
Understanding Parallelism-Inhibiting Dependences in Sequential Java Programs

Atanas (Nasko) Rountev

Kevin Van Valkenburgh

Dacong Yan

P. Sadayappan

Ohio State University

Overview and Motivation

- Multi-core hardware presents a great software evolution challenge
- Our goals
 - Characterize how much **inherent method-level parallelism** exists in a given sequential Java program
 - Identify **run-time data dependences** that reduce this inherent parallelism
 - Do this before the actual parallelization is attempted
- The technique
 - Analyzes dependences observed during a program run
 - Similarly to existing work on ILP/loop analysis
 - “Turns off” dependences to find bottlenecks

Abstract Model of Best-Case Parallelism

- Method-level parallelism
 - Code in a method executes sequentially
 - When a call occurs, the **callee** method starts executing concurrently with the **caller** method
- Best-case (hypothetical) scenario
 - Each statement executes as soon as possible
 - Must respect data dependences from the run of the sequential program: e.g. if statement instance **s4** is
 - data dependent on **s1** and **s2**
 - preceded by **s3** in the same method

$$time(\mathbf{s4}) = 1 + \max(time(\mathbf{s1}) , time(\mathbf{s2}) , time(\mathbf{s3}))$$

Available Parallelism

- Compute a **best-case parallel timestamp** *time(s)* for each statement instance *s* during the sequential run
 - N = number of statement instances in the run
 - T = largest timestamp
- N/T is the **available parallelism** in the program
 - Austin-Sohi [ISCA'92] for instruction-level parallelism
- What does it mean?
 - Independent of number of processors, cost of thread creation and synchronization, thread scheduling, etc.
 - Impossible to achieve
 - Allows comparisons between programs, and searching for parallelism bottlenecks within a program

Specific Contributions

- A **run-time analysis** algorithm for measuring the available parallelism
 - Bytecode instrumentation
 - Run-time computation of timestamps
- A **bottleneck analysis** built on top of it
 - Finds problematic instance fields
- Measurements on a large set of programs
 - Overall, **low available parallelism**; data dependences through instance fields are partly to blame
- Three **case studies**

Bytecode Instrumentation (1/2)

- Done on three-address IR (Soot framework)
- Memory locations of interest: static fields (class fields), instance fields (object fields), array elements, some locals
- Example: for an instance field **f**, add two new **shadow instance fields** **f_w** and **f_r** to the class
 - For a run-time object **O**, **O.f_w** stores the timestamp of the last statement instance that wrote to **O.f**
 - In **O.f_r**, store the largest timestamp among all reads of **O.f** since the last write to **O.f**
- Special local variable **control** in each method

Bytecode Instrumentation (2/2)

- Example: for an assignment **$x.f = y$** , add
 - **control** = $1 + \max(x.f_r, x.f_w, \text{control}, x_w, y_w)$
 - **$x.f_r$** : read-before-write
 - **$x.f_w$** : write-before-write
 - **control** : previous statement in the method
 - **x_w** and **y_w** : if their values come as return values of calls that have already occurred
- Example: for a call **$x = y.m(z)$**
 - **control** = $1 + \max(y_w, z_w)$
 - Initialize the **callee's control** with the **caller's control**
 - Make the call
 - Set **x_w** to be the **callee's control**

What Next?

- We can measure the largest timestamp T and the available parallelism N/T , but what does it mean?
 - By itself, not much, although “low” is likely to be bad
- First client: characterize the memory locations
 - E.g., ignore (“turn off”) all dependences through static fields, and see the increase in available parallelism
- Second client: hunting for bad instance fields
 - Variant 1: **turn off all dependences through a field f** ; the higher the increase, the more suspicious the field
 - Variant 2: **turn off all dependences *except* through f** ; compare with a run with all instance fields turned off

Experimental Evaluation

- 26 single-threaded Java programs from 4 benchmark suites: SPEC JVM98, Java Grande 2.0, Java rewrite of Olden, and DaCapo
- Instrumented both the application and the standard Java libraries
- Question 1: what is the run-time overhead?
 - Median slowdown: 29 ×, which is comparable with similar existing work; definitely room for improvement
 - If the standard Java libraries are not instrumented, the overhead is reduced by about 56% but the precision suffers

- Question 2: what is the available parallelism?
 - Quite low: **median value is 2.1**
 - Need code analysis and changes
- Question 3: who is to blame?
 - Turning off **static fields** makes almost no difference
 - Turning off **array elements** makes little difference
 - Turning off **instance fields** significantly increases the available parallelism

antlr	1.6
bh	47.4
bisort	1.1
bloat	5.1
chart	2.0
compress	1.2
db	1.2
em3d	4.0
euler	1.6
fop	2.4
health	116.9
jack	1.7
javac	2.7
jess	5.2
ython	2.2
luindex	4.0
moldyn	1.8
montecarlo	1.3
mpegaudio	1.6
pmd	2.0
power	134.1
raytrace	2.4
raytracer	1.2
search	1.6
tsp	38.2
voronoi	32.1

Case Studies (1/2)

- Study 1: **moldyn** shows two orders of magnitude increase when static fields are ignored
 - Static fields `epot`, `vir`, and `interactions` are updated in method `force` (e.g. `epot=epot+expr`)
 - All calls to `force` conflict with each other, but in reality are **independent**; a simple code change increases the available parallelism from **1.8** to **102.7**
- Study 2: Java Grande **raytrace** shows lower than expected available parallelism
 - Five different parallelism bottlenecks
 - Examine a highly-ranked field, change the code, recompute rankings (did the field become harmless?)

Case Studies (2/2)

- Study 3: **em3d** shows the greatest increase when instance fields are ignored
 - Problem – during the building of a directed graph
 - When adding a node **n2** to **n1.successor_list**, the code also increments **n2.num_predecessors**
 - If **n2** is also added to **n3.successor_list**, the two increments of **n2.num_predecessors** are serialized
- Solution: break up the computation
 - Populate all **successor_list** fields – *highly parallel*
 - Next, traverse all **successor_list** and compute all **num_predecessors** fields – *very little parallelism*
 - Available parallelism jumps from **4.0** to **48.9**

Summary

- A measure of **inherent method-level parallelism** in a sequential Java program
 - Will be easy to generalize to loop-level parallelism
- Run-time analysis algorithm: bytecode instrumentation computes best-case timestamps
- Interesting applications
 - **How harmful are the data dependences through certain memory locations?**
 - **What are the effects of code changes?**

Questions?