# Milepost GCC: Machine Learning Enabled Self-tuning Compiler

Presented by Kartikeya Kandula and Tarun Gogineni

# Background

New processor architectures

- Higher performance
- Lower power
- Time to market as short as possible

Static compilers fail to deliver satisfactory levels of performance

- Cannot keep pace with hardware evolution
- Fixed heuristics based on simplistic hardware models and lack of run-time information means that much manual retuning of the compiler is needed
- Systems have multiple heterogeneous reconfigurable cores

# Classical Approach - Iterative Compilation

Applying automatic compiler tuning based on feedback-directed compilation

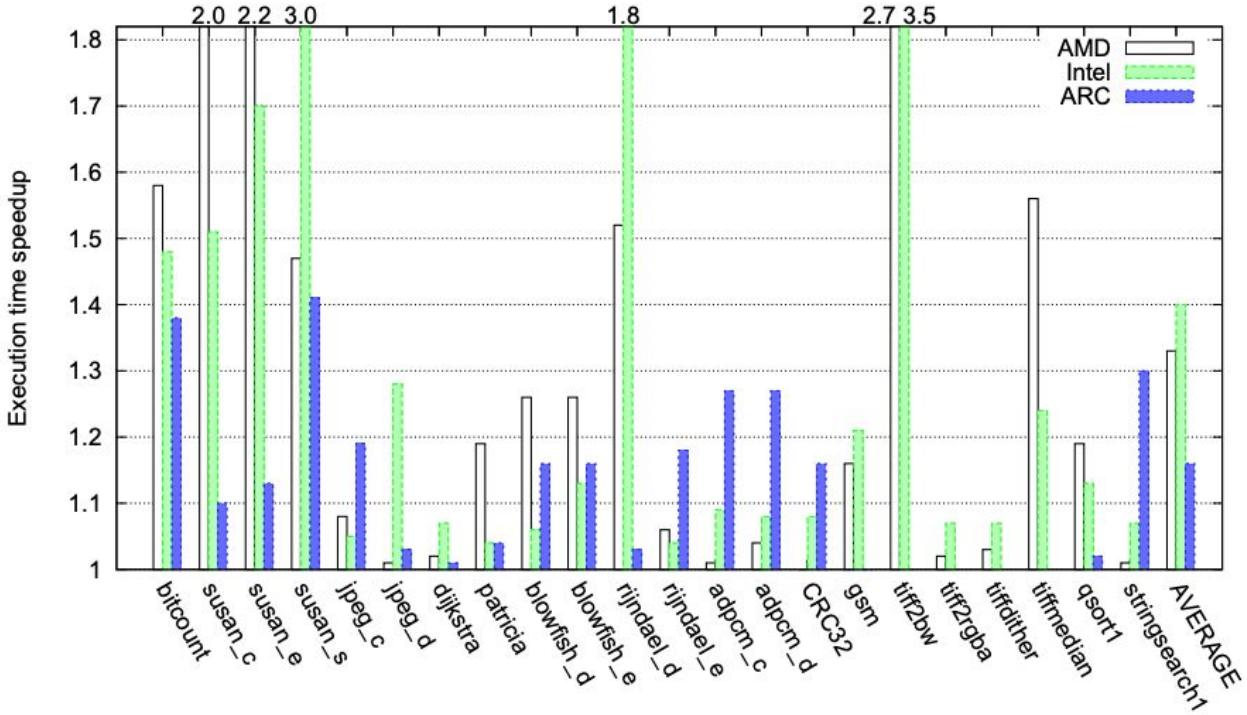
Iterative search of optimization space to find most profitable solutions to improve:

- execution time
- compilation time
- code size
- power use
- arbitrary metrics

Usable only within relatively narrow search spaces

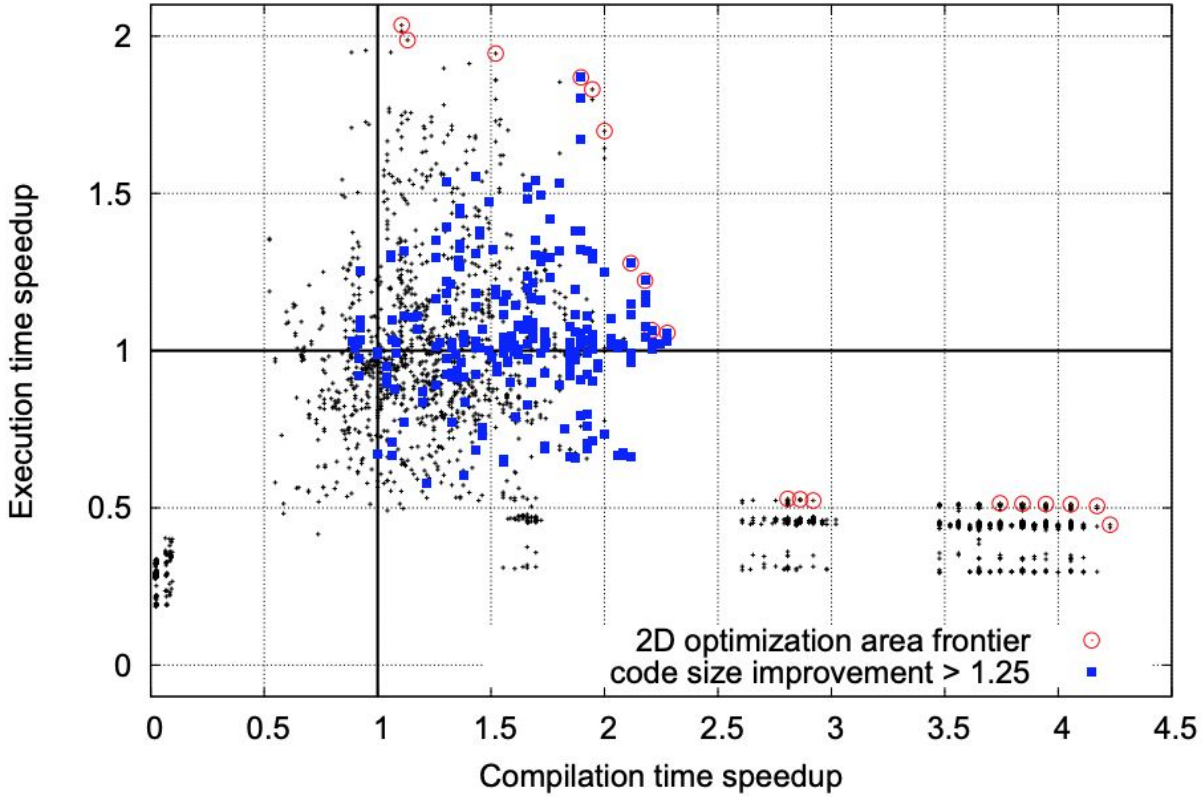
- excessive compile time needed to optimize each program, too slow in most cases

# Motivation - 1000 Random Trials




Random Search can beat GCC -O3

# Pareto Optimality



# Pareto Best

| |
|---|
| -O1 -fcse-follow-jumps -fno-tree-ter -ftree-vectorize |
| -O1 -fno-cprop-registers -fno-dce -fno-move-loop-invariants -frename-registers -fno-tree-copy-prop -fno-tree-copyrename |
| -O1 -freorder-blocks -fschedule-insns -fno-tree-ccp -fno-tree-dominator-opts |
| -O2 |
| -O2 -falign-loops -fno-cse-follow-jumps -fno-dce -fno-gcse-lm -fno-inline-functions-called-once -fno-schedule-insns2 -fno-tree-ccp -fno-tree-copyrename -funroll-all-loops |
| -O2 -finline-functions -fno-omit-frame-pointer -fschedule-insns -fno-split-ivs-in-unroller -fno-tree-sink -funroll-all-loops |
| -O2 -fno-align-jumps -fno-early-inlining -fno-gcse -fno-inline-functions-called-once -fno-move-loop-invariants -fschedule-insns -fno-tree-copyrename -fno-tree-loop-optimize -fno-tree-ter -fno-tree-vrp |
| -O2 -fno-caller-saves -fno-guess-branch-probability -fno-ira-share-spill-slots -fno-tree-reassoc -funroll-all-loops -fno-web |
| -O2 -fno-caller-saves -fno-ivopts -fno-reorder-blocks -fno-strict-overflow -funroll-all-loops |
| -O2 -fno-cprop-registers -fno-move-loop-invariants -fno-omit-frame-pointer -fpeel-loops |
| -O2 -fno-dce -fno-guess-branch-probability -fno-strict-overflow -fno-tree-dominator-opts -fno-tree-loop-optimize -fno-tree-reassoc -fno-tree-sink |
| -O2 -fno-ivopts -fpeel-loops -fschedule-insns |
| -O2 -fno-tree-loop-im -fno-tree-pre |
| -O3 -falign-loops -fno-caller-saves -fno-cprop-registers -fno-if-conversion -fno-ivopts -freorder-blocks-and-partition -fno-tree-pre -funroll-all-loops |
| -O3 -falign-loops -fno-cprop-registers -fno-if-conversion -fno-peephole2 -funroll-all-loops |
| -O3 -falign-loops -fno-delete-null-pointer-checks -fno-gcse-lm -fira-coalesce -floop-interchange -fsched2-use-superblocks -fno-tree-pre -fno-tree-vectorize -funroll-all-loops -funsafe-loop-optimizations -fno-web |
| -O3 -fno-gcse -floop-strip-mine -fno-move-loop-invariants -fno-predictive-commoning -ftracer |
| -O3 -fno-inline-functions-called-once -fno-regmove -frename-registers -fno-tree-copyrename |
| -O3 -fno-inline-functions -fno-move-loop-invariants |

