

# 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)



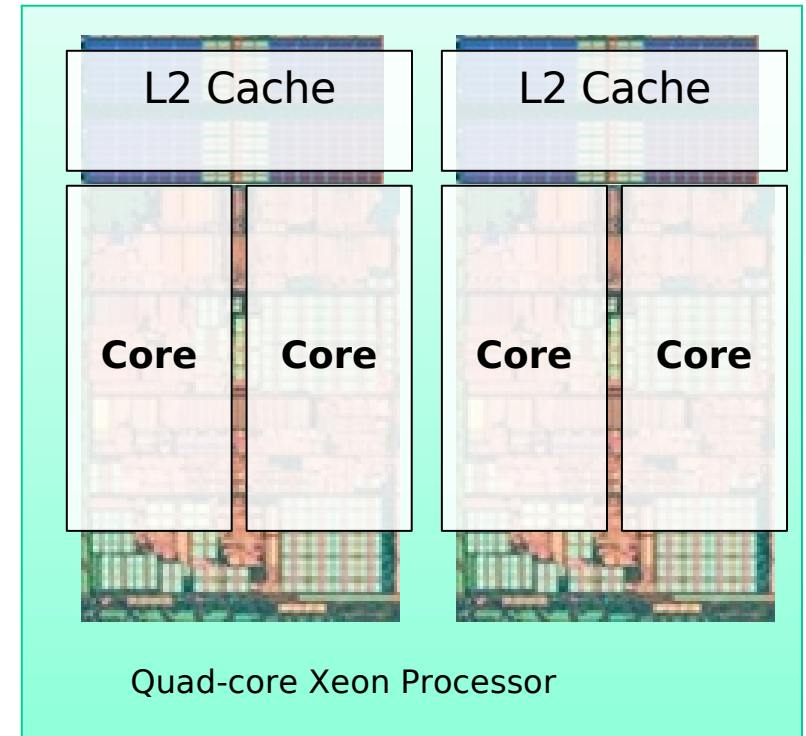
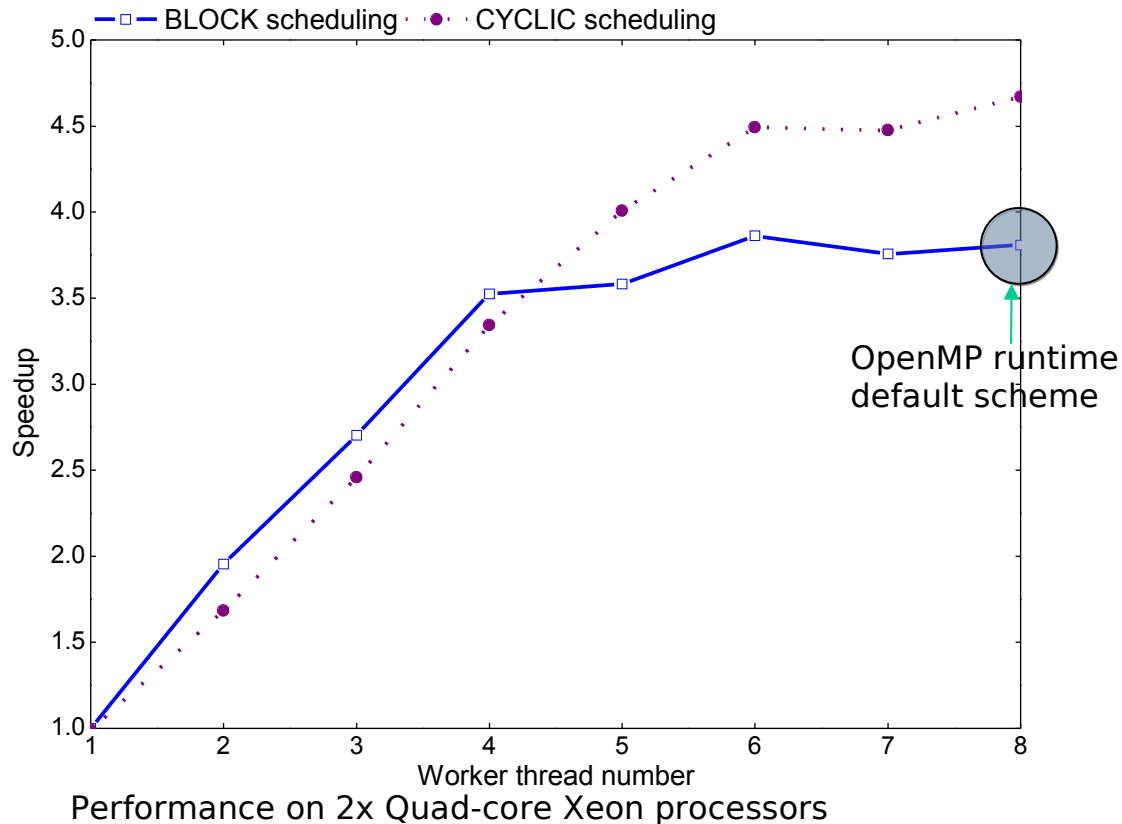
# Exploiting Parallelism on Multi-cores

- 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

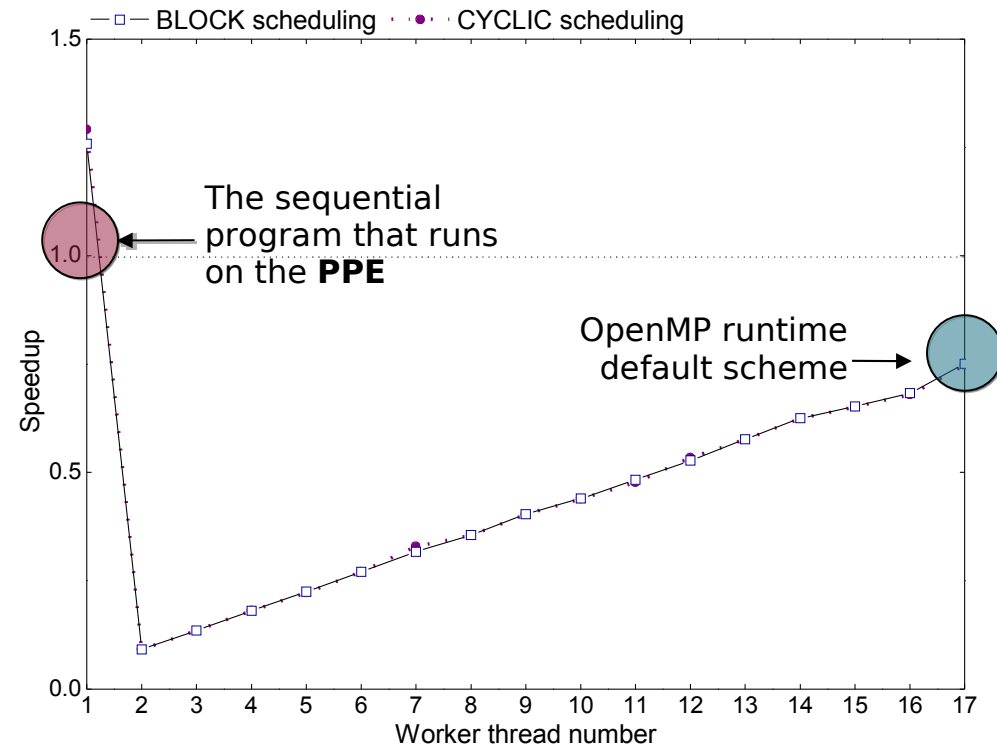
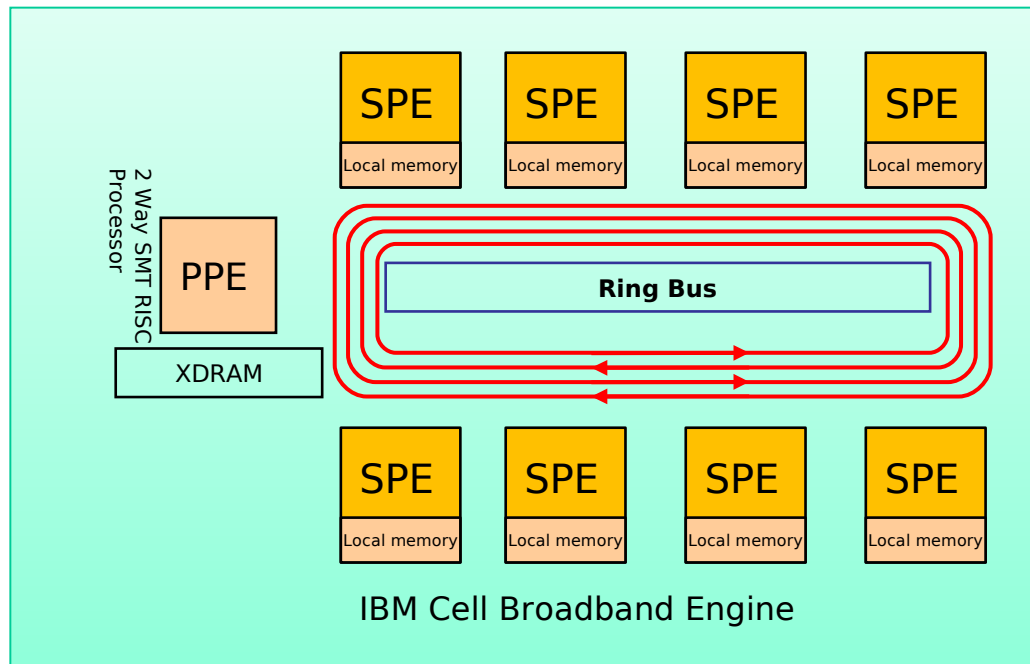
- Fixed heuristics
  - Likely to fail when things changed
- Runtime adaption
  - Target-specific runtime adaptation
- Analytical models
  - Require low-level hardware and program details
- Online learning models
  - Expensive learning cost

