Compiling Sequential Code for a Speculative Parallel Architecture

VICTOR A. YING
MARK C. JEFFREY
DANIEL SANCHEZ

ACM STUDENT RESEARCH COMPETITION @ PLDI 2019

How to parallelize sequential code?

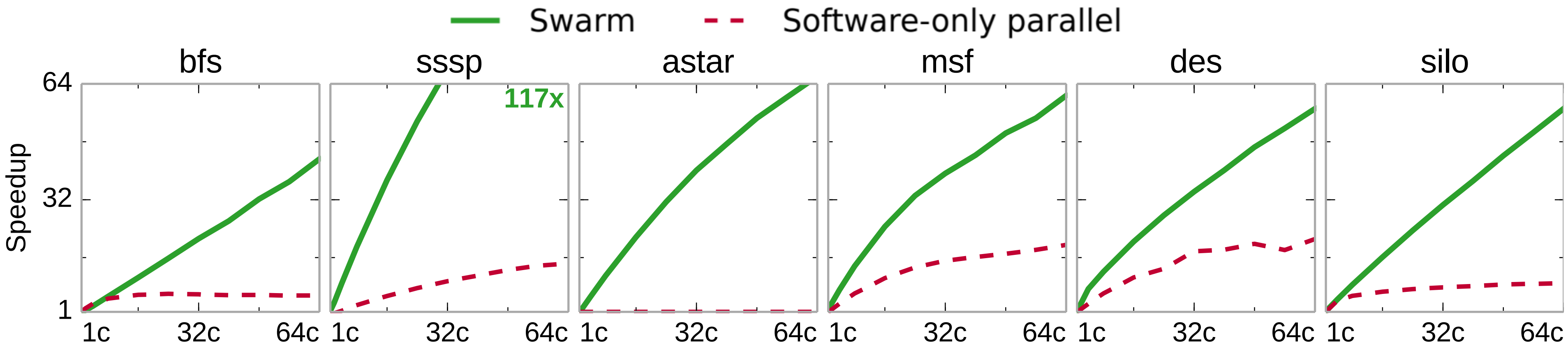
Multicores are everywhere.

Parallel programming is hard.

We should parallelize sequential code to use multicores.

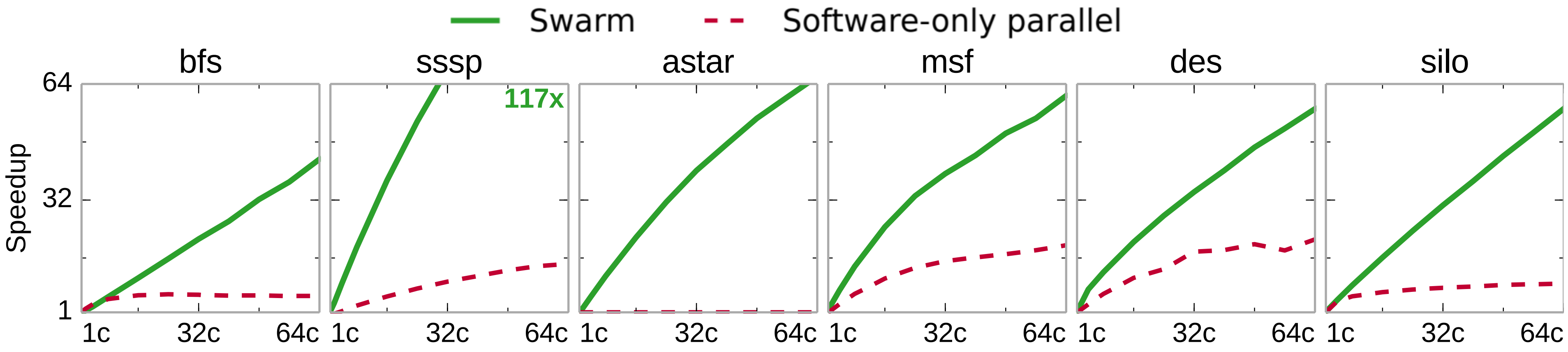
Background: Swarm architecture

Recent **Swarm architecture** [MICRO'15, MICRO'16, ISCA'17] parallelizes programs that were hard to parallelize



Background: Swarm architecture

Recent **Swarm architecture** [MICRO’15, MICRO’16, ISCA’17] parallelizes programs that were hard to parallelize



SCC: compile sequential C/C++ to exploit parallelism on Swarm.

Swarm hardware attributes [Jeffrey et al. MICRO’15]

Execution model:

- Program comprises timestamped tasks.
- Tasks spawn children with greater or equal timestamp.
- Tasks appear to execute in timestamp order.

Swarm hardware attributes [Jeffrey et al. MICRO’15]

Execution model:

- Program comprises timestamped tasks.
- Tasks spawn children with greater or equal timestamp.
- Tasks appear to execute in timestamp order.

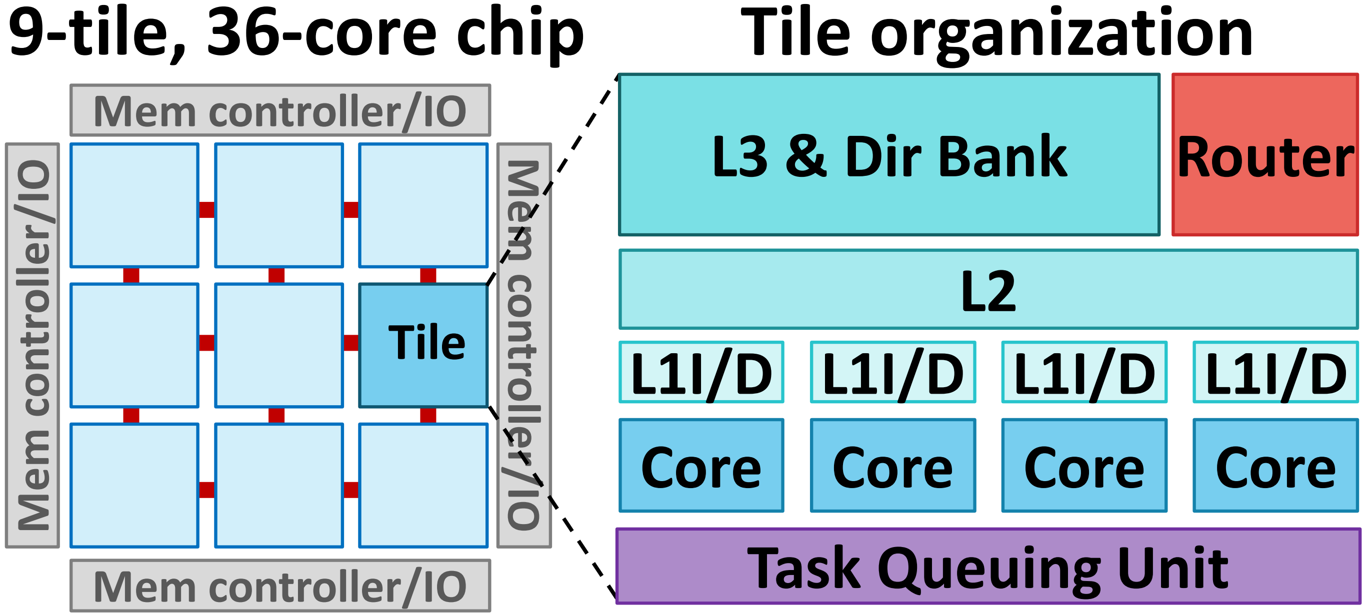
Executes tasks speculatively
and out of order.

Swarm hardware attributes [Jeffrey et al. MICRO’15]

Execution model:

- Program comprises timestamped tasks.
- Tasks spawn children with greater or equal timestamp.
- Tasks appear to execute in timestamp order.

Executes tasks speculatively and out of order.



9-tile, 36-core chip

Tile organization

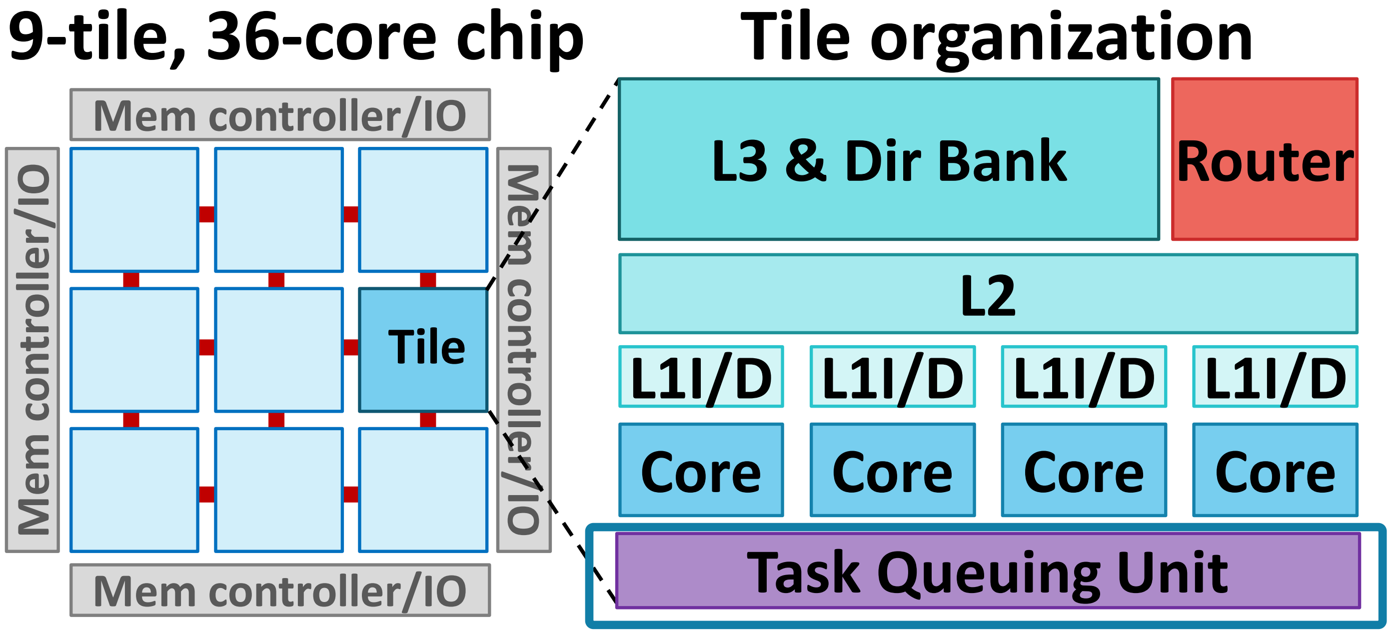
Swarm hardware attributes [Jeffrey et al. MICRO'15]

Execution model:

- Program comprises timestamped tasks.
- Tasks spawn children with greater or equal timestamp.
- Tasks appear to execute in timestamp order.

