RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS 516 Compilers and Programming Languages II

Approximation and Prallel Computing-1

RUTGERS Colloquium Talk – Tuesday Feb. 25

Yipeng Huang – PostDoc @ Princeton -- **Attend talk / no class**

10:30am, CoRE 301

Free coffee and pastry!

Faculty Candidate Talk

Emerging Architectures for Humanity's Grand Challenges

Tuesday, February 25, 2020, 10:30am

- Analog, approximate computing
- Quantum computing

Speaker:

Yipeng Huang, Princeton University

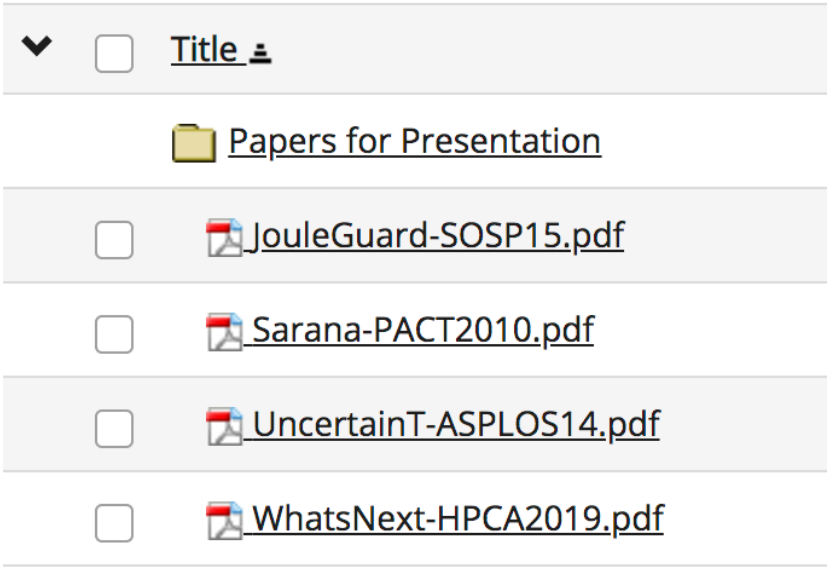
Bio

Yipeng is a postdoctoral research associate at Princeton University, working with Prof. Margaret Martonosi. He received his PhD in computer science in 2018 from Columbia University, working with Prof. Simha Sethumadhavan. His research interest is in building, programming, and in identifying applications for emerging architectures. These include quantum and analog computer architectures that may uniquely address challenges in scientific computing, but would require new programming tools and architectural abstractions. His work has been recognized as an IEEE Micro Top Pick among computer architecture conference papers. He has also received support for his research work through DARPA, in the form of a Small Business Technology Transfer grant to investigate commercial applications of his research.

Location : CoRE A 301

Paper List

These four papers are available under resources on sakai



- Uncertain<T> : Language extensions and runtime system for approximate computing
- Architecture/compiler for intermittent energy sources
- Sarana – Opportunistic networks with approximate answers
- JouleGuard – approximate computing with power/energy constraints

Need four volunteers to present these papers. Please send me and email if you want to present one of these papers.

Classification of Approximations

Underlying assumption: There is **a ground truth** consisting of the **perfect/correct/optimal answer**. This answer may not be effectively computable. An **approximate answer** comes "close" to the ground truth, where "closeness" is an application-specific metric.

Model approximation (solving a real-world problem through an algorithm):

- use a simpler view of the world (ground truth) that still works for a particular application
  (example: assume the world is flat for a car-navigation application)
- even optimal solution of the model is approximation to ground truth (e.g.: simplified physical interactions)

Data approximations :

- less precise data (e.g.: single vs. double)
- use a subset of data (e.g.: statistical representation, subset probing)
- probabilistic data values

Computation approximation:

- solve a model non-optimally (e.g.: use a heuristic)
  - + relaxed convergence criteria
  - + imprecise solution strategies (e.g.: simpler stencil for PDE solvers)
  - + probabilistic solution strategies

Approximation: Summary

There are different aspects of approximation that will finally determine the precision/quality of your answer.

What does "optimal" really mean in the presence of approximation?

Example: An optimal solution to a problem that uses approximations in its model and data representation may be less effective than a sub-optimal solution strategy that uses are more precise data representation.

