Towards Dynamically Adaptive Programs a.k.a. autotuning

Rudi Eigenmann

Purdue University

Why Autotuning ?
my bias

- Runtime decisions for compilers are necessary because compile-time decisions are too conservative
  - Insufficient information about program input, architecture
  - When to apply what transformation, in which flavor?
  - Polaris compiler has some 200 switches

    Example of an important switch: parallelism threshold

- My goals:
  - Looking for tuning parameters and evidence of performance difference
  - Go beyond the “usual”: unrolling, blocking, reordering
  - Show performance on real programs

Other Motivators

- Architectures and environments get increasingly heterogeneous, distributed and dynamic

  -> Adapt behavior

- Information Power Grid

  -> Roles for the compiler ?

Is there Potential ?

You bet!

- Imagine you (the compiler) had full knowledge of input data and execution platform of the program

"Amdahl's law of dynamic optimization"

Very Early Work in the Polaris Parallelizer

- Multi-version code

  IF expression=true run serially

  ELSE run in parallel

- Runtime data-dependence analysis

  Run in parallel, track data accesses

  IF there was a conflict, backtrack, run serially

Early Results on Fully-Dynamic Adaptation

- ADAPT system (Michael Voss - 2000)
- Features:
  - AL - adapt language
  - Used remote compilation
    - Allowed standard compilers and all options to be used
  - Triage & shelter
    - tune the most deserving program sections first
    - reduce tuning on un-deserving sections
- Issues:
  - Scalability to large number of optimizations
  - Runtime overheads

Recent Work

Offline Tuning - "Profile-time" tuning

Zhelong Pan

Challenges:

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 - PACT 2006

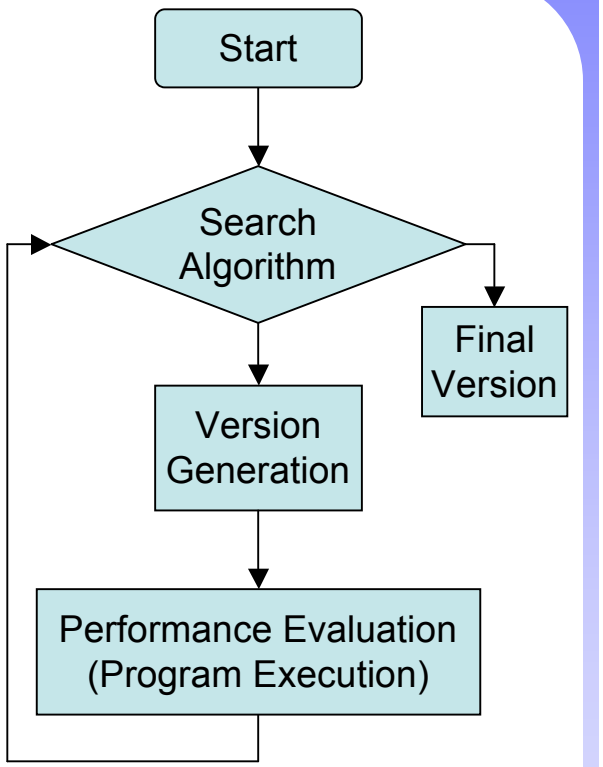
   - Program partitioning into tuning sections

