POSH:
A Profiler-Enhanced TLS Compiler that Leverages Program Structure

Wei Liu, James Tuck, Luis Ceze, Karin Strauss, and Josep Torrellas

University of Illinois at Urbana-Champaign

http://iacoma.cs.uiuc.edu

Jose Renau

University of California at Santa Cruz
http://masc.soe.ucsc.edu

This work was funded by DARPA-IBM under PERCS.

Introduction

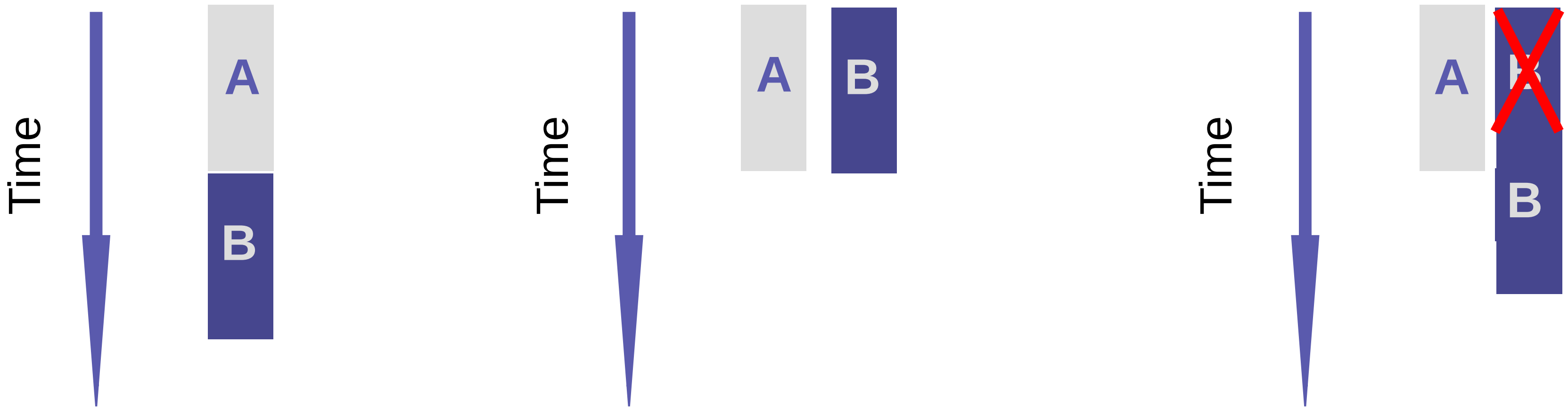
- Chip multiprocessors (CMPs) have arrived
  - Pentium D
  - Opteron dual core
  - Power5
  - Niagara
- How do we speedup hard-to-parallelize single-threaded applications on CMPs?

➡ Thread-Level Speculation

Thread-Level Speculation (TLS)

- TLS Hardware
  - Tracks data accesses at run-time
  - Detects dependence violations
  - Squashes and restarts tasks

Sequential **TLS (no dep violations)** **TLS (dep violations)**



Motivation

- Most TLS work consists of proposing architectural variations
  - We need to turn our focus to thread extraction
- Automated TLS compilers are key to TLS acceptance

