# POSH: A TLS Compiler that Exploits Program Structure

**Wei Liu**, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau[†] and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

[†]Computer Engineering Department
University of California, Santa Cruz

# CMPs are available now!

- Chip multiprocessors (CMPs) have arrived
  - Pentium D, Opteron, Power5, Niagara, ...

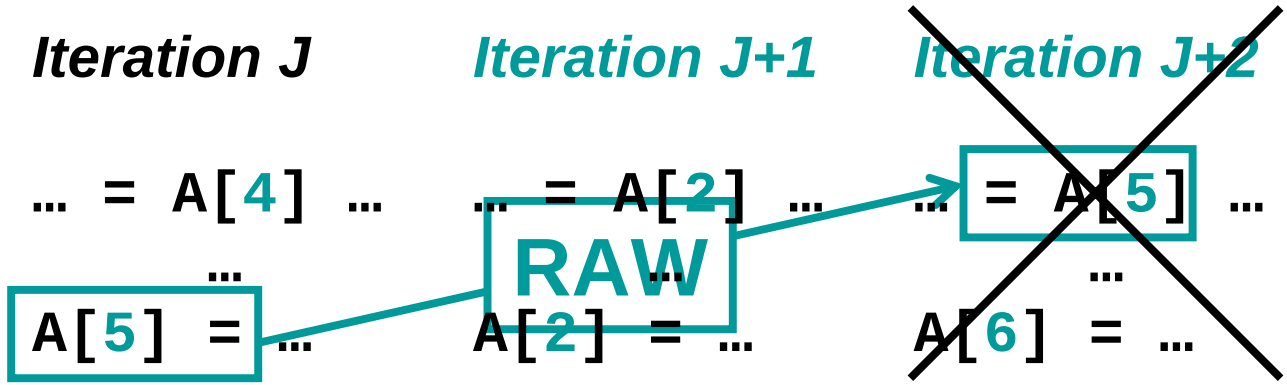
- Good speedups for parallel programs
- How to speed up single-threaded applications?
  - Especially the hard-to-parallelize applications, such as SPECint.

➡ Thread-Level Speculation (TLS)

# What is Thread Level Speculation (TLS)?

- Execute potentially-dependent tasks in parallel
  - Assume no dependence across tasks will be violated
- **TLS hardware**
  - Track memory accesses; buffer unsafe state
  - Detect any violation (RAW)
  - Squash offending tasks, repair polluted state, restart tasks

```
for(i=0;i<n;i++){
  … = A[B[i]] …
        …
  A[C[i]] = …
}
```



# Key to the Acceptance of TLS

- Fully automated TLS compilers
  - Do not need to prove the independence between tasks
  - Crucial impact on performance
    - How to break the code into tasks
    - When to spawn the tasks

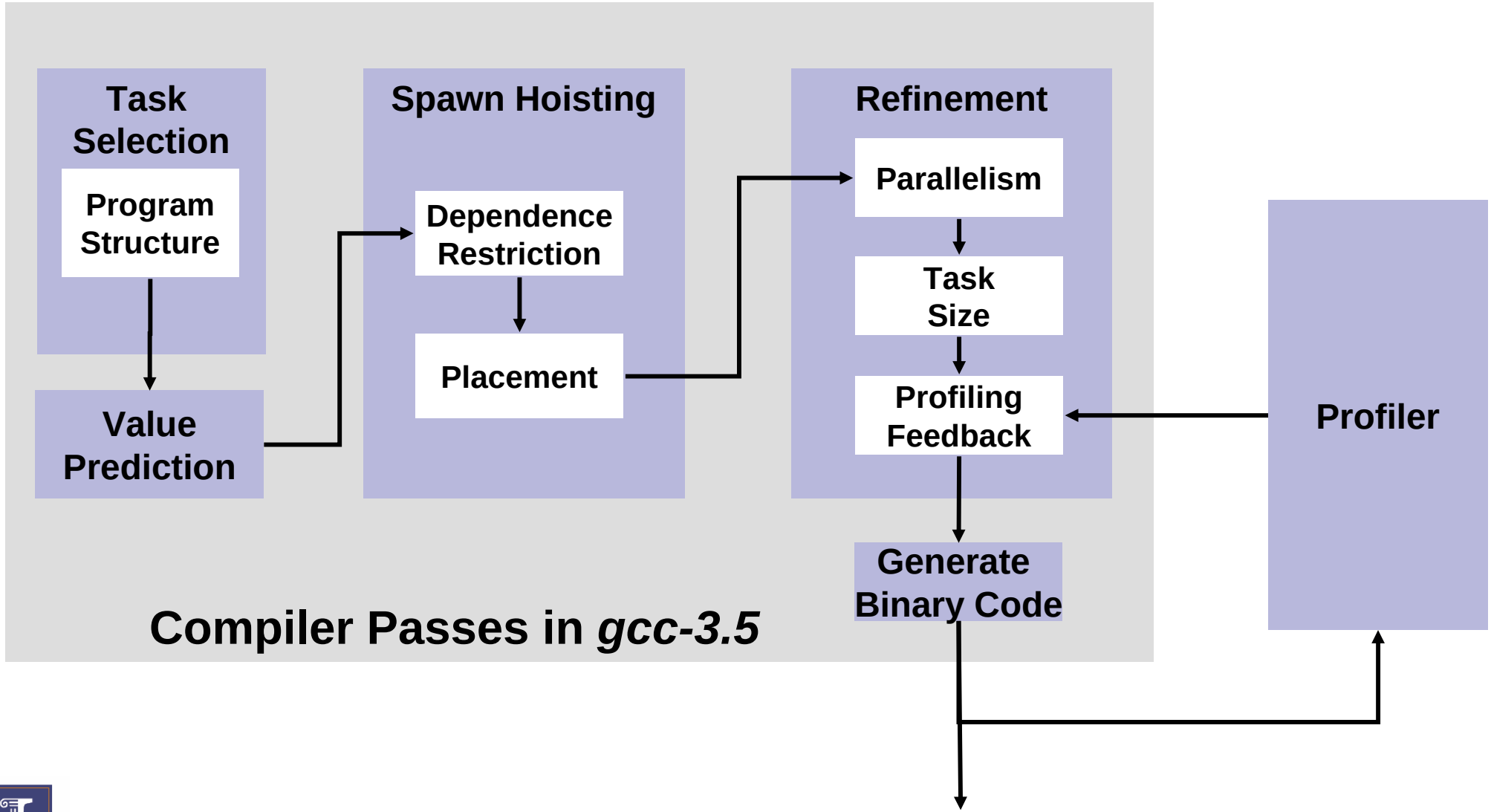
# Contributions

- *POSH*, a fully automated TLS compiler infrastructure
  - Exploits code structure (loop iterations and subroutines of any nesting level) for tasks
  - Uses a simple profiling pass to discard ineffective tasks
    - Leverages both parallelism and **data prefetching**
- Speedup: 1.30 on average for SPECint 2000
- Detailed characterization of speedup sources and task behavior.
  - 26% of the speedup is from prefetching (i.e. 8% w.r.t. the baseline)