Two goals : increase program performance and reduce tuning time

Whole-Program Tuning

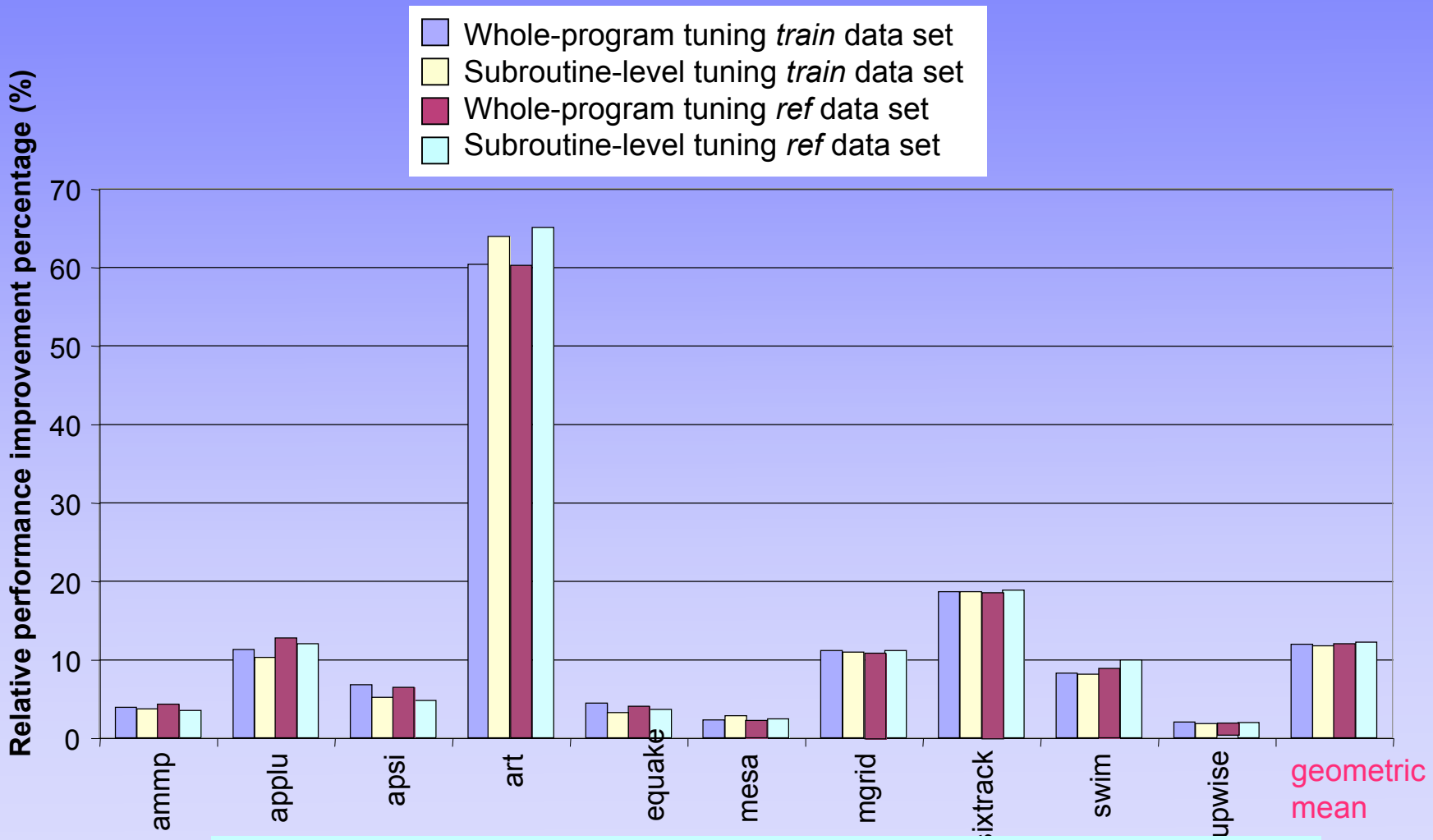
Search Algorithms

- 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 => slow
  - Considers interaction => effective
- **CE: combined elimination (final algorithm)**
  - **Eliminates a few "bad" optimizations at a time**
- Other algorithms
  - optimization space exploration, statistical selection, genetic algorithm, random search