**Table 1** Best found combinations of Milepost GCC flags to improve execution time, code size and compilation time after iterative compilation (1000 iterations) across all evaluated benchmarks and platforms.

# Number of Iterations

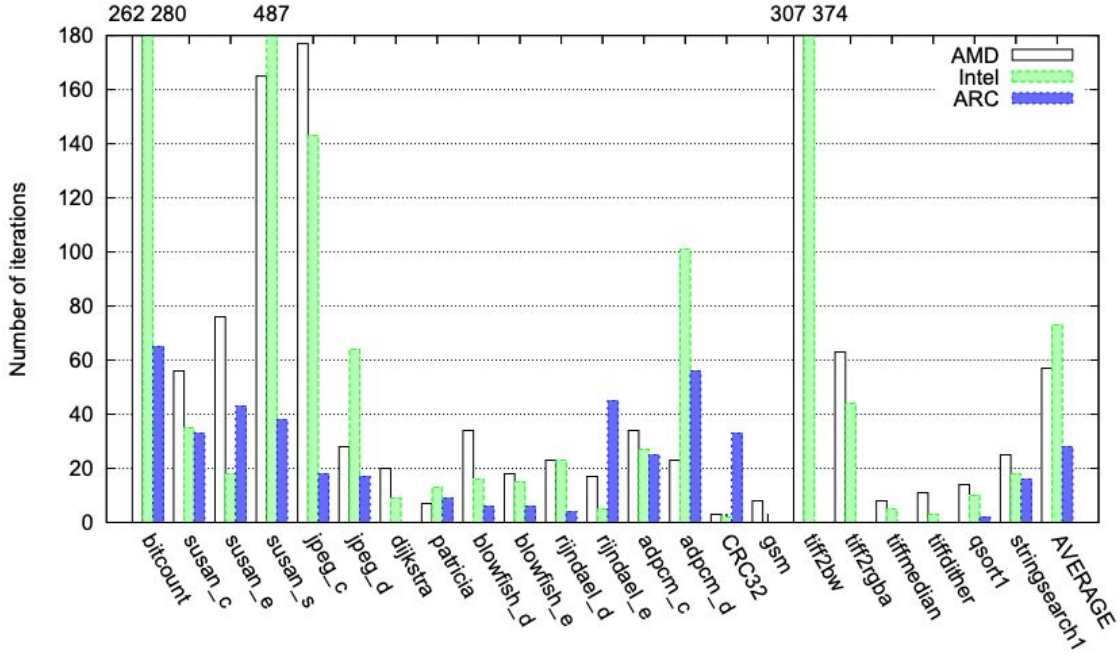



**Fig. 5** *Number of iterations needed to obtain 95% of the available speedup using iterative compilation with uniform random distribution.*

# Solution - Machine Learning

Allows for the potential of reusing knowledge across iterative compilation runs

- benefits of iterative compilation
- reducing the number of execution trials needed to achieve good solution

# Milepost

optimizes programs for configurable heterogeneous processors from correlation between program features, run-time behavior, optimizations

Interactive Compilation Interface separates optimization from compiler

- middleware between compilers (GCC) and user definable research plugins
- allows feature extraction module, selecting arbitrary optimization passes
- compiler independent, will allow transfer of Milepost to other compilers