# Outline

- Background of TLS
- Contributions of POSH
- POSH: A TLS Compiler
- Experimental Results
- Conclusions

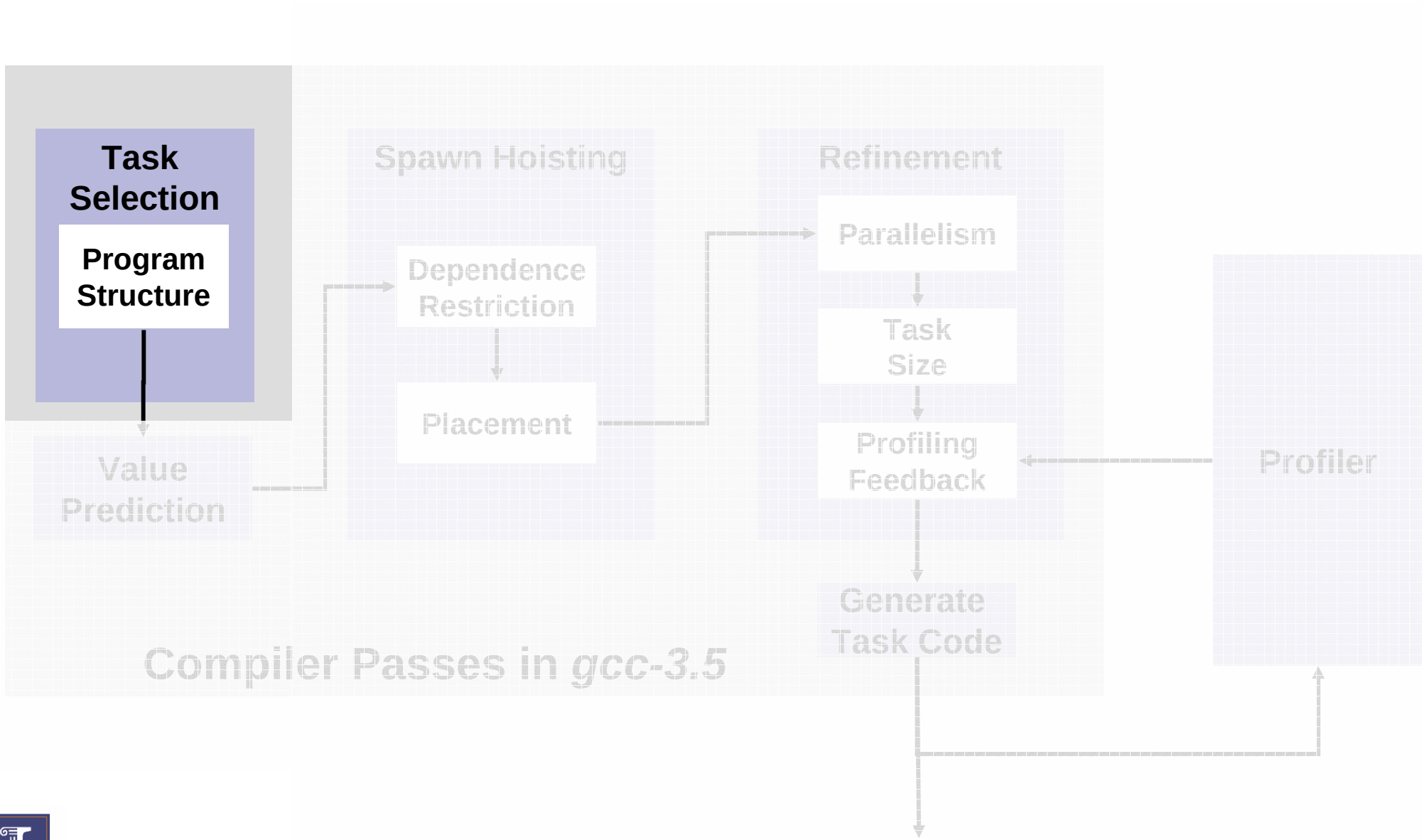
# Flowchart of the POSH Framework



# Hardware Assumptions

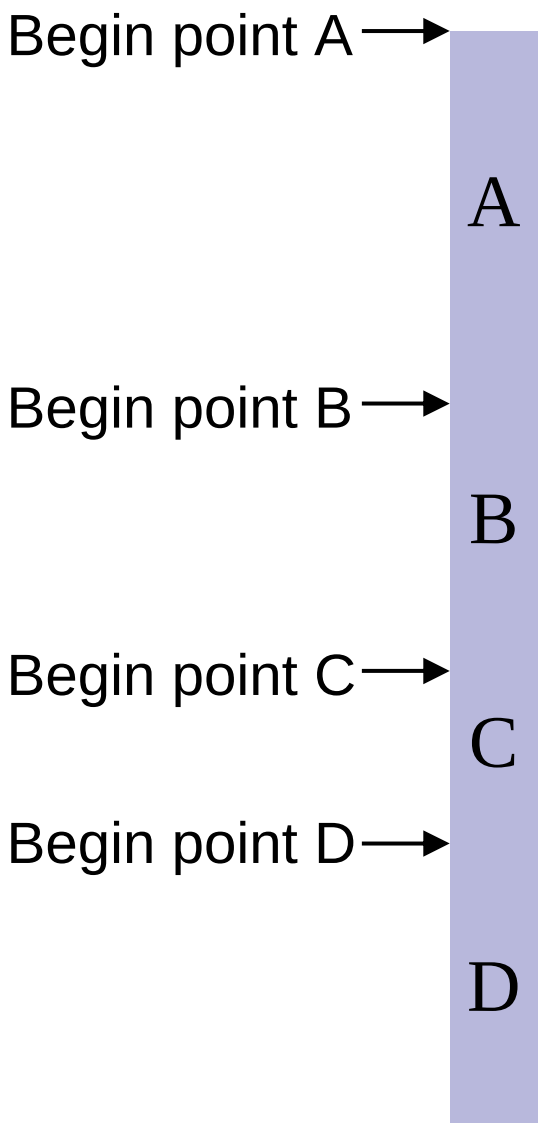
- No special hardware for register communication
  - Dependence detection through memory
- Support *spawn* and *commit* instructions