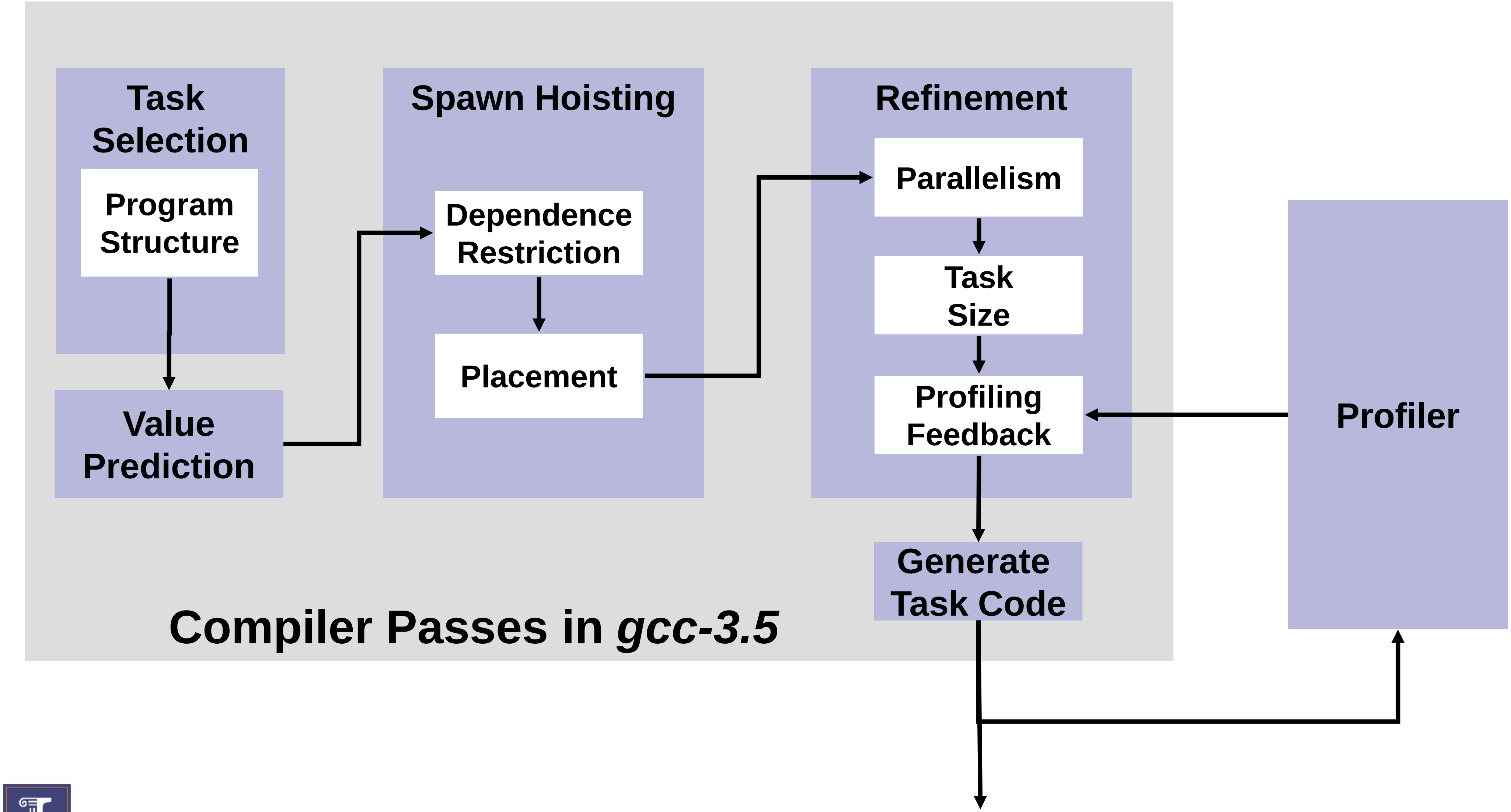
Main Contributions

- POSH is a fully automated TLS compiler for SPECint class applications
    - Simple
        - Leverages code structure (loops, subroutine continuations) for tasks
    - Effective
        - Average speedup for SPECint applications of 1.28 over sequential on a 4-core CMP
    - Leverages both parallelism and data prefetching
- We evaluated the impact of several key decisions:
    - Task structures
    - Use of a profiler
    - Value prediction

Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation

Flowchart of the POSH Framework



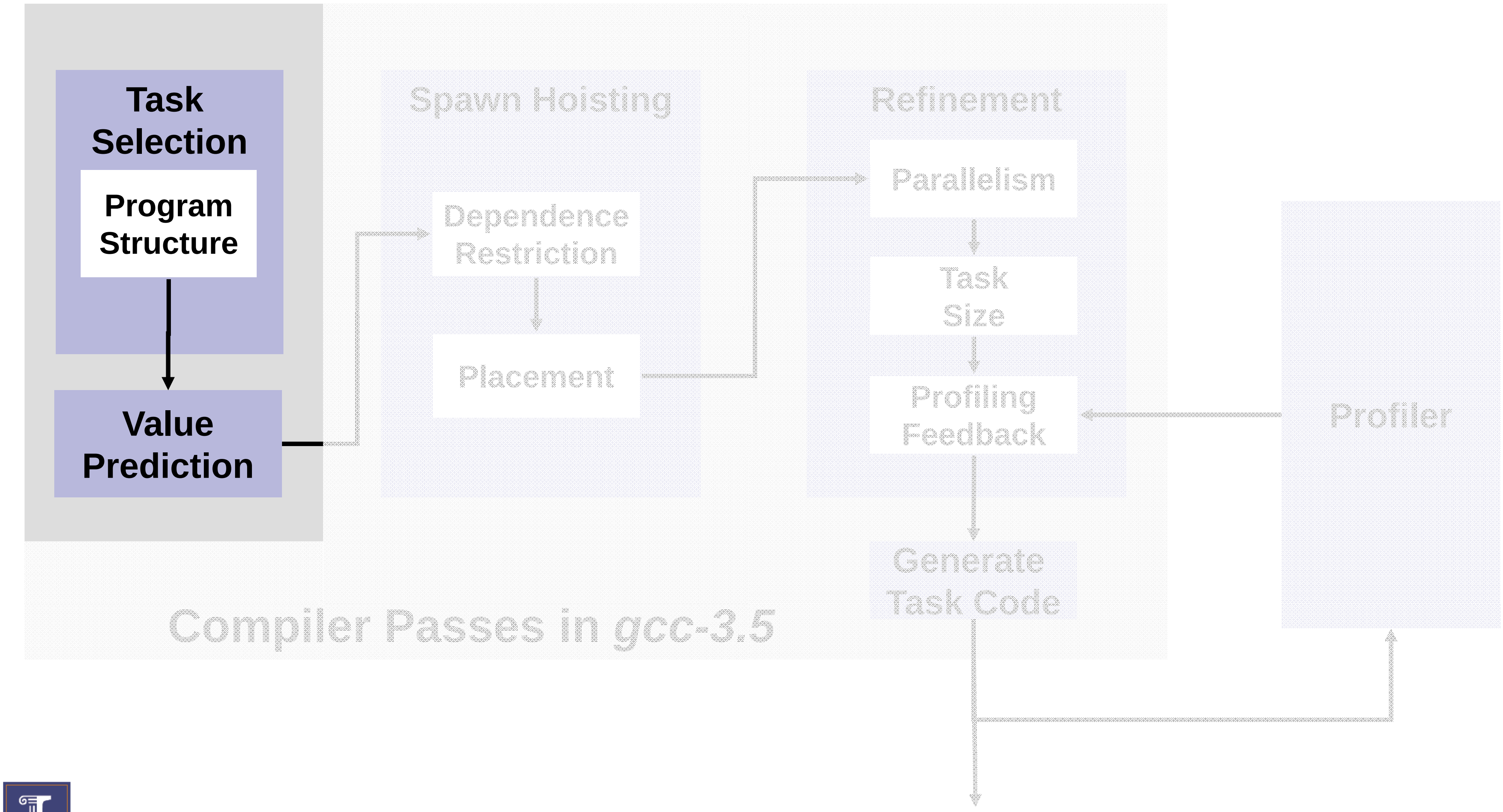
CMP Hardware for TLS

- All data dependences are carried through memory
  - No special hardware for register communication
- Support task *spawn* and *commit* instructions

Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation

Task Selection



Task Selection

Program Structure

Value Prediction

Spawn Hoisting

Dependence Restriction

Placement

Refinement

Parallelism

Task Size

Profiling Feedback

Generate Task Code

Profiler

Compiler Passes in *gcc-3.5*

Task Selection

- Break the sequential program into tasks
- Select tasks without regard for nesting
- Select the following structures
  - Subroutine continuations
    - Restrict based on return value and size
  - Loops
    - Consider all loops found using strongly connected components

