Mapping Parallelism to Multi-cores: A Machine Learning Based Approach

Zheng Wang and Michael O’Boyle
School of Informatics, Edinburgh University UK
European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC)
Exploiting Parallelism on Multi-cores

- Multi-core systems are here and exploiting parallelism is important
- Three steps for exploiting parallelism
  - 1. Discovering parallelism
  - 2. Expressing parallelism
  - 3. Mapping parallelism
    - Fixed Heuristics?
      - In practice, heuristics perform well on a architecture that it is particularly tuned for
      - ...but, requires a lot of tweaking when shifting to a new hardware
Previous Works

- Fixed heuristics
  - Likely to fail when things changed
- Runtime adaptation
  - Target-specific runtime adaptation
- Analytical models
  - Require low-level hardware and program details
- Online learning models
  - Expensive learning cost
Complex Mapping Decisions

Performance on 2x Quad-core Xeon processors

* A parallel loop example from FT (NAS parallel benchmark)
Complex Mapping Decisions

The sequential program that runs on the PPE

OpenMP runtime default scheme

*A parallel loop example from FT (NAS parallel benchmark)
Complex Mapping Decisions

Performance on 2x Quad-core Xeon processors

Performance on 2x Cell processors

*A parallel loop example from FT (NAS parallel benchmark)
Motivation

- Parallelism mapping is important but challenging
- Compiler heuristics rely on detailed knowledge of the system
- Architectures are complex
- Interactions among programs, runtime systems, and architectures are not well understood
The Problem

- Finding optimal parallelism schemes
  - The number of threads
  - Selecting four OpenMP scheduling policies: BLOCK, CYCLIC, DYNAMIC and GUIDED
    - **BLOCK**: Iterations are divided into chunks of size \( \text{ceiling}(\text{number_of_iterations}/\text{number_of_threads}) \). *Each thread is assigned a separate chunk.*
    - **CYCLIC**: Iterations are divided into chunks of size 1 and each chunk is assigned to a thread in *round-robin* fashion.
    - **DYNAMIC**: Iterations are divided into chunks of size \( \text{ceiling}(\text{number_of_iterations}/\text{number_of_threads}) \). *Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available.*
    - **GUIDED**: Chunks are made progressively smaller until the default minimum chunk size (1) is reached.
An analytical model for IBM Cell BE (F. Blagojevic, HiPEAC’2008)

\[ T = a \cdot T_{\text{HPU}} + \frac{T_{\text{APU}}}{p} + C_{\text{APU}} + p \cdot (O_L + T_S + O_C + p \cdot g) \]

- 9 parameters for typical parallel programs
  - Assuming load is balanced
  - Need to find new parameters/values for a new platform
  - Could we do better with an automatic and systematic approach to tune heuristics?
Supervised Learning

- Supervised learning algorithms try to find a function $F(X) \rightarrow Y$
  - $X$: vector of program and architecture characteristics (features)
  - $Y$: empirically found best parallel mappings (thread numbers and scheduling policies)
Machine Learning Models

- Two learning algorithms fit our problems well
  - Artificial Neural Network (ANN)
    - Predicts the scalability -> selects the number of threads
  - Support Vector Machine (SVM)
    - Classifies scheduling policies -> selects the scheduling policy
- Both algorithms solve problems quickly
  - Train at the factory
The Artificial Neural Network Model

The predictive best thread number

Worker thread number

Actual performance --- Predicted performance

Speedup

features

hidden neurons

output
The Support Vector Machine Model

- Map the original feature space into a higher-dimensional space
- Find hyper-planes that maximally separate the data

Select BLOCK scheduling  Don’t select Block scheduling

# feature 1 (i.e. FP operations ratio)
# feature 2 (i.e. D1 Cache miss rate)

Hyper-plane 1

# feature 1 (i.e. $f' = f \times X - b$)
# feature 2

(i.e. $f' = f \times X - b$)
Features

- Features are the inputs of a machine learning model
- Start with features that might be important
- Small feature sets are better
  - Learning algorithms run faster
  - Are less prone to over-fitting the training data
  - Useless features can confuse learning algorithms
Selected Features

Static Code Features
- Operations
- Branches
- Memory operations

Dynamic features
- Loop iteration count
- L1 Dcache miss rate

Runtime feature
- Parallel execution time

Extracted from LLVM IR

Profiled with the smallest input data set
Our Approach

- **Training**
  - Off-line training at the factory

- **Deployment**
  - Apply trained models for prediction
Two Predictors

- A data sensitive (DS) predictor
  - Profile the program once with each input data set
- A data insensitive (DI) predictor
  - Profile the program once with the *smallest* input data set
- Profiling runs for each predictor with $N$ input data sets

<table>
<thead>
<tr>
<th>Model</th>
<th>Profiling with the sequential program</th>
<th>Profiling with the parallel program</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DS</td>
<td>$N$</td>
<td>1</td>
</tr>
</tbody>
</table>
Evaluation Metrics

- **Stability**
  - Can our approach have stable performance across programs?