# Task Selection

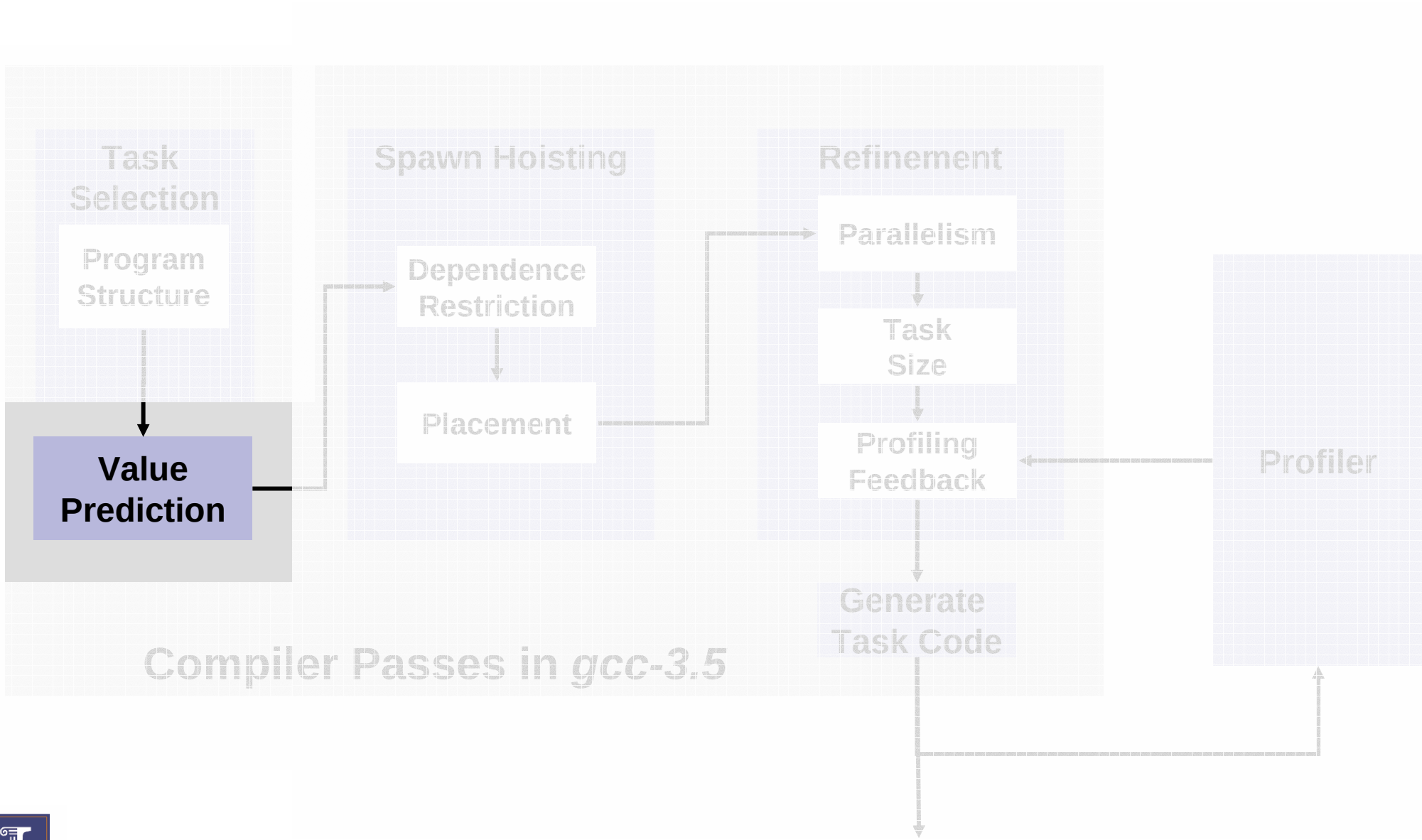


# Task Selection

- Break the sequential program into tasks
- Select the following structures
  - Subroutines
  - Subroutine continuations
  - Loop iterations
  - Loop continuations
  - Continuation == Code region that follows subroutines or loops
- Mark the begin point for each task