# Complex Mapping Decisions



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

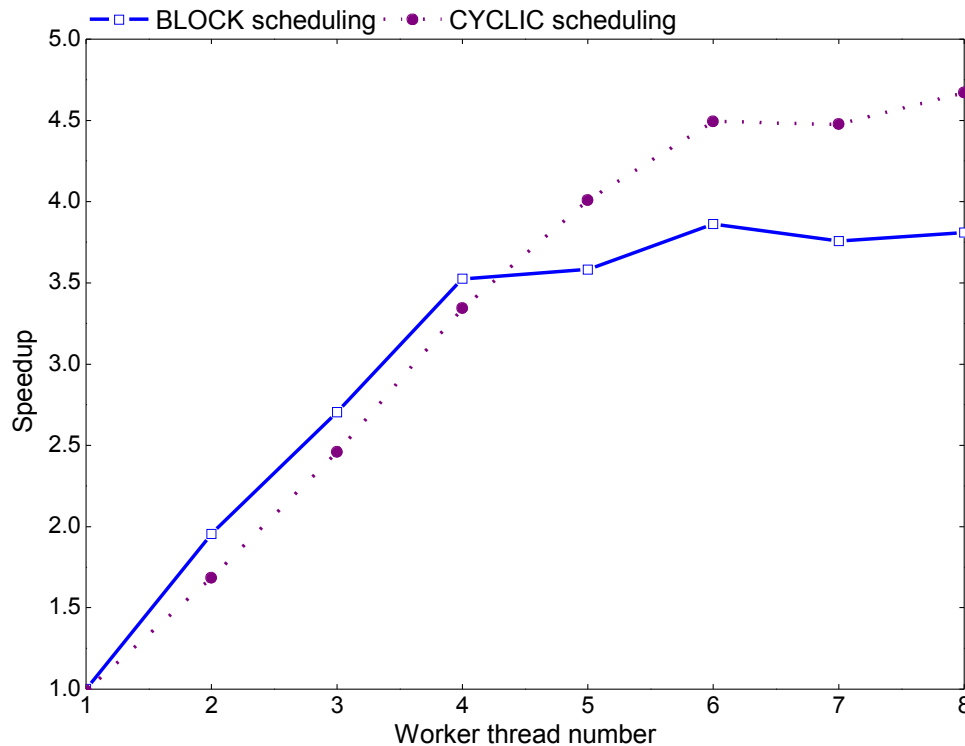
# Complex Mapping Decisions



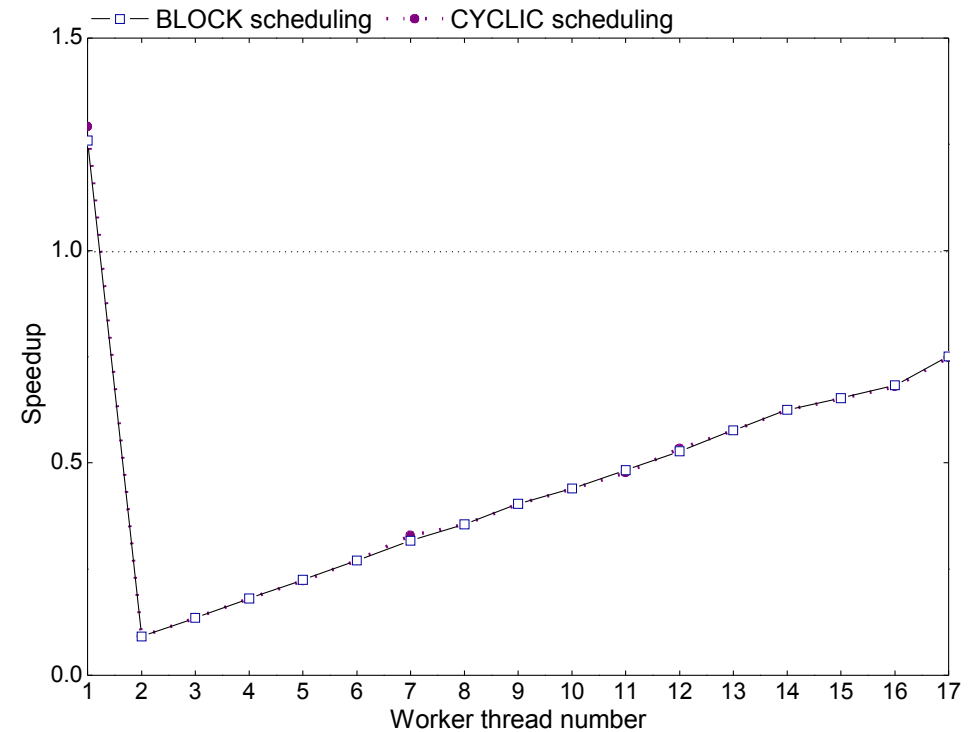
Performance on 2x Cell processors

\*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.