Executes tasks speculatively and out of order.

Task units manage hundreds of tiny speculative tasks.



Parallelizing sequential code

Example: maximal independent set:

- Iterates through vertices in graph.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Parallelizing sequential code

Example: maximal independent set:

- Iterates through vertices in graph.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Parallelizing sequential code

Example: maximal independent set:

- Iterates through vertices in graph.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Indirect memory accesses

Parallelizing sequential code

Example: maximal independent set:

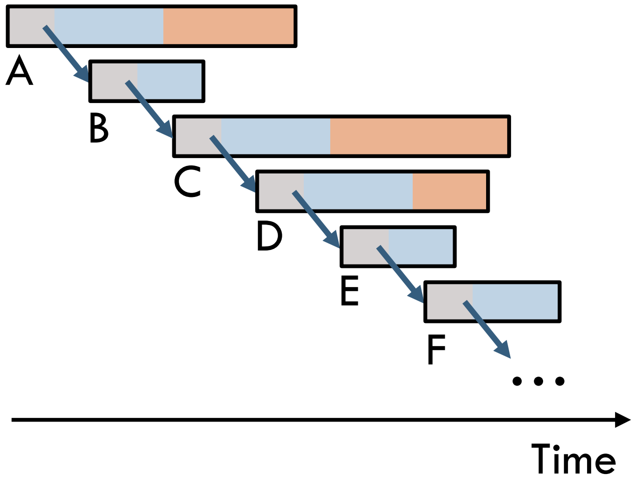
- Iterates through vertices in graph.

One task per outer-loop iteration.

- Each tasks spawns the next.
- Hardware tries to run tasks in parallel.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Indirect memory accesses



Parallelizing sequential code

Example: maximal independent set:

- Iterates through vertices in graph.

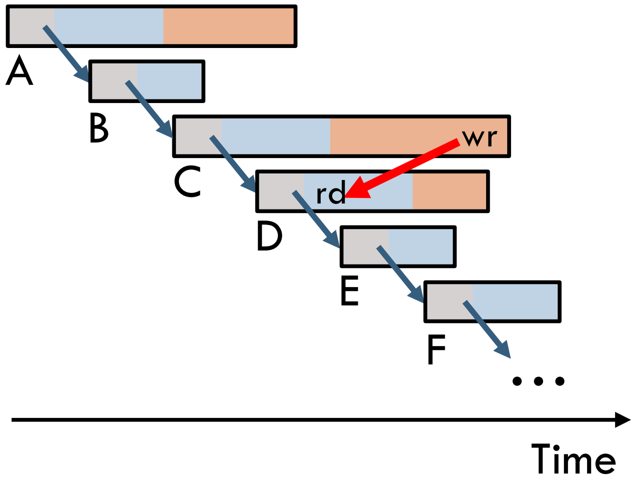
One task per outer-loop iteration.

- Each tasks spawns the next.
- Hardware tries to run tasks in parallel.

Hardware tracks memory accesses to discover data dependences.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Indirect memory accesses

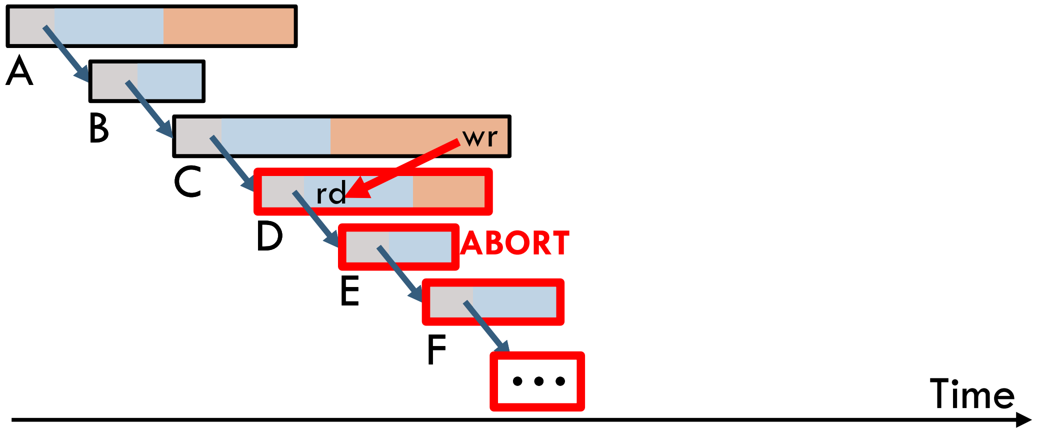


Task chains incur costly misspeculation recovery

Tasks abort if they violated data dependence.

Tasks that abort must roll back their effects, including children they spawned.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

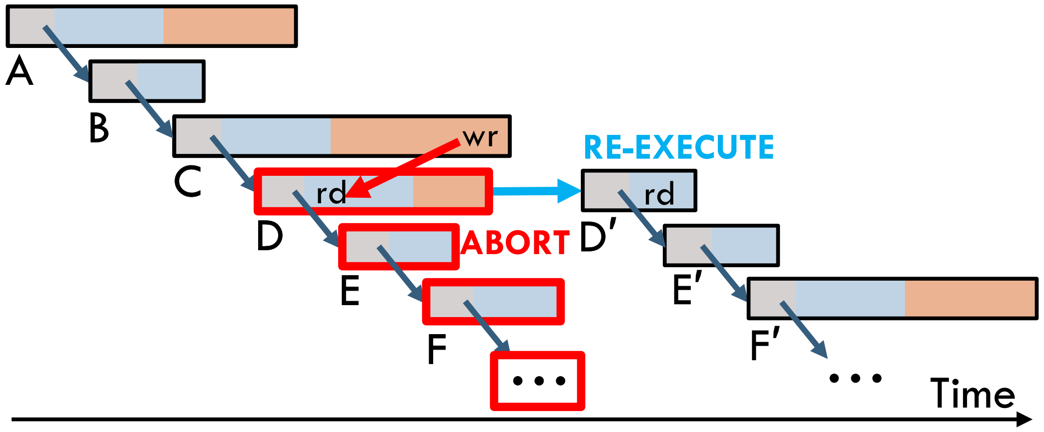


Task chains incur costly misspeculation recovery

Tasks abort if they violated data dependence.

Tasks that abort must roll back their effects, including children they spawned.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

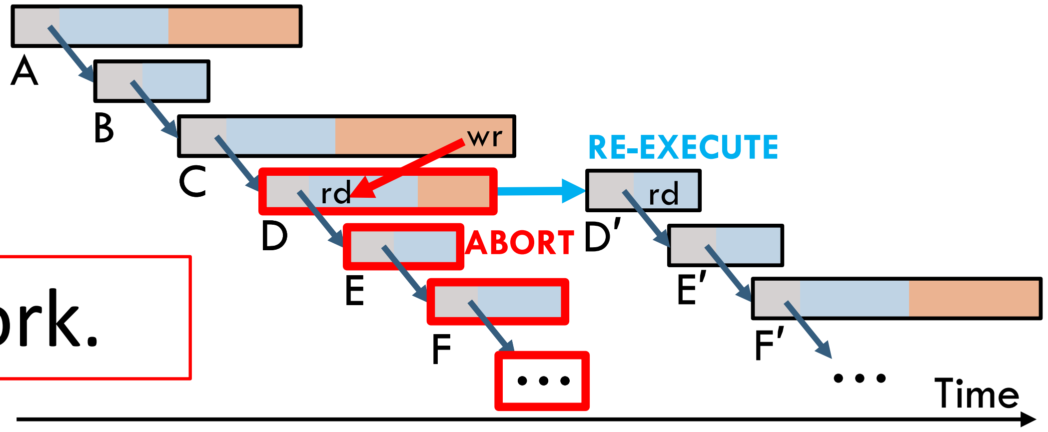


Task chains incur costly misspeculation recovery

Tasks abort if they violated data dependence.

Tasks that abort must roll back their effects, including children they spawned.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```



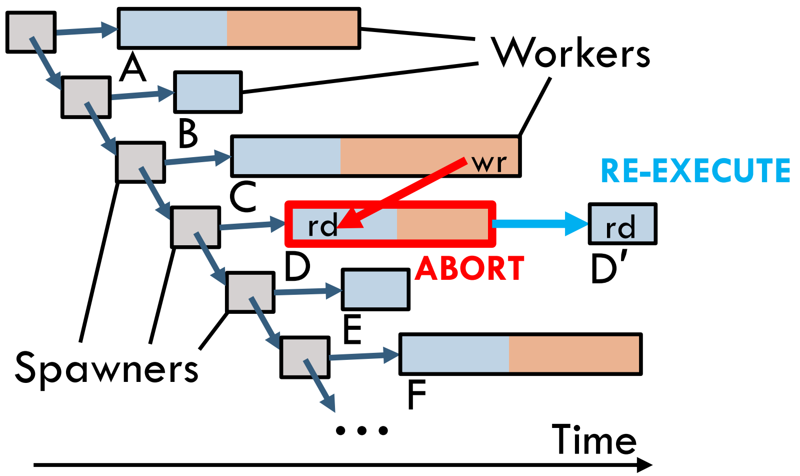
Cascading aborts waste a lot of work.