# Value Prediction

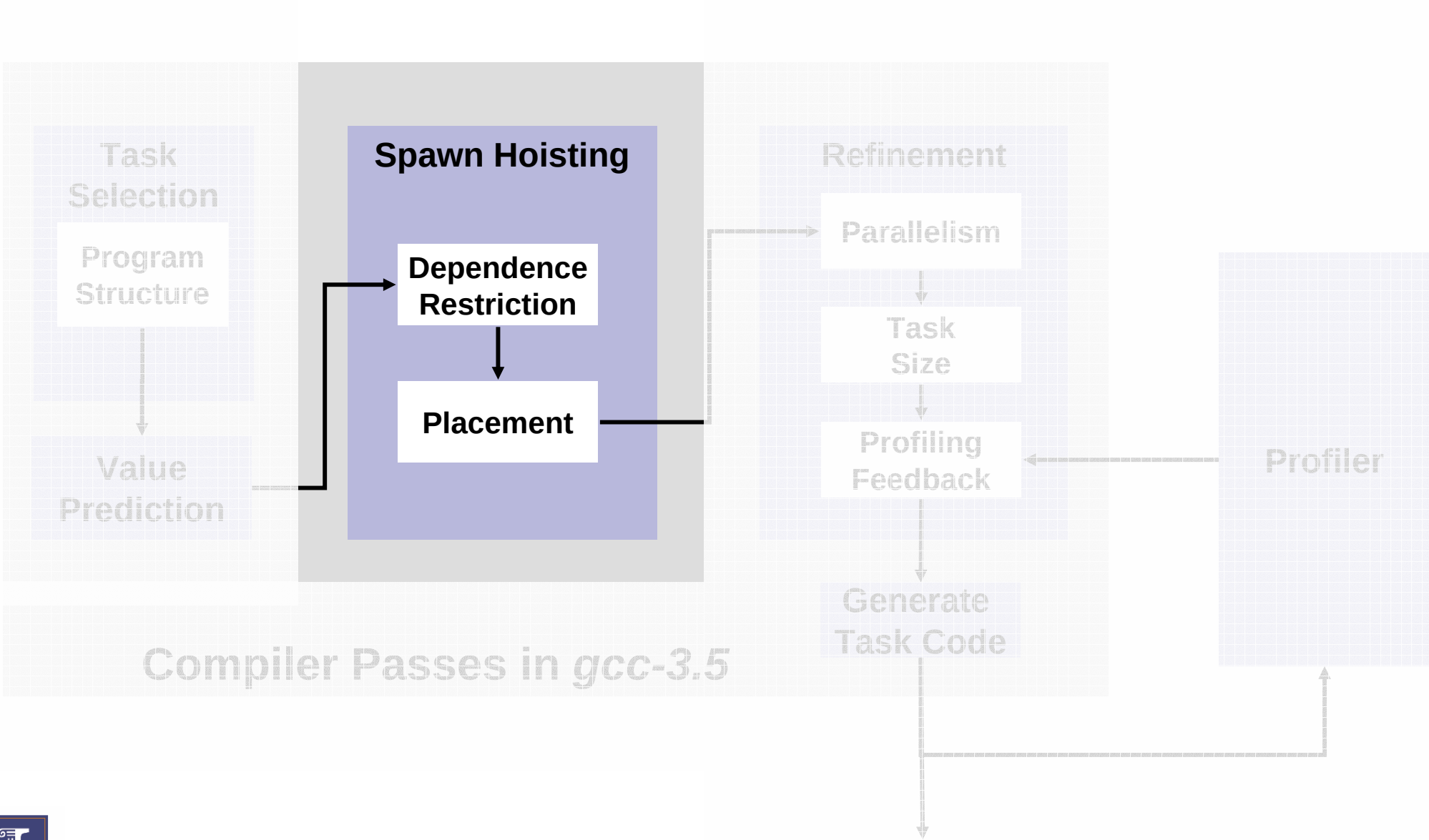


# Value Prediction

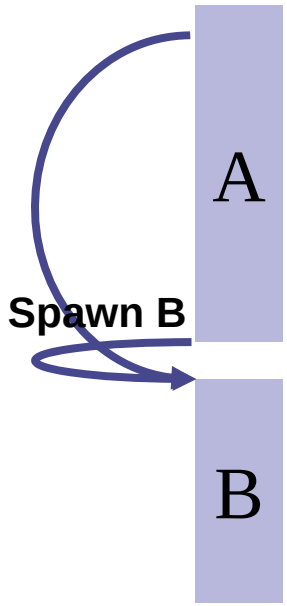
- Predict the values of variables that cross task boundaries
  - reduce violations
- Software Value Predictor
  - Function return variables
  - Loop induction variables
  - Induction-like variables

```
for (i = 0; i < 100; i++) {
   …
   if (result == 0) {
      j++;  /* induction-like variable */
   }
   …
}
```