# Fixed Heuristics

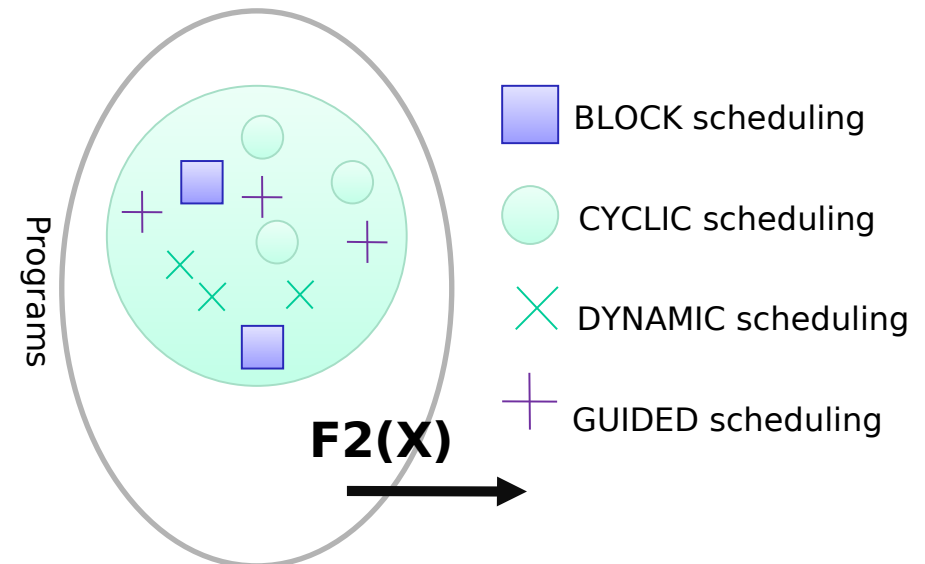
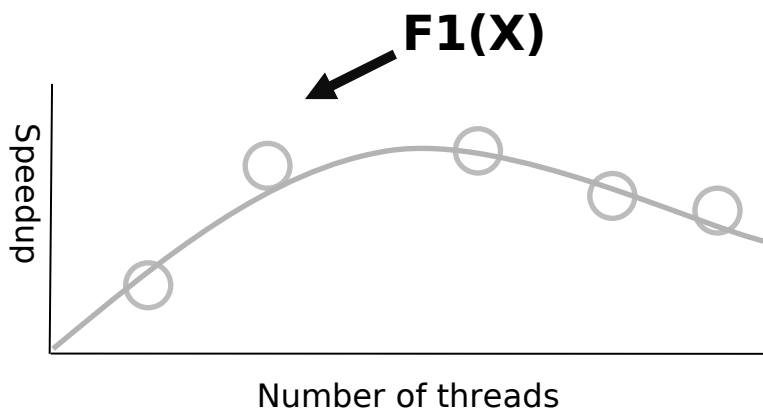
- An analytical model for IBM Cell BE (F. Blagojevic, HiPEAC'2008)

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

- 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

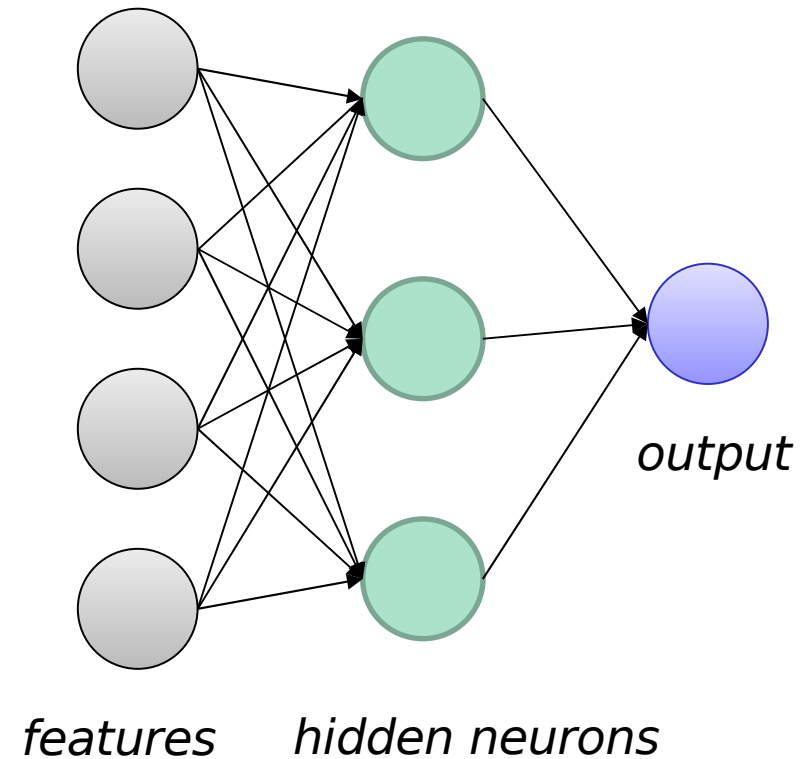
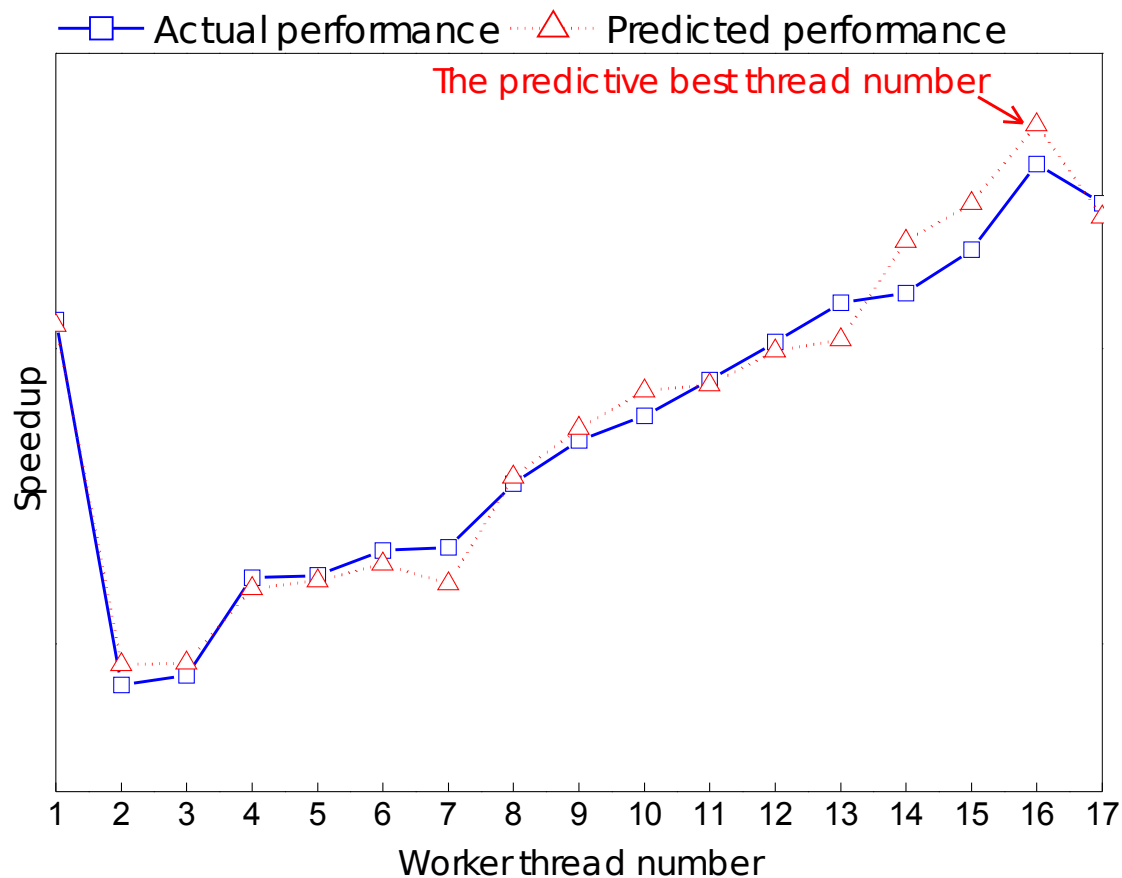
- Supervised learning algorithms try to find a function  $F(X) \rightarrow Y$ 
  - $X$  : vector of program and architecture characteristics (features)
  - $Y$  : empirically found best parallel mappings (thread numbers and scheduling policies)