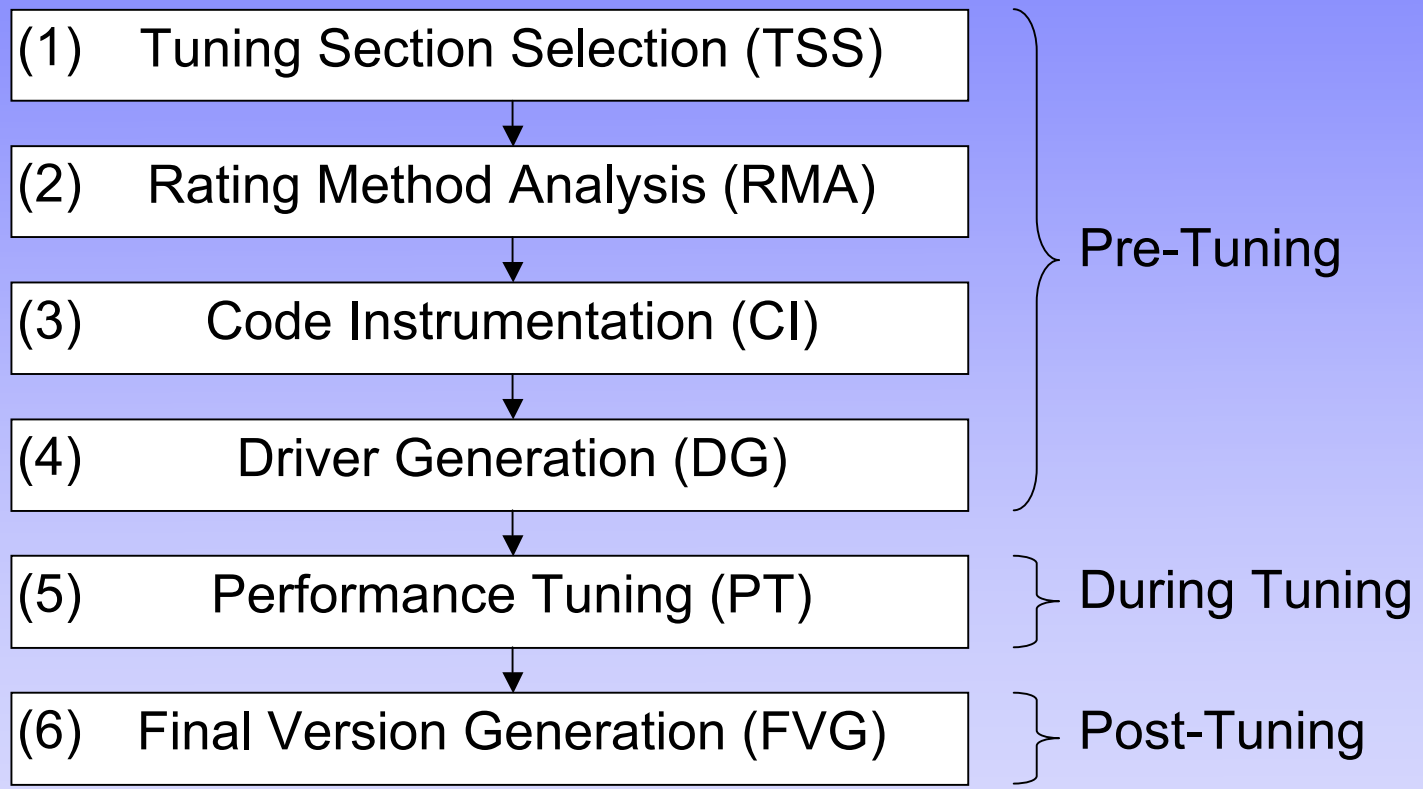
Empirical search

Performance Improvement

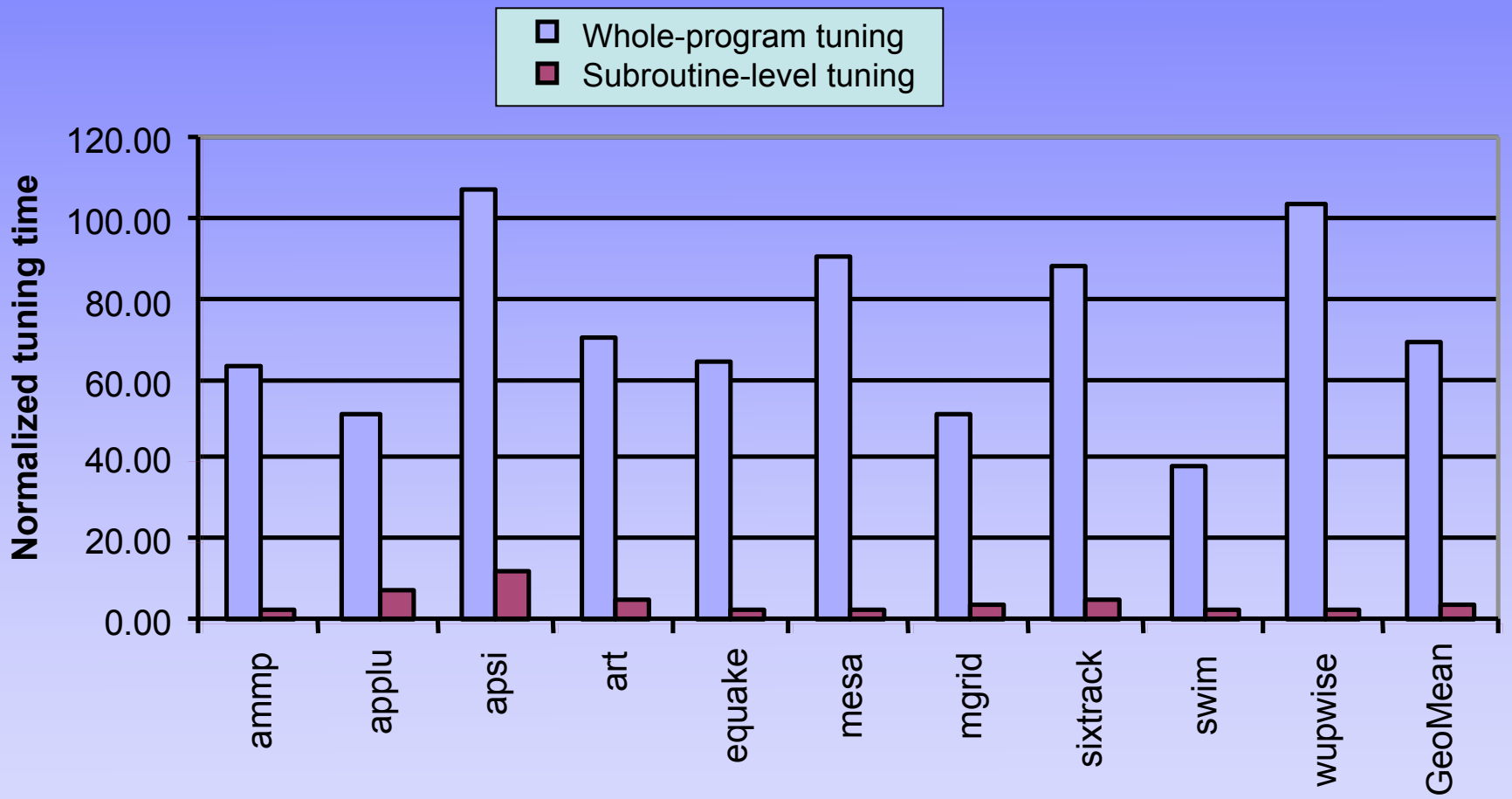


Tuning Goal: determine the best combination of GCC options

A Mechanism for Subroutine-level Tuning



Reduction of Tuning Time through Subroutine-level Tuning



