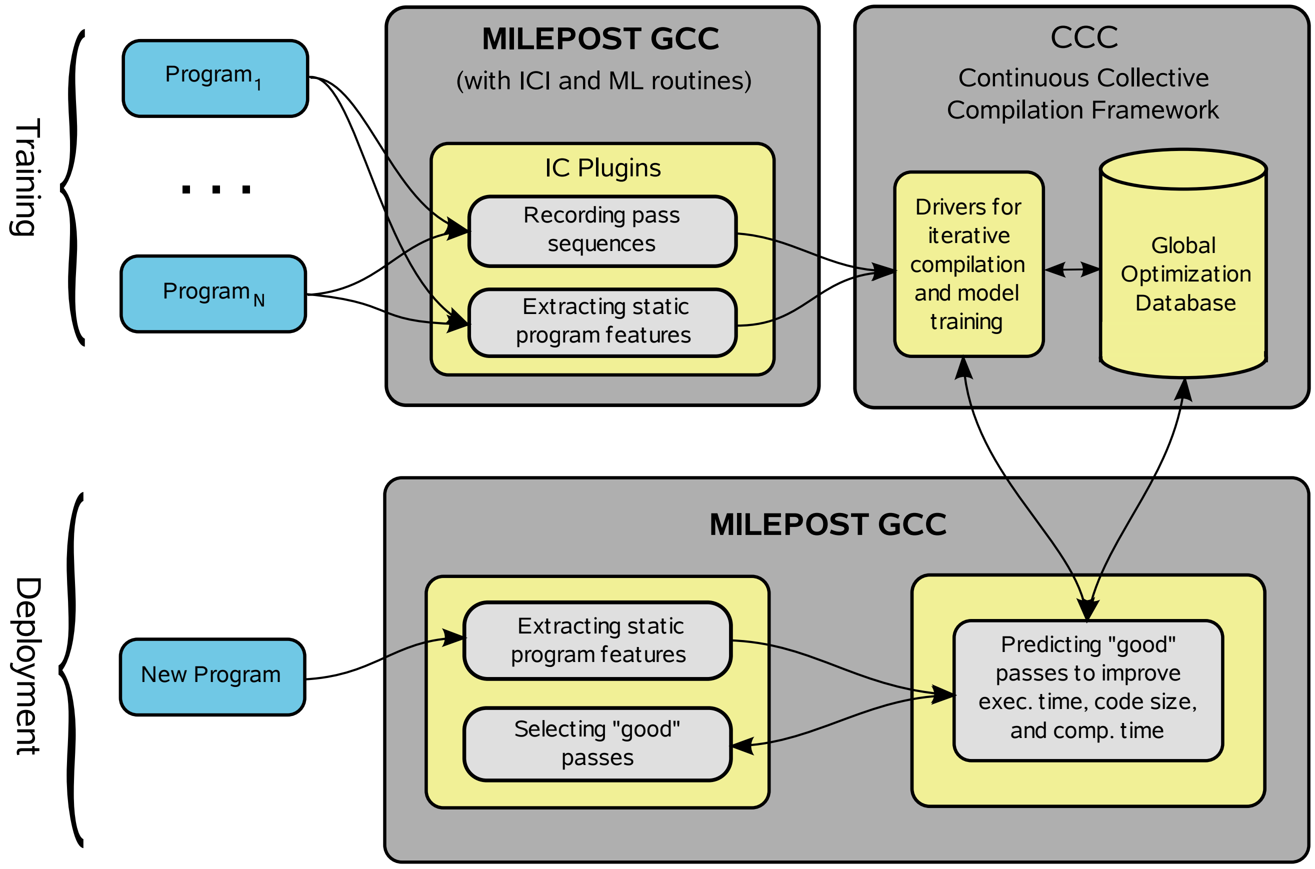

Training
Program1
. . .
ProgramN
MILEPOST GCC
(with ICI and ML routines)
IC Plugins
Recording pass sequences
Extracting static program features
CCC
Continuous Collective Compilation Framework
Drivers for iterative compilation and model training
Global Optimization Database
Deployment
New Program
MILEPOST GCC
Extracting static program features
Selecting "good" passes
Predicting "good" passes to improve exec. time, code size, and comp. time

Figure 1: Framework to automatically tune programs and improve default optimization heuristics using machine learning techniques, MILEPOST GCC with Interactive Compilation Interface (ICI) and program features extractor, and Continuous Collective Compilation Framework to train ML model and predict good optimization passes

The next section describes the new ICI structure and explains how program features can be extracted for later machine learning in Section 4.

## 3 Interactive Compilation Interface

This section describes the Interactive Compilation Interface (ICI). The ICI provides opportunities for external control and examination of the compiler. Optimization settings at a fine-grained level, beyond the capabilities of command line options or pragmas, can be managed through external shared libraries, leaving the compiler uncluttered.

The first version of ICI [20] was reactive and required minimal changes to GCC. It was, however, unable to modify the order of optimization passes within the compiler and so large opportunities for speedup were closed to it. The new version of ICI expands on the capabilities of its predecessor, permitting the pass order to be modified. This version of ICI is used in the MILEPOST GCC to automatically learn good sequences of optimization passes. In replacing default optimization heuristics, execution time, code size, and compilation time can be improved.