# Spawn Hoisting

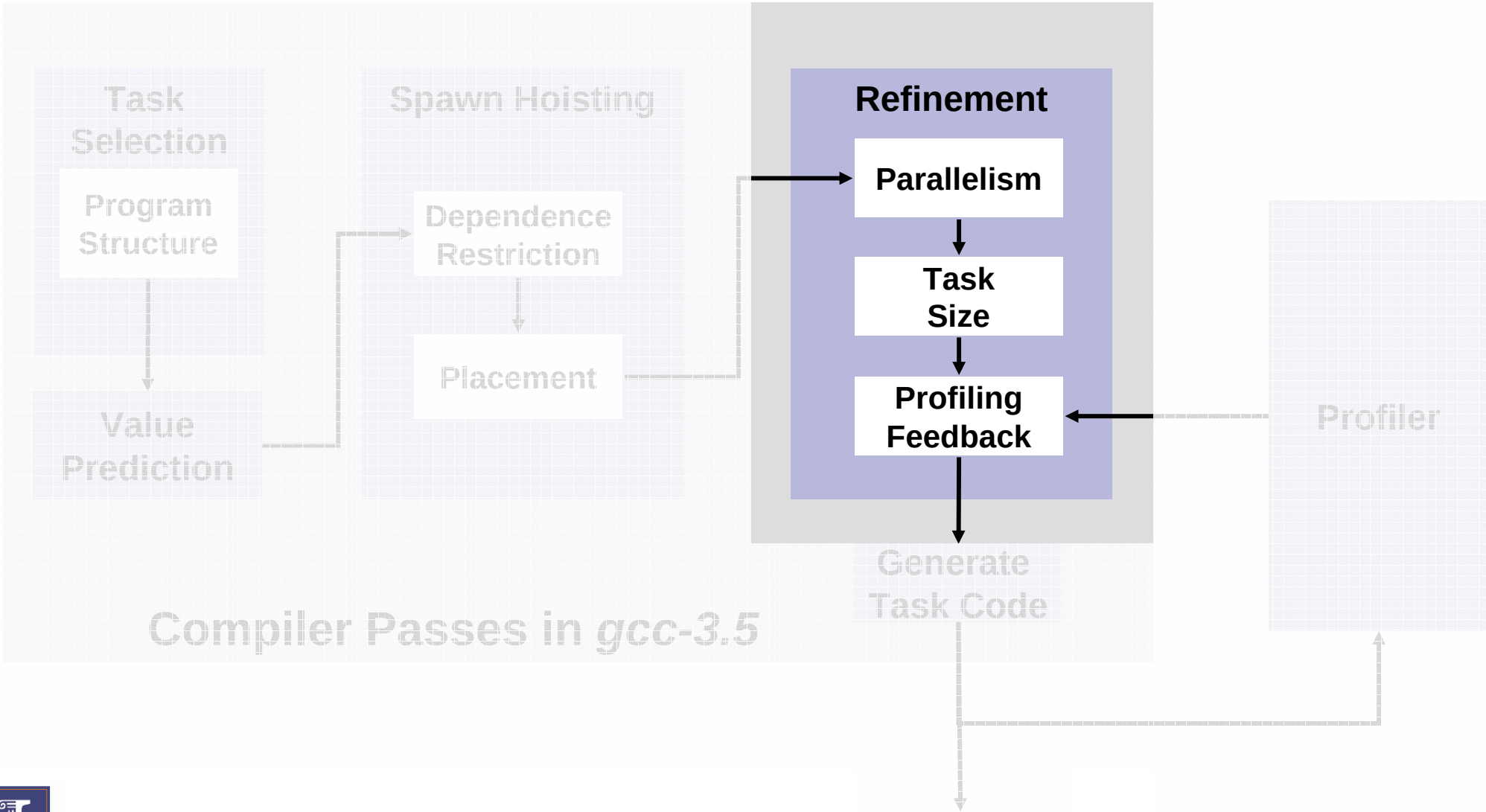


# Spawn Hoisting

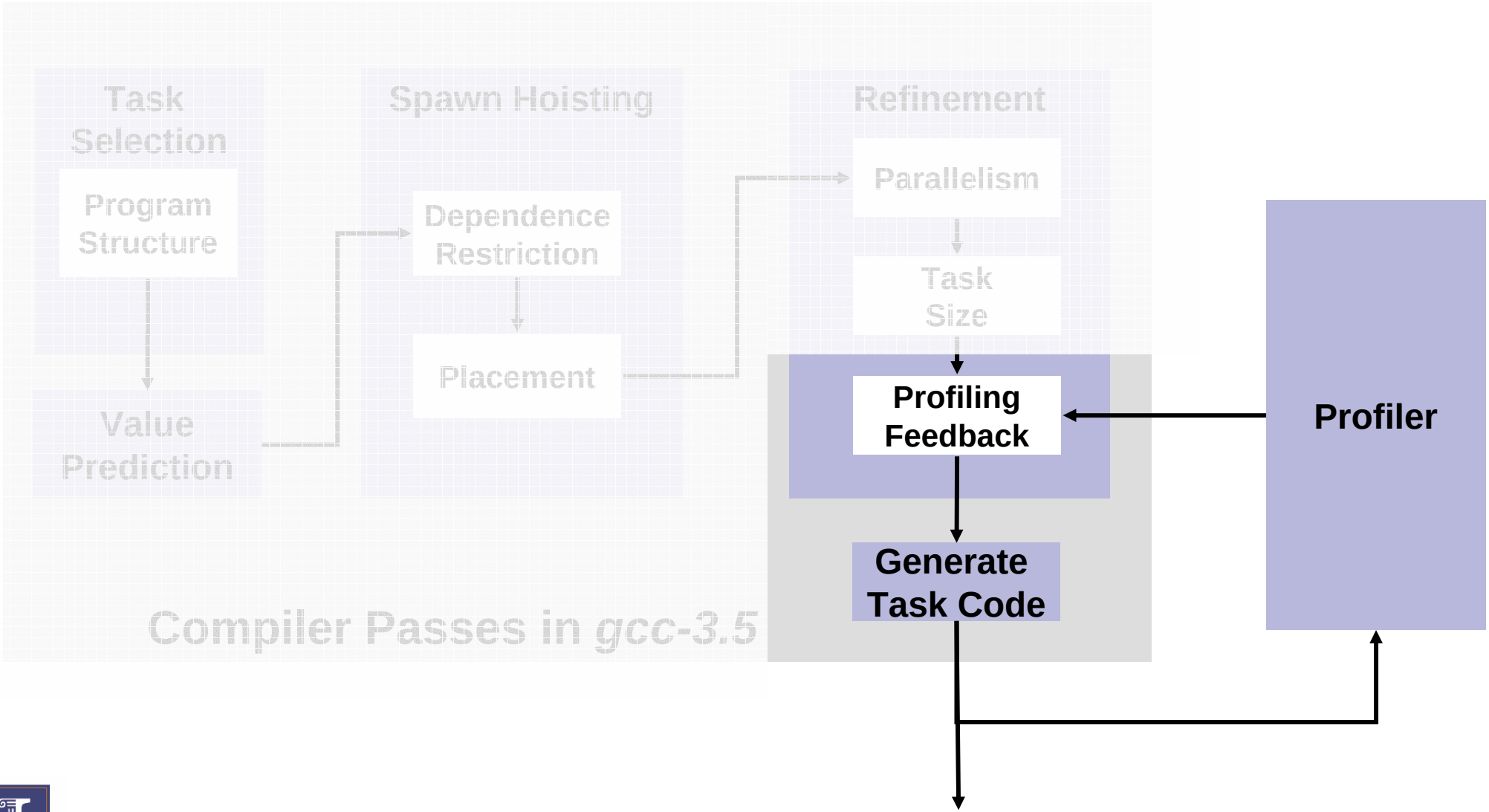



- The spawn instruction is *hoisted* relative to the beginning of a task
- Restrictions on hoisting (See paper for more details)
  - Definition of any variables in the task
  - Control safe location
  - Reverse sequential order for multiple spawn instructions

# Refinement


Task Selection
Program Structure
Value Prediction
Spawn Hoisting
Dependence Restriction
Placement
Refinement
Parallelism
Task Size
Profiling Feedback
Profiler
Generate Task Code
Compiler Passes in gcc-3.5

