Kremlin: Like gprof, but for Parallelization

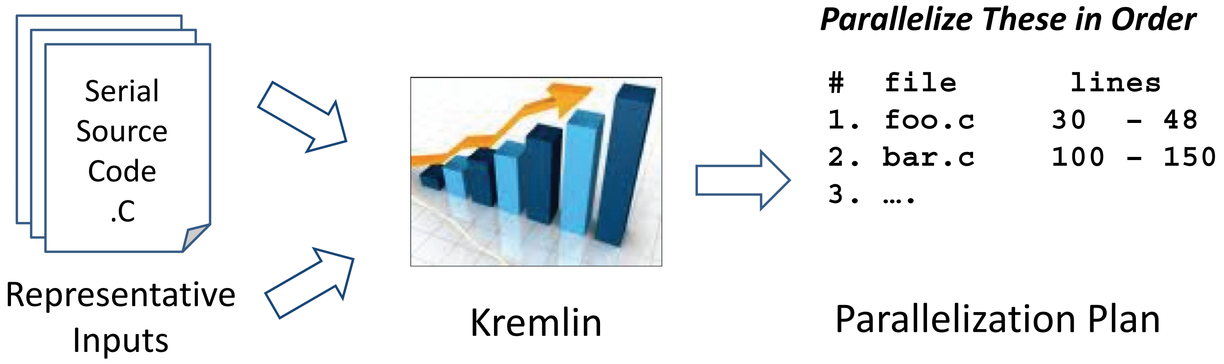
Donghwan Jeon, Saturnino Garcia, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor, UC San Diego

Overview

Multicore Programmers Wish They Knew…

"Which parts of the program should I spend time parallelizing?"

Our Approach: Kremlin



Three Key Contributions We Introduce

1. **Hierarchical Critical Path Analysis (HCPA)**
   a dynamic analysis that measures parallelism across many nested regions using multi-versioned shadow memory.
2. **Self Parallelism**
   a metric that localizes parallelism to a code region, separating a parent region's innate parallelism from its children's.
3. **The Concept of a Planner**
   based on self parallelism, work, and target platform constraints (e.g. OpenMP, 32-core NUMA), a planner produces a parallelization plan that focuses the programmer's efforts.

Sample Usage (similar to `gprof`)

```
$> make CC=kremlin-cc
$> <program> <data>
$> kremlin <program> --openmp
```

	File (lines)	Self Parallelism	Coverage(%)	Time Reduction(%)
1	imageBlur.c (49-58)	145.3	9.7	9.5
2	imageBlur.c (37-45)	145.3	8.7	8.5
3	getIPatch.c (26-35)	25.3	8.9	4.2
…	…			

The Details

Hierarchical Critical Path Analysis

Goal

Quantify parallelism within program regions without requiring deep or intractable program analysis.

Challenges

Previous approaches track the earliest possible execution time of each operation in the program's dynamic execution graph, based only on control and data dependences. However, these schedules smear operations from many different regions into each unit of time. To provide actionable information for the user, we need an analysis that is better at localizing parallelism to concrete code regions in the program.

Our Approach

We extend *critical path analysis* so that, rather than computing parallelism information for just one region, we compute it for every loop and function in the program, using one pass through the program.

We do this through the use of efficient algorithms that implement a hierarchical, multi-versioned shadow memory.

Self Parallelism

Goal

Estimate a program's new execution time as the set of regions that are parallelized is varied.

Challenges

To exactly evaluate this, we would need to capture the entire dynamic execution graph of the program, which is prohibitive in terms of space and time.

Our Approach

We created a heuristic, *self-parallelism*, that can be computed as the HCPA analysis runs, and does not require storage of the program graph.

Self-parallelism estimates the amount of parallelism that is attributable to a parent region separate from its children. Using self-parallelism, we can set an upper bound on the speedup attainable by parallelizing a region. Our results show that this heuristic works extremely well across a wide variety of parallelism structures.

$$SP(R) = \frac{\sum_{k=1}^{n} cp(child(R,k)) + SW(R)}{cp(R)}$$

The Planner

Goal

Select among the many possible parallelization plans, choosing the one that will yield the best speedup, while keeping programmer effort low.

Challenges

Speedup is sensitive to machine characteristics (e.g. synchronization costs); parallelization system (e.g. OpenMP, Cilk); and, of course, program characteristics.

Our Approach

We employ planner algorithms that create plans quickly and accurately, by combining self-parallelism with a *planner personality* for the target platform.

Two Sample Personalities

Planner Personalities	OpenMP	Cilk++
Nested Parallelization	Usually bad	Yes
Algorithm	Dynamic Programming	Greedy
Sync Overhead	Higher	Lower

Results

Reduced Parallelization Effort

Kremlin eliminates unnecessary parallelization, reducing the parallelized region count by ***1.57x***.

Improved Results

Kremlin attained much higher (**1.86x** and **1.46x)** speedup on two benchmarks. On the other nine, it averaged **.96x**, with an average reduction in region count of **1.62x**. (The saved time could be applied to more profitable serial optimization!)

Methodology

We parallelized 11 benchmarks according to Kremlin's parallelization plan, and compared them against manually parallelized versions from experts.

11 Benchmarks

All NAS Parallel Bench: *{bt, cg, ep, ft, is, lu, mg, sp}*
SpecOMP in C: *{ammp, art, equake}*

Target Platform

32 Core System (8 * Quad-core AMD 8380)
GCC 4.1 with OpenMP Library

Read our paper in PLDI 2011!