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

Phil Barnard
Elton Ashton

ARC International UK

Eric Courtois
Francois Bodin

CAPS Enterprise France

Edwin Bonilla
John Thomson
Hugh Leather
Chris Williams
Michael O'Boyle

University of Edinburgh, UK

Contact: grigori.fursin@inria.fr

Motivation

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 poster we describe MILEPOST** 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.

* MILEPOST Project - MachIne Learning for Embedded PrOgramS opTimization
http://www.milepost.eu

Challenges

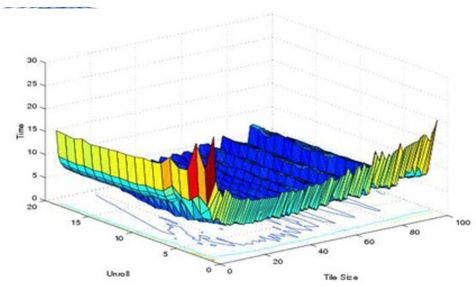
Current state-of-the-art compilers often fail to deliver best performance due to:

- *hardwired optimization heuristics (cost models) for rapidly evolving hardware (often impossible to fine-tune programs externally)*
- *interaction between optimizations*
- *large irregular optimization spaces*
- *difficult to add new transformations to already tuned optimization heuristics*
- *inability to reuse optimization knowledge among different programs and architectures*
- *lack of run-time information and inability to adapt to varying program and system behavior at run-time with low overhead*

Need modular self-tuning compilers that can continuously and automatically learn how to optimize programs, and have an ability to make program adaptable at run-time for different behavior and constraints

Iterative compilation

Optimization spaces (set of all possible program transformations) are large, non-linear with many local minima



Finding a good solution may be long and non-trivial

matmul, 2 transformations, search space = 2000

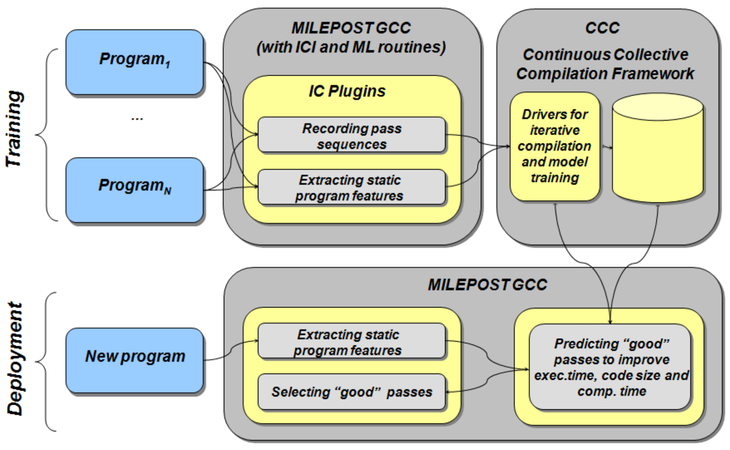
swim, 3 transformations, search space = 10^{52}

Iterative compilation:
learn program behavior across executions

High potential (O'Boyle, Cooper since 1998), but:
- slow
- the same dataset is used
- no run-time adaptation
- no optimization knowledge reuse

Solving these problems is non-trivial

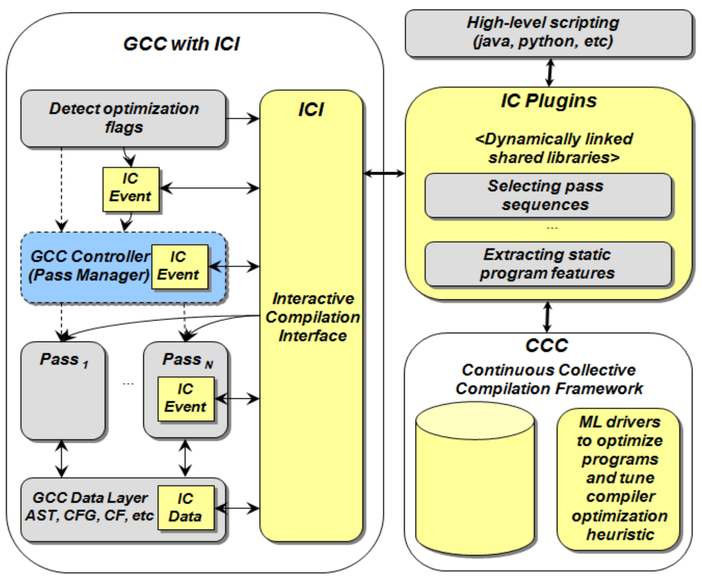
MILEPOST Framework



Training: Gathering information about the structure of programs and record how they behave when compiled under different optimization settingsto build machine learning models.

Deployment: ML model is able to predict good optimization strategies for a given set of program features and is built as a plugin within MILEPOST GCC.

Interactive Compilation Interface



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 IC plugins.