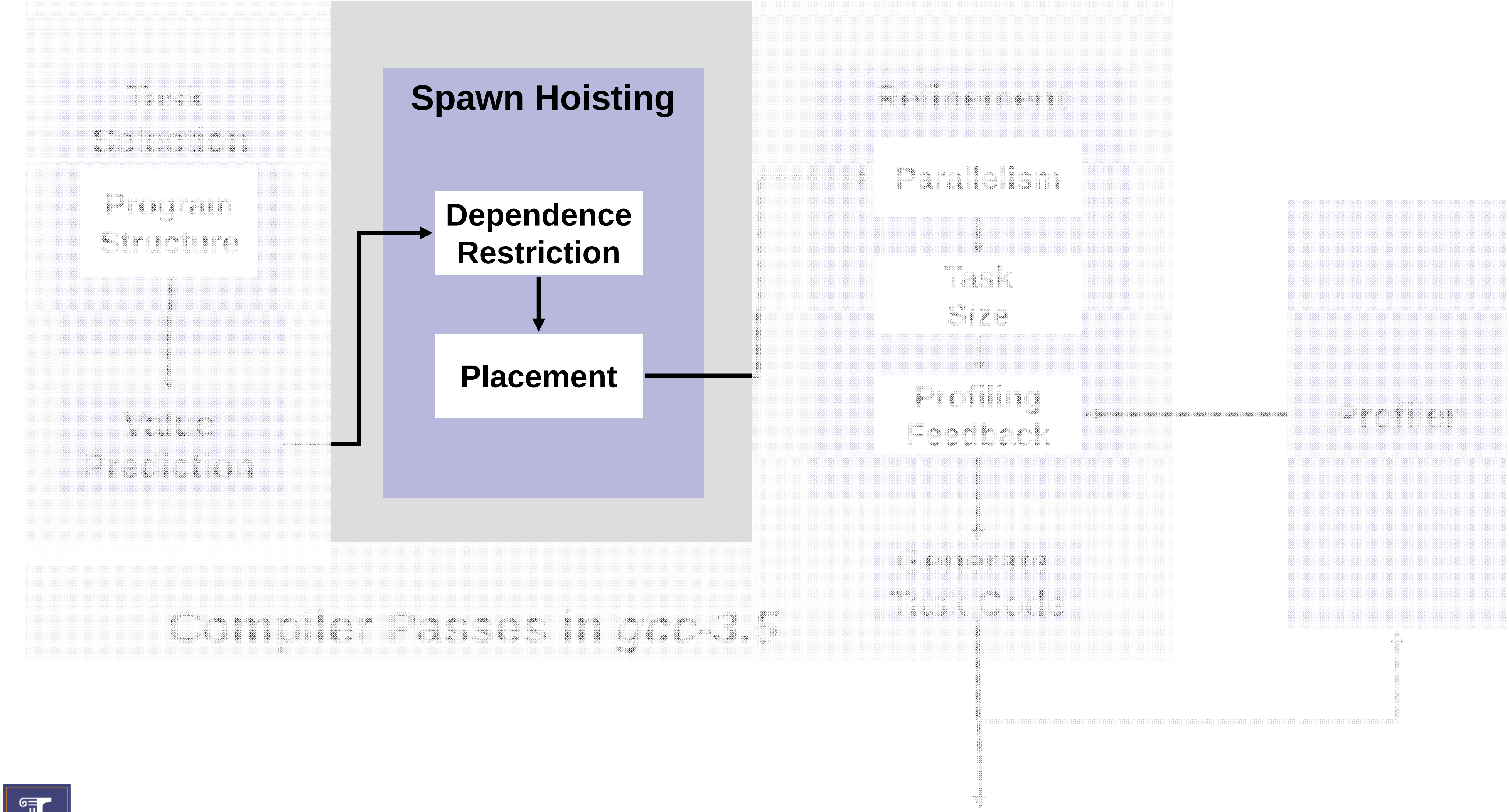
A

B

C

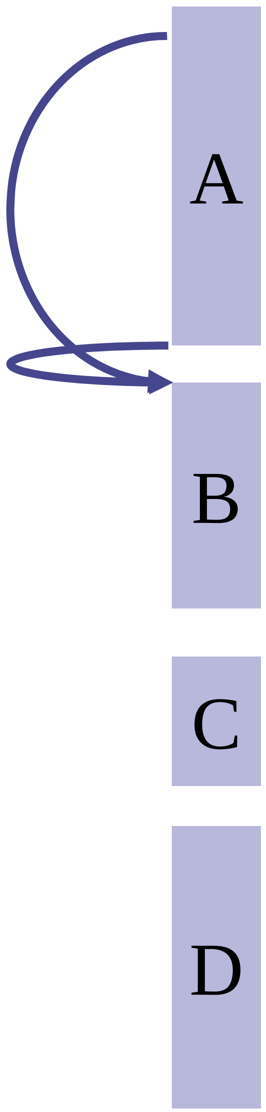
D

Spawn Hoisting

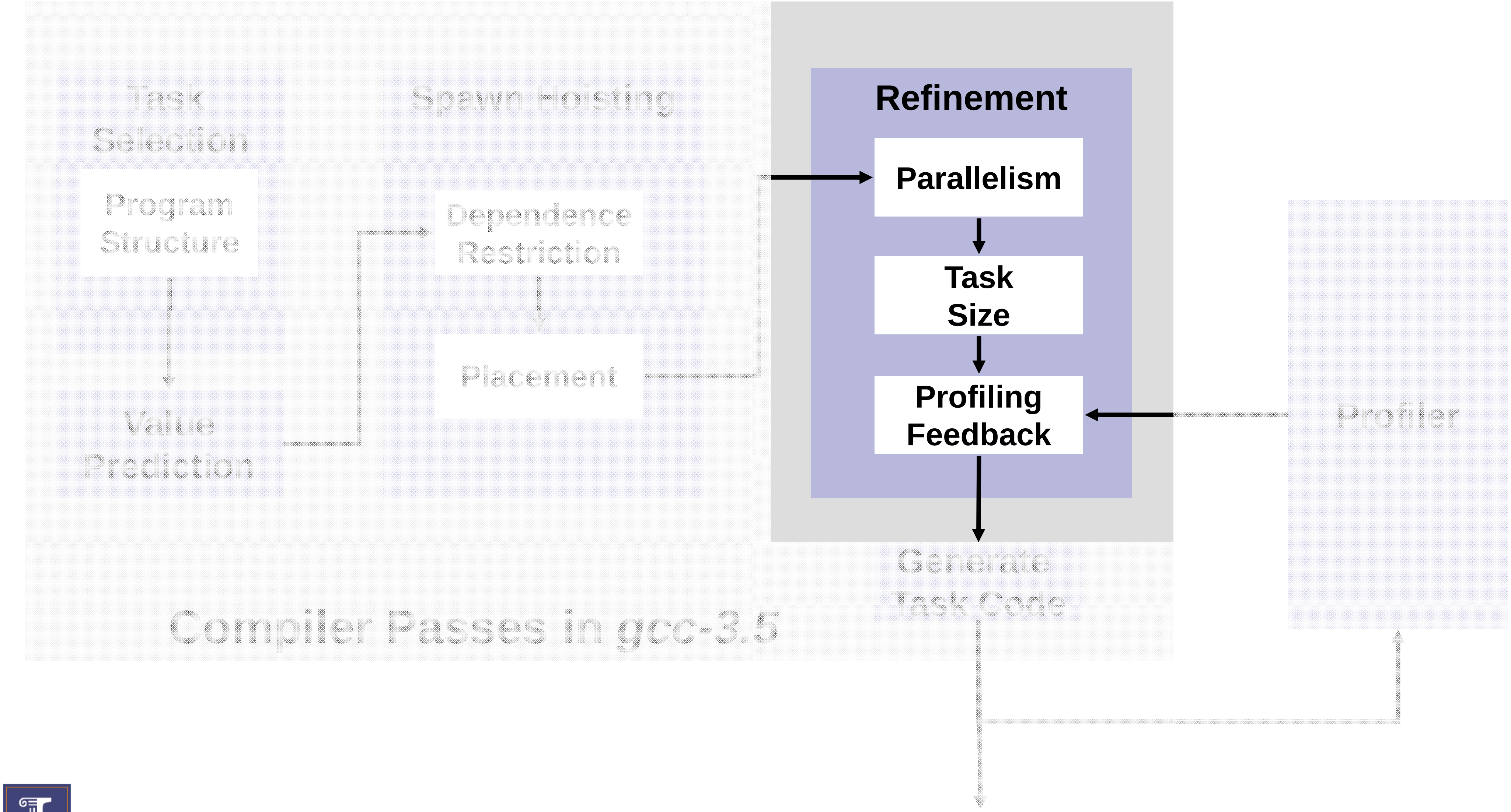


Spawn Hoisting

- The spawn instruction is *hoisted* relative to the beginning of a task
- Goal is to provide parallelism and prefetching
- Restrictions on hoisting
  - Spawn after the definition of variables used in the task