# Machine Learning Models

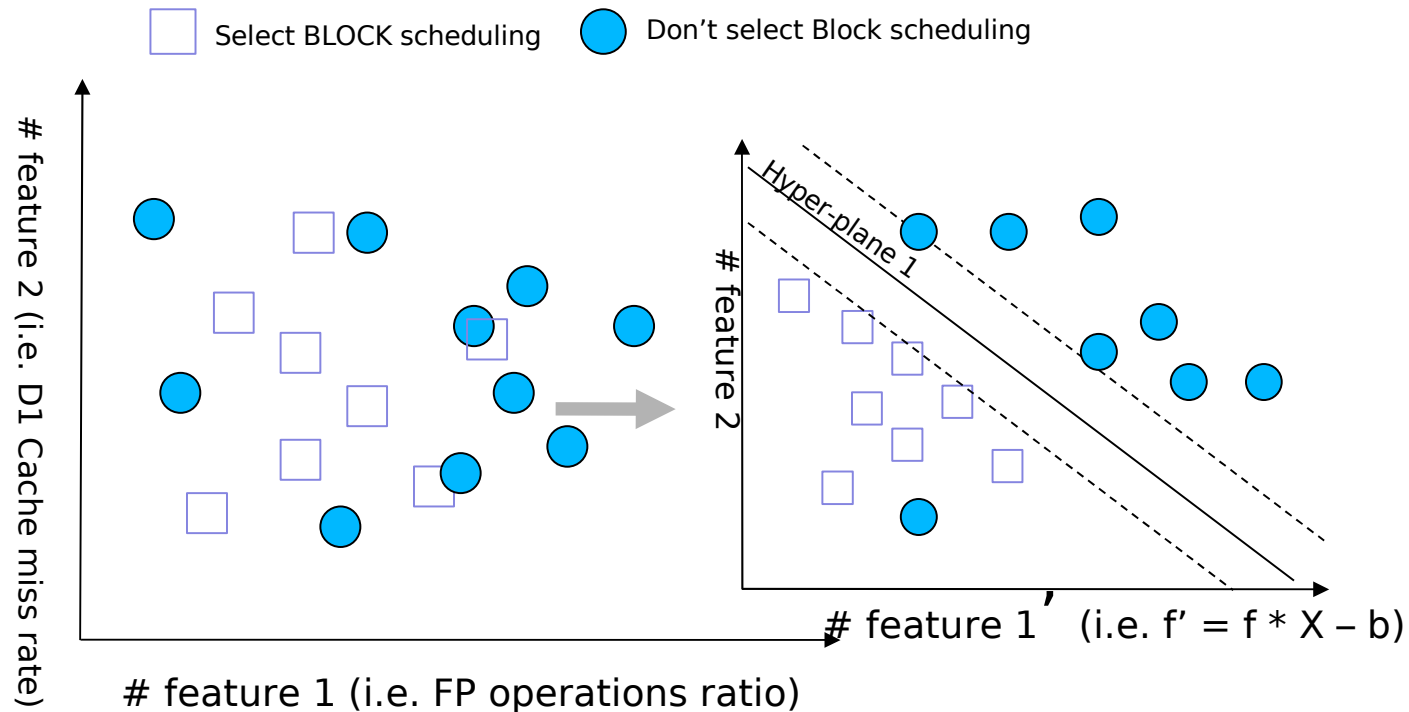
- Two learning algorithms fit our problems well
  - Artificial Neural Network (ANN)
    - Predicts the scalability -> selects the number of threads
  - Support Vector Machine (SVM)
    - Classifies scheduling policies -> selects the scheduling policy
- Both algorithms solve problems quickly
  - Train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

# The Artificial Neural Network Model



# The Support Vector Machine Model

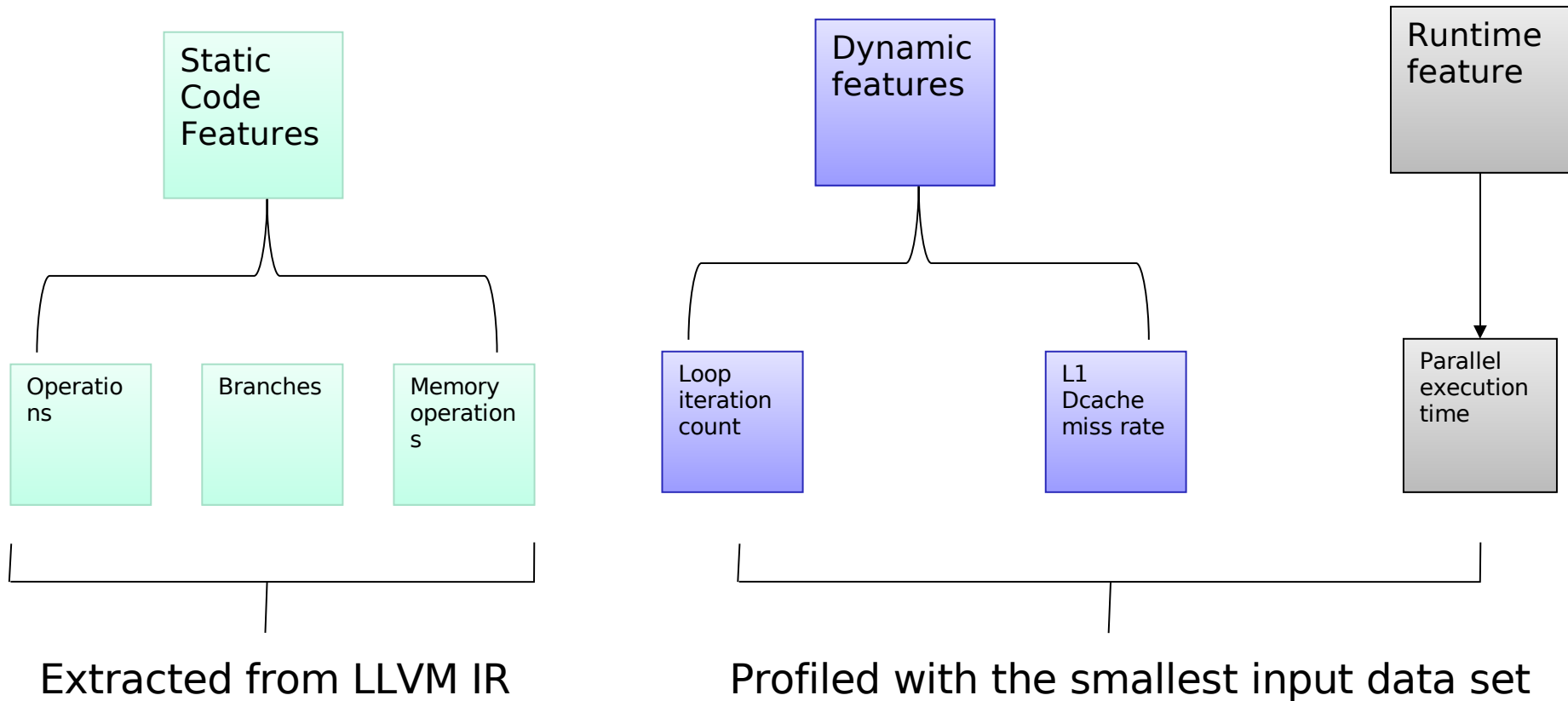
- Map the original feature space into a higher-dimensional space
- Find hyper-planes that maximally separate the data