Connected Milepost GCC to cTuning.org

- continuously updated training data from multiple users, environments

# GCC

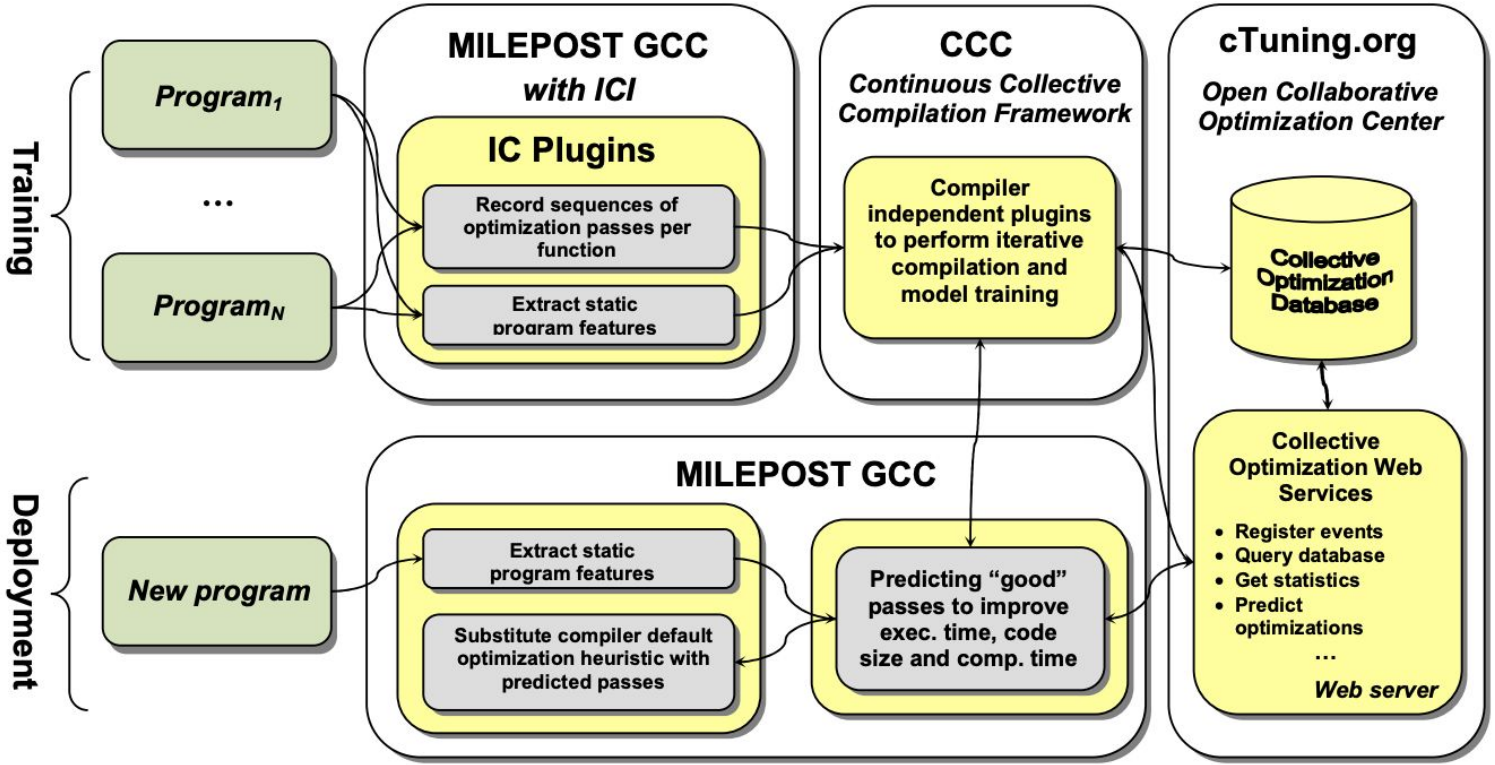
Mature and popular open-source optimizing compiler

Supports many languages, competitive with best commercial compilers

Features large number of program transformation techniques

Developed techniques are not compiler dependent

# Milepost Schema



# Features (Expert Intuition)

| | |
|---|---|
| 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 then 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 then 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 then two successors and more then two predecessors |
| ft13 | Number of basic blocks with number of instructions less then 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 |
| 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] |

# Probabilistic Model vs Transductive Model

Two machine learning techniques to select combinations of optimization passes

Probabilistic model

- Assumes each attribute is independent
- Finds closest program from the training set to the test program

Transductive model

- Analyzes interdependencies between attributes
- generalizes and identifies good combinations of flags, program attributes

# Probabilistic Inference over Distributions

set the predictive distribution q(x|t, θ) to be the distribution corresponding to the training program that is closest in feature space to the new (test) program.