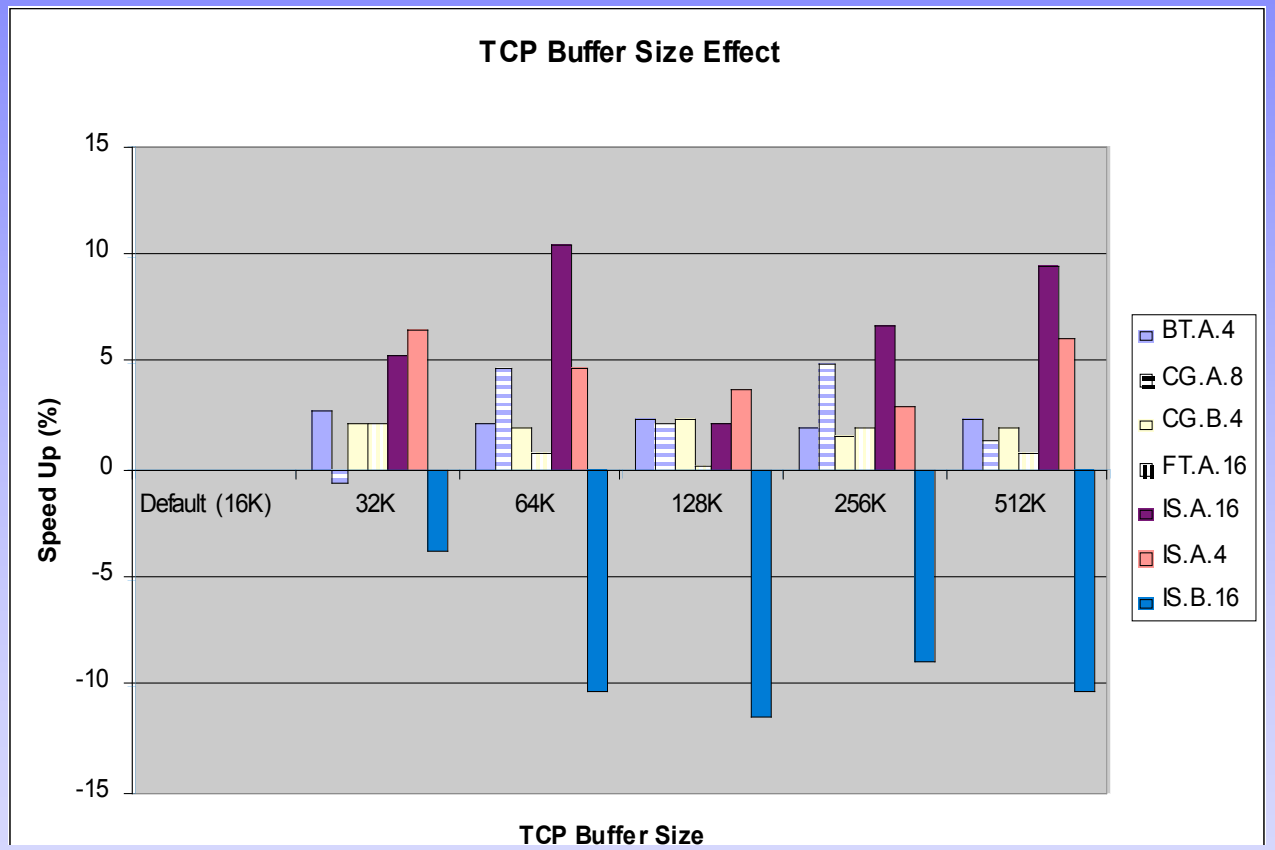
Ongoing Work

Seyong Lee

Beyond autotuning of compiler options

- New applications of the tuning system
  - MPI parameter tuning
  - Tuning library selection - (ScalaPack, ...)
  - OpenMP to MPI translator
  - Tuning SPMUL on clusters