  - Spawn must be control safe
    - We avoid complications of control misspeculation



Refinement

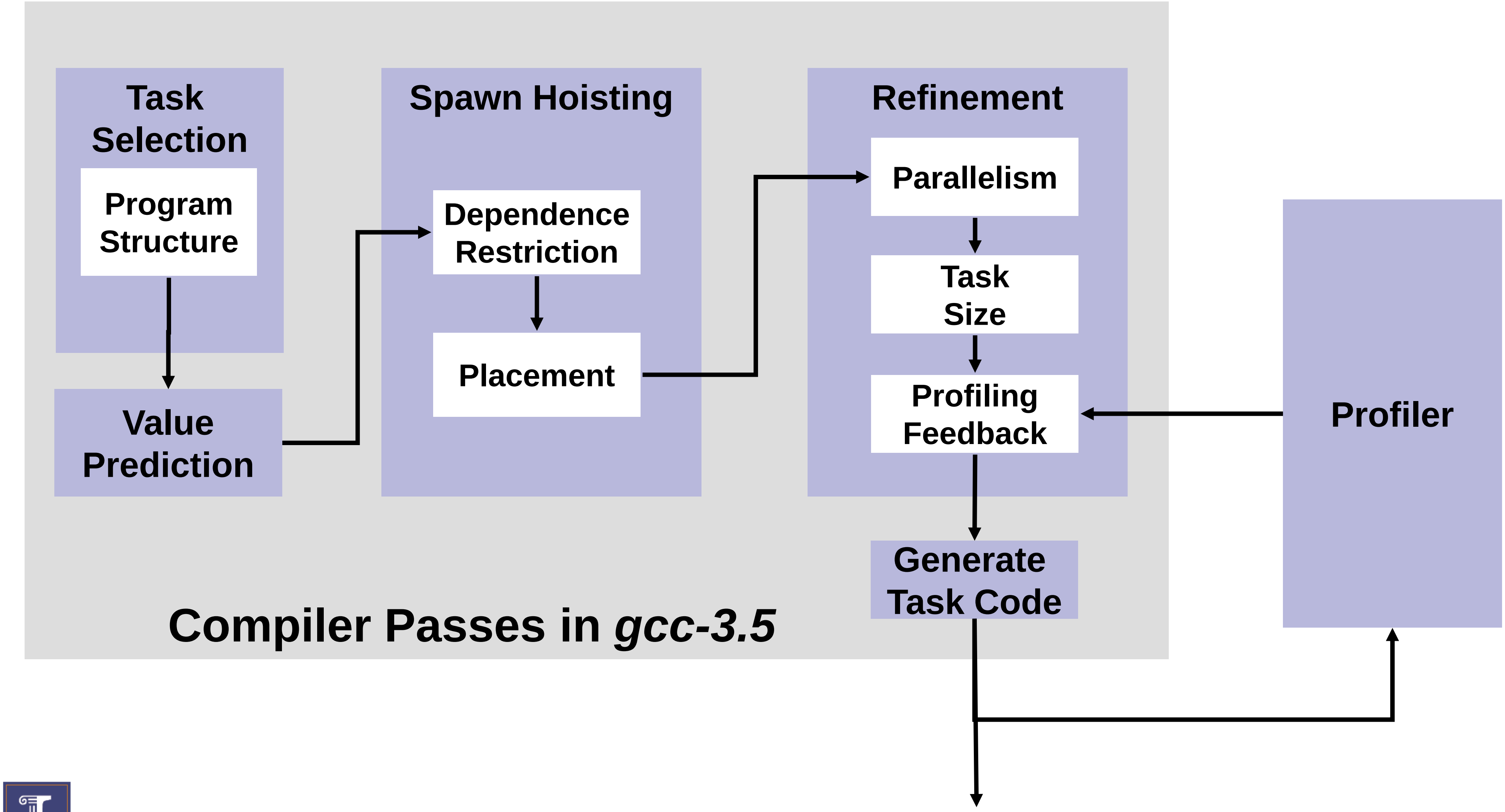

Task Selection
Program Structure
Value Prediction
Spawn Hoisting
Dependence Restriction
Placement
Refinement
Parallelism
Task Size
Profiling Feedback
Profiler
Generate Task Code
Compiler Passes in *gcc-3.5*