# The POSH Profiler



# POSH Profiler

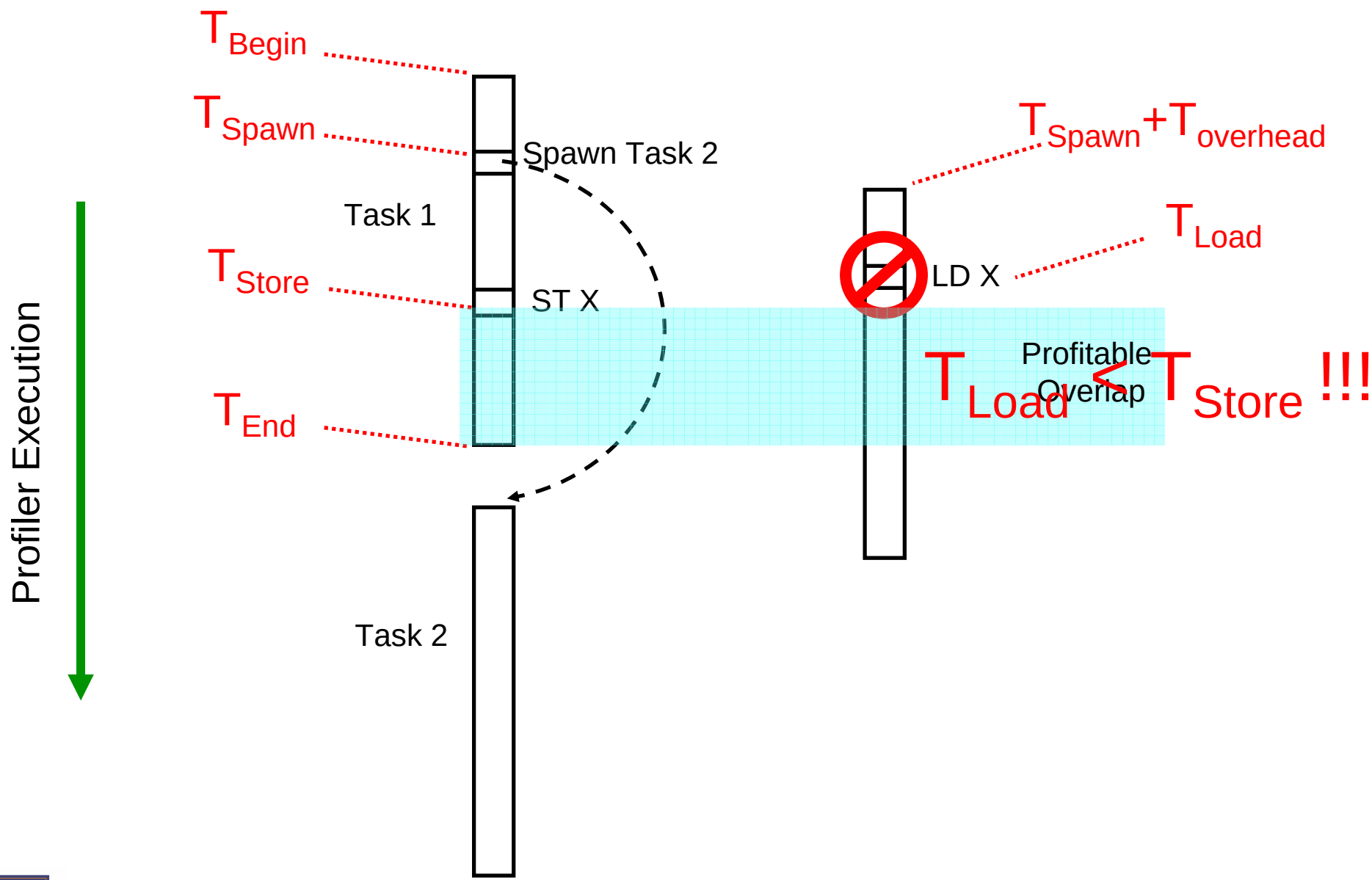
- Runs a TLS binary with tasks selected by the compiler
  - Uses *train* input set
- Executes the TLS binary *sequentially*
  - No TLS architectural support assumed for generality
  - With rudimentary timing, it can collect information about potential run-time violations
- Provides a simple L2 cache model to estimate misses
- Not tied to a fixed number of processors

# Profiling Pass

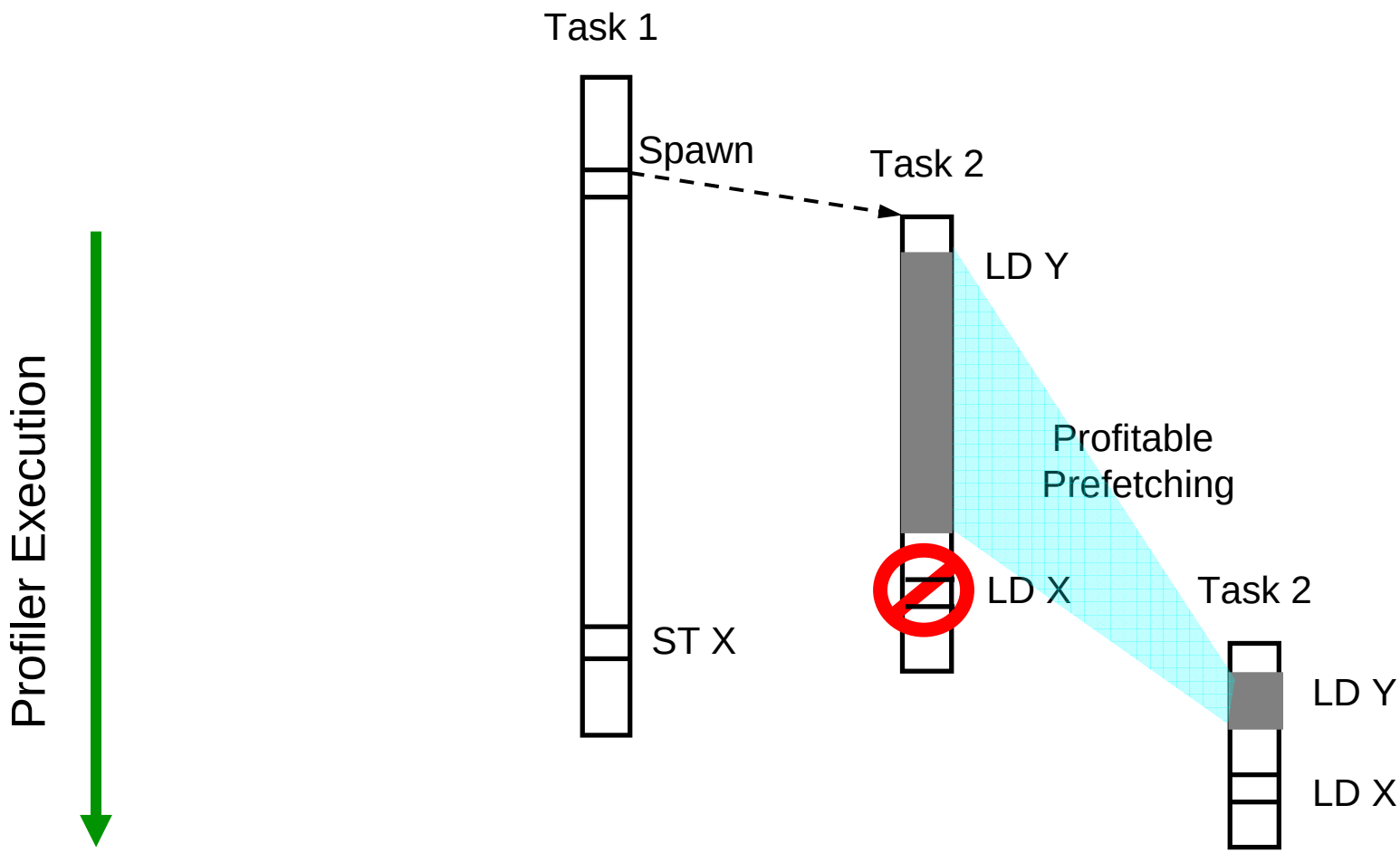
- Goal: Minimize *Execution Time*
- Preserve parallelism
  - Find tasks with the best overlap
- Reward prefetching due to task squashes
  - Keep squashed tasks that prefetch even with small or no overlap

$$Benefit = Overlap + Prefetching$$

# Estimate Overlap



# Estimate Prefetching

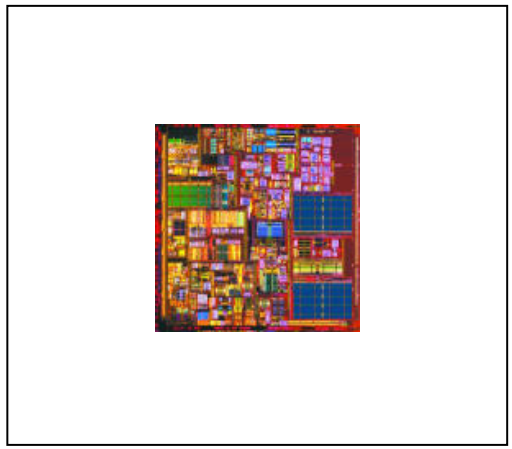


# Outline

- Background of TLS
- Contributions of POSH
- POSH: A TLS Compiler
- Experimental Results
- Conclusions