### 3.1 Internal structure

To avoid the drawbacks of the first version of the ICI, we designed a new version, as shown in Figure 2. This version can now transparently monitor execution of passes or replace the GCC Controller (Pass Manager), if desired. Passes can be selected by an external plugin which may choose to drive them in a very different order to that currently used in GCC, even choosing different pass orderings for each and every function in program being compiled. Furthermore, the plugin can provide its own passes, implemented entirely outside of GCC.

In an additional set of enhancements, a coherent event and data passing mechanism enables external plugins to

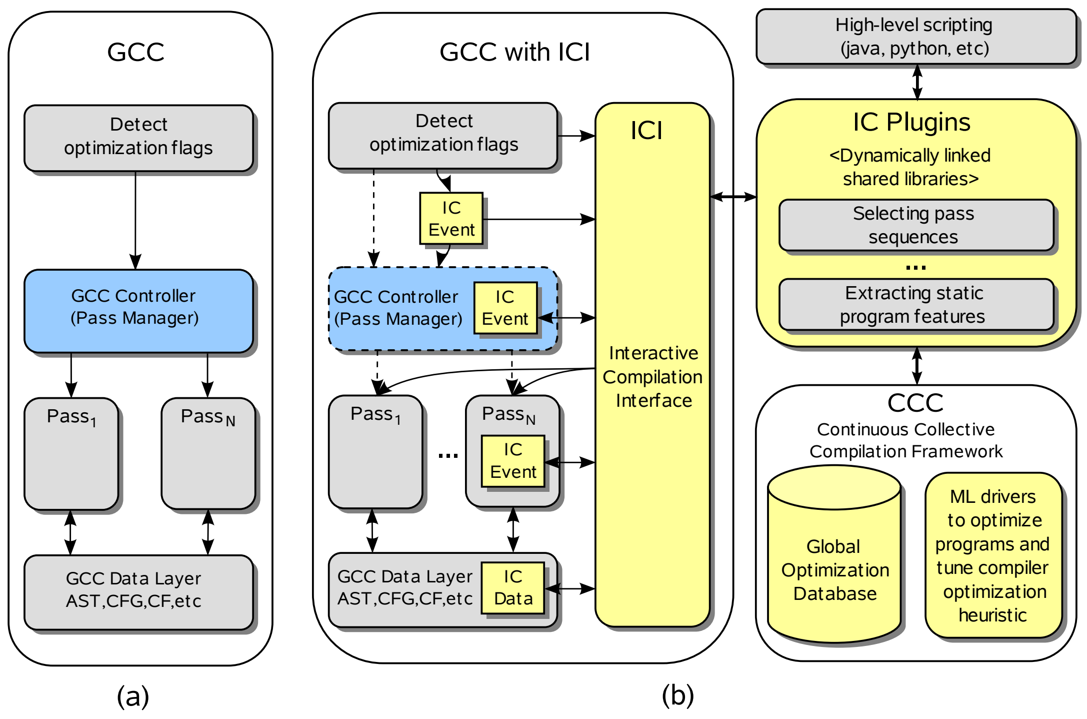

GCC
Detect optimization flags
GCC Controller (Pass Manager)
Pass$_1$
Pass$_N$
GCC Data Layer AST,CFG,CF,etc
(a)
GCC with ICI
Detect optimization flags
IC Event
GCC Controller (Pass Manager)
IC Event
Pass$_1$
...
Pass$_N$
IC Event
GCC Data Layer AST,CFG,CF,etc
IC Data
ICI
Interactive Compilation Interface
High-level scripting (java, python, etc)
IC Plugins
<Dynamically linked shared libraries>
Selecting pass sequences
...
Extracting static program features
CCC
Continuous Collective Compilation Framework
Global Optimization Database
ML drivers to optimize programs and tune compiler optimization heuristic
(b)

Figure 2: GCC Interactive Compilation Interface: a) original GCC, b) GCC with ICI and plugins

discover the state of the compiler and to be informed as it changes. At various points in the compilation process, events (IC Event) are raised, indicating decisions about transformations. Auxiliary data (IC Data) is registered if needed.

Since plugins now extend GCC through external shared libraries, experiments can be built with no further modifications to the underlying compiler. Modifications for different analysis, optimization, and monitoring scenarios proceed in a tight engineering environment. These plugins communicate with external drivers and can allow both high-level scripting and communication with machine learning frameworks such as MILEPOST GCC.