Static Task Refinement

- Static refinement eliminates tasks:
  - Determined to be very small
  - Offering little parallelism due to small hoist distance

- Feedback by profiling with small data set size:
  - List of most beneficial tasks

Generate Code For Tasks



Task Selection

Program Structure

Value Prediction

Spawn Hoisting

Dependence Restriction

Placement

Refinement

Parallelism

Task Size

Profiling Feedback

Generate Task Code

Profiler

Compiler Passes in *gcc-3.5*

An Example of the Compiler at Work

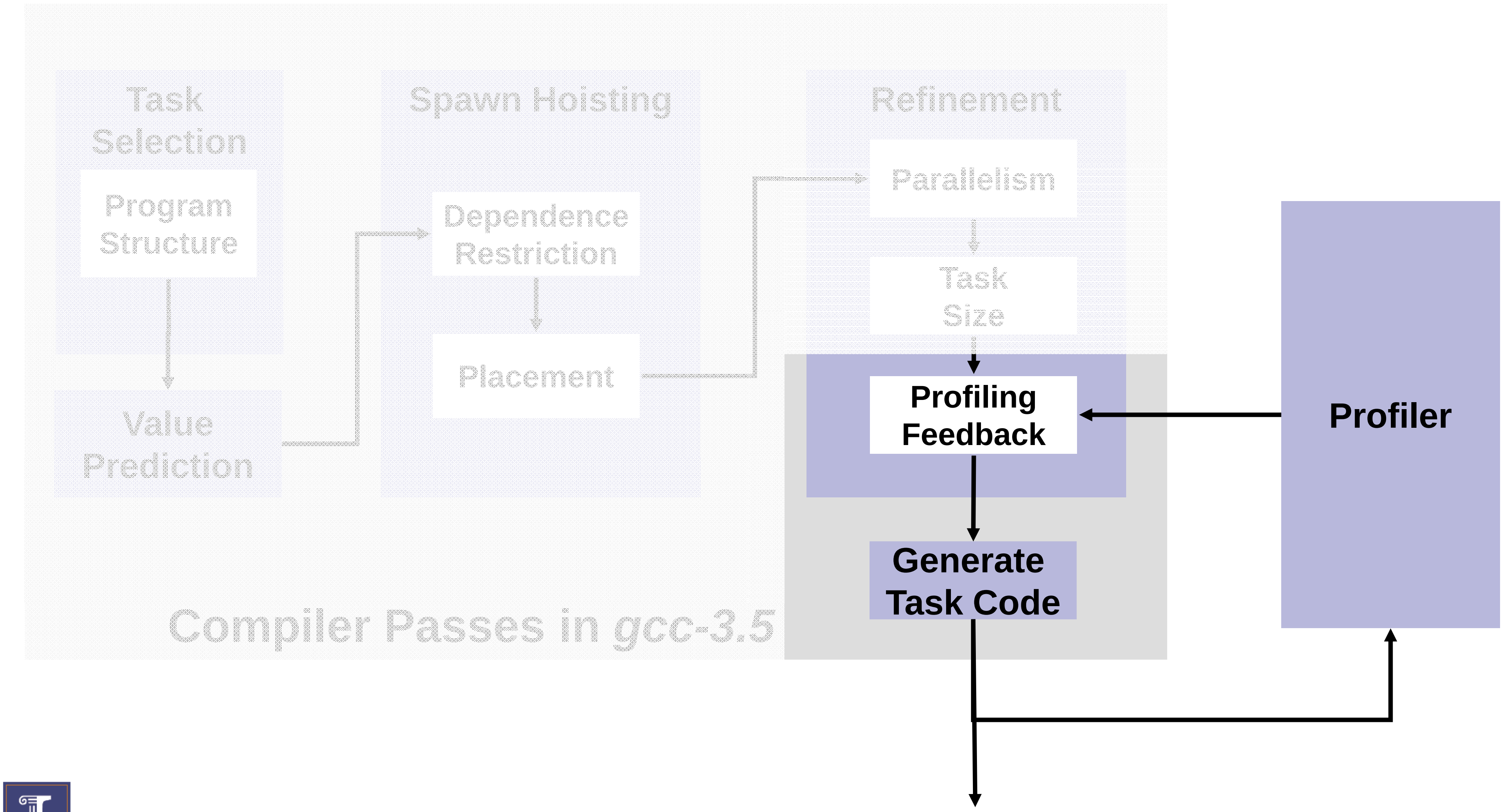
- First select the task
    - Task label, commit
- Identify spawn location
- Pass dependences through memory

```
       int i=0;

       …

Loop:
       if( i > 99 )
         goto Lend;

       <LOOP BODY>

       i = i + 1;

       goto Loop;
Lend:
```

Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation

The POSH Profiler

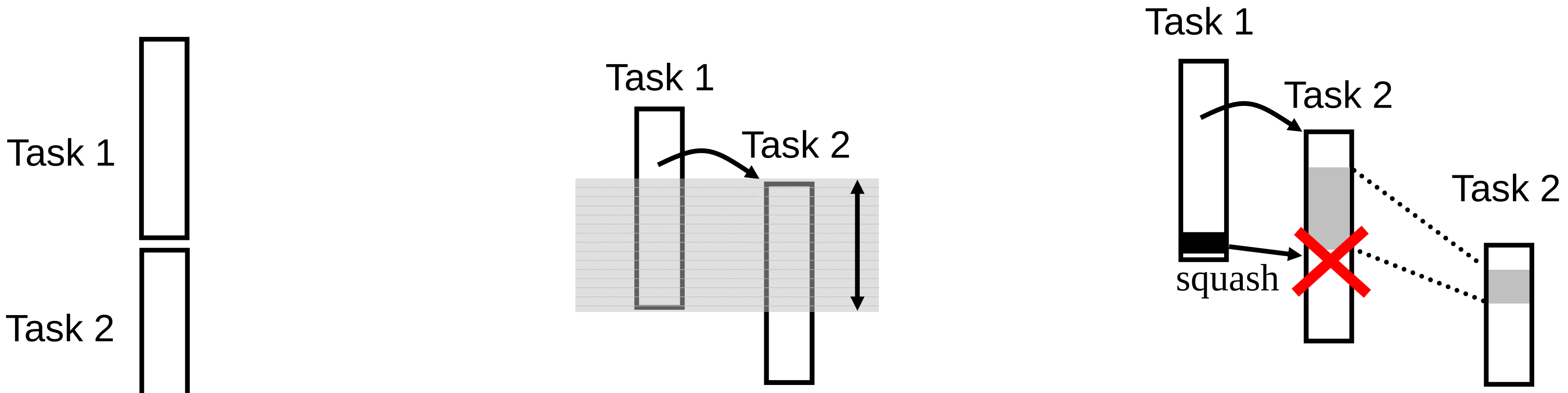


Profiler Details

- Runs a TLS binary with all tasks selected by the compiler
  - Uses train input set for SPECint
- Executes the TLS binary *sequentially*
  - No TLS architectural support assumed for generality
- Provides a simple L2 cache model to estimate misses
- Not tied to a fixed number of processors

The Profiler’s Goals

- Preserves parallelism
  - Keeps tasks with good overlap
- Rewards prefetching
  - Keeps tasks that prefetch for themselves



Benefit = *Overlap* + *Prefetching*

Profiler Phases

- Eliminate tasks using dynamic information
  - Small dynamic task size
    - BUT, if a small task spawns a successor, treat it with care
  - Small or large spawn hoist distance
    - If too small, then no overlap
    - If too large, then potentially too many data dependences
  - Too many squashes per invocation
    - Apply a *prefetching correction* to this rule
    - If *Benefit* is high, keep task regardless of squash rate

Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation

Methodology

- Simulated Architecture
  - 4-core chip multiprocessor
  - 4 GHz, 3-issue core
  - Detailed cache hierarchy model
    - Per core private 16k L1
    - 1MB Shared L2
    - Speculative data kept in the L1
    - Cache coherence protocol aware of versions
  - Main memory latency of 500 cycles
  - 12 cycle spawn, 20 cycle squash overhead
- Full program simulation (not just loops)

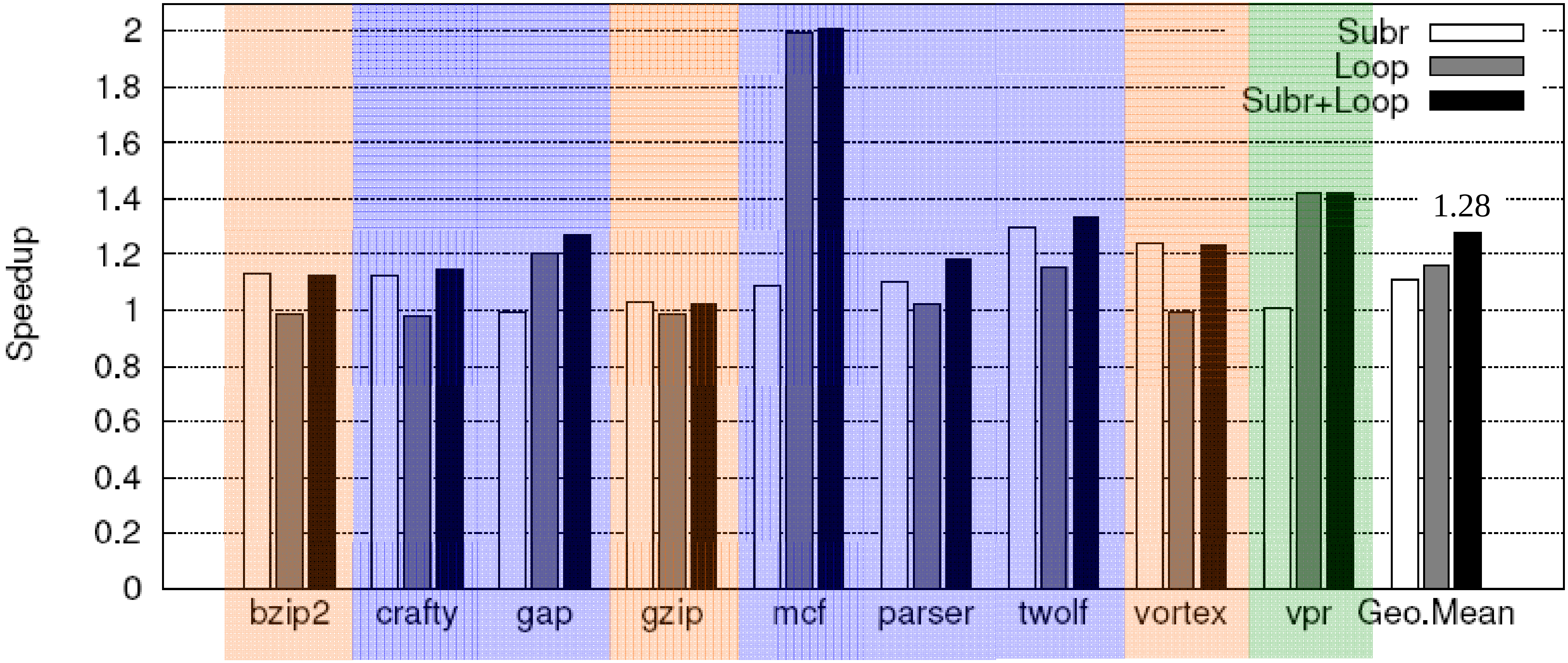
Applications

- Evaluated SPECint 2000
    - Except *gcc*, *perlbmk*, *eon*
- First, apply SGI's source-to-source optimizer (copt)
- Non-TLS binary compiled with gcc-3.5 using –O2
- TLS binary compiled with POSH

Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation

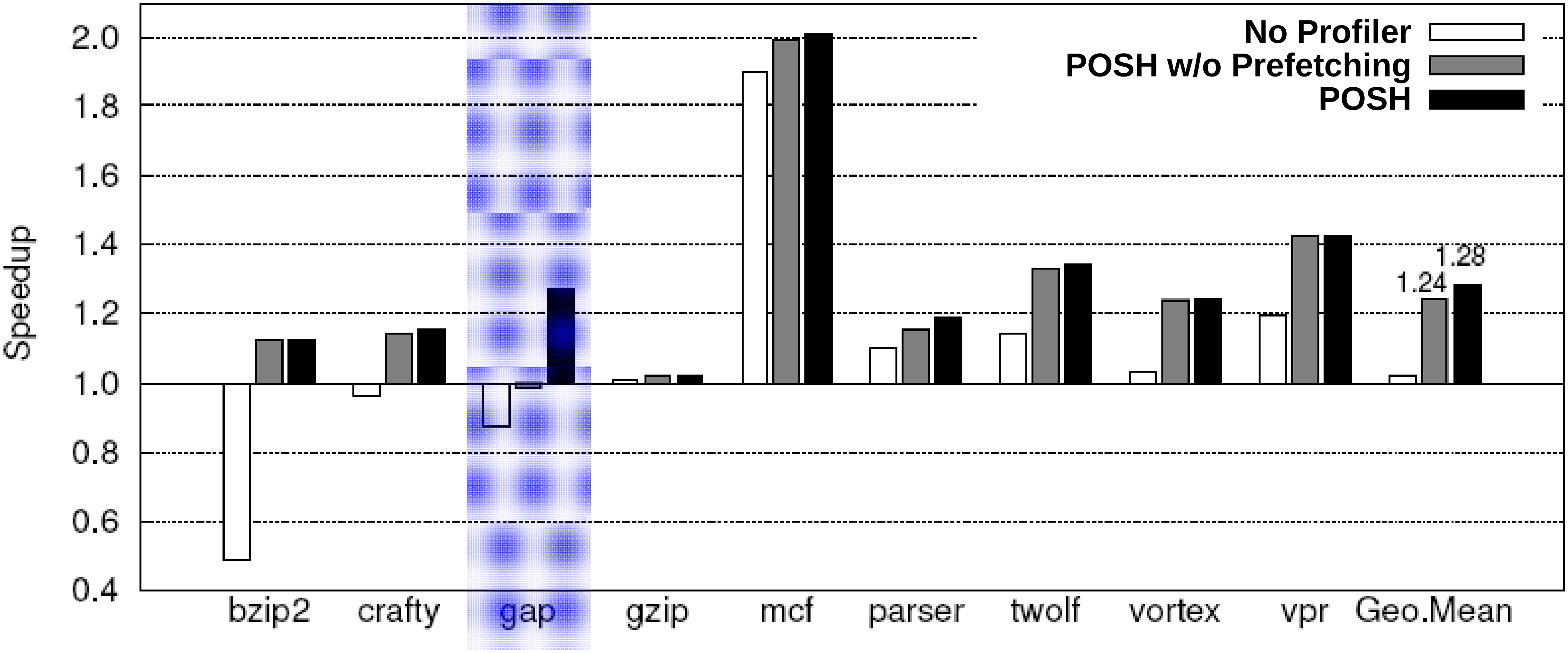
TLS Speedup Over Sequential Execution



Achieve an average speedup of 1.28 on SPECints!

Using both forms of tasks is simple and effective

Profiler Effectiveness



Targeting prefetching improves performance

Conclusions

- A TLS compiler utilizing program structure and a profiler delivers good speedups
    - 1.28 average speedup for SPECints
- For better performance, consider both parallelism and prefetching
- We evaluated the impact of several important design decisions
    - Employ both subroutine continuations and loops for best performance
    - Usefulness of the profiler
    - Value Prediction

Thank You

POSH:

A Profiler-Enhanced TLS Compiler that Leverages Program Structure

Wei Liu, James Tuck, Luis Ceze, Karin Strauss, and Josep Torrellas

University of Illinois at Urbana-Champaign

http://iacoma.cs.uiuc.edu

Jose Renau

University of California at Santa Cruz

http://masc.soe.ucsc.edu

This work was funded by DARPA-IBM under PERCS.

Backup
Slides

Selection and Hoisting Example

```
       int i=0;

       …

Loop:
       if( i > 99 )
         goto Lend;

       <LOOP BODY>



       i = i + 1;

       goto Loop;
Lend:
```

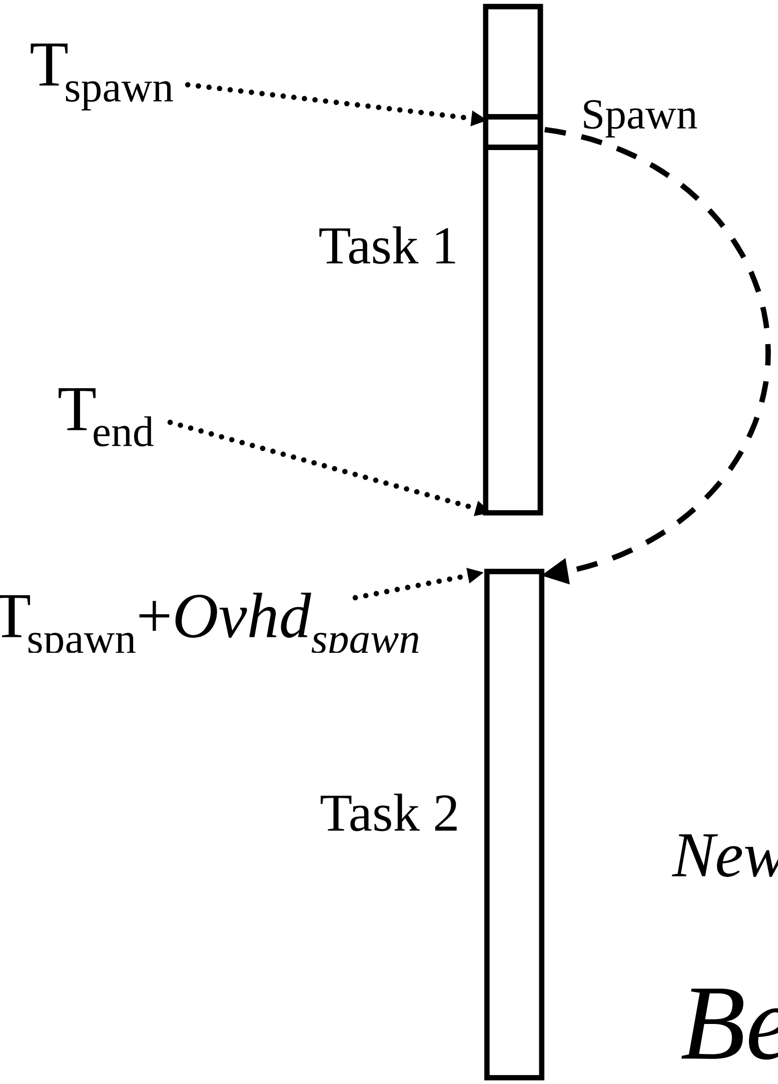
```
int x=0;

…


x = f(x);





do_something(x);
```