# Features

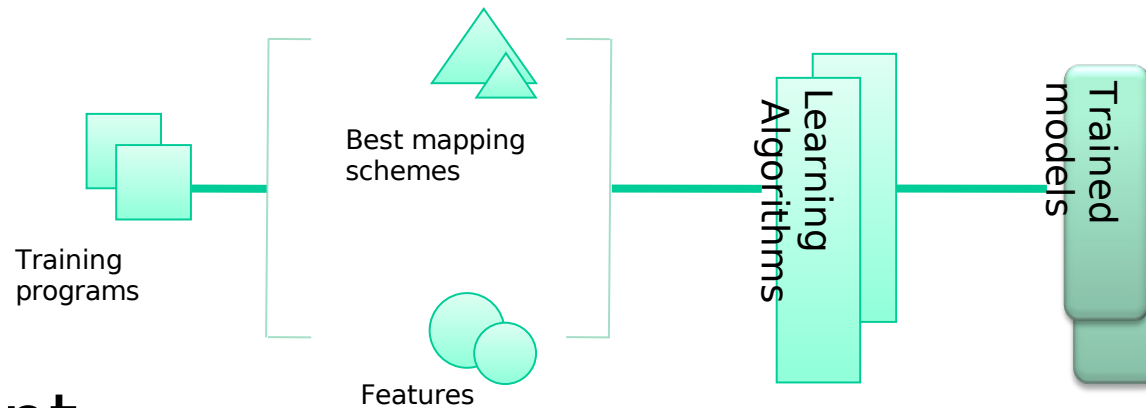
- Features are the inputs of a machine learning model
- Start with features that might be important
- Small feature sets are better
  - Learning algorithms run faster
  - Are less prone to over-fitting the training data
  - Useless features can confuse learning algorithms

# Selected Features

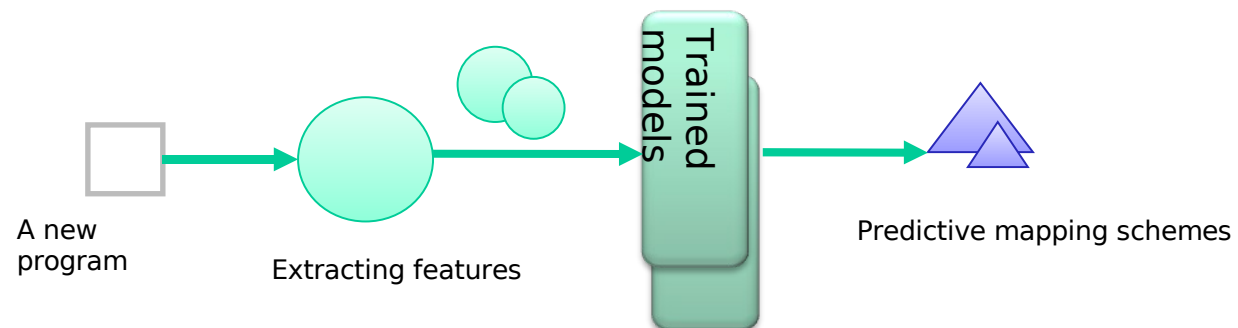


# Our Approach

- 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

Model	Profiling with the sequential program	Profiling with the parallel program
DI	1	1
DS	$N$	1

# 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

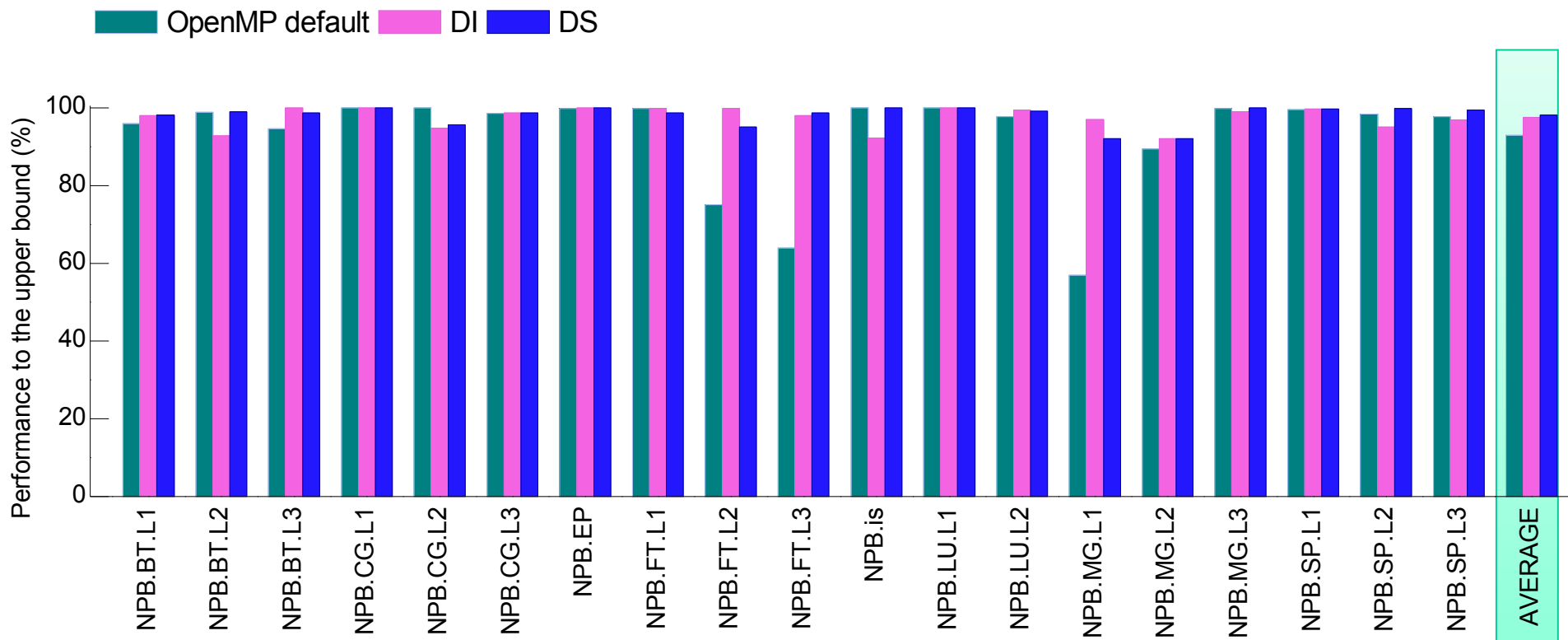
- Platforms
    - 2x Quad-core 3.0 GHz Intel Xeon processors
    - 2x 3.2 GHz Cell processors
  - Compilers and runtimes
    - Intel icc 10.1
    - IBM xlc single source compiler for Cell v0.9
  - Benchmarks
    - Programs from NAS parallel benchmark, UTDSP and Mibench
  - **Leave-one-out cross validation**
  - 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