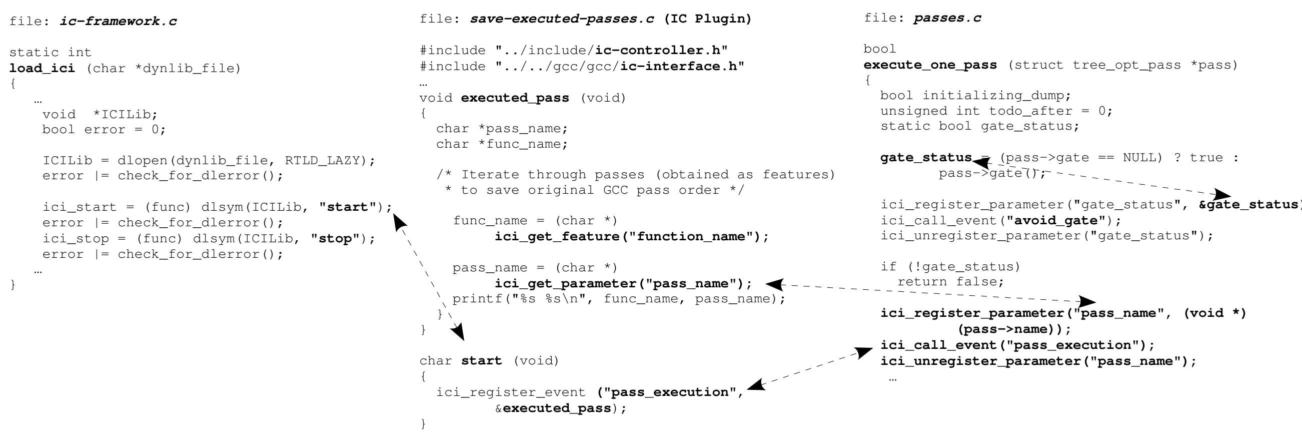
Note that it is not the goal of this project to develop a full-fledged plugin system. Rather, we show the utility of such approaches for iterative compilation and machine learning in compilers. We may later utilize GCC plugin systems currently in development, for example [7] and [12].

Figure 3 shows some of the modifications needed to enable ICI in GCC with an example of a passive plugin to monitor executed passes. The plugin is invoked by the new `-fici` GCC flag or by setting `ICI_USE` environment variable to 1 (to enable non-intrusive optimizations without changes to Makefiles). When GCC detects these options, it loads a plugin (dynamic library) with a name specified by `ICI_PLUGIN` environment variable and checks for two functions, `start` and `stop`, as shown in Figure 3a.

The `start` function of the example plugin registers an event handler function `executed_pass` on an *IC-Event* called *pass_execution*.

Figure 3c shows simple modifications in GCC Controller (Pass Manager) to enable monitoring of executed passes. When the GCC Controller function `execute_one_pass` is invoked, we register an *IC-Parameter* called `pass_name` giving the real name of the executed pass and trigger an *IC-Event pass_execution*. This in turn invokes the plugin function `executed_pass` where we can obtain the current name of the compiled function using `ici_get_feature("function_name")` and the pass name using `ici_get_parameter("pass_name")`.

*IC-Features* provide read only data about the compilation state. *IC-Parameters*, on the other hand, can be dynamically changed by plugins to change the subsequent behavior of the compiler. Such behavior modification is demonstrated in the next subsection using an example with an `avoid_gate` parameter needed for dynamic pass manipulation.



```
file: ic-framework.c

static int
load_ici (char *dynlib_file)
{
   …
    void  *ICILib;
    bool error = 0;

    ICILib = dlopen(dynlib_file, RTLD_LAZY);
    error |= check_for_dlerror();

    ici_start = (func) dlsym(ICILib, "start");
    error |= check_for_dlerror();
    ici_stop = (func) dlsym(ICILib, "stop");
    error |= check_for_dlerror();
   …
}
```

```
file: save-executed-passes.c (IC Plugin)

#include "../include/ic-controller.h"
#include "../../gcc/gcc/ic-interface.h"
…
void executed_pass (void)
{
  char *pass_name;
  char *func_name;

  /* Iterate through passes (obtained as features)
   * to save original GCC pass order */

    func_name = (char *)
         ici_get_feature("function_name");

    pass_name = (char *)
         ici_get_parameter("pass_name");
    printf("%s %s\n", func_name, pass_name);
  }
}

char start (void)
{
  ici_register_event ("pass_execution",
         &executed_pass);
}
```

```
file: passes.c

bool
execute_one_pass (struct tree_opt_pass *pass)
{
  bool initializing_dump;
  unsigned int todo_after = 0;
  static bool gate_status;

  gate_status = (pass->gate == NULL) ? true :
         pass->gate();

  ici_register_parameter("gate_status", &gate_status);
  ici_call_event("avoid_gate");
  ici_unregister_parameter("gate_status");

  if (!gate_status)
    return false;

  ici_register_parameter("pass_name", (void *)
         (pass->name));
  ici_call_event("pass_execution");
  ici_unregister_parameter("pass_name");
   …
```

Figure 3: Some GCC modifications to enable ICI and an example of a plugin to monitor executed passes: a) IC Framework within GCC, b) IC Plugin to monitor executed passes, c) GCC Controller (pass manager) modification

Since we use the `name` field from the GCC `pass` structure to identify passes, we have had to ensure that each pass has a unique name. Previously, some passes have had no name at all and we suggest that in the future a good development practice of always having unique names would be sensible.

## 3.2 Dynamic Manipulation of GCC Passes

Previous research shows a great potential to improve program execution time or reduce code size by carefully selecting global compiler flags or transformation parameters using iterative compilation. The quality of generated code can also be improved by selecting different optimization orders as shown in [15, 14, 16, 25]. Our approach combines the selection of optimal optimization orders and tuning parameters of transformations at the same time.