SCC's decoupled spawn enables selective aborts

Put most work into **worker** tasks at the leaves of the task tree.

- Use Swarm's mechanisms for cheap selective aborts.

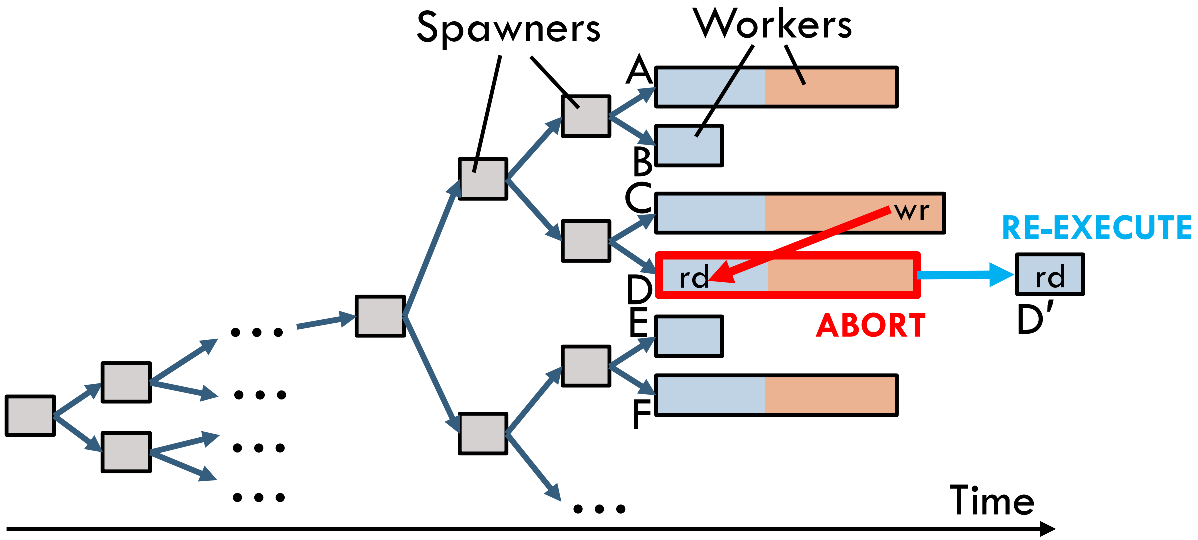
```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```



SCC’s balanced task trees enable scalability

Spawners recursively divide the range of iterations.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

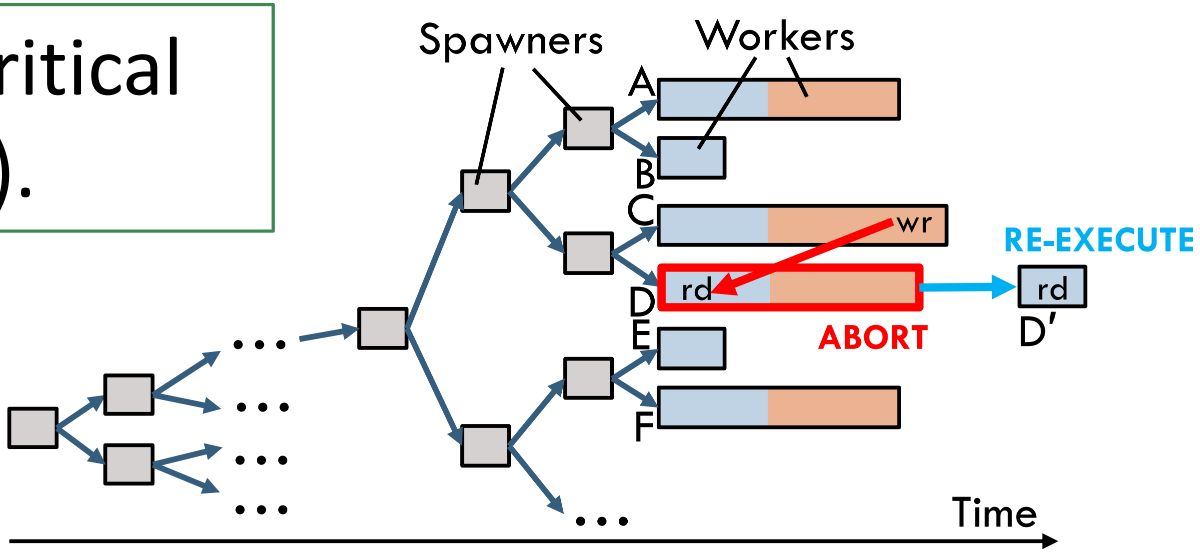


SCC's balanced task trees enable scalability

Spawners recursively divide the range of iterations.

```
for (int v = 0; v < numVertices; v++) {
  if (state[v] == UNVISITED) {
    state[v] = INCLUDED;
    for (int nbr = 0; nbr < numNeighbors(v); nbr++)
      state[neighbors(v)[nbr]] = EXCLUDED;
  }
}
```

Balanced spawner trees reduce critical path length to O(log(# iterations)).



Progressive expansion: parallelizing irregular loops

Progressive expansion generates balanced spawner trees for loops with unknown tripcount.

Source code:

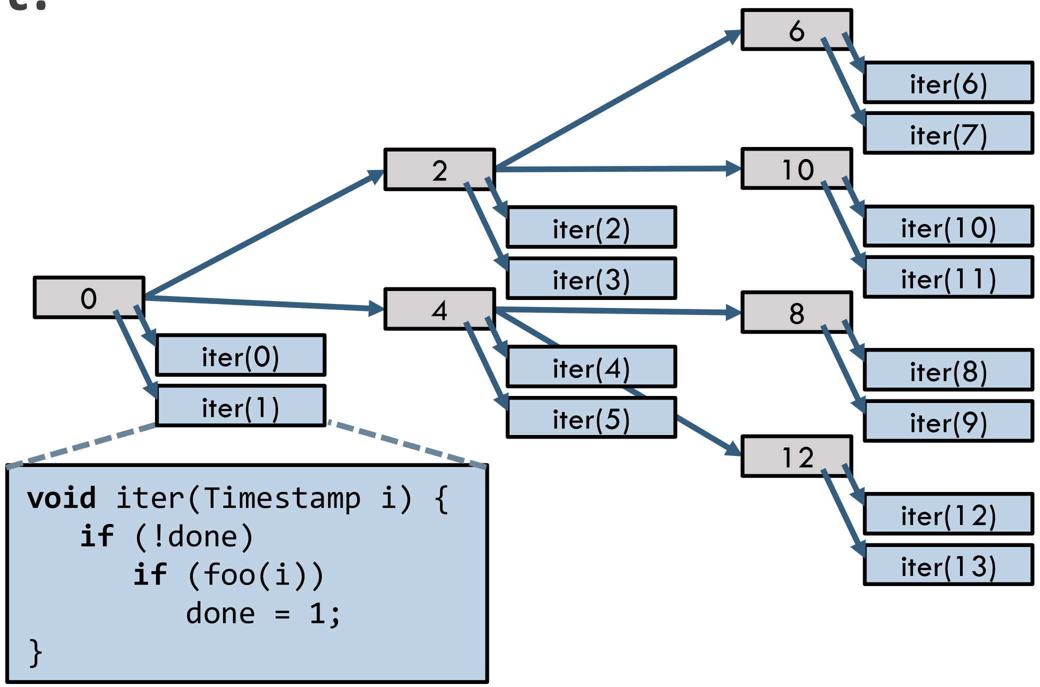
```
for (i = 0; ; i++)
   if (foo(i))
      break;
```

Progressive expansion: parallelizing irregular loops

Progressive expansion generates balanced spawner trees for loops with unknown tripcount.

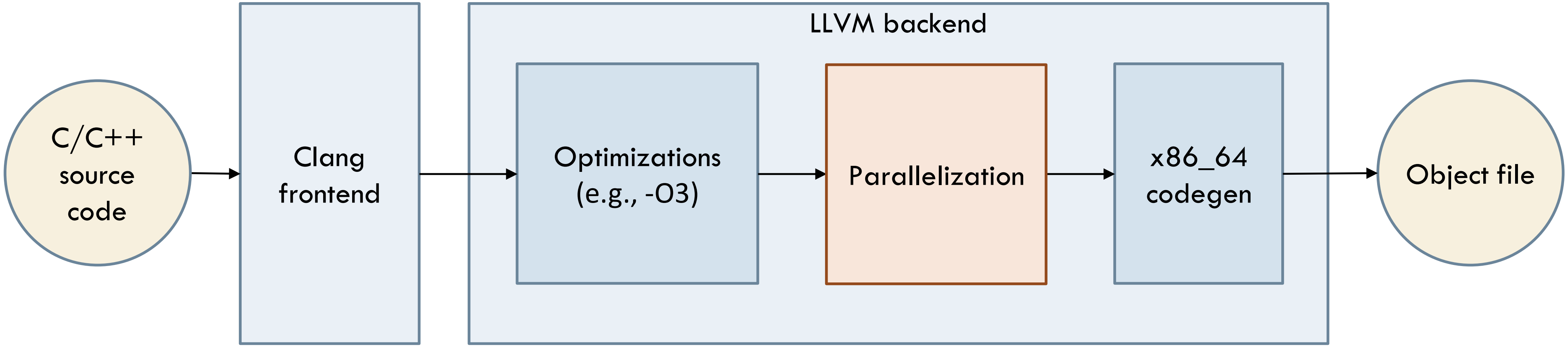
Source code:

```
for (i = 0; ; i++)
    if (foo(i))
        break;
```



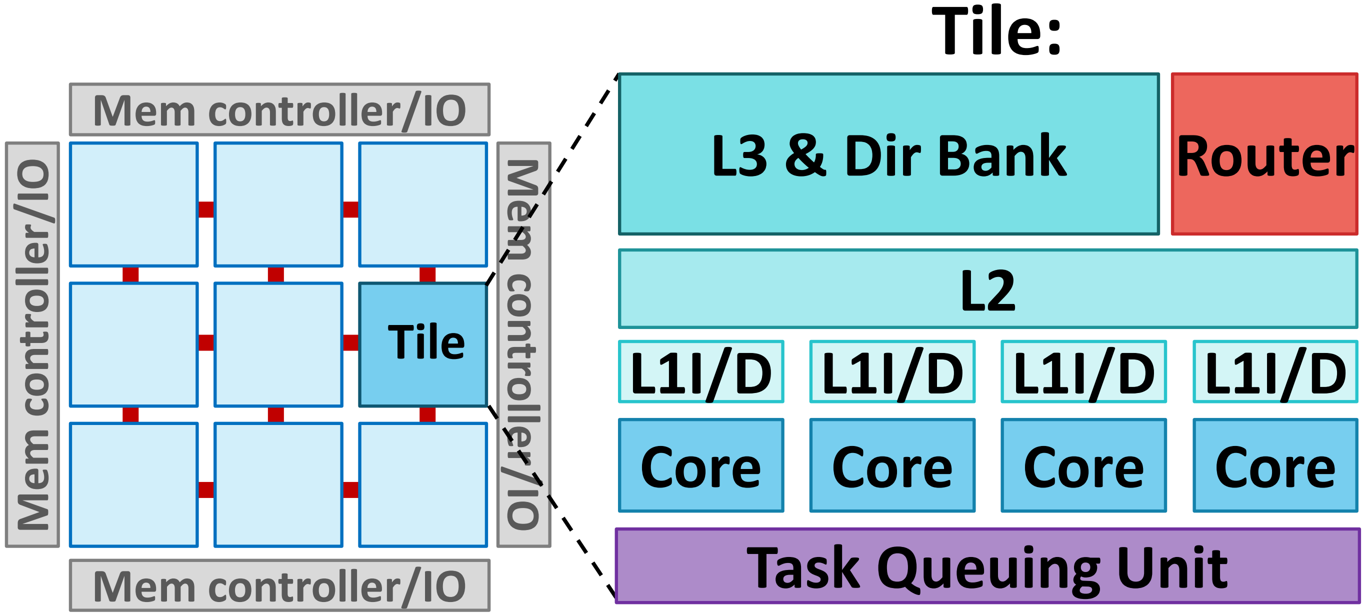
```
void iter(Timestamp i) {
   if (!done)
      if (foo(i))
         done = 1;
}
```

SCC implementation in LLVM/Clang



Evaluation Methodology

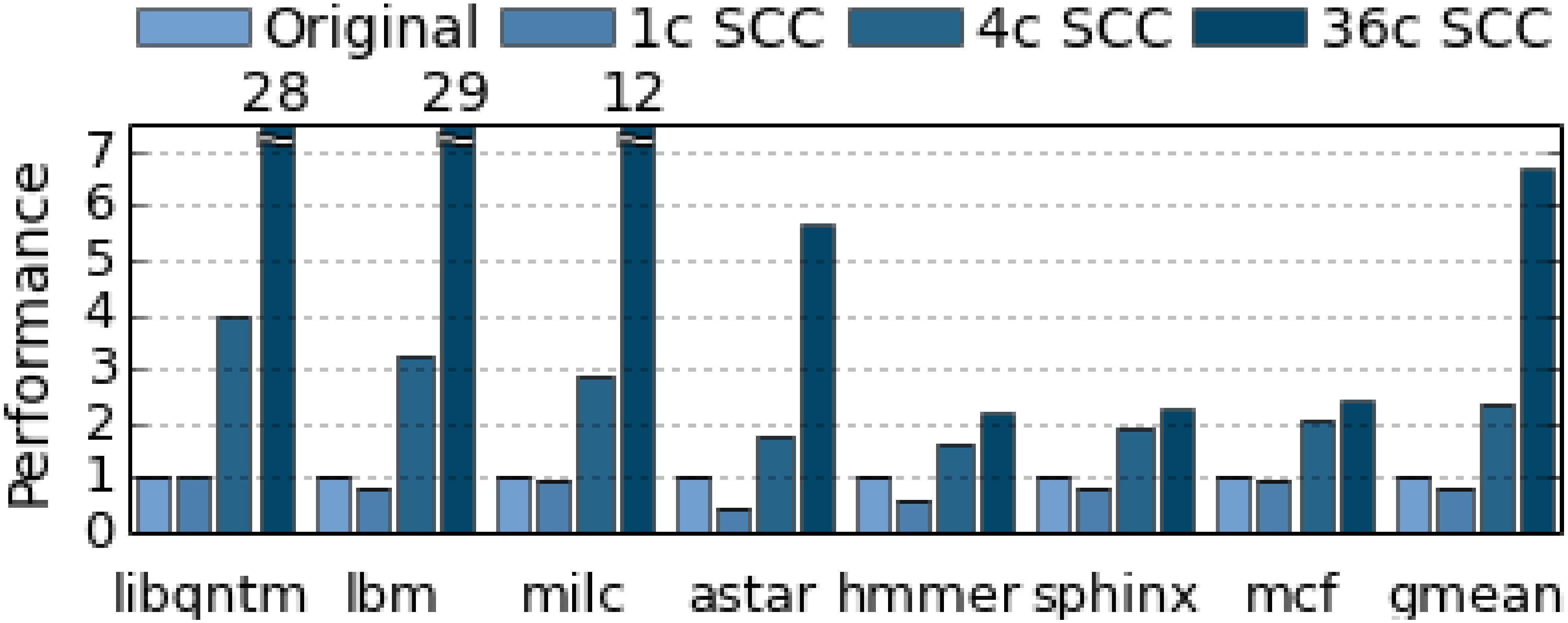
Simulated 1-, 4-, and 36-core systems.



Simulated hardware consistent with prior work [Jeffrey et al. MICRO'15].

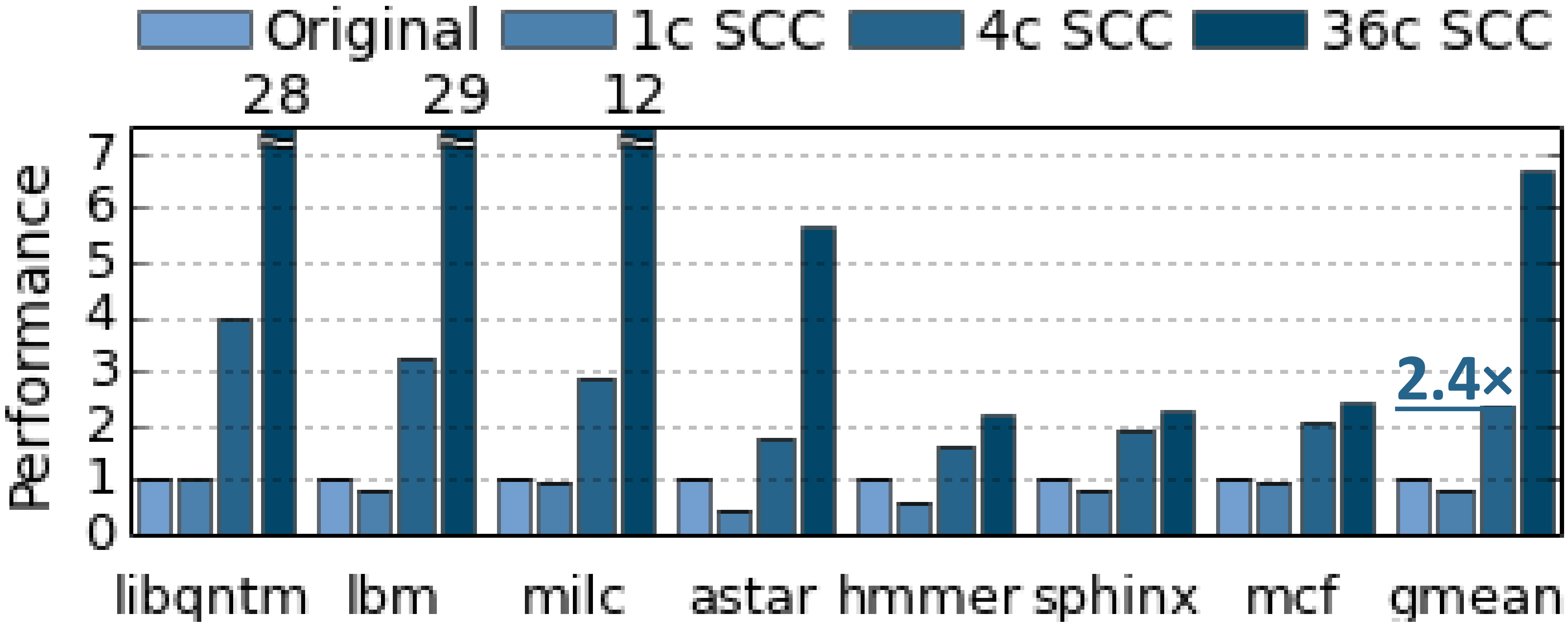
Results

SPEC CPU2006 C/C++ benchmarks compiled with -O3.



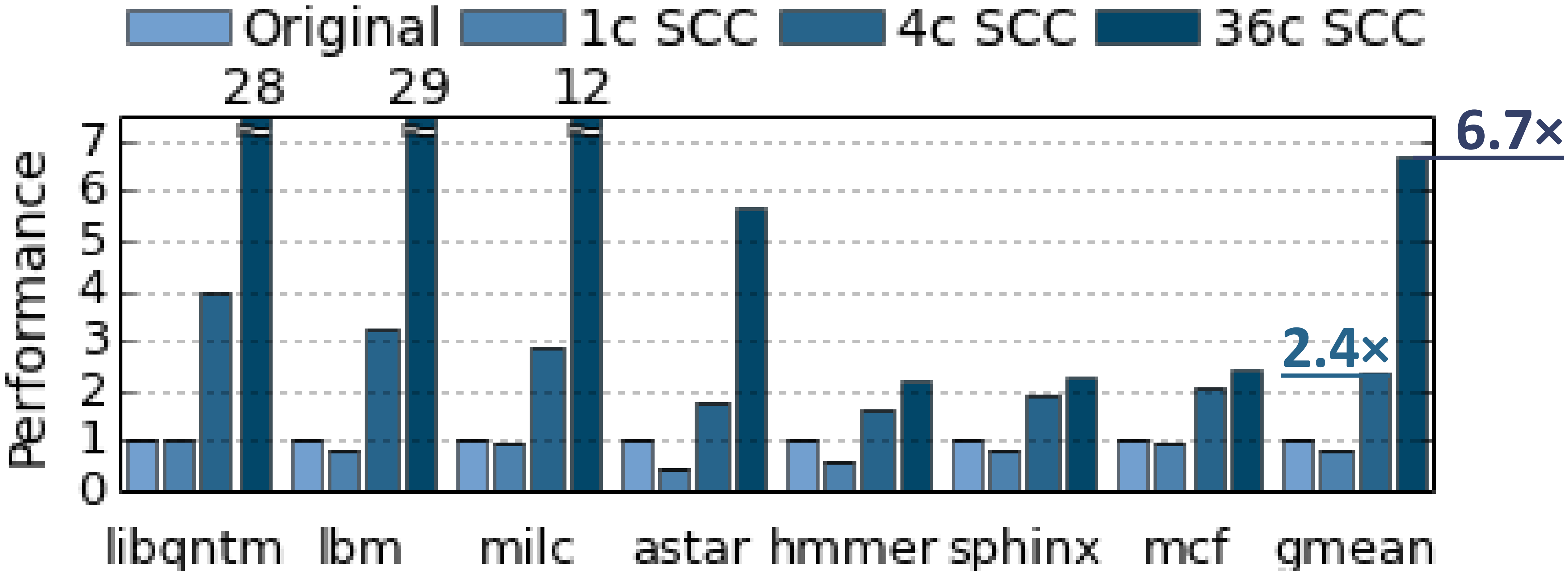
Results

SPEC CPU2006 C/C++ benchmarks compiled with -O3.



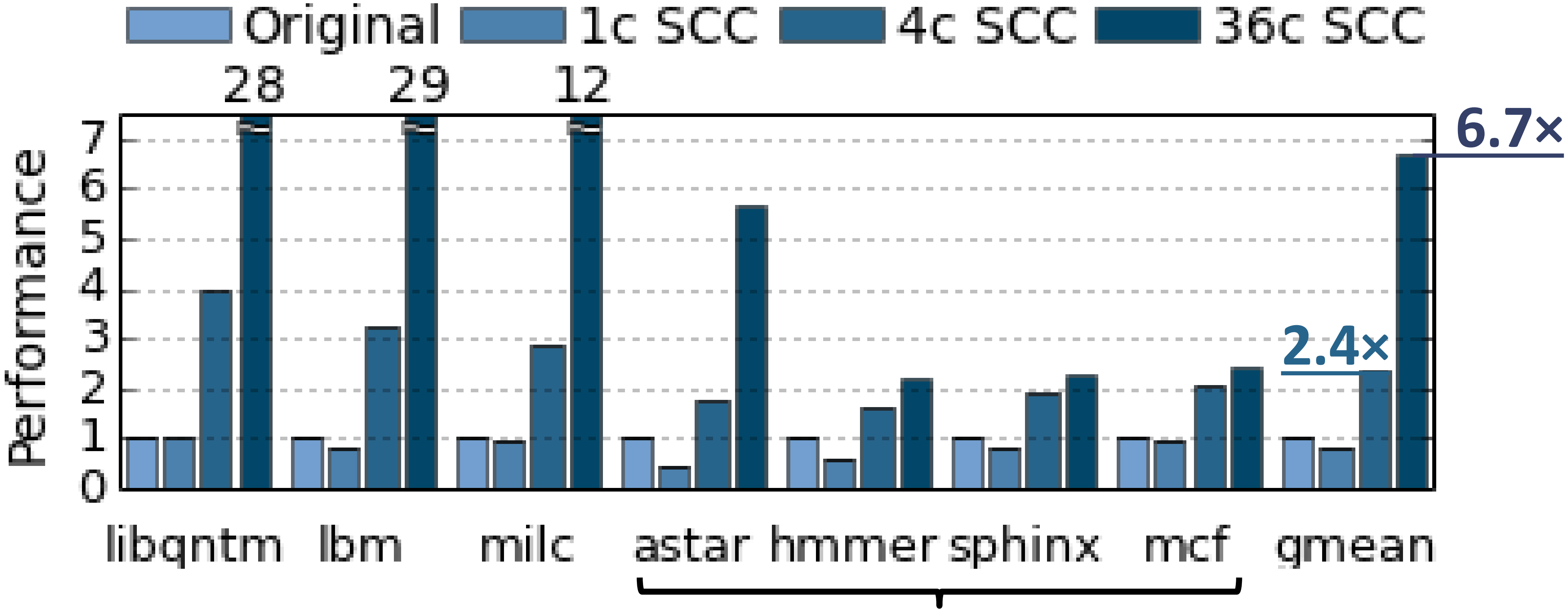
Results

SPEC CPU2006 C/C++ benchmarks compiled with -O3.



Results

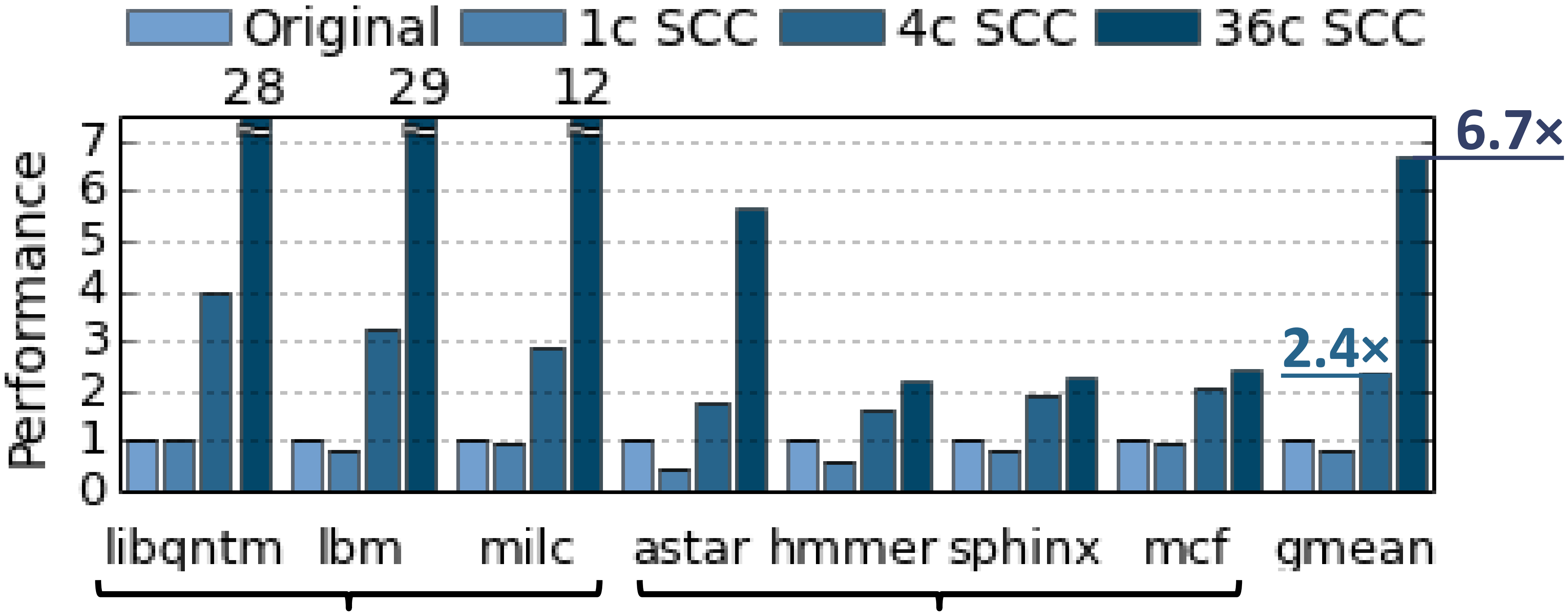
SPEC CPU2006 C/C++ benchmarks compiled with -O3.



Frequent data dependences, modest scalability