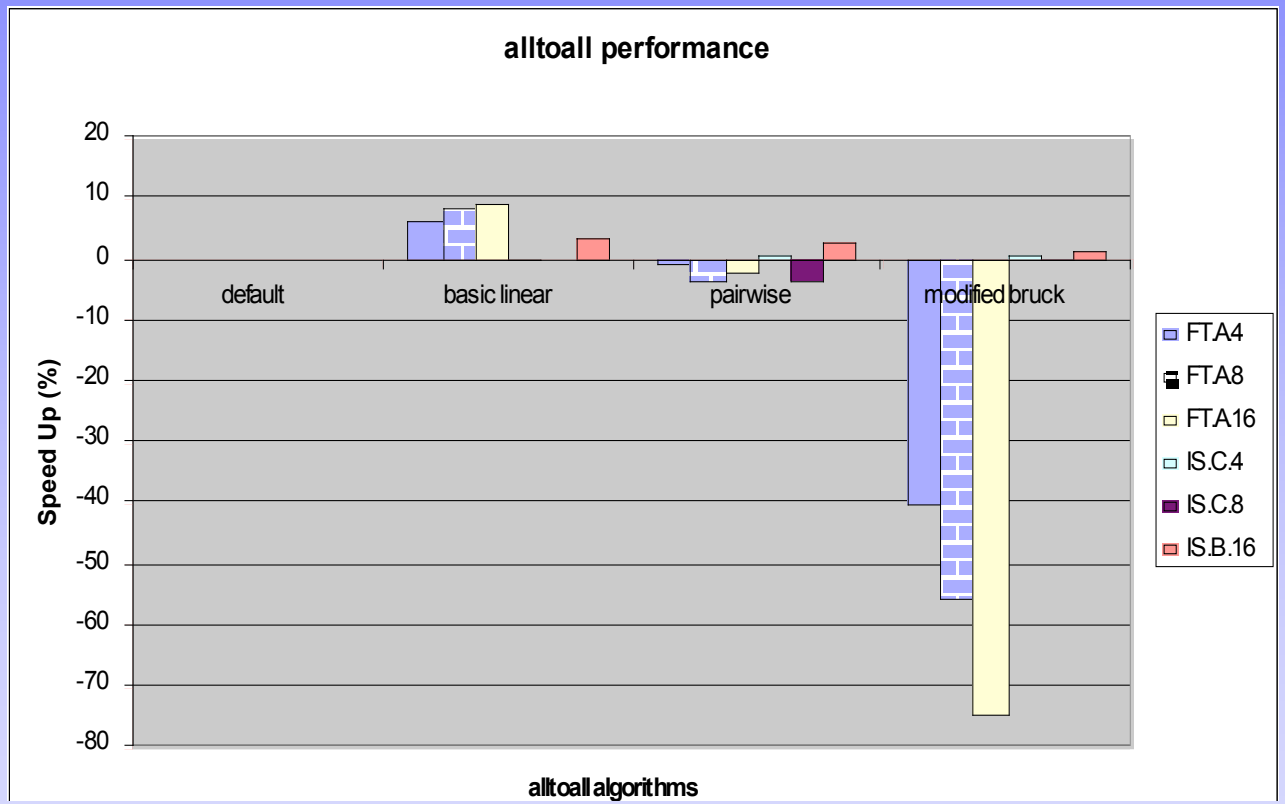
TCP Buffer Size Effect on NPB



Target system: Dell IA-32 P4 nodes cluster

Used MPI: MPICH1

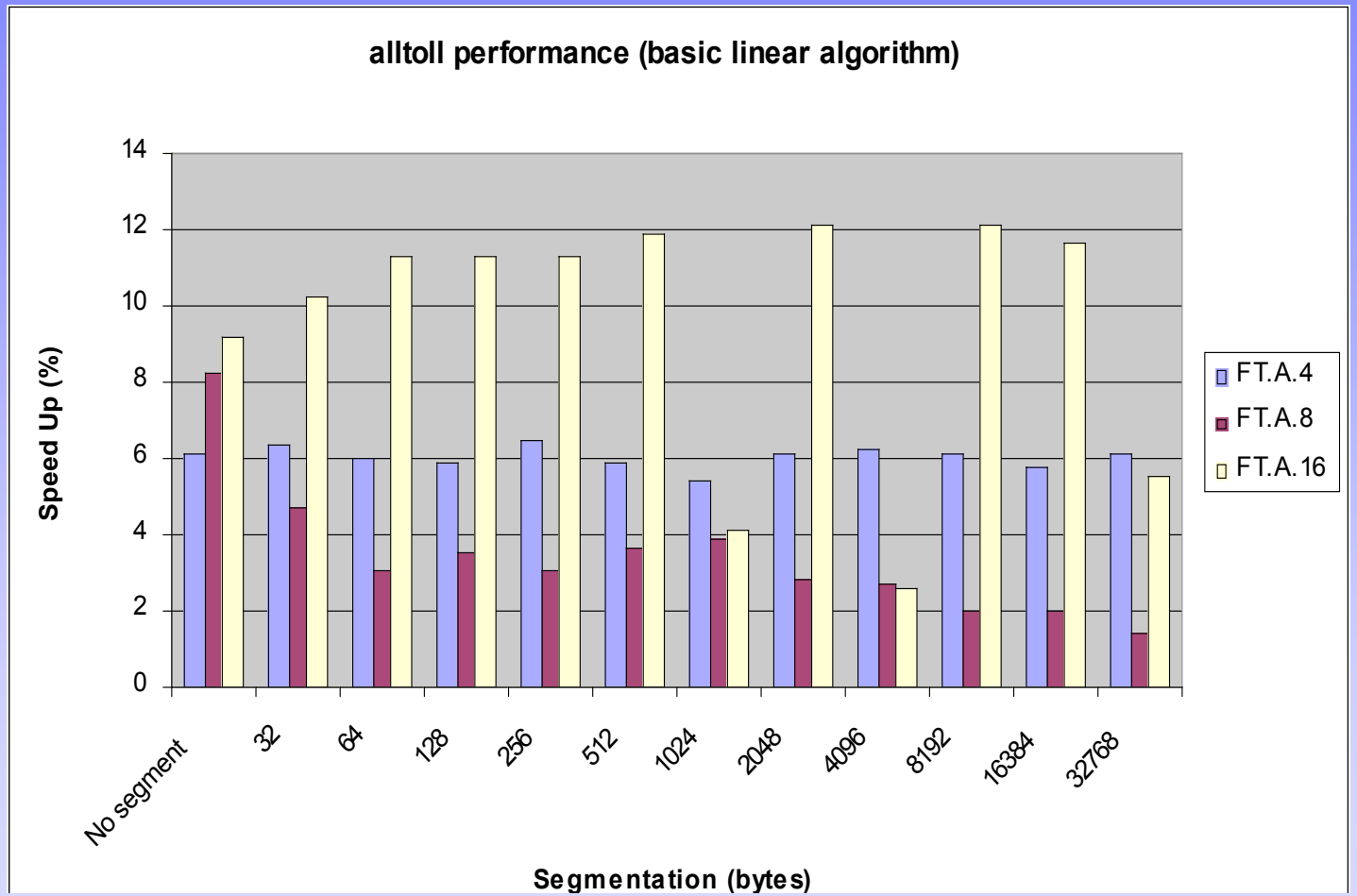
All-to-all collective call performance



Target system: Dell IA-32 P4 nodes cluster

Used MPI: Open MPI 1.2.2

Segmentation Effect on Basic Linear Alltoall Algorithm



Target system: Dell IA-32 P4 nodes cluster

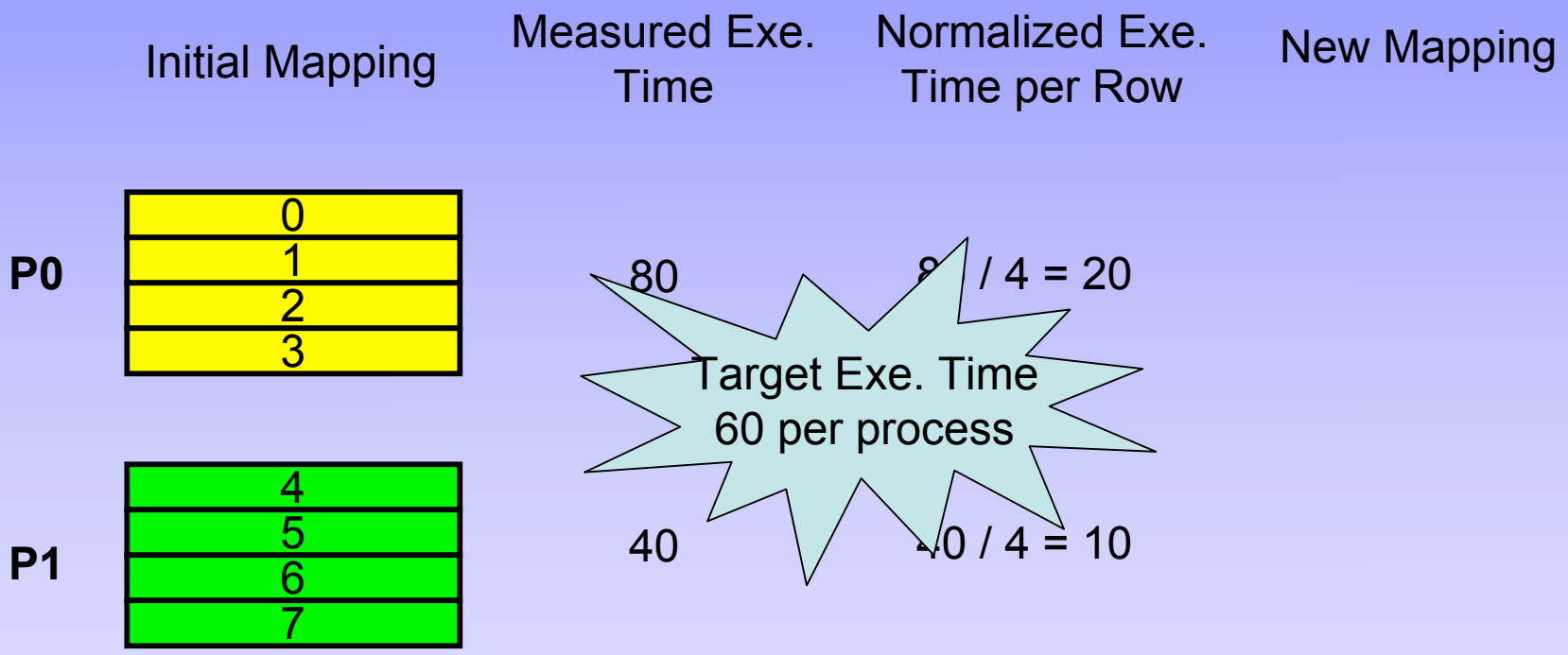
Used MPI: Open MPI 1.2.2

Tuning SPMUL on Clusters
(towards petascale)

- Adaptive Iteration-to-Process Mapping
  - Automatically adapt *Iteration-to-Process mapping* considering computational load imbalance and dynamic runtime-system performance
- Run-time Communication Selection
  - Packed data vs. Block data communication
  - Broadcasting vs. Point-to-point exchange