The new version of ICI enables arbitrary selection of legal optimization passes and has a mechanism to change parameters or transformations within passes. Since GCC currently does not provide enough information about dependencies between passes to detect legal orders, and the optimization space is too large to check all possible combinations, we focused on detecting influential passes and legal orders of optimizations. We examined the pass orders generated by compiler flags that improved program execution time or code size using iterative compilation.

Before we attempt to learn good optimization settings and pass orders, we first confirm that there is indeed performance to be gained within GCC from such actions—otherwise there is no point in trying to learn. By using the Continuous Collective Compilation Framework [2] to randomly search though the optimization flag space (50% probability of selecting each optimization flag) and MILEPOST GCC 4.2.2 on AMD Athlon64 3700+ and Intel Xeon 2800MHz, we could improve execution time of *susan_corners* by around 16%, compile time by 22%, and code size by 13% using Pareto optimal points as described in the previous work [23, 24]. Note that the same combination of flags degrades execution time of this benchmark on Itanium-2 1.3GHz by 80%, thus demonstrating the importance of adapting compilers to each new architecture. Figure 4a shows the combination of flags found for this benchmark on the AMD platform while Figures 4b,c show the passes invoked and monitored by MILEPOST GCC for the default `-O3` level and for the best combination of flags respectively.

Given that there is good performance to be gained by searching for good compiler flags, we now wish to automatically select good optimization passes and transformation parameters. These should enable fine-grained program and compiler tuning as well as non-intrusive continuous program optimizations without modifications to Makefiles, etc. The current version of ICI allows passes to be called directly using the `ici_run_pass` function that in turn invokes GCC function `execute_one_pass`. Therefore, we can circumvent the default GCC Pass Manager and execute good sequences of passes previously found by the CCC Framework as

```
-O3 -fsched-stalled-insns-dep=28 -falign-functions=64 -falign-jumps=64 -falign-loops=19 -fno-branch-count-reg
-fbranch-target-load-optimize -fno-crossjumping -fno-cse-skip-blocks -fdefer-pop -fno-defer-pop -fno-force-addr
-fgcse-after-reload -fno-gcse-las -fno-gcse -fif-conversion -finline-functions -fno-ivopts -fno-math-errno
-foptimize-register-move -fno-optimize-sibling-calls -fpeel-loops -fno-prefetch-loop-arrays -fregmove -freorder-blocks
-frerun-loop-opt -fno-sched-interblock -fsched-spec-load-dangerous -fsched-spec-load -fno-sched2-use-superblocks
-fschedule-insns -fschedule-insns2 -fno-signaling-nans -fno-split-ivs-in-unroller -fstrength-reduce -fstrict-aliasing
-fno-tree-copyrename -fno-tree-dominator-opts -fno-tree-dse -fno-tree-loop-im -fno-tree-loop-linear -ftree-loop-optimize
-fno-tree-lrs -ftree-pre -ftree-sra -fno-tree-vect-loop-version
```

**(a)**

fixupcfg,init_datastructures,all_optimizations,referenced_vars,reset_cc_flags,salias,ssa,alias,retslot,**copyrename**,ccp,fre,
dce,forwprop,copyprop,mergephi,vrp,dce,**dom**,**phicprop**,phiopt,alias,**tailr**,profile,ch,,cplxlower,sra,alias,**copyrename**,**dom**,
**phicprop**,reassoc,dce,**dse**,alias,forwprop,phiopt,objsz,store_ccp,store_copyprop,fab,alias,crited,pre,alias,sink,loop,loopinit,
copyprop,**lim**,unswitch,sccp,empty,record_bounds,ivcanon,cunroll,**ivopts**,loopdone,reassoc,vrp,**dom**,**phicprop**,cddce,**dse**,
forwprop,phiopt,**tailc**,**copyrename**,**uncprop**,optimized,nrv,blocks,final_cleanup,warn_function_noreturn,free_datastructure,
free_cfg_annotations,expand,rest_of_compilation,init_function,sibling,locators,initvals,unshared,vregs,jump,cse1,**gcse1**,
**bypass**,ce1,loop2,loop2_init,loop2_invariant,loop2_unswitch,loop2_done,cse2,life1,combine,ce2,regmove,split1,
mode-sw,life2,lreg,greg,postreload,postreload_cse,gcse2,flow2,csa,peephole2,ce3,rnreg,bbro,leaf_regs,sched2,stack,
compute_alignments,compgotos,free_cfg,mach,elnotes,barriers,eh-anges,shorten,set_nothrow_function_flags,final,
clean_state