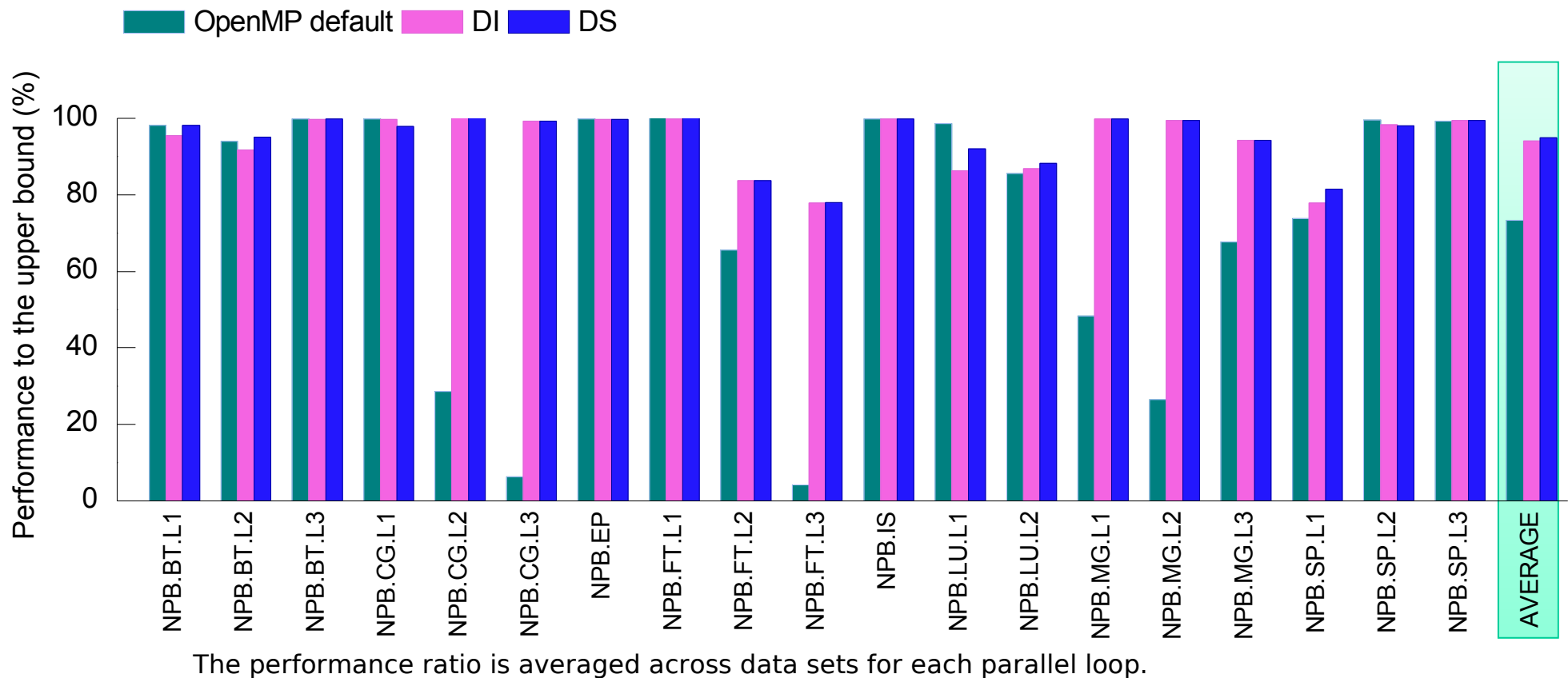
- Performance on the Xeon platform



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

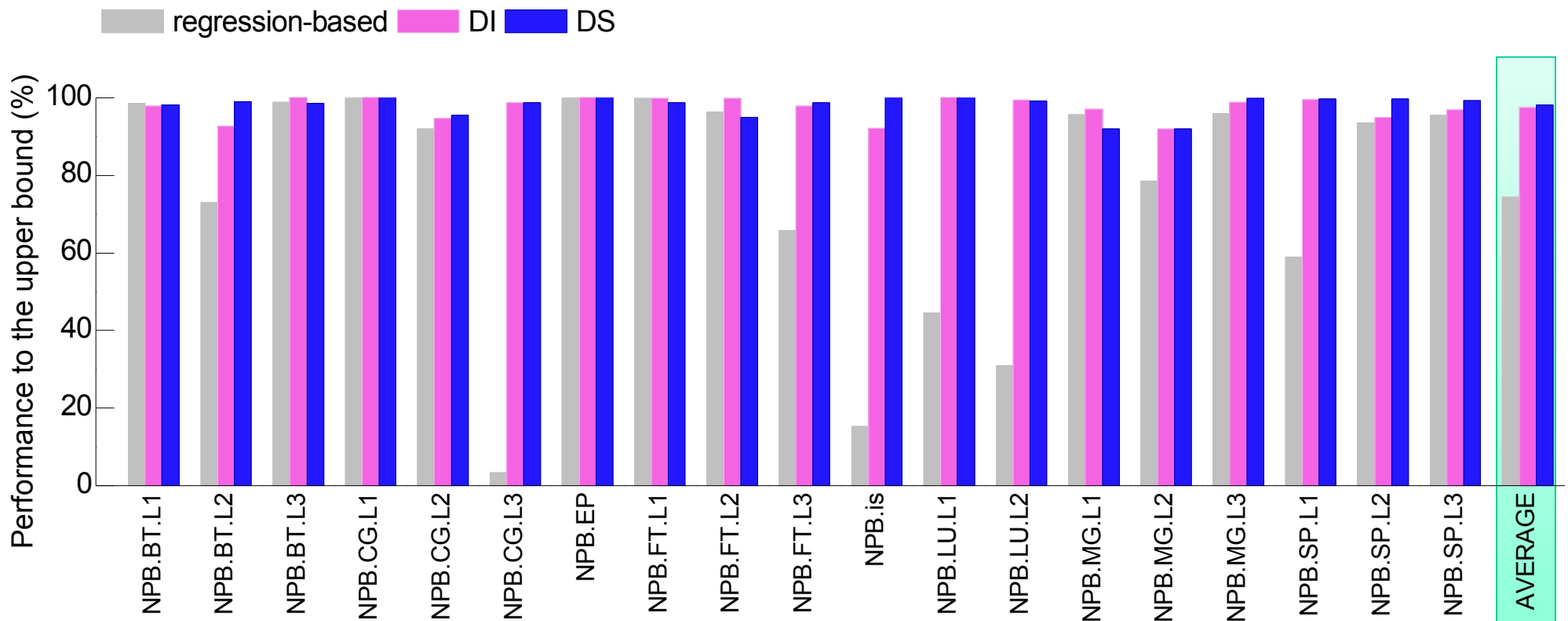
# Comparison with the Default Scheme

- Performance on the Cell platform



# Comparison with Other Techniques

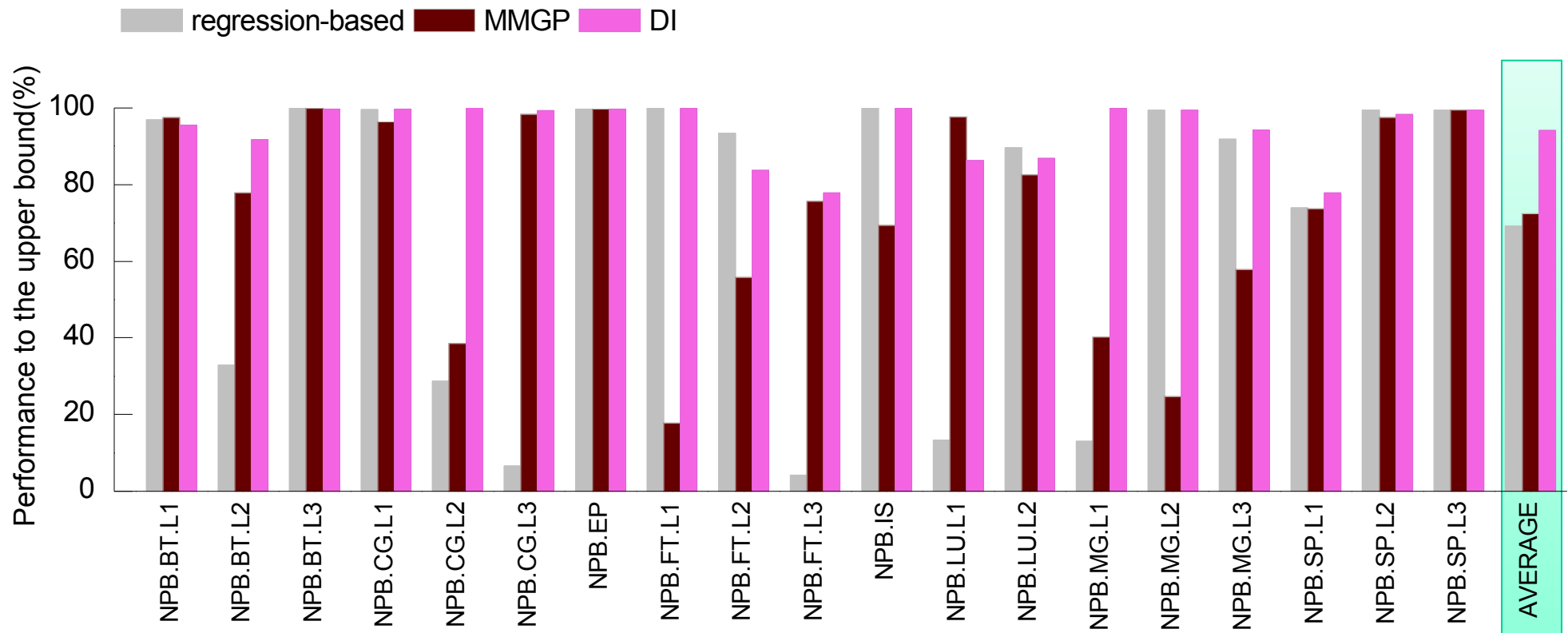