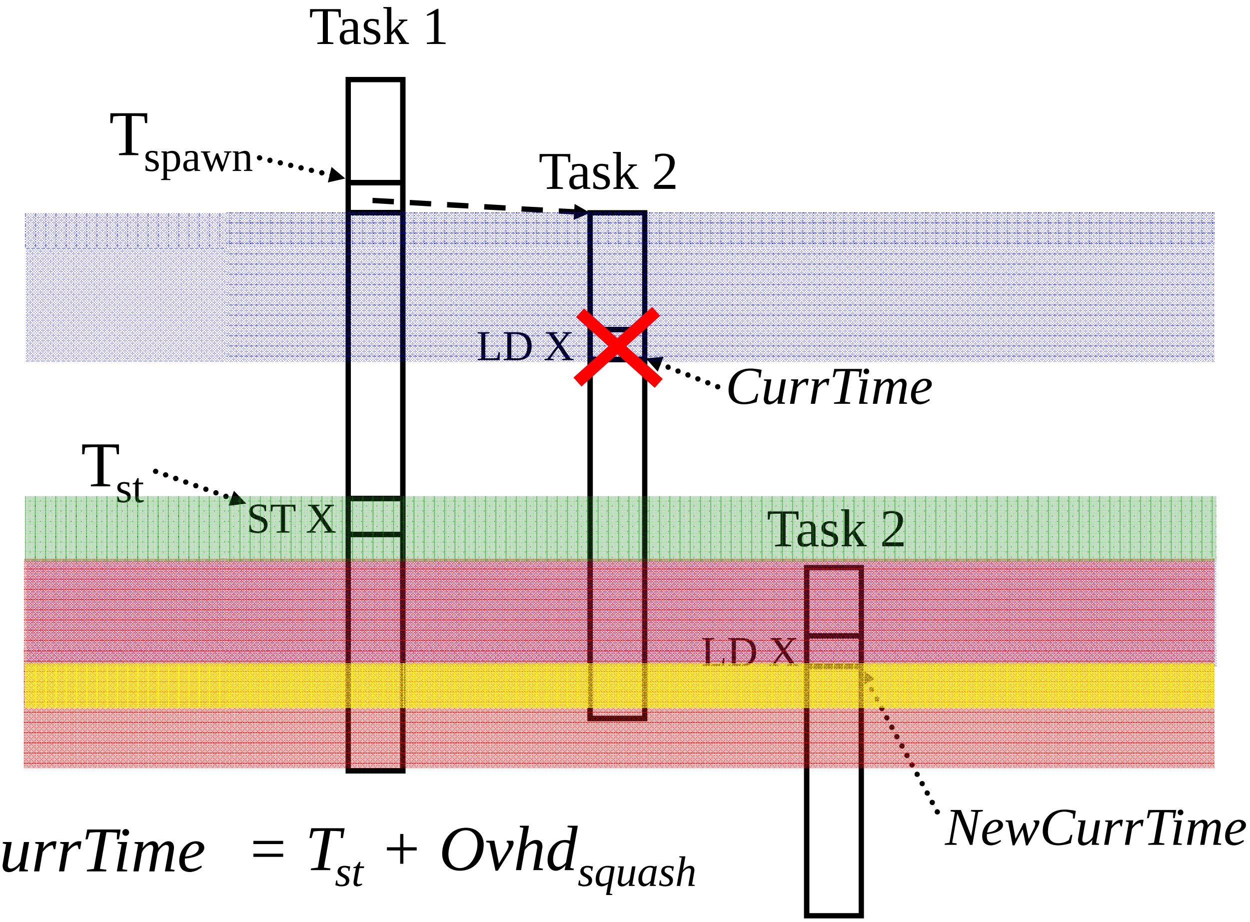
What is a “Task” in POSH?

- From the perspective of POSH:
  - *Begin point*
  - *Spawn point*
- Runtime perspective
  - Dynamic execution between *begin points*

```
 spawn Task1;
 Do_some_work();
Task1:

 while( i < 1000 ) {
  spawn Task2;
  …

  …

Task2:
  i++;
 }
 Do_some_more_work();
```

Estimating Task Performance and Benefit





$$NewCurrTime = T_{st} + Ovhd_{squash} + CurrTime - T_{spawn} - Ovhd_{spawn}$$

$$Benefit = Overlap + Prefetch - MissOvhd$$

Characterization of Profiling

App.	#Tasks Before	# Elim. For Size	# Elim. For Hoist	# Elim. For Squashing	#Saved for Pref.	# Tasks After
gap	36	0	13	11	**7**	12
mcf	17	2	6	4	1	5
parser	464	47	78	158	6	181
vpr	21	0	0	18	0	**3**
Average	**139.7**	14.8	33.7	**55.8**	2.1	**35.4**

Thread-Level Speculation (TLS)

Sequential ⇨

```
for(i=0;i<n;i++) {
  X[Y[i]] = X[Z[i]]...
}
```

- With TLS:
  - Assume no dependences
  - Run in parallel
  - Auto detect violations

TLS Task A ⇨

```
for(i=0;i<n/2;i++) {
  X[Y[i]] = X[Z[i]]...
}
```

TLS Task B ⇨

```
for(i=n/2;i<n;i++) {
  X[Y[i]] = X[Z[i]]...
}
```