**(b)**

fixupcfg,init_datastructures,all_optimizations,referenced_vars,reset_cc_flags,salias,ssa,alias,retslot,ccp,fre,dce,forwprop,
copyprop,mergephi,vrp,dce,phiopt,alias,profile,ch,cplxlower,sra,alias,reassoc,dce,alias,forwprop,phiopt,objsz,store_ccp,
store_copyprop,fab,alias,crited,pre,alias,sink,loop,loopinit,copyprop,unswitch,sccp,empty,record_bounds,ivcanon,cunroll,
loopdone,reassoc,vrp,cddce,forwprop,phiopt,optimized,nrv,blocks,final_cleanup,warn_function_noreturn,
free_datastructures,free_cfg_annotations,expand,rest_of_compilation,init_function,sibling,locators,initvals,unshared,vregs,
jump,cse1,ce1,loop2,loop2_init,loop2_invariant,loop2_unswitch,**loop2_unroll**,loop2_done,**web**,cse2,life1,combine,ce2,
regmove,split1,mode-sw,life2,**sched1**,lreg,greg,postreload,postreload_cse,gcse2,flow2,csa,peephole2,ce3,rnreg,bbro,
leaf_regs,sched2,stack,compute_alignments,compgotos,free_cfg,mach,elnotes,barriers,eh-ranges,shorten,
set_nothrow_function_flags,final,clean_state

**(c)**

Figure 4: a) Selection of compiler flags found using CCC Framework with uniform random search strategy that improve execution and compilation time for *susan_corners* benchmark over `-O3`, b) recorded compiler passes for `-O3` using ICI, c) recorded compiler passes for the good selection of flags (a)

shown in Figure 2b or search for new, good orders of optimizations. However, we leave the determination of their interaction and dependencies for future work.

To verify that we can change the default optimization pass orders using ICI, we recompiled the same benchmark with the `-O3` flag, but selected passes as shown in Figure 4c. However, note that the GCC internal function `execute_one_pass` shown in Figure 3c has gate control (*pass->gate()*) to execute the pass only if the associated optimization flag is selected. To avoid this gate control we use *IC-Parameter "gate_status"* and *IC-Event "avoid_gate"* so that we can set `gate_status` to `TRUE` within plugins and thus force its execution. The execution of the generated binary shows that we improve its execution time by 13% instead of 16% and the reason is that some compiler flags not only invoke the associated pass, such as `-funroll-loops`, but also select specific fine-grain transformation parameters and influence code generation in other passes. Thus, at this point we recompile programs with such flags always enabled, and in the future plan to add support for such cases explicitly.

## 3.3 Adding program feature extractor pass

Our machine-learnt model predicts the best GCC optimization to apply to an input program based on its program structure or *program features*. The program features are typically a summary of the internal program representation and characterize the essential aspects of a program needed by the model to distinguish between good and bad optimizations.

The current version of ICI allows invoking auxiliary passes that are not a part of the default GCC compiler. These passes can monitor and profile the compilation process or extract data structures needed to generate program features.

During the compilation, the program is represented by several data structures, implementing the intermediate representation (tree-SSA, RTL, etc.), control flow graph (CFG), def-use chains, the loop hierarchy, etc. The data structures available depend on the compilation pass currently being performed. For statistical machine learning, the information about these data structures is encoded as a vector of constant size of numbers (i.e., features). This process is called feature extraction and is needed to enable optimization knowledge reuse among different programs.

Therefore, we implemented an additional GCC pass *ml-feat* to extract static program features. This pass is not invoked during default compilation, but can be called using an `extract_program_static_features` plugin after any arbitrary pass starting from *FRE*, when all the GCC data necessary to produce features is ready.

In the MILEPOST GCC, the feature extraction is performed in two stages. In the first stage, a relational representation of the program is extracted; in the second stage, the vector of features is computed from this representation.

In the first stage, the program is considered to be characterized by a number of entities and relations over these entities. The entities are a direct mapping of similar entities defined by the language reference, or generated during the compilation. Such examples of entities are variables, types, instructions, basic blocks, temporary variables, etc.

A relation over a set of entities is a subset of their Cartesian product. The relations specify properties of the entities or the connections between them. For describing the relations we used a notation based on logic—Datalog is a Prolog-like language but with simpler semantics, suitable for expressing relations and operations between them [34, 33].

For extracting the relational representation of the program, we used a simple method, based on the examination of the *include* files. The compiler main data structures are *struct* data types, having a number of *fields*. Each such *struct* data type may introduce an entity, and its *fields* may introduce relations over the entity representing the including *struct* data type and the entity representing the data type of the *field*. This data is collected using the *ml-feat* pass.

In the second stage, we provide a Prolog program defining the features to be computed from the Datalog relational extracted from the compiler internal data structures in the first stage. The `extract_program_static_features` plugin invokes a Prolog compiler to execute this program, the result being the vector of features shown in Table 1 that can later be used by the Continuous Collective Compilation Framework to build machine learning models and predict best sequences of passes for new programs. This example shows the flexibility and capabilities of the new version of ICI.

## 4 Using Machine Learning to Select Good Optimization Passes

The previous sections have described the infrastructure necessary to build a learning compiler. In this section we describe how this infrastructure is used in building a model.

Our approach to selecting good passes for programs is based upon the construction of a *probabilistic model* on a set of $M$ training programs and the use of this model in order to make predictions of "good" optimization passes on unseen programs.

Our specific machine learning method is similar to that of [9] where a probability distribution over "good" solutions (i.e., optimization passes or compiler flags) is learnt across different programs. This approach has been referred to in the literature as Predictive Search Distributions (PSD) [11]. However, unlike [9, 11] where such a distribution is used to focus the search of compiler optimizations on a new program, we use the distribution learned to make *one-shot* predictions on unseen programs. Thus we do not search for the best optimization, we automatically predict it.