GCC IC Plugins

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");
  ...
```

Modifications needed to enable GCC ICI:

ic-framework.c: GCC plugin (dynamic library) invocation

save-executed-passes.c: Plugin to monitor executed passes. It registers an event handler function executed_pass on an IC-Event called pass_execution.

passes.c: Modifications in GCC Controller (Pass Manager) to enable manipulation with optimization passes.

Optimization selection

Using Interactive Compilation Interface we can conduct research on optimization pass selection and reordering

*fixupcfg,init_datastructures,all_optimizations, ... ,retslot,**copyrename**,ccp, ...,dce,**dom**,**phicprop**,phiopt,alias,**tailr,** profile,ch,...,alias,**copyrename**,**dom**,**phicprop**,reassoc,dce,**dse**,alias, ...,copyprop,**lim**,unswitch,...,cunroll,**ivopts**, loopdone,reassoc,vrp,**dom**,**phicprop**,cddce,**dse**,forwprop,phiopt,**tailc**,**copyrename**,**uncprop**,optimized, ...,cse1, **gcse1**,**bypass**,ce1,...,final, clean_state*

Sequence of compiler passes for –O3

*fixupcfg,init_datastructures,all_optimizations, ...,retslot,ccp,...,dce,phiopt,alias,profile,ch,...,alias,reassoc,dce,alias,..., cunroll, loopdone,reassoc,vrp,cddce,forwprop,phiopt,optimized, ..., cse1,ce1,...,**loop2_unroll**,loop2_done,**web**,cse2, life1,combine,ce2, regmove,split1,mode-sw,life2,**sched1**,...,final,clean_state*

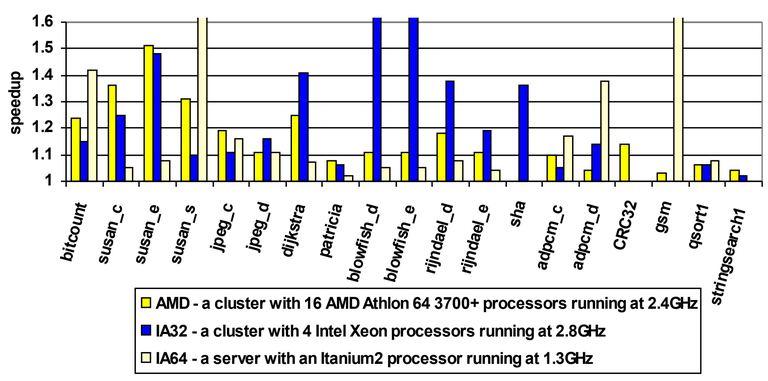
Sequence of GCC passes for the "good" set of compiler flags (ISSUE with UNROLLING, PEELING)

Feature extraction

We can now add new passes that are not included into default optimization heuristic but called through ICI and plugins:

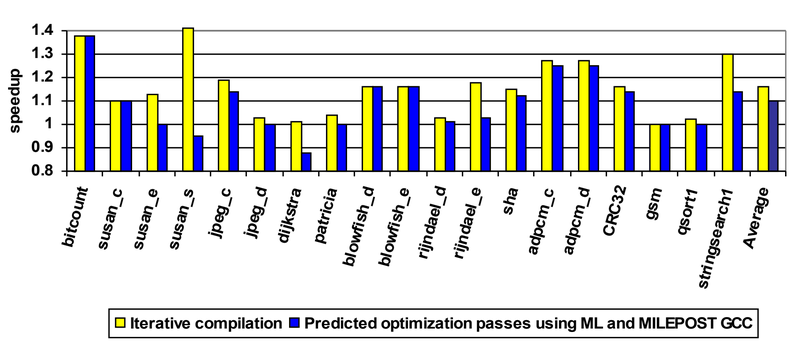
ft1 - Number of basic blocks in the method
...
ft20 - Number of conditional branches in the method
ft21 - Number of assignment instructions in the method
...
ft25 - Average of number of instructions in basic blocks
...
ft55 - Number of static/extern variables that are pointers in the method

Training



We generate training data using Continuous Collective Compilation Framework. We select 500 random sequences of flags (or associated passes) either turned on or off. We can already achieve considerable speedups across all platforms, however it is very time-consuming process motivating the use of machine learning to automatically build specialized compilers and predict the best optimization flags or sequences of passes for different architectures.

Predicting



Once ML model is built, we can evaluate it by introducing a new program to a system and measuring how well the prediction performs. This graphs shows the speedups obtained on ARC725D after iterative compilation (500 iterations) and after 1 prediction. It demonstrates that except a few pathological cases using CCC Framework, MILEPOST GCC and Machine Learning Models we can improve original ARC GCC by around 11%.

Future work

- continue research on pass selection and reordering for reconfigurable processors and design space exploration
- improve and automate selection of static and dynamic (hardware counters) program features
- use ICI for adaptive libraries
- enable run-time adaptation and parallelization for static programs

MILEPOST

More information

Project news: http://www.milepost.eu
GCC-ICI: http://gcc-ici.sourceforge.net
UNIDAPT: http://unidapt.org

HiPEAC COMPILATION ARCHITECTURE

Extended version (GCC Summit'08): http://unidapt.org/papers/fmtp2008.pdf