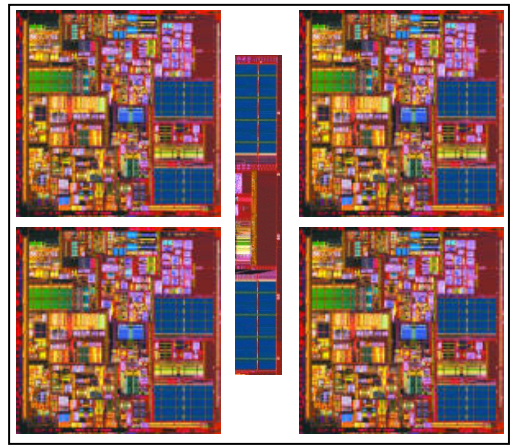
# Methodology

**Serial**



One 3-issue core

**TLS CMP**



Four 3-issue cores with TLS

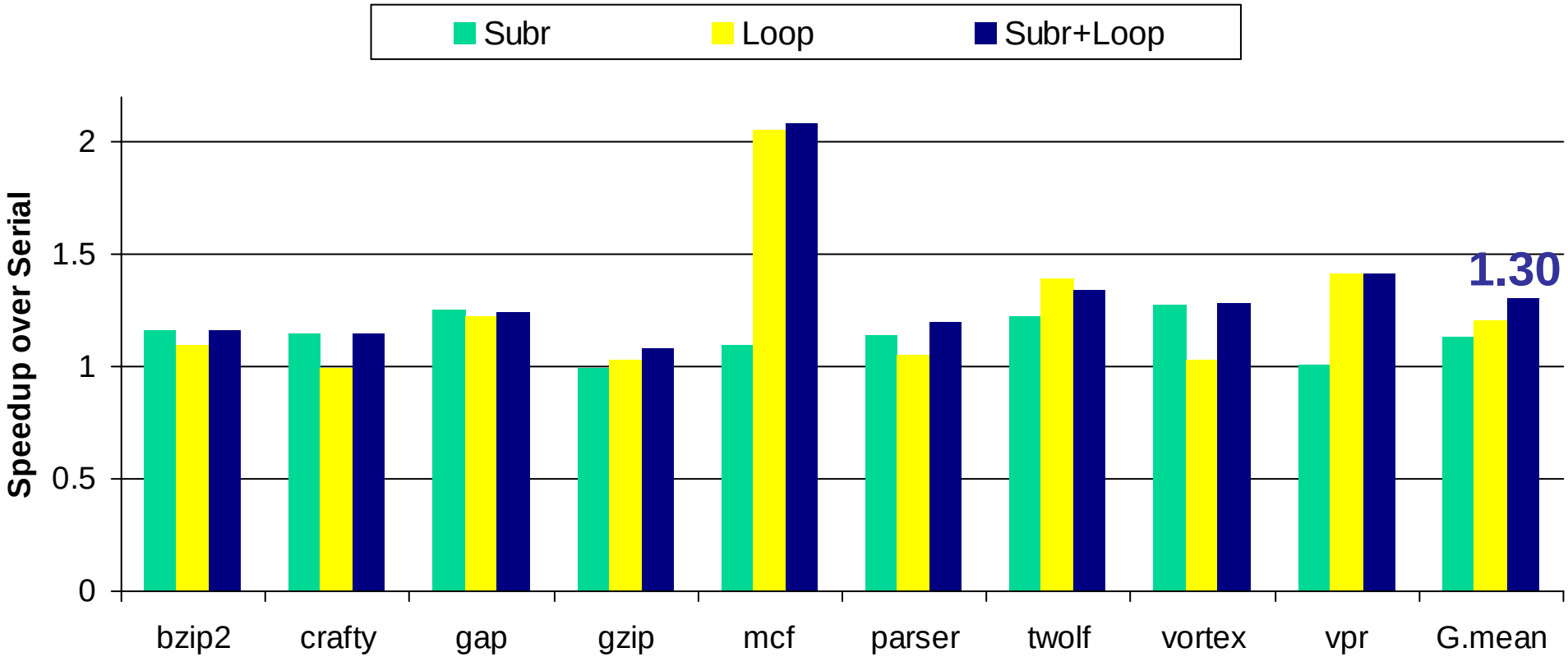
- Simulated CMP architecture
  - 4 GHz
  - Private 16k L1 per core; 1 MB Shared L2
- Unmodified SPECint 2000 applications

# Task Characterization

- Static Information
  - On average 32.2 *Subr* Tasks
  - On average 7.3 loops with *Loop* Tasks
  - About 6 live-ins and 6 live-outs per task

- Dynamic Information
  - Average task size: 476 instructions
  - Most tasks have 50-1000 dynamic instructions
  - 45.7% tasks commit