### 4.1 The Machine Learning Model

Given a set of training programs $T^1, \ldots, T^M$, which can be described by (vectors of) features $\mathbf{t}^1 \ldots, \mathbf{t}^M$, and for which we have evaluated different sequences of optimization passes ($\mathbf{x}$) and their corresponding execution times (or speed-ups $y$) so that we have for each program $M^j$ an associated dataset $\mathscr{D}^j = \{(\mathbf{x}^i, y^i)\}_{i=1}^{Nj}$, with $j = 1, \ldots M$, our goal is to predict a good sequence of optimization passes $\mathbf{x}^*$ when a new program $T^*$ is presented.

We approach this problem by learning the mapping from the features of a program $\mathbf{t}$ to a *distribution over good solutions* $q(\mathbf{x}|\mathbf{t}, \theta)$, where $\theta$ are the parameters of the distribution. Once this distribution has been learnt, predictions on a new program $T^*$ is straightforward and it is achieved by sampling at the mode of the distribution. In other words, we obtain the predicted sequence of passes by computing:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}}\, q(\mathbf{x}|\mathbf{t}, \theta). \qquad (1)$$

### 4.2 Continuous Collective Compilation Framework

We used Continuous Collective Compilation Framework [2] and MILEPOST GCC shown in Figure 1 to

| **Feature number:** | **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 then 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 |
| ft21 | Number of unconditional branches in the method |
| ft22 | Number of binary integer operations in the method |
| ft23 | Number of binary floating point operations in the method |
| ft24 | Number of instructions in the method |
| ft25 | Average of number of instructions in basic blocks |
| ft26 | Average of number of phi-nodes at the beginning of a basic block |
| ft27 | Average of arguments for a phi-node |
| ft28 | Number of basic blocks with no phi nodes |
| ft29 | Number of basic blocks with phi nodes in the interval [0, 3] |
| ft30 | Number of basic blocks with more then 3 phi nodes |
| ft31 | Number of basic block where total number of arguments for all phi-nodes is in greater then 5 |
| ft32 | Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5] |
| ft33 | Number of switch instructions in the method |
| ft34 | Number of unary operations in the method |
| ft35 | Number of instruction that do pointer arithmetic in the method |
| ft36 | Number of indirect references via pointers ("*" in C) |
| ft37 | Number of times the address of a variables is taken ("&" in C) |
| ft38 | Number of times the address of a function is taken ("&" in C) |
| ft39 | Number of indirect calls (i.e., done via pointers) in the method |
| ft40 | Number of assignment instructions with the left operand an integer constant in the method |
| ft41 | Number of binary operations with one of the operands an integer constant in the method |
| ft42 | Number of calls with pointers as arguments |
| ft42 | Number of calls with the number of arguments is greater than 4 |
| ft44 | Number of calls that return a pointer |
| ft45 | Number of calls that return an integer |
| ft46 | Number of occurrences of integer constant zero |
| ft47 | Number of occurrences of 32-bit integer constants |
| ft48 | Number of occurrences of integer constant one |
| ft49 | Number of occurrences of 64-bit integer constants |
| ft50 | Number of references of a local variables in the method |
| ft51 | Number of references (def/use) of static/extern variables in the method |
| ft52 | Number of local variables referred in the method |
| ft53 | Number of static/extern variables referred in the method |
| ft54 | Number of local variables that are pointers in the method |
| ft55 | Number of static/extern variables that are pointers in the method |

Table 1: List of static program features currently available in the MILEPOST GCC

generate a training set of programs together with compiler flags selected uniformly at random, associated sequences of passes, program features and speedups (code size, compilation time) that is stored in the externally accessible Global Optimization Database. We use this training set to build machine learning model described in the next section which in turn is used to predict the best sequence of passes for a new program given its feature vector. Current version of CCC Framework requires minimal changes to the Makefile to pass optimization flags or sequences of passes and has a support to verify the correctness of the binary by comparing program output with the reference one to avoid illegal combinations of optimizations.

### 4.3 Learning and Predicting

In order to learn the model, it is necessary to fit a distribution over good solutions to each training program beforehand. These solutions can be obtained, for example, by using uniform sampling or by running an estimation of distribution algorithm (EDA, see [26] for an overview) on each of the training programs. In our experiments we use uniform sampling and we choose the set of good solutions to be those optimization settings that achieve at least 98% of the maximum speedup available in the corresponding program-dependent dataset.

Let us denote the distribution over good solutions on each training program by $P(\mathbf{x}|T^j)$ with $j = 1,\ldots,M$. In principle, these distributions can belong to any parametric family. However, In our experiments we use an IID model where each of the elements of the sequence are considered independently. In other words, the probability of a "good" sequence of passes is simply the product of each of the individual probabilities corresponding to how likely each pass is to belong to a good solution:

$$P(\mathbf{x}|T^j) = \prod_{\ell=1}^{L} P(x_\ell|T^j), \tag{2}$$

where $L$ is the length of the sequence.