Adaptive Iteration-to-Process Mapping

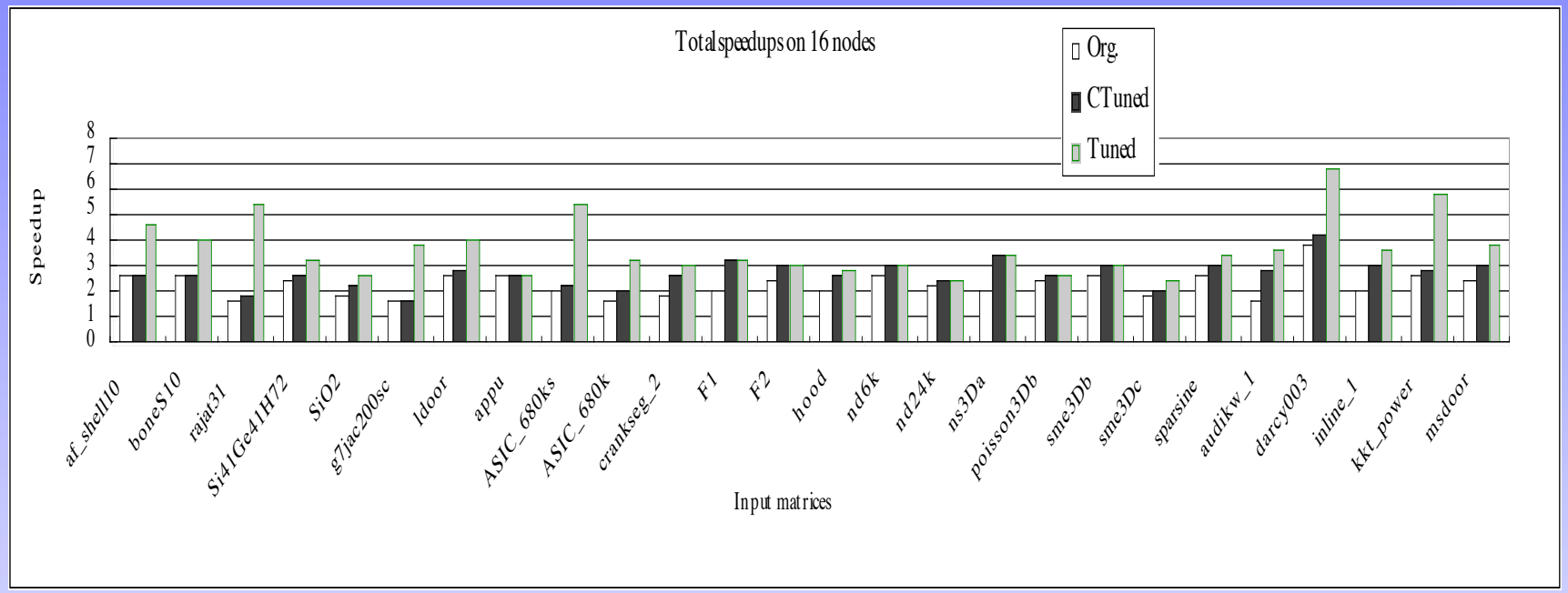
- Main Idea



Runtime Communication Selection

- Communication Methods
  - *Block broadcasting method (CM1):* each process broadcasts locally written output blocks.
  - *Block point-to-point exchange method (CM2):* processes exchange bounding blocks containing needed elements through point-to-point communication.
  - *Packed point-to-point exchange method (CM3):* processes exchange exactly needed element through point-to-point communication.
- Optimal method is re-selected at runtime whenever the adaptive mapping system changes the distribution.

Performance Improvement Through Adaptive Runtime Tuning



Speedups of the base parallel version (*Org*), computation-tuning only version (*CTuned*), and tuned version (*Tuned*) on 16 nodes.