My philosophy:

- Only use approximations if you have to
- Optimal choices in one domain (model, data, computation) help you understand what approximation levels are needed in the others to implement an effective overall system
  - ➔ One reason to use gurobi to avoid computation approximation

Existing/possible research thrusts

- Extending existing languages to become approximation-aware [Examples: Uncertain<T>, loop perforation, …]

-Techniques to prove/guarantee approximate application properties [Example Thu et al. - Hadoop]
  + error bounds
  + execution time
  + power, energy, thermal
  + …

- Systematic construction of approximate algorithms/applications, with approximation as a first-order design principle (no "dusty decks")
  + model approximation
  + data approximation
  + computation approximation

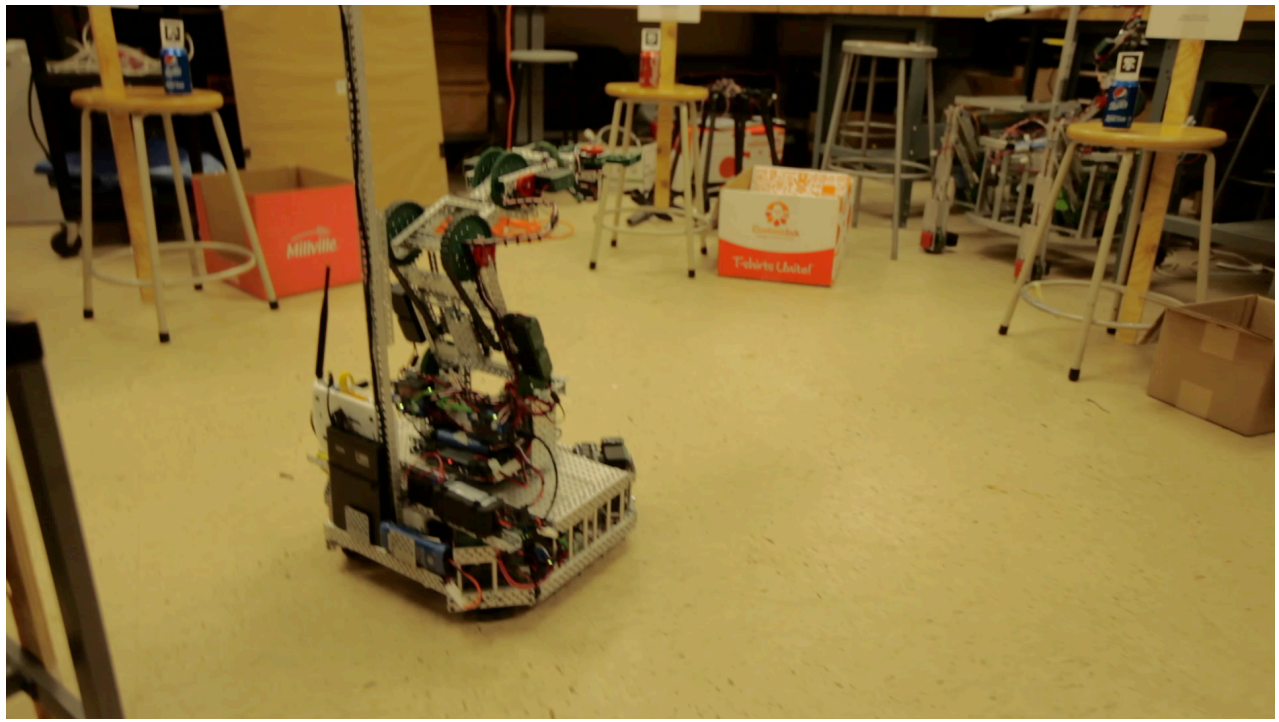
- Composition of applications consisting of approximate components, and select the "semantically best" combination of component approximation levels [RSDG]; tools for tradeoff space exploration

Existing/possible research thrusts

- Compilation of approximate applications onto heterogeneous architectures with approximate hardware components (e.g.: cache, CPU, memory)

- Application investigations and benefit analyses of different forms of approximation
  - + parallel computing, high performance computing [Example: LU factorization, Jack Dongarra]
  - + web search (Hadoop)
  - + data analytics and big data
  - + mobile applications (smart phones)
  - + robotics (vision, path planning, motion planning, risk management)

Example: Robotics



Robot solves a traveling salesman problem every 2 seconds under a user specified preference of Coke over Pepsi, and a limited resource budget (battery energy). Correctness condition: Robot has to "return home" to its starting position.

Two ways of thinking about concurrency:

data-centric view: **partition the data** that can be worked on in parallel (data-level parallelism);

→ your work is determined by the data that you are assigned to work on.

task-centric view: **partition the work** that can be done concurrently (task-level parallelism);

→ your data is determined by the work that you have to do

These are two symmetric problems:

What tasks have "to travel" to what data (data-centric)
or what data has "to travel" to what tasks (task-centric)

Task-Centric Concurrency

Different levels/granularities to exploit concurrency:

1. Instruction-level parallelism (ILP) -
   typically exploited by hardware or compiler

2. Loop-level parallelism -
   single loop iterations are considered individual tasks

3. Procedure-level parallelsim -
   different procedures calls may be executed concurrently

4. Process-level parallelism -
   different programs maybe executed concurrently

Will concentrate mostly on loop-level parallelism/concurrency

Goal of parallelization / concurrent execution

Starting point: A sequential application

Goal: Identify "independent pieces of work" that can be executed concurrently without changing the semantics of the application, i.e., do not change application input/output behavior (safety of parallelization).

performance impact → reduction in the length of the critical path

Outcome: A (partially) parallelized application where parts of an application are identified as executable in parallel. Examples: An application may contain parallel regions, parallel loops, fork-join, VLIW, …

Key Challenge: How to decide whether parallel executions of parts of the application are safe? → notion of **data and control dependence**

Dependence Overview

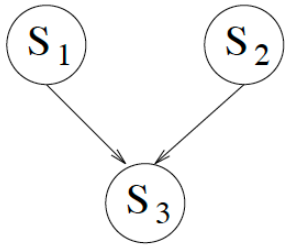
Dependence relation: Describes all program regions (e.g.: statements) execution orderings for a sequential program that must be preserved if the meaning of the program is to remain the same.