- Performance on the Xeon platform



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

# Comparison with Other Techniques(cont.)

- Performance on the Cell platform

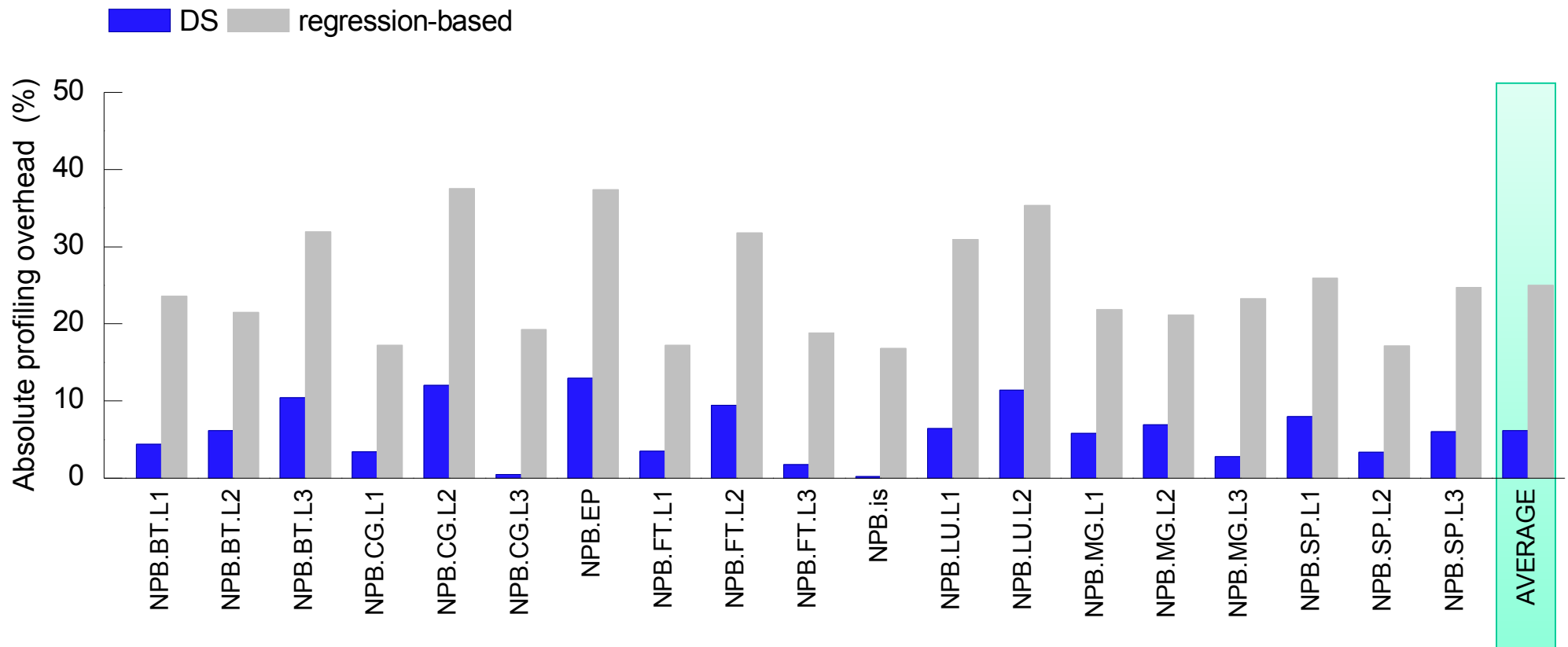


The performance ratio is averaged across data sets for each parallel loop.