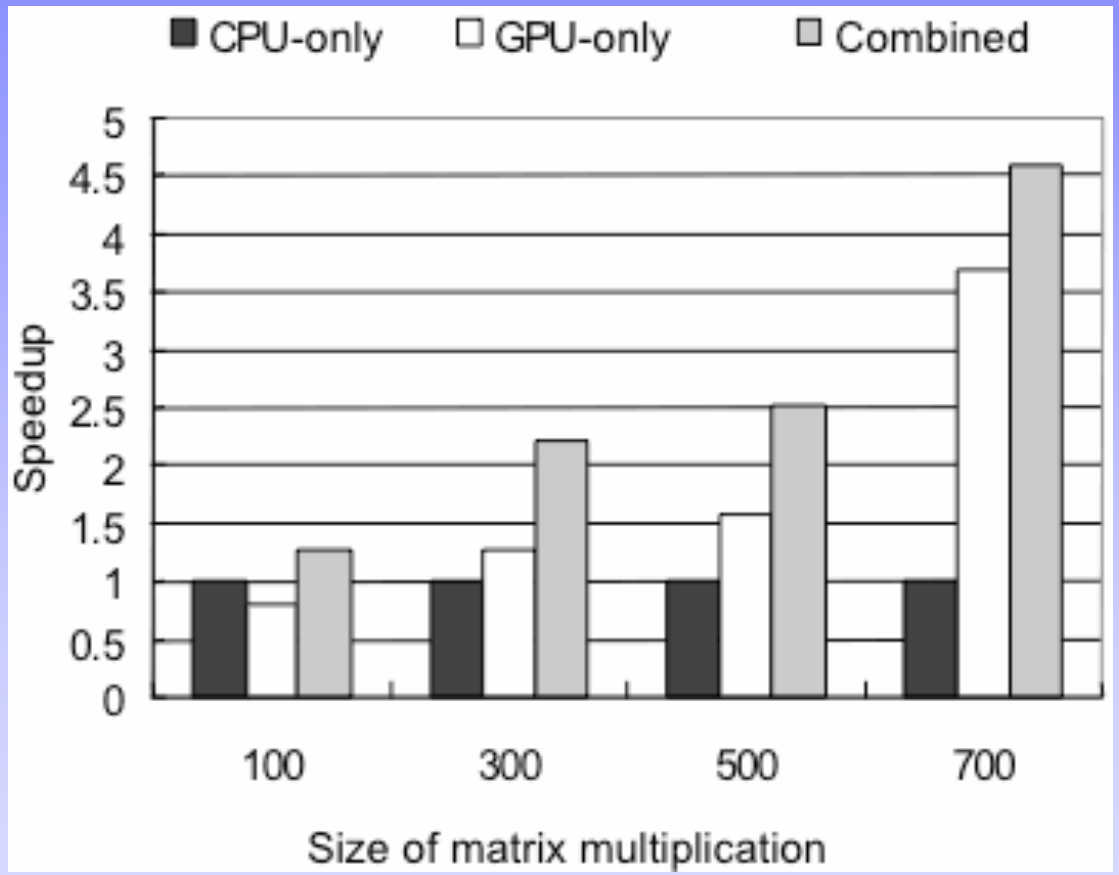
Our adaptive mapping (*CTuned*) reduces execution time up to 37.8% (14% avg)

Overall tuning system (*CTuned*) reduces execution time up to 66.7% (33.3% avg)

OpenMP2GPU: Automatic Translation and Tuning of OpenMP Programs on GPU

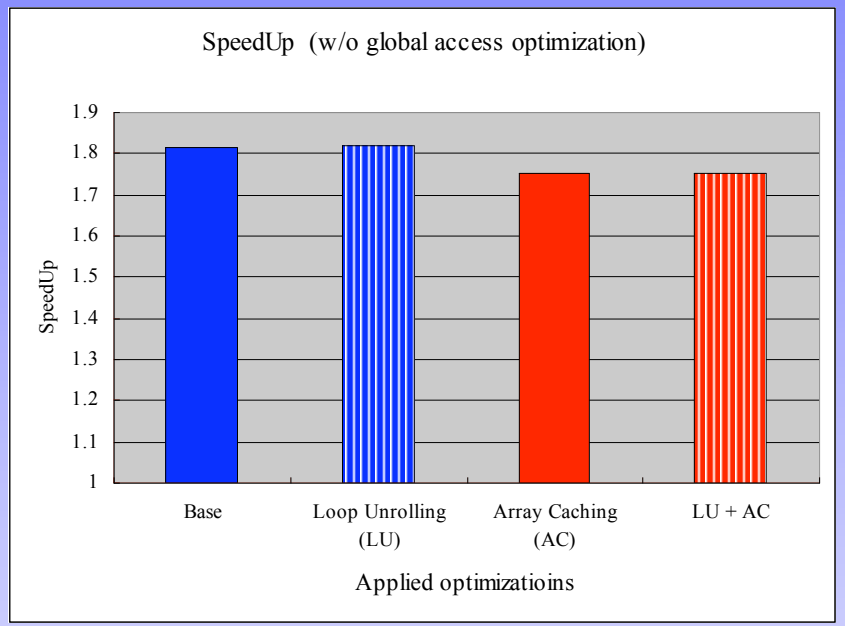
- We are developing an automatic OpenMP-to-GPU translator,
  - takes OpenMP programs as inputs and generates GPU codes targeting on NVIDIA CUDA-enabled GPUs.
  - which also provides several code variants that can be tuned at runtime.
- Challenging Issues for Performance Tuning
  - Two major tuning targets:
    - To minimize and align global memory accesses
    - To minimize control flow divergence.
  - Several compiler techniques, such as tiling and loop unrolling, can be applied.
  - However, optimizations may conflict with each other due to hardware resource limits (ex: # of registers and shared memory size)
  - Runtime tuning is necessary to strike a balance among conflicting optimizations.