Results

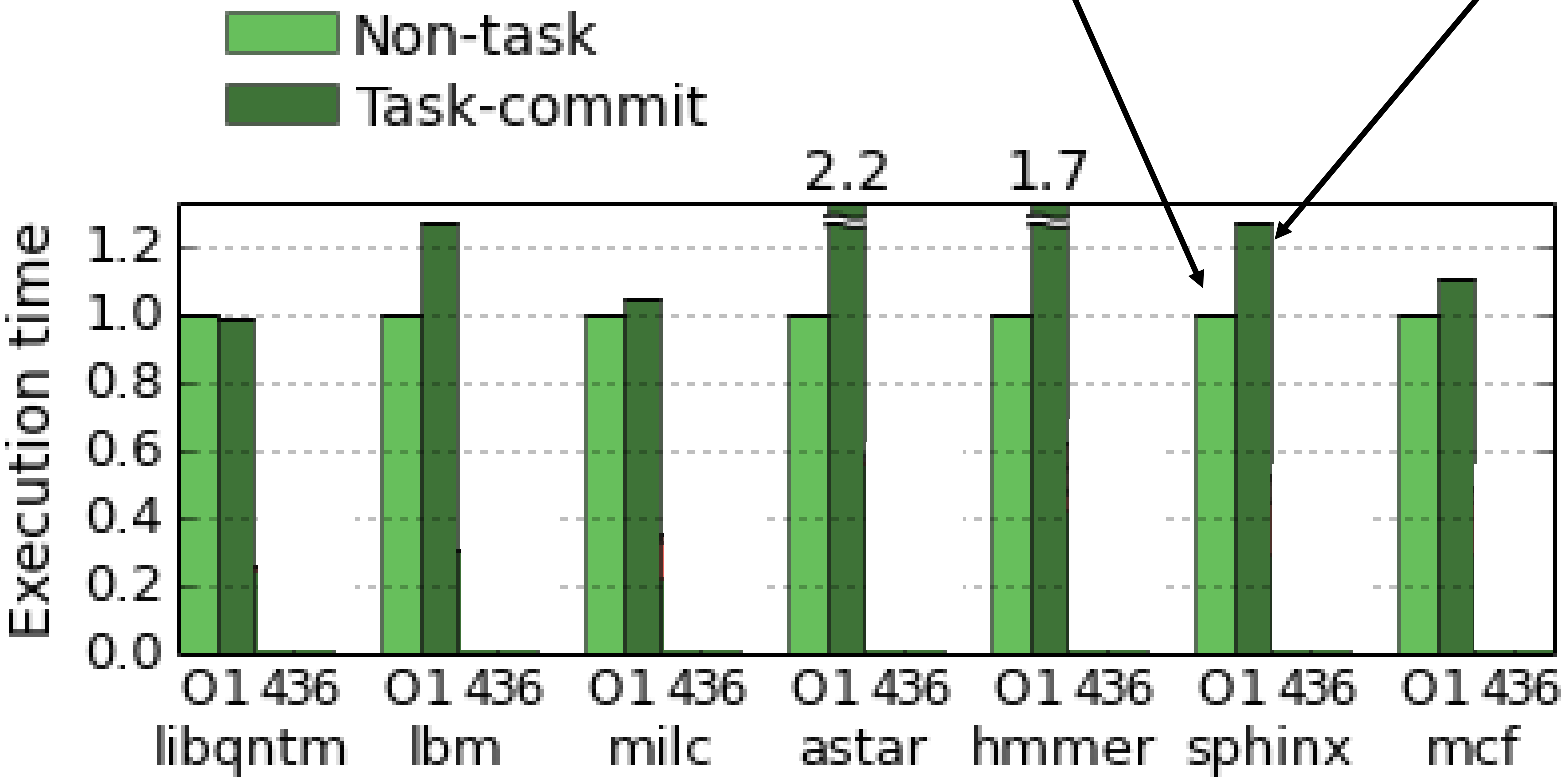
SPEC CPU2006 C/C++ benchmarks compiled with -O3.



Rare data dependences, high scalability

Frequent data dependences, modest scalability

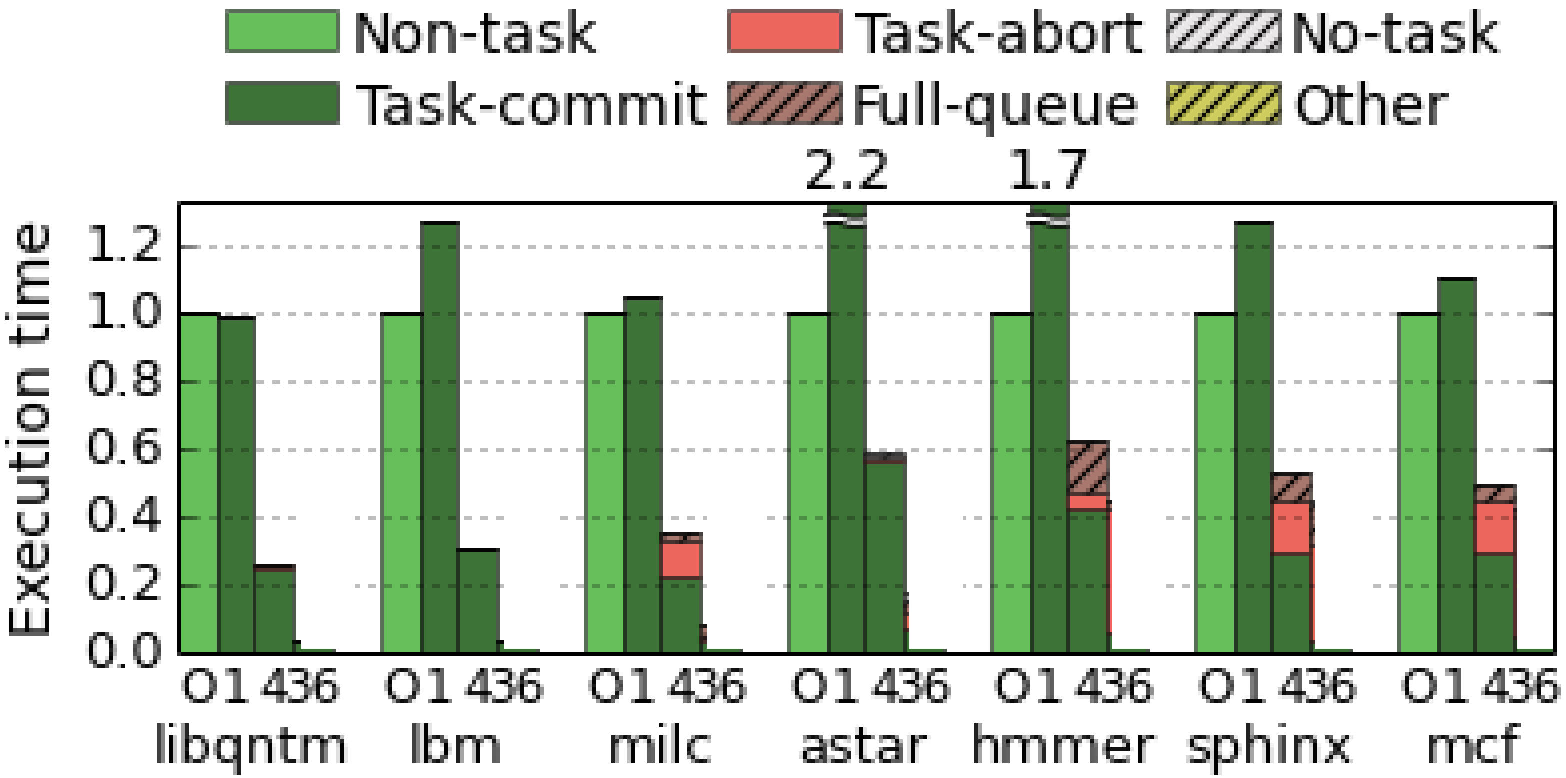
Results: Overheads are moderate



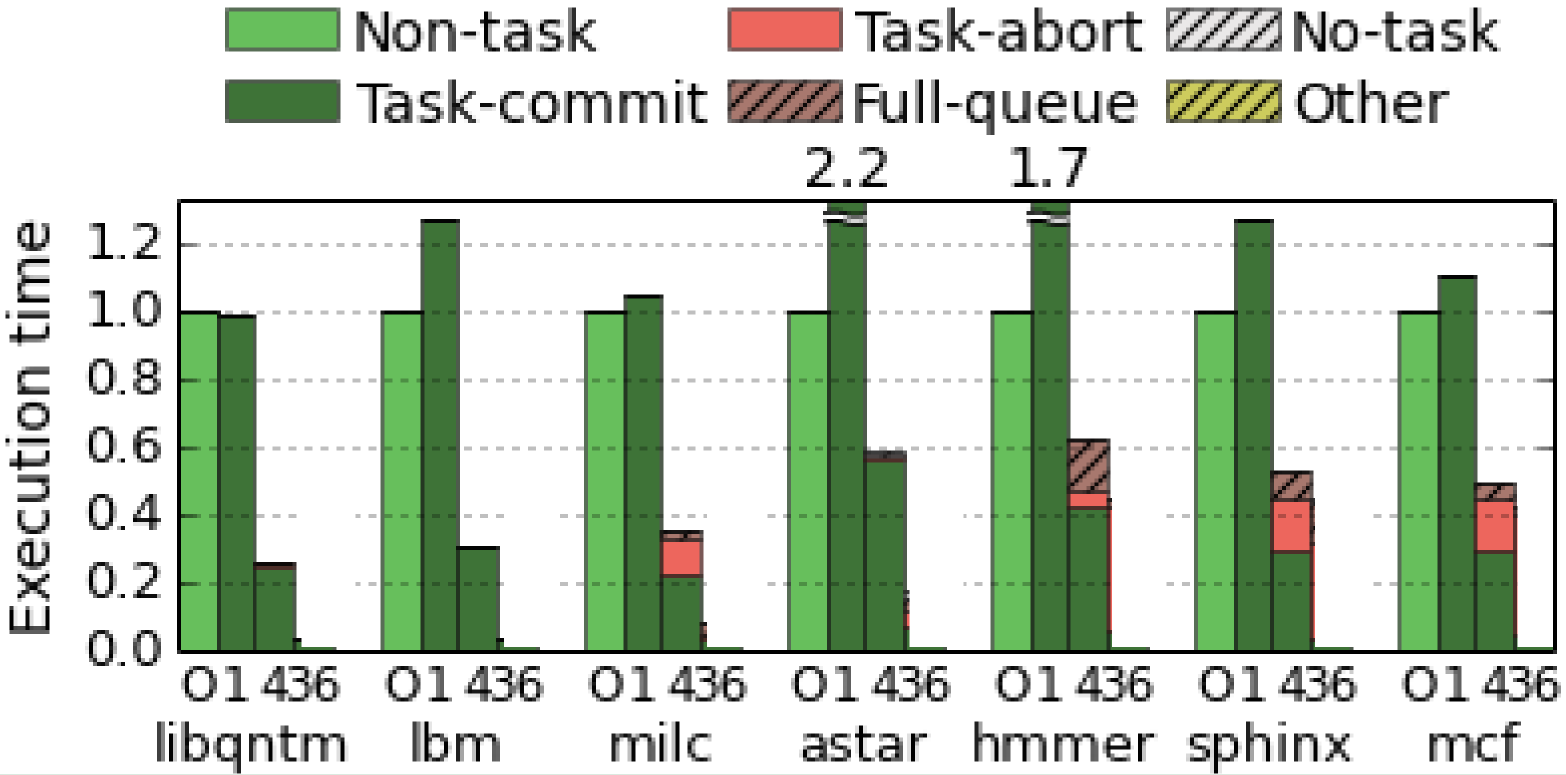
Serial code, -O3.

Parallelized with SCC.

Results: Cores busy most of the time

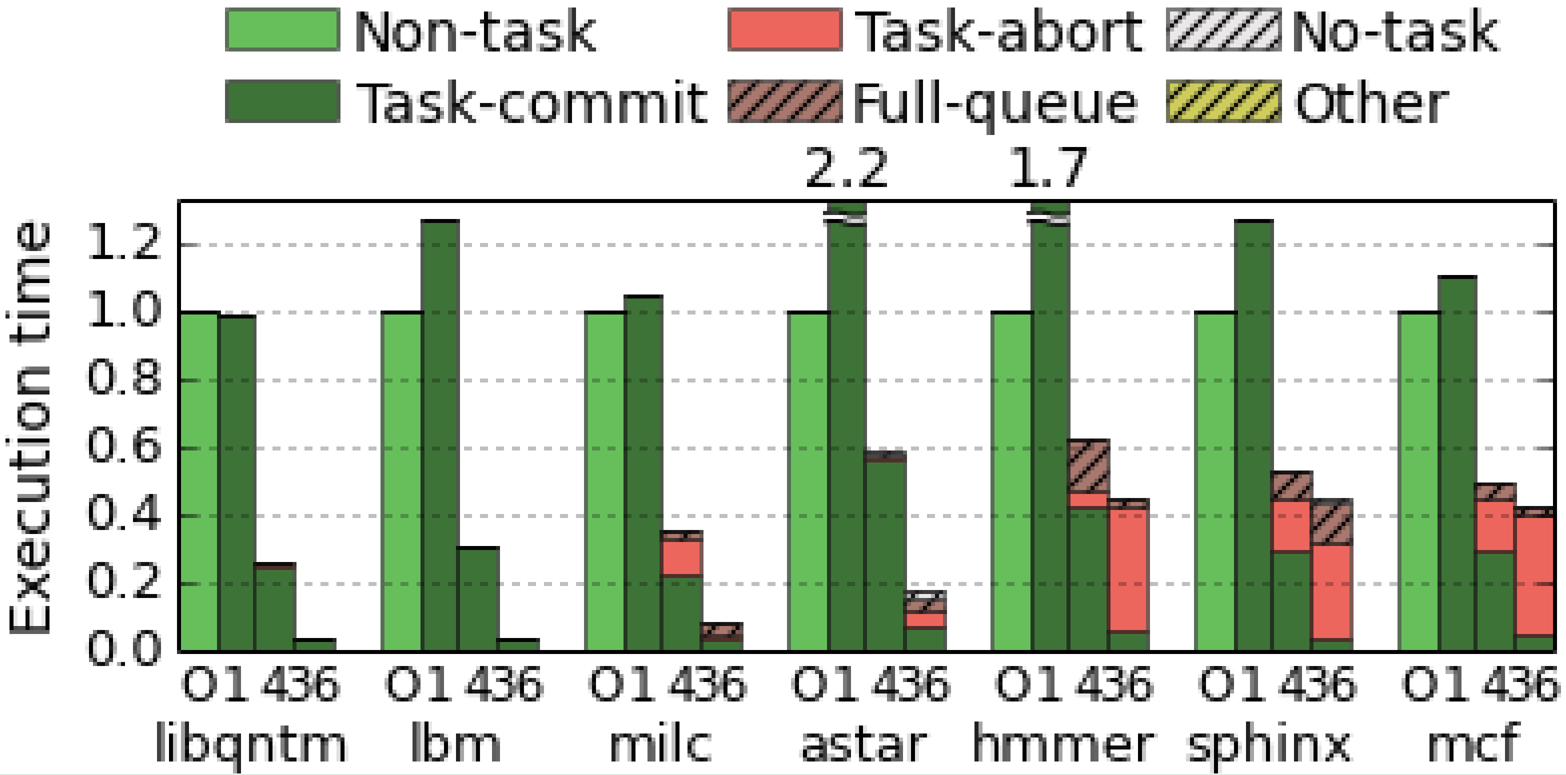


Results: Cores busy most of the time



Cores spend most time executing useful work, not aborting.

Results: Cores busy most of the time



Cores spend most time executing useful work, not aborting.

Contributions

Parallel programming is hard. Sequential code has latent parallelism.

Contributions

Parallel programming is hard. Sequential code has latent parallelism.