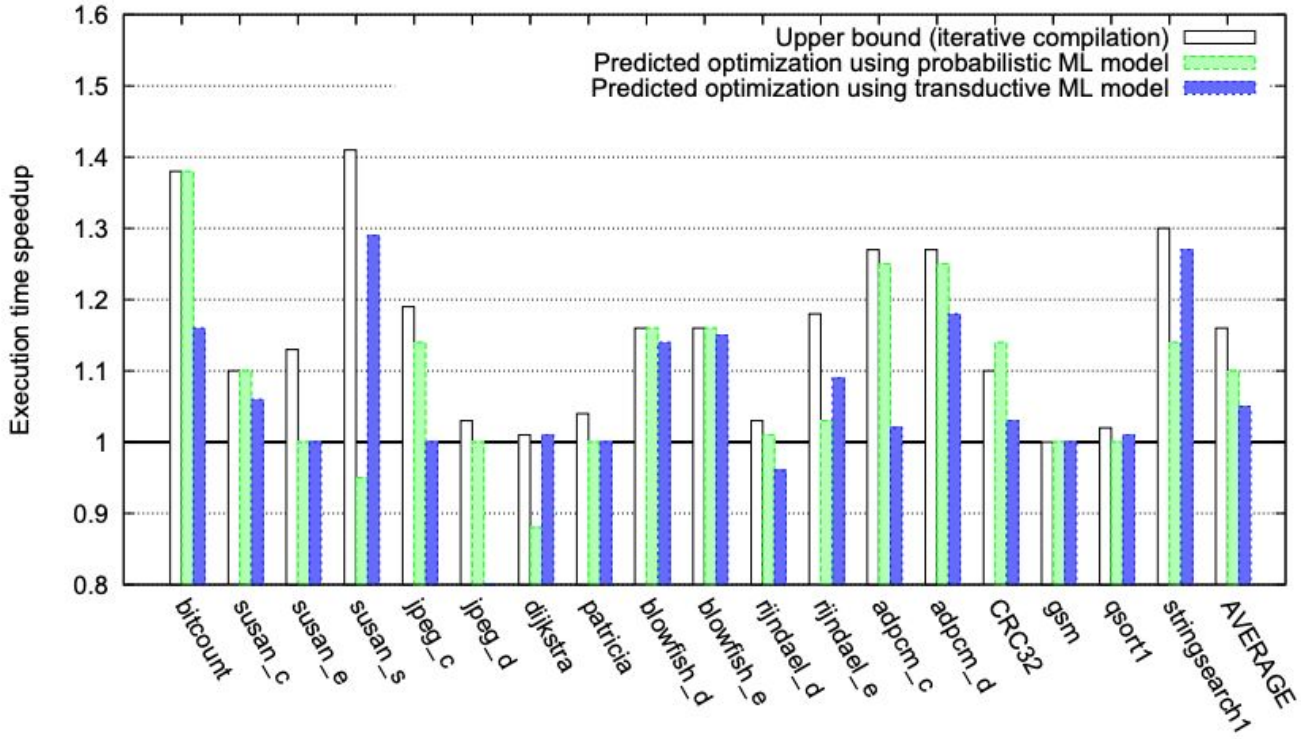
$$P(\mathbf{x}|T^j) = \prod_{\ell=1}^{L} P(x_\ell|T^j),$$

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x}} q(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta}).$$

# Transductive inference

- variables:
  - **x** (compiler settings vector)
  - **t** (task vector)
  - **y** (optimizer target e.g. run time)
- train regression tree model **y = f(concat(x,t))**

# Results


Upper bound (iterative compilation)
Predicted optimization using probabilistic ML model
Predicted optimization using transductive ML model
Execution time speedup
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
bitcount
susan_c
susan_e
susan_s
jpeg_c
jpeg_d
dijkstra
patricia
blowfish_d
blowfish_e
rijndael_d
rijndael_e
adpcm_c
adpcm_d
CRC32
gsm
qsort1
stringsearch1
AVERAGE