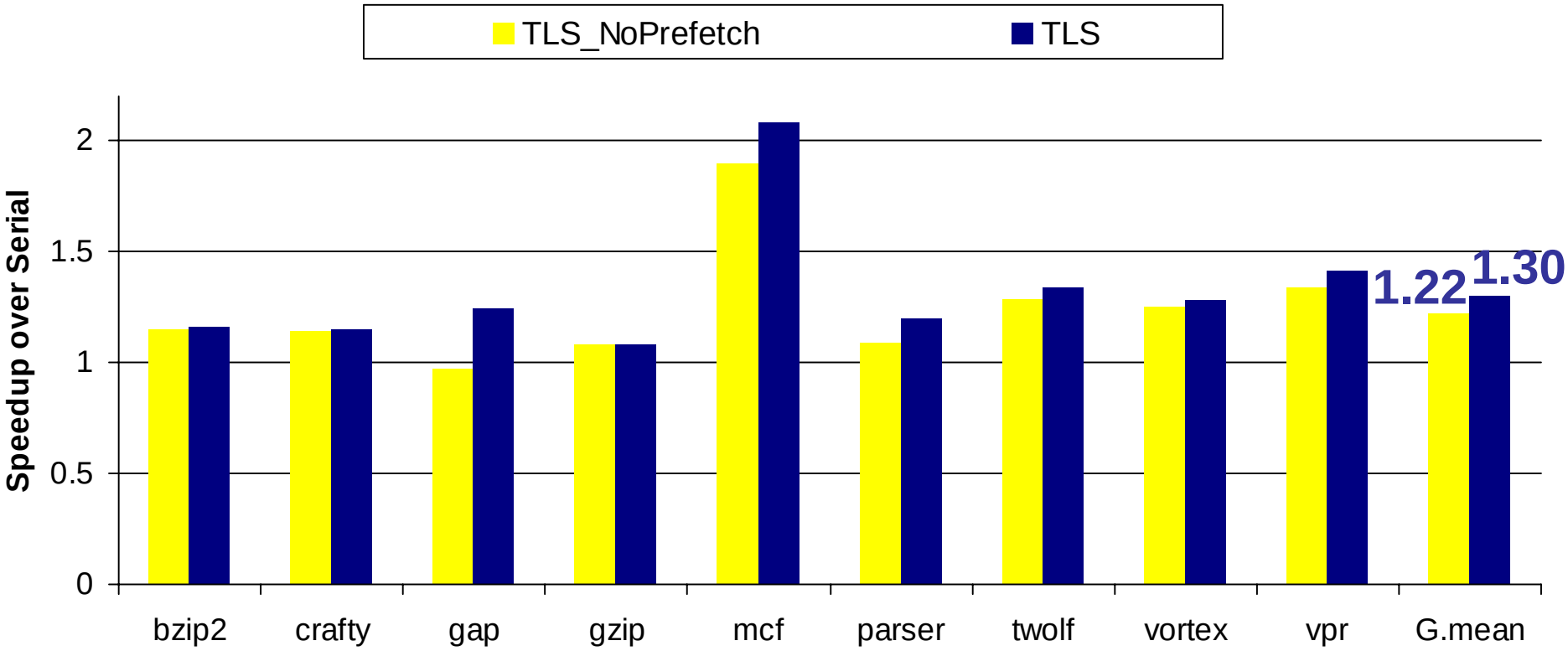
# Application Speedups




TLS: an average speedup of **1.30**!

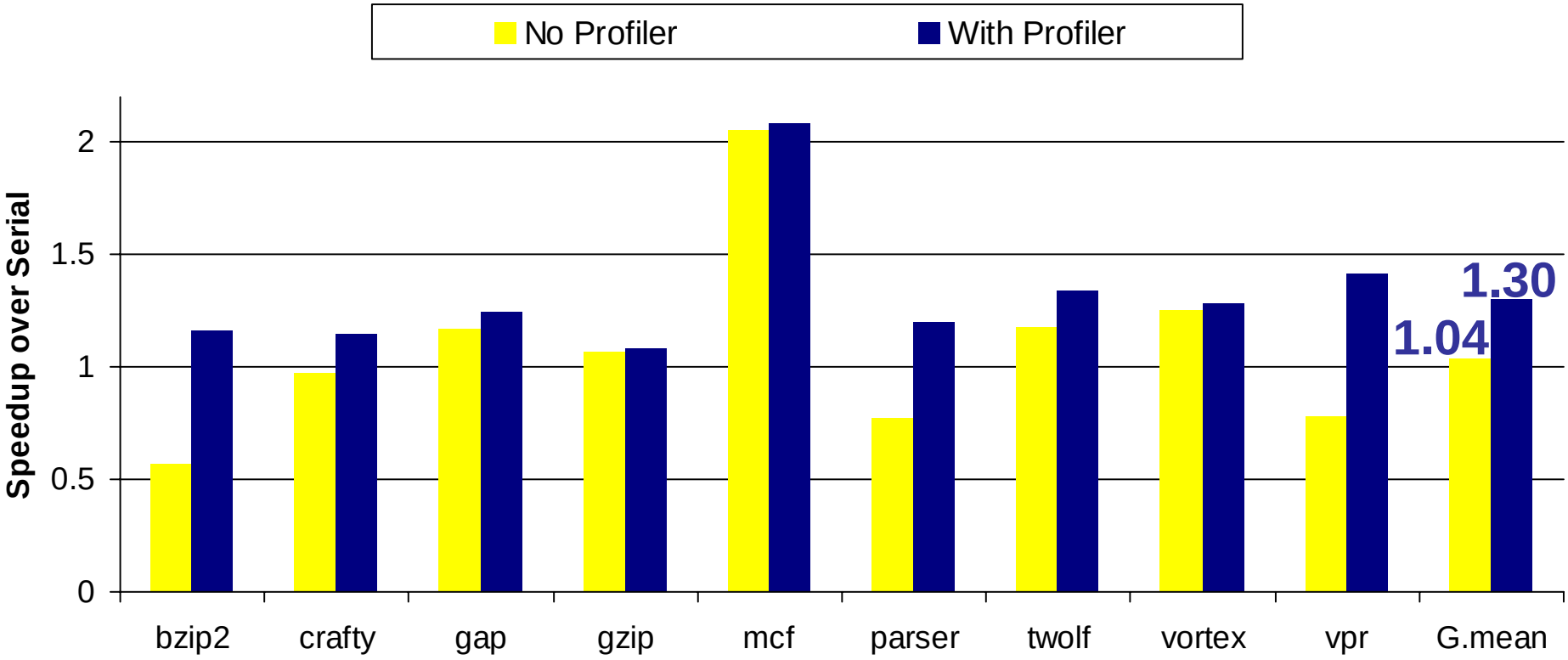
Using both forms of tasks is simple and effective

# Contribution of Prefetching




About a quarter of the speedup is from prefetching

# Effectiveness of Profiler




- Profiling is needed for good performance
- Profiler reduces the average number of tasks from 198 to 39

# Conclusions

- POSH: A fully automated TLS compiler
  - Simple: Use program structure
  - Effective: Prefetching + Parallelism
  - Speedup: on average **1.30** for SPECint 2000

# POSH: A TLS Compiler that Exploits Program Structure

**Wei Liu**, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau[†] and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

[†]Computer Engineering Department
University of California, Santa Cruz