As proposed in [11], once the individual training distributions $P(\mathbf{x}|T^j)$ have been obtained, the predictive distribution $q(\mathbf{x}|\mathbf{t},\theta)$ can be learnt by maximization of the conditional likelihood or by using *k-nearest neighbor* methods. In our experiments we use a 1-nearest neighbor approach. In other words, we set the predictive distribution $q(\mathbf{x}|\mathbf{t},\theta)$ to be the distribution corresponding to the training program that is closest in feature space to the new (test) program.

Note that we currently predict "good" sequences of optimization passes that are associated to the best combination of compiler flags. We will investigate in future work the application of our models to the general problem of determining "optimal" order of optimization passes for programs.

## 5 Experiments

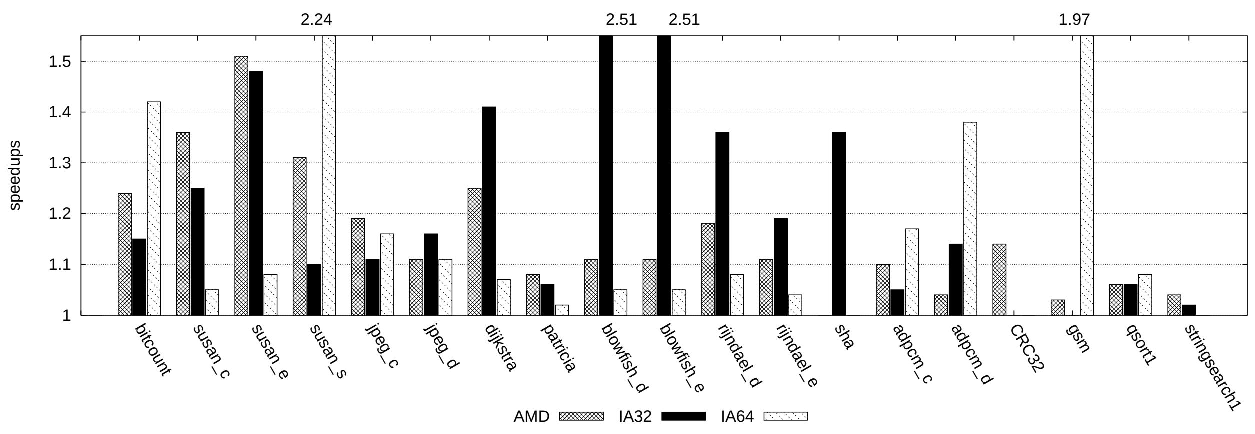
We performed our experiments on four different platforms:

- *AMD* – a cluster with 16 AMD Athlon 64 3700+ processors running at 2.4GHz
- *IA32* – a cluster with 4 Intel Xeon processors running at 2.8GHz
- *IA64* – a server with an Itanium2 processor running at 1.3GHz
- *ARC* – FPGA implementation of the ARC 725D processor running GNU/Linux with a 2.4.29 kernel.

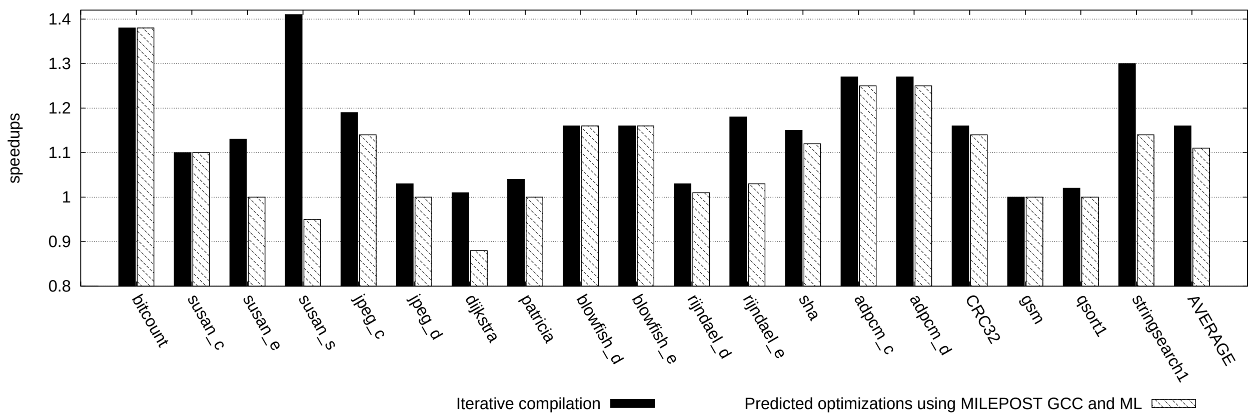
In all case the compiler used is MILEPOST GCC 4.2.x with ICI version 0.9.6. We decided to use the open-source MiBench benchmark with MiDataSets [19, 6] (dataset No. 1 in all cases) due to its applicability to both general-purpose and embedded domains.

### 5.1 Generating Training Data

In order to build a machine learning model, we need training data. This was generated by a random exploration of a vast optimization search space using the CCC Framework. It generated 500 random sequences of flags either turned on or off. These flag settings can be readily associated with different sequences of optimization passes. Although such a number of runs is very small in respect to the optimisation space, we have shown that sufficient information can be gleaned from this to allow significant speedup. Indeed, the size of the space left unexplored serves to highlight our lack of knowledge in this area, and the need for further work.



