A Bayesian Network Approach to Compiler Auto-Tuning for Embedded Processors

Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Cristina Silvano
Politecnico di Milano
Compilers are usually considered as black boxes due to their complexity
  ► Including configurable knobs

Pre-build configurations exists
  ► E.g. -O1 -O2 -Os -O3 ...

Configuration tuning can give great energy/performance advantages in embedded systems

Problems
  ► Huge space ($2^n - n$ optimizations to enable/disable)
  ► Impact of compiler optimization
    • Have complex effects that only compiler guys can understand
    • Is strongly application and architecture dependent
To give a more Application-Specific meaning to

“On average it works well”
Related Works (1)

- State of the art approaches faced the problem by using auto-tuning techniques

- Iterative compilation strategies
  - Several recompilation of the application with different optimization flags and than choosing the one providing the best performance
  - High overhead due to a time consuming procedure
    - Optimization Algorithms
    - Statistical methods
Related Works (2)

- Machine learning techniques
  - Use predictive models to decide the optimization flags to be activated
  - The performance usually worse than in the previous case

- Phase Ordering
  - Not only deciding the compiler flag inclusions (as for GCC)
  - But also ordering and possible repetitions (as for LLVM)
  - Huge design space exploration space
    - Mixed Iterative-ML approach
PROPOSED APPROACH
High-level View

- 2 Level approach:
  - Feature Extraction
  - Prediction Model

- Generates a parametric representation of the target application

- Model trained after an architecture-compiler dependent characterization phase

Diagram:
- Applications
- Data Sets
- Application Features Extraction
- Probabilistic Model
- Sampling Technique
- Application-specific sequences of Compiler transformations
Programs can be classified thanks to features that can be extracted statically or dynamically.

Dynamically extracted features are data-set dependent but are able to extract the actual behavior of a program.

Possibility to exploit application similarities.
Features Extraction - MICA

- Micro-architectural Independent Characterization of Applications (MICA)
  - PIN-tool for characterizing applications in a cross-platform manner without relying to HW counters

- Instruction Mix
- Instruction-level parallelism
- Register traffic
- Working-set size
- Data stream strides
- Branch predictability

- E.g. Percentage of loads, stores, branches, ALUs ...
- E.g. Distance between production and consumption of a register instance
- E.g. Difference in memory addresses between two consecutive accesses
MICA analysis includes too many application features

*Principal Component Analysis (PCA)* to reduce the number of parameters to characterize the application

- Obtains uncorrelated features
- Hard to interpret in terms of original program characteristics
Bayesian Framework

- Problem: Does not exists a function
  - $f(\alpha(\text{App})) = o'$ where
  - $D(\text{App}, o') = \min_i(D(\text{App}, o_i))$

- There are unknown features not captured by $\alpha(\text{App})$
- However $f'(\alpha(\text{App})) = o'$

- Def. Bayesian networks are a powerful tool to model a *certain phenomenon* by means of the probability distributions of variables and their conditional dependencies

- **Certain phenomenon** -> Compiler Opt. Sequence
Bayesian networks have the following features of interest for the target problem:

- Include the possibility to manage heterogeneous variables
  - Boolean Variables – e.g. optimization vector $\mathbf{o}$
  - Continuous Variables – e.g. application characterization $\mathbf{\alpha}$

- Suitable to model cause-effect dependencies
  - E.g. Some compiler optimizations benefits by some features or by the presence of other compiler optimizations

- It's possible to include some a priori knowledge
  - E.g. Compiler Designer Knowledge

- The probability distribution of some variable (e.g. $\mathbf{o}$) depends on evidence on other variables (e.g. $\mathbf{\alpha}$)
  - Application-specific (depending $\mathbf{\alpha}$) distribution of the vector $\mathbf{o}$
Bayesian Model – Structure

- Direct Acyclic Graph (DAG)
- Nodes = Variables
- Edges = Dependencies

The dependency between 2 variables is modeled as a conditional probability: \( P(o_1 = \text{TRUE} \mid \alpha_1 = x) \)

Application Features
- “Controllable Variables”
  - E.g. Number of loop instructions

Compiler Optimizations
- “Observable Variables”
  - E.g. \( O_1 \) Loop Unrolling
  - \( O_2 \) Loop Tiling
Bayesian Model – Inference

Video App

MICA

\[ \alpha_1 = x \]

\[ P(o_1 = \text{TRUE} \mid \alpha_1 = x) \]

\[ P(o_1 = \text{FALSE} \mid \alpha_1 = x) \]

\[ P(o_2 = \text{TRUE} \mid \alpha_1 = x \land o_1 = \text{TRUE}) \]

\[ P(o_2 = \text{FALSE} \mid \alpha_1 = x \land o_1 = \text{TRUE}) \]

Flip a Coin

\[ o_1 = \text{TRUE} \]

\[ o_2 = \text{FALSE} \]
Proposed Framework – Training

- Features Extraction
- Bayesian net. training
- Uniform distribution
- Optimal Solution
- Sampling
- 15% Selection

Proposed framework

Training application set

Trained model
Proposed Framework – Optimization
EXPERIMENTAL RESULTS
Experimental Setup

- ARMv7 Cortex A9 architecture within a TI- OMAP4430
- Gcc-ARM 4.6.3 and ArchLinux
- cBench as Benchmark suite
  - 24 apps and 5 datasets
- 7 Compiler Transformations
  - Selected from [Chen12taco] where the speed-up > 1.1
  - Baseline -O3
  - 10 executions
    - *Linux-Perf* tool for performance evaluation

Principal Components of Application Features

PC.1 → PC.2 → PC.3
PC.4 → PC.5 → PC.6
PC.7 → PC.8 → PC.9 → PC.10

Compiler Transformations

- Loop unrolling impacts on branch probability
- Tree-based optimizations are interdependent
Optimization Results

- 8 Extractions
- Speedup averaged on the 5 datasets
Comparative Analysis – Iterative Compilation (1)

Optimal Solution

-03

Normalized speed-up w.r.t. -O3

proposed opt

iterative compilation

number of extractions
Comparative Analysis – Iterative Compilation (2)

- Effect of the application characterization
  - With and without MICA

![Graph showing normalized speed-up with and without MICA](image)
Comparative Analysis – Iterative Compilation (3)

- Compilation effort
  - 8 extractions for the proposed approach
Conclusions

- We presented a framework for compiler auto-tuning
  - Not iterative & Based on a Bayesian model
- For small extractions provides great advantages
- A new meaning to “on average it works well”

Limitations:
- It requires a training phase for each target architecture
- Dataset-dependent since it is based on dynamic profiling
- Does not solve completely the problem
  - even with a small search space