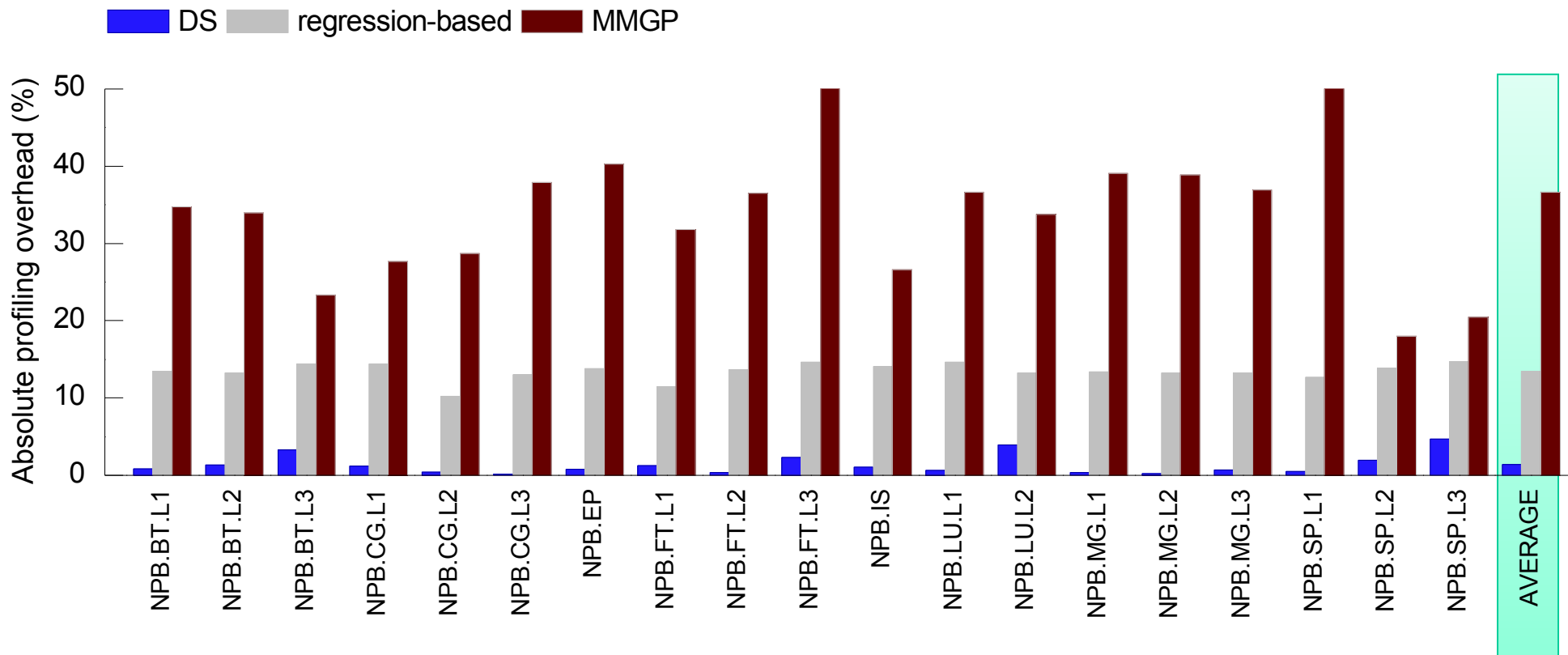
# Profiling Overhead

- Profiling overhead on the Xeon platform



# Profiling Overhead (cont.)

- Profiling overhead on the Cell platform

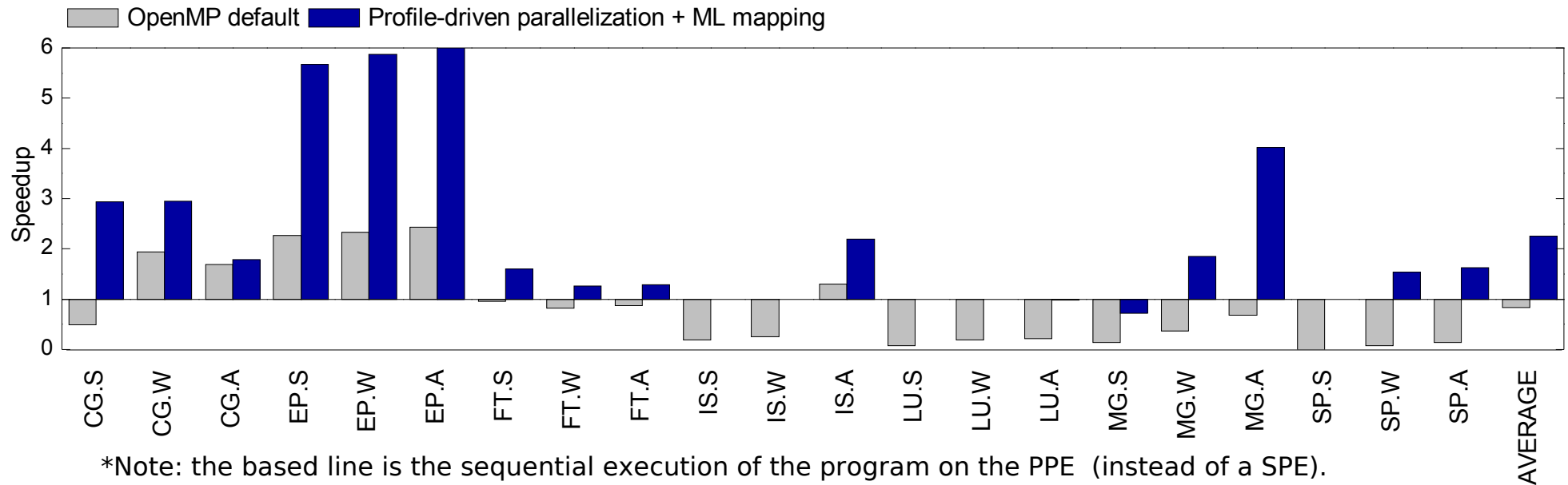


# Conclusions

- 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.

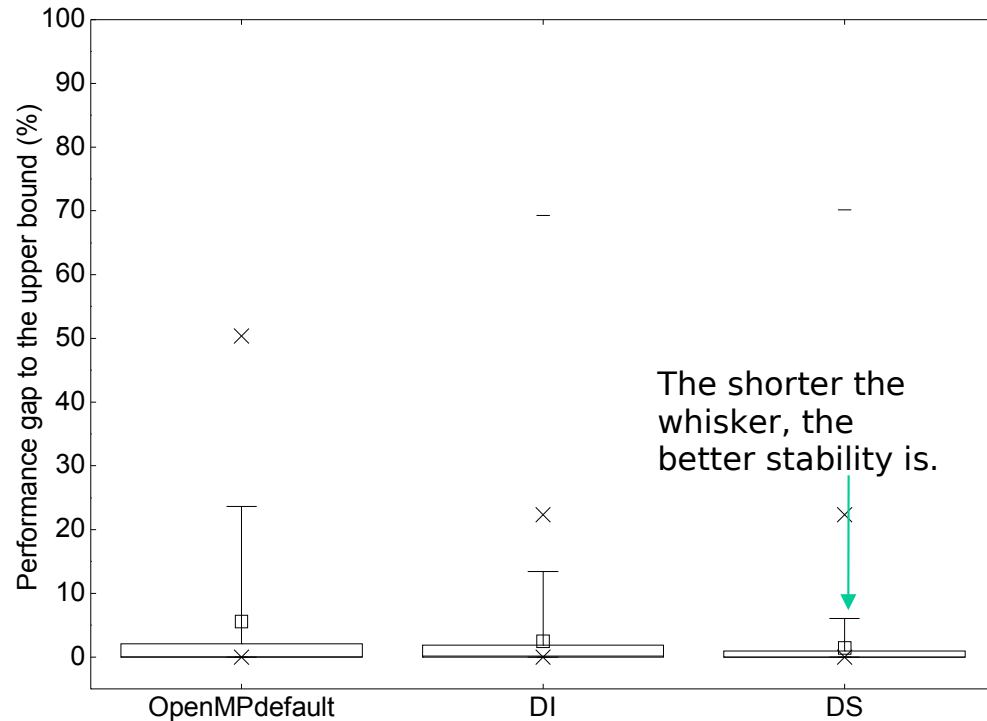
# QUESTIONS?

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

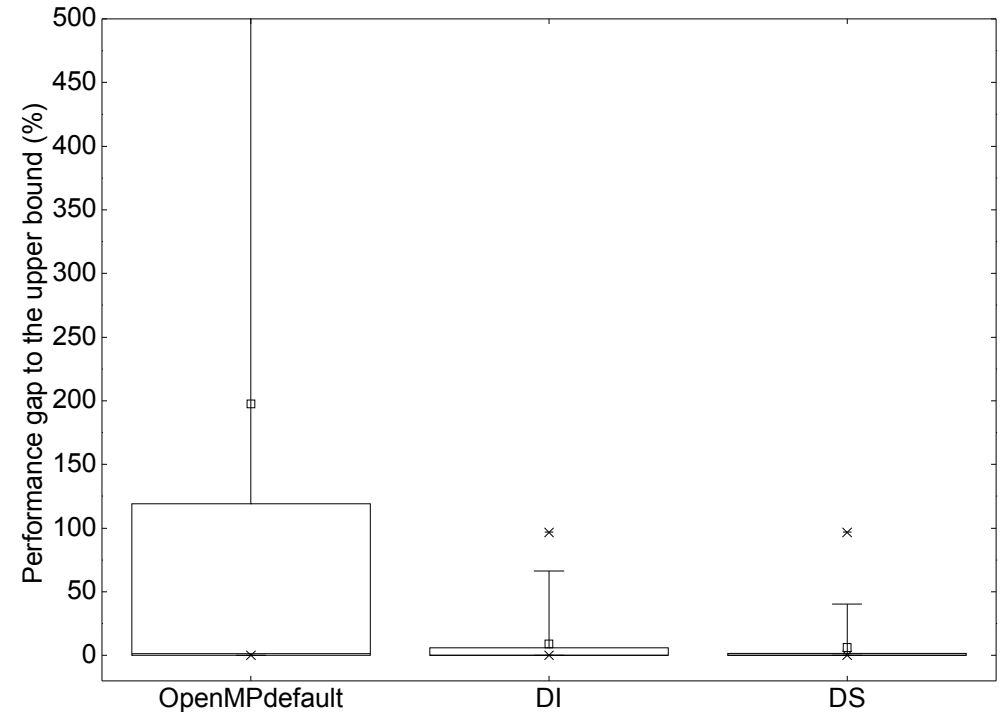


Georgios Tournavitis, Zheng Wang, Bjorn Franke and Michael O'Boyle.  
**Towards a Holistic Approach to Auto-Parallelization** - *Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping*, In PLDI 2009 (to appear).

# Stability



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

Model	Profiling with the sequential program	Profiling with the parallel program
DI	1	1
DS	N	1
Regression-based	N	$M * N$
MMGP	N	$M * N$

# Profiling Cost

- Absolute profiling cost

Model	Intel	Cell
DI	4.79sec	4.80mins
DS	13.10mins	1.75hours
Regression-based	59mins	16.45hours
MMGP	N.A.	41hours



# 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