**(a)**



**(b)**

Figure 5: Experimental results when using iterative compilation with random search strategy (500 iterations; 50% probability to select each flags; AMD,IA32,IA64) and when predicting best optimization passes based on program features and ML model (ARC)

Firstly, features for each benchmark were extracted from programs using the new pass within MILEPOST GCC, and these features then sent to the Global Optimization Database within CCC Framework. An ML model for each benchmark was built, using the execution time gathered from 500 separate runs using different random sequences of passes, and a fixed data set. Each run was repeated 5 times so speedups were not caused by cache priming, *etc.*). After each run, the experimental results including execution time, compilation time, code size, and program features are sent to the database, where they are stored for future reference.

Figure 5a shows that considerable speedups can be already obtained after iterative compilation on all platforms. However, this is a time-consuming process and different speedups across different platforms motivates the use of machine learning to automatically build specialized compilers and predict the best optimization flags or sequences of passes for different architectures.

## 5.2 Evaluating Model Performance

Once a model has been built for each of our benchmarks, we can evaluate the results by introducing a new program to the system, and measuring how well the prediction provided by our model performs. In this work we did not use a separate testing benchmark suite due to time constraints, so instead *leave-one-out-cross-validation* was used. Using this method, all training data relating to the benchmark being tested is excluded from the training process, and the models rebuilt. When a second benchmark is tested, the training data pertaining to the first benchmark is returned to the training set, that of the second benchmark excluded, and so on. In this way we ensure that each benchmark is tested as a new program entering the system for the first time—of course, in real-world usage, this process is unnecessary.

When a new program is compiled, features are first generated using MILEPOST GCC. These features are then sent to our ML model within the CCC Framework (implemented as a MATLAB server), which processes them and returns a predicted sequence of passes which should either improve execution time, reduce code size, or both. We then evaluate the prediction by compiling the program with the suggested sequence of passes, measure the execution time, and compare with the original time for the default `-O3` optimization level. It is important to note that only one compilation occurs at evaluation—there is no search involved. Figure 5b shows these results for the ARC725D. It demonstrates that except a few pathological cases where predicted flags degraded performance and which analysis we leave for future work, using CCC Framework, MILEPOST GCC and Machine Learning Models we can improve original ARC GCC by around 11%.

This suggests that our techniques and tools can be efficient to build future iterative adaptive specialized compilers.

## 6 Conclusions and Future Work

In this paper we have shown that MILEPOST GCC has significant potential in the automatic tuning of GCC optimization. We plan to use these techniques and tools to further investigate the automatic selection of optimal orders of optimization passes and fine-grain tuning of transformation parameters. The overall framework will also allow the analysis of interactions between optimizations and investigation of the influence of program inputs and run-time state on program optimizations. Future work will also include fine-grain runtime adaptation for multiple program inputs on heterogeneous multi-core architectures.

## 7 Acknowledgments

The authors would like to thank Sebastian Pop for the help with modifying and debugging GCC and John W. Lockhart for the help with editing this paper. This research is supported by the European Project MILEPOST (Machine Learning for Embedded Programs Optimisation) [4] and by the HiPEAC European Network of Excellence (High-Performance Embedded Architecture and Compilation) [5].

## References

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

[2] Continuous Collective Compilation Framework. `http://cccpf.sourceforge.net`.

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

[4] EU Milepost project (MachIne Learning for Embedded PrOgramS opTimization). `http://www.milepost.eu`.

[5] European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). `http://www.hipeac.net`.

[6] MiDataSets SourceForge development site. `http://midatasets.sourceforge.net`.

[7] Plugin project. `http://libplugin.sourceforge.net`.

[8] QLogic PathScale EKOPath Compilers. `http://www.pathscale.com`.

[9] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[10] F. Bodin, T. Kisuki, P. Knijnenburg, M. 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.

[11] E. V. Bonilla, C. K. I. Williams, F. V. Agakov, J. Cavazos, J. Thomson, and M. F. P. O'Boyle. Predictive search distributions. In W. W. Cohen and A. Moore, editors, *Proceedings of the 23rd International Conference on Machine learning*, pages 121–128, New York, NY, USA, 2006. ACM.

[12] S. Callanan, D. J. Dean, and E. Zadok. Extending gcc with modular gimple optimizations. In *Proceedings of the GCC Developers' Summit'2007*, 2007.

[13] 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.

[14] 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.

[15] K. Cooper, P. 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.

[16] B. Elliston. Studying optimisation sequences in the gnu compiler collection. Technical Report ZITE8199, School of Information Technology and Electical Engineering, Australian Defence Force Academy, 2005.

[17] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[18] G. Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.

[19] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.

[20] G. Fursin and A. Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007.

[21] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[23] 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.

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

[25] 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.

[26] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[27] 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.

[28] 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.

[29] 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.

[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. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

[33] J. Ullman. Principles of database and knowledge systems. *Computer Science Press*, 1, 1988.

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

[35] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.