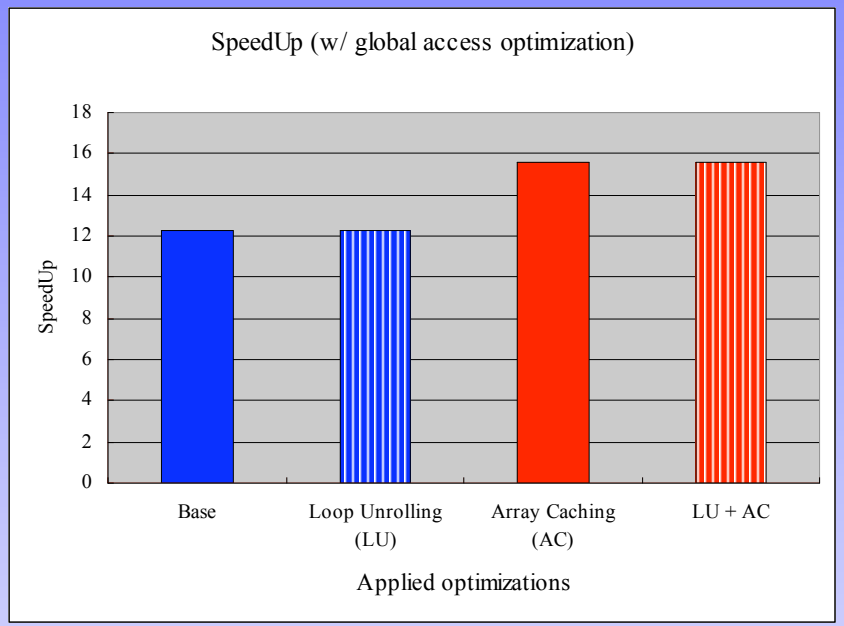
Offloading Computation to GPU



In all cases, tuned work sharing on CPPU and GPU is fastest.

Performance of NPB OMP EP





Performance of translated version of NPB OMP EP, executed on NVIDIA GeForce 8400 GS. The speedups are over serial version of EP, run on Intel Core™2 Duo CPU @2.2GHz. The left graph shows speedups when global memory optimization is not applied, and the right graph shows speedups when the global optimization is applied. The results show that local array caching technique (AC) behaves differently depending on other optimizations.

Automatic Tuning for Multicore

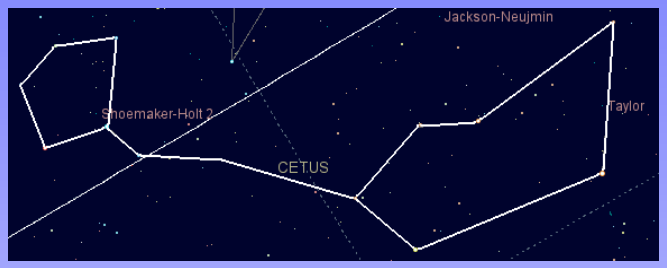
- Starting point was the Polaris compiler - 200 switches
- Early results on dynamic serialization
- Goal: parallelizing compiler that never lowers the performance of a program
- OpenMP to MPI translation
- Tuning NICA architectures
  - Multicore + niche capabilities (accelerators and more)

    => Purdue 

Project

Compiler Infrastructure

Cetus



- Successor to Polaris
- Source-to-source translator for C, C++, Java
- Supported by the U.S. National Science Foundation
- Written in Java
- cetus.ecn.purdue.edu

Questions and Propositions

Autotuning compilers are not just compilers.

- The core is a *runtime adaptation engine* that picks from among variants.
- Compiler-generated code is one of several ways of creating code variants (others are libraries and user code)
- Compiler is used to analyze and instrument the code. But this is not classical compilation.
- The Engine finds the best through models and experiments
  - In our case: focus on experiments - empirical search
  - Empirical search is extremely powerful but can be slow. Pruning through model is future

Questions and Propositions

Big issue: where/when to generate and prune code variants

- Offline - maybe even at system generation time
- fully dynamically
- Offline training, runtime selection
- Parameterized code variants
- staged compilers
- library variants
- user-generated (autotuning language)

Questions and Propositions

- How to do fine-grained autotuning?
  - Access to more detailed optimization parameters
  - Tuning individual code sections
- How to detect when to re-tune (recognize phase or environment changes)
  - shelter code (near-zero tuning overheads) as long as possible
  - retune when important changes happen

  => Combine the benefits of offline and adaptive tuning
- How to create an autotuning architecture?

  what's a good architecture to integrate multiple contributions?
  - this is a general issue in all research
  - Agreeing on terminology may be a starting point

Questions and Propositions

- What is special about petascale?
  - tuning for parallelism
  - tuning for high performance is different from JIT compilation
- Performance metrics:
  - Use ordinary real benchmarks
  - Long running benchmarks are especially important, as they show phase behavior
  - Use the same metrics: performance, productivity, power
- What architectures/platforms should we target?
  - All. Heterogeneous architecture are especially interesting.
- What improvements should we expect from autotuning, at both the compiler level and the library level?
  - Up to orders of magnitude.
  - So far, we achieved < 2x

Questions and Propositions

- Will we allow future compilers to change data structures and algorithms?

  Yes, note this is a global optimization.

- The traditional boundaries between applications, libraries, compilers, and operating systems is too rigid and needs to be changed.

  Yes. Compilation optinos, library variants, program parameters can all be orchestrated by a smart tuning engine.

- What can we do to build common tool bases for compiler-based autotuning and for construction of self-tuning or autotuning libraries?

  This is important and should be discussed further.

Conclusions

- Dynamic Adaptation is one of the most exciting research topics with tremendous potential
- It will only get more important, as architectures/environments get more complex and languages get higher level
- There is room for 10s of PhD theses
- Perhaps the biggest issue: how to synergize the community
  - Can we create an autotuning architecure, in which we can all plug in our contributions?