There are two sources of dependences, data and control dependencies:

Example:
(statement level)

data dependence

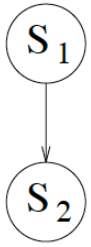
```
S1   pi = 3.14
S2   r = 5.0
S3   area = pi * r**2
```



S_3 cannot be executed before S_1 or S_2

control dependence

```
S1   if (t .ne.  0.0) then
S2      a = a/t
     endif
```



S_2 cannot be executed before S_1

Basic Dependence Theorem

Theorem

Any reordering transformation that preserves every dependence (i.e., visits first the source, and then the sink of the dependence) in a program preserves the meaning of that program.

Note:

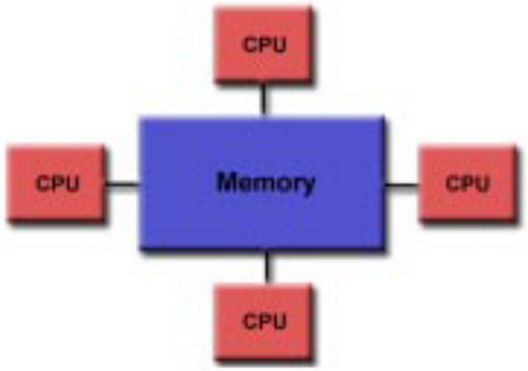
(1) Dependence starts with the notion of a sequential execution, i.e., starts with a sequential program.

(2) Control dependencies can be converted to data dependencies. From now on, we only look at data dependencies,

Loop / Statement-level Dependencies

We will concentrate on compilation issues for compiling scientific codes on shared-memory parallel/vector architectures. Some of the basic ideas can be applied to other application domains as well. Typically, scientific codes

- Use arrays as their main data structures.
- Have loops that contain most of the computation in the program.



As a result, advanced optimizing transformations concentrate on loop level optimizations. Most loop level optimizations are source-to-source, i.e., reshape loops at the source level.

We will talk about

- Dependence analysis
- Automatic vectorization
- Automatic parallelization
- Heterogenous parallel architectures

```
#pragma omp parallel for private(i, hash)
     for (j = 0; j < num_hf; j++) {
          for (i = 0; i < wl_size; i++) {
               hash = hf[j] (get_word(wl, i));
               hash %= bv_size;
               bv[hash] = 1;
          }
     }
```

Loop / Statement-level Dependencies

Definition

There is a data dependence from statement S_1 to statement S_2 (S_2 depends on S_1) if:

1. Both statements access the same memory location and at least one of them stores/writes into it, and
2. There is a feasible run-time execution path from S_1 to S_2

Data dependence classification: "S2 depends on S1" — $S_1 \delta S_2$

true (flow) dependence (RAW hazard)

S_1 writes a memory location that S_2 later reads

anti dependence (WAR hazard)

S_1 reads a memory location that S_2 later writes

output dependence (WAW hazard)

S_1 writes a memory location that S_2 later writes

input dependence

S_1 reads a memory location that S_2 later reads.

Note: Input dependences do not restrict statement (load/store) order!

Dependence: Identify Concurrent Loops

We restrict our discussion to data dependence for scalar and subscripted variables (no pointers and no control dependence).

Sequential source code

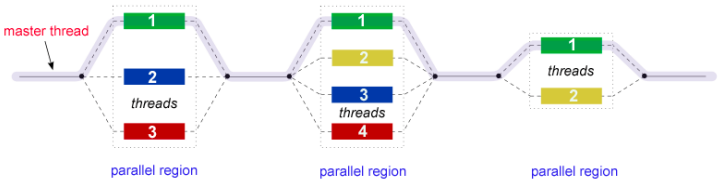
```
do I = 1, 100
  do J = 1, 100
    A(I,J) = A(I,J) + 1
  enddo
enddo
```

```
do I = 1, 99
   do J = 1, 100
      A(I,J) = A(I+1,J) + 1
   enddo
enddo
```

vectorization


```
A(1:100:1,1:100:1) = A(1:100:1,1:100:1) + 1
A(1:99,1:100) = A(2:100,1:100) + 1
```

parallelization




```
doall I = 1, 100
  doall J = 1, 100
    A(I,J) = A(I,J) + 1
  enddo
  implicit barrier sync.
enddo
implicit barrier sync.
```

```
do I = 1, 99
   doall J = 1, 100
      A(I,J) = A(I+1,J) + 1
   enddo
   implicit barrier sync.
enddo
```