Procedure-Level Performance Tuning of Whole Programs

Rudolf Eigenmann
ECE, Purdue University

Zhelong Pan
VMware Inc.
Why would we want to do this?

- Biggest barrier for compilers: lack of compile-time information
  
  => need techniques that gather data at runtime and dynamically optimize the program

- Performance potential: huge
- State of the art: a few 10%
1. Explore the optimization space
   (*Empirical optimization algorithm* - CGO 2006)

2. Comparing performance
   (*Fair Rating methods* - SC 2004)
   - Comparing two (differently optimized) subroutine invocations

3. Choosing procedures as tuning candidates
   (*Tuning section selection*)
   - Program partitioning into tuning sections

Two goals: increase program performance and reduce tuning time
1. Algorithms for Empirical Optimization - *Orchestration*

- **BE**: batch elimination
  - Eliminates “bad” optimizations in a batch => fast
  - Does not consider interaction => not effective
- **IE**: iterative elimination
  - Eliminates one “bad” optimization at a time greedily => slow
  - Considers interaction => effective
- **CE**: combined elimination (final algorithm)
  - Eliminates a few “bad” optimizations at a time greedily
- **Other algorithms**
  - optimization space exploration, statistical selection, genetic algorithm, random search

```
Start

Orchestration Algorithm

Version Generation

Performance Evaluation (Program Execution)

Final Version
```
CE has the fastest tuning speed, achieving equal or better program performance improvement.
2. Procedure-Level, Fair Rating

- **Rating**: Evaluating the performance of an optimized procedure version

- Tuning scenario:
  1. Execute the program
  2. Different procedure invocations use different optimizations
  3. Compare these procedure execution times, choose the best
    - Need to factor the workload into measured execution times
  4. Repeat this process until tuning complete
Fair Rating Methods

- CBR (Context Based Rating)
  - Average execution times under the same workload
- MBR (Model Based Rating)
  - Formulate relationship between different workloads
- RBR (Re-execution Based Rating)
  - Restore input and re-execute
  - Compare two versions under the same input

The compiler’s tasks
- determine workload, execution time model, input
- instrument program
3. Tuning Section Selection

- Carve tuning sections out of a program, so as to achieve
  - high program coverage => high performance
  - Large number of invocations => fast tuning
**gprof output**

- Number of invocations to callee
- Execution time spent in callee and its descendants

**TS:** (1) b (including d and e) 
(2) f

Coverage = \(\frac{80+18}{100} = 0.98\)

\[N_{\text{min}} = \min(1000,200) = 200\]
Implementation: The PEAK System

1. Tuning Section Selection (TSS)
2. Rating Method Analysis (RMA)
3. Code Instrumentation (CI)
4. Driver Generation (DG)
5. Performance Tuning (PT)
6. Final Version Generation (FVG)

Pre-Tuning:
- (1) Tuning Section Selection (TSS)
- (2) Rating Method Analysis (RMA)
- (3) Code Instrumentation (CI)
- (4) Driver Generation (DG)
- (5) Performance Tuning (PT)
- (6) Final Version Generation (FVG)

During Tuning:

Post-Tuning:
PEAK System Components

- PEAK compiler (Polaris, SUIF + Cetus)
  - Rating method analysis
  - Source-to-source translation
  - Add instrumentation which calls PEAK runtime
- PEAK runtime (library)
  - Dynamic code generation and loading
  - Performance rating
  - Optimization orchestration
- Tuning driver
  - Execute the program multiple times
  - Under a training input
  - Until the best combination of compiler flags is found for each tuning section
Experiments

- Train dataset to tune performance
- Ref dataset to measure performance
- SPEC 2000 benchmarks
- 38 GCC optimizations
PEAK is 20 times as fast as the whole-program tuning.

On average, PEAK reduces tuning time from 2.19 hours to 5.85 minutes.
Tuning Time Component

Percentage of the total time spent in tuning

- amp
- applu
- apsi
- art
- equake
- mesa
- mgrid
- sixtrack
- swim
- wupwise
- Average
Program Performance

The performance is the same.

SPEC 2000 FP
Conclusions

- Automatic tuning of compiler optimizations lead to 10% (avg) and up to 60% performance improvement.
- Procedure-level tuning methods tune the 32 GCC options for the average SPEC 2000 benchmarks in two application runs.
- Ongoing work:
  - Integer applications
  - Parallel applications
  - Continuous experiments
  - Library tuning