- **Portability**
  - Is our approach portable to different architectures?

- **Overhead**
  - How much overhead do we have compared to analytical and online learning models?
Experimental Setup

- **Platforms**
  - 2x Quad-core 3.0 GHz Intel Xeon processors
  - 2x 3.2 GHz Cell processors

- **Compilers and runtimes**
  - Intel icc 10.1
  - IBM xlc single source compiler for Cell v0.9

- **Benchmarks**
  - Programs from NAS parallel benchmark, UTDSP and Mibench

  **Leave-one-out cross validation**

- **Compare with two recent models**
  - An analytical model for the Cell processor (F. Blagojevic, HiPEAC’08)
  - A regression-based model (B. Barnes, ICS’08)
Outline of Results

• Stability and Portability
  • Consistently stable performance across programs
  • On average, above 96% performance of the upper bound on both platforms

• Lower overhead
  • Reduce the profiling cost for a new program by a factor between 4x and 512x

• On average, the data insensitive (DI) predictor performs as well as the data sensitive (DS) predictor
  • Adapt to input data sets with very low profiling cost
Comparison with the Default Scheme

• Performance on the Xeon platform

The performance ratio is averaged across data sets for each parallel loop.
Comparison with the Default Scheme

- Performance on the Cell platform

The performance ratio is averaged across data sets for each parallel loop.
Comparison with Other Techniques

- Performance on the Xeon platform

The performance ratio is averaged across data sets for each parallel loop.
Comparison with Other Techniques (cont.)

Performance on the Cell platform

The performance ratio is averaged across data sets for each parallel loop.
Profiling Overhead

- Profiling overhead on the Xeon platform

![Graph showing profiling overhead for various benchmarks](image)

- **DS** vs **regression-based**

- Absolute profiling overhead (%)

- Benchmarks: NPB.BT.L1, NPB.BT.L2, NPB.BT.L3, NPB.CG.L1, NPB.CG.L2, NPB.CG.L3, NPB.EP, NPB.FT.L1, NPB.FT.L2, NPB.FT.L3, NPB.is, NPB.LU.L1, NPB.LU.L2, NPB.LU.L3, NPB.MG.L1, NPB.MG.L2, NPB.MG.L3, NPB.SP.L1, NPB.SP.L2, NPB.SP.L3, AVERAGE
Profiling Overhead (cont.)

- Profiling overhead on the Cell platform

![Graph showing profiling overheads for various benchmarks with DS, regression-based, and MMGP categories.](image-url)
Conclusions

• A portable, and automatic compiler-based approach
  • Models are automatically constructed and trained off-line.
  • Stable performance across programs and architectures
• Low profiling cost relative to other techniques
• Let an off-line machine learning model build heuristics for us.
QUESTIONS?

- Acknowledgements
  - European MILEPOST and SARC projects
  - Barcelona Supercomputing Centre
  - The Edinburgh Compute and Data Facility
Stability

The shorter the whisker, the better stability is.

Performance gap to the upper bound on the Xeon platform

Performance gap to the upper bound on the Cell platform
Profiling Runs

- Profiling runs for each model with N input data sets and M scheduling policies

<table>
<thead>
<tr>
<th>Model</th>
<th>Profiling with the sequential program</th>
<th>Profiling with the parallel program</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DS</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Regression-based</td>
<td>N</td>
<td>M * N</td>
</tr>
<tr>
<td>MMGP</td>
<td>N</td>
<td>M * N</td>
</tr>
</tbody>
</table>
## Profiling Cost

- **Absolute profiling cost**

<table>
<thead>
<tr>
<th>Model</th>
<th>Intel</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>4.79sec</td>
<td>4.80mins</td>
</tr>
<tr>
<td>DS</td>
<td>13.10mins</td>
<td>1.75hours</td>
</tr>
<tr>
<td>Regression-based</td>
<td>59mins</td>
<td>16.45hours</td>
</tr>
<tr>
<td>MMGP</td>
<td>N.A.</td>
<td>41hours</td>
</tr>
</tbody>
</table>
Hurdles

• Compiler writer must extract features
• Generating data and model training need time
  • ~2 days to collect data on four machines
  • ~2 weeks to build models
• Prediction quality is highly dependent on the training data
• Have to tweak learning algorithms