We present SCC: A C/C++ compiler that effectively parallelizes sequential code by exploiting the recent Swarm architecture.

- Speedups of 6.7× gmean and up to 29× on 36 cores.

Contributions

Parallel programming is hard. Sequential code has latent parallelism.

We present SCC: A C/C++ compiler that effectively parallelizes sequential code by exploiting the recent Swarm architecture.

- Speedups of 6.7× gmean and up to 29× on 36 cores.

Techniques:

- Balanced spawner trees: decouple task spawn from most work.
- Progressive expansion: speculative spawners for irregular loops.

Questions?

Parallel programming is hard. Sequential code has latent parallelism.

We present SCC: A C/C++ compiler that effectively parallelizes sequential code by exploiting recent Swarm architecture.

- Speedups of 6.7× gmean and up to 29× on 36 cores.

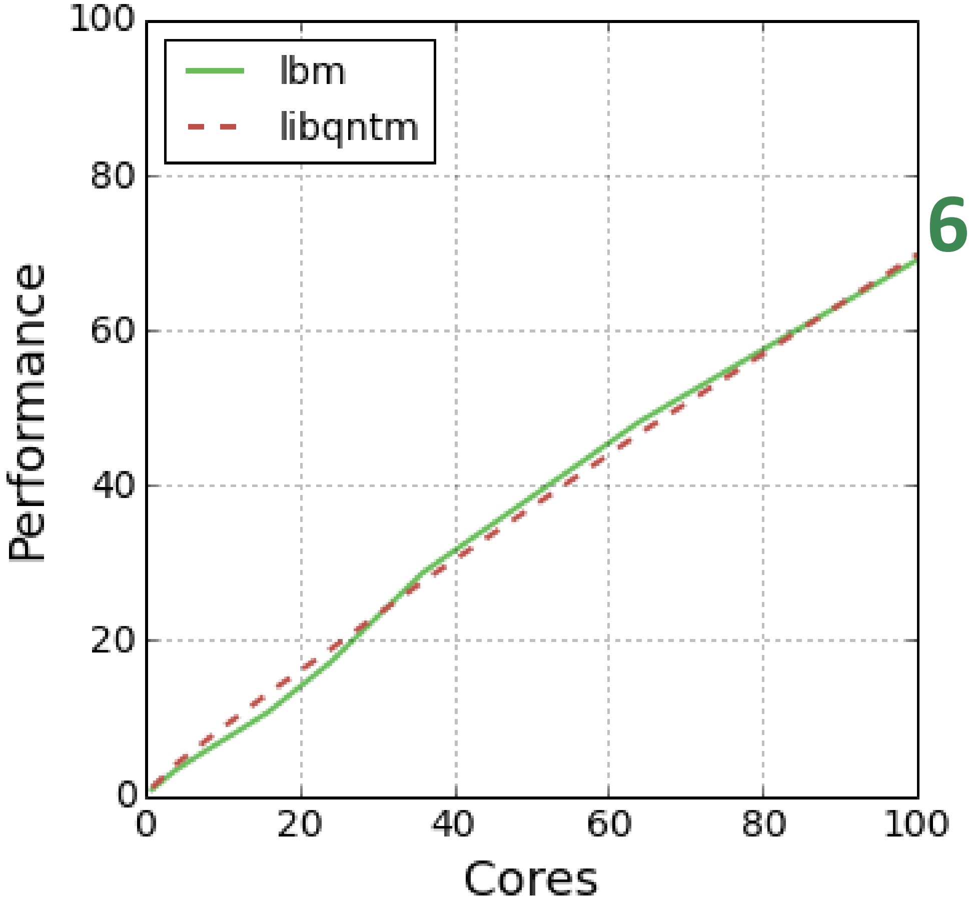
Techniques:

- Balanced spawner trees: decouple task spawn from most work.
- Progressive expansion: speculative spawners for irregular loops.

Email: victory@csail.mit.edu

Backup Slides

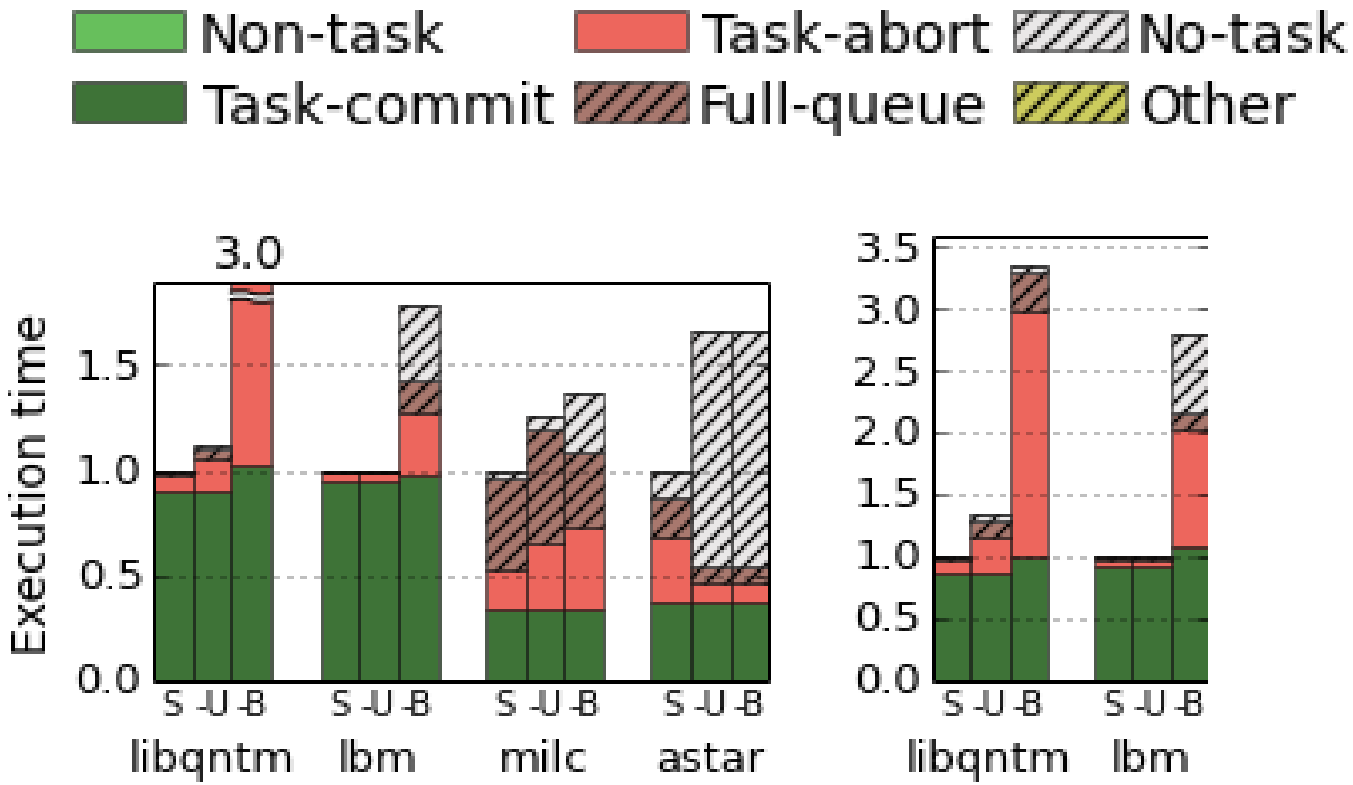
Results: Scalability



69×

Linear scalability in 462.libquantum and 470.lbm up to 100 cores!

Balanced spawner trees are key to scalability



4 Cores

36 Cores

Fine-Grained Task Selection

Task heuristics:

- Split loop iterations
- Split non-inlined function calls and their continuations.

Manual annotations suggest additional task boundaries without affecting semantics.

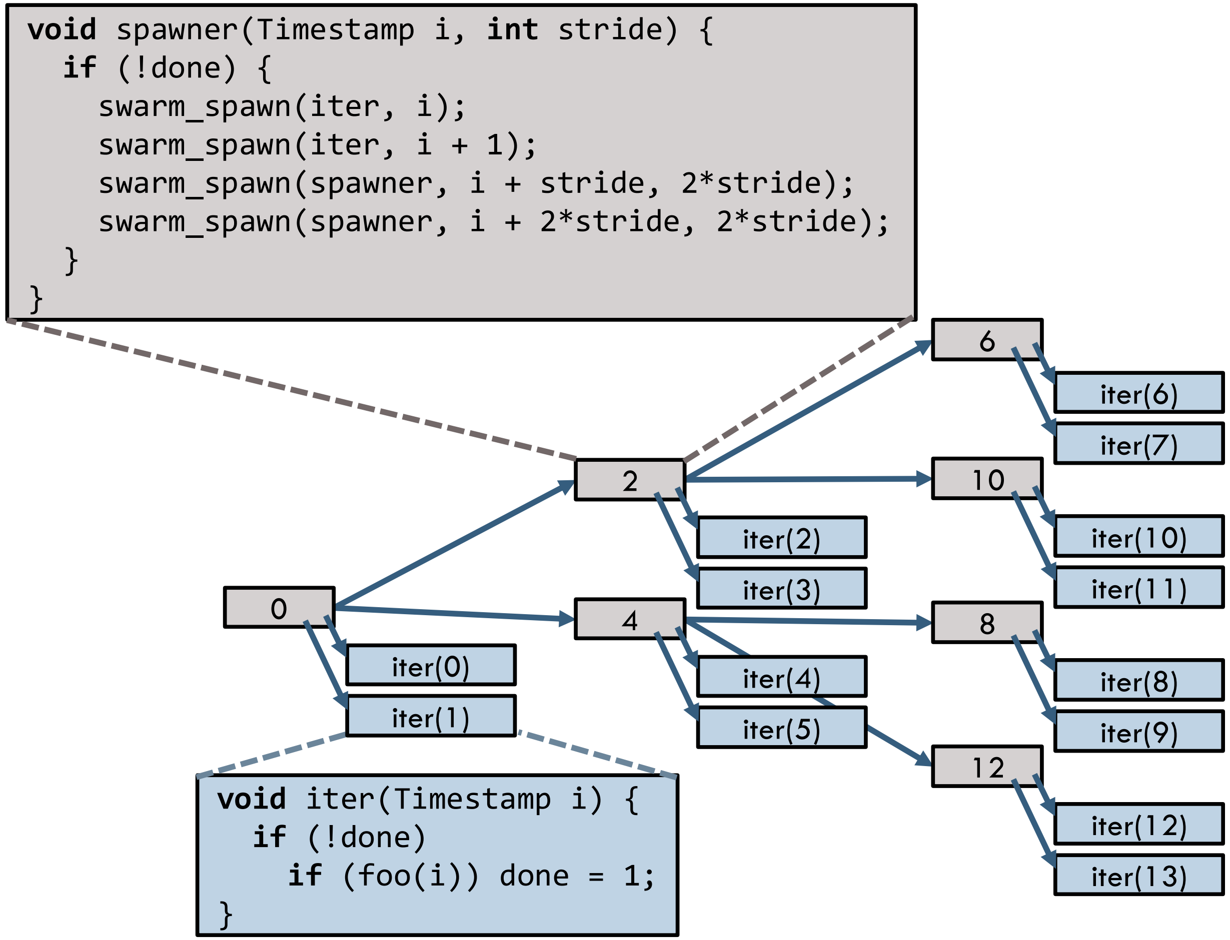
- Ongoing work: heuristics to fully automate task selection

Progressive expansion: parallelizing irregular loops

Progressive expansion generates balanced spawner trees for loops with unknown tripcount.

Source code:

```
for (i = 0; ; i++)
   if (foo(i)) break;
```



```
void spawner(Timestamp i, int stride) {
  if (!done) {
    swarm_spawn(iter, i);
    swarm_spawn(iter, i + 1);
    swarm_spawn(spawner, i + stride, 2*stride);
    swarm_spawn(spawner, i + 2*stride, 2*stride);
  }
}
```

```
void iter(Timestamp i) {
  if (!done)
    if (foo(i)) done = 1;
}
```

Hierarchical timestamps preserve program order

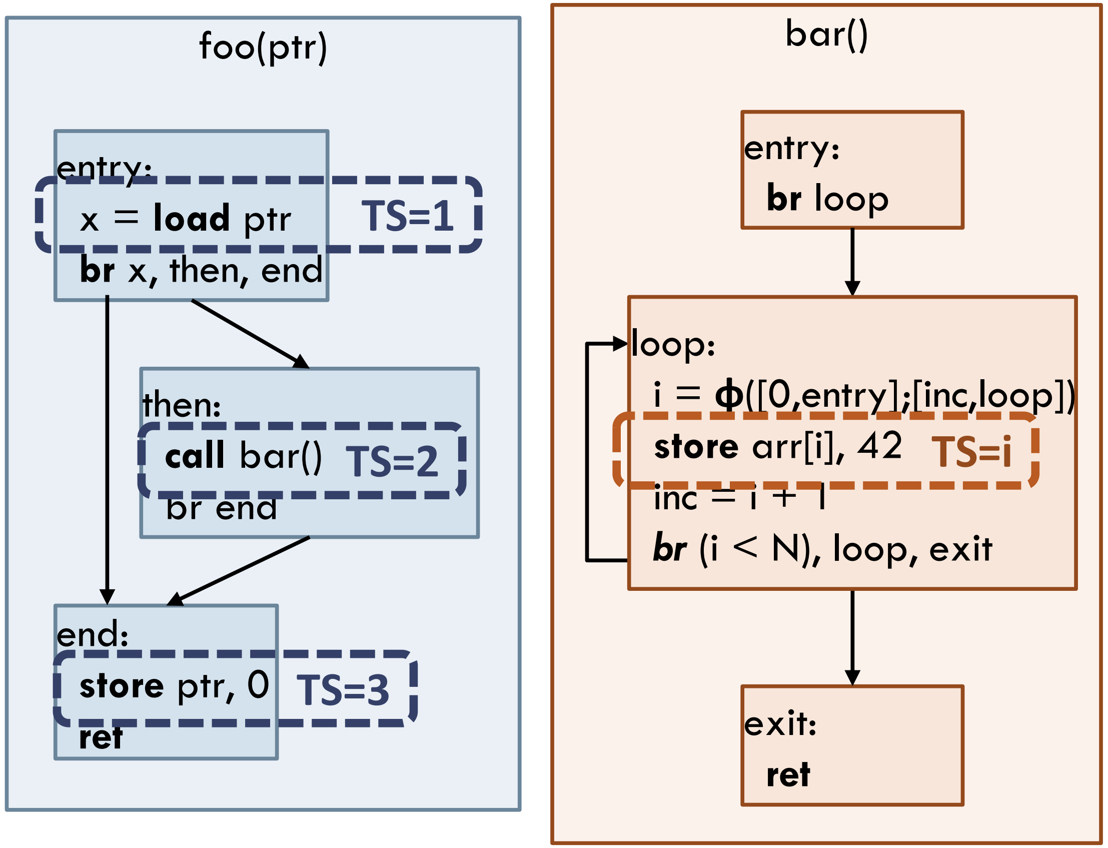
Must generate timestamps for arbitrary control-flow graph.

Topological sorting gives timestamps for acyclic control-flow (sub)graphs.

Targeting Fractal extension of Swarm hardware architecture [ISCA’17].

Loops, function calls handled by creating subdomains of timestamps.

Fractal hardware tracks hierarchy of timestamps, preserves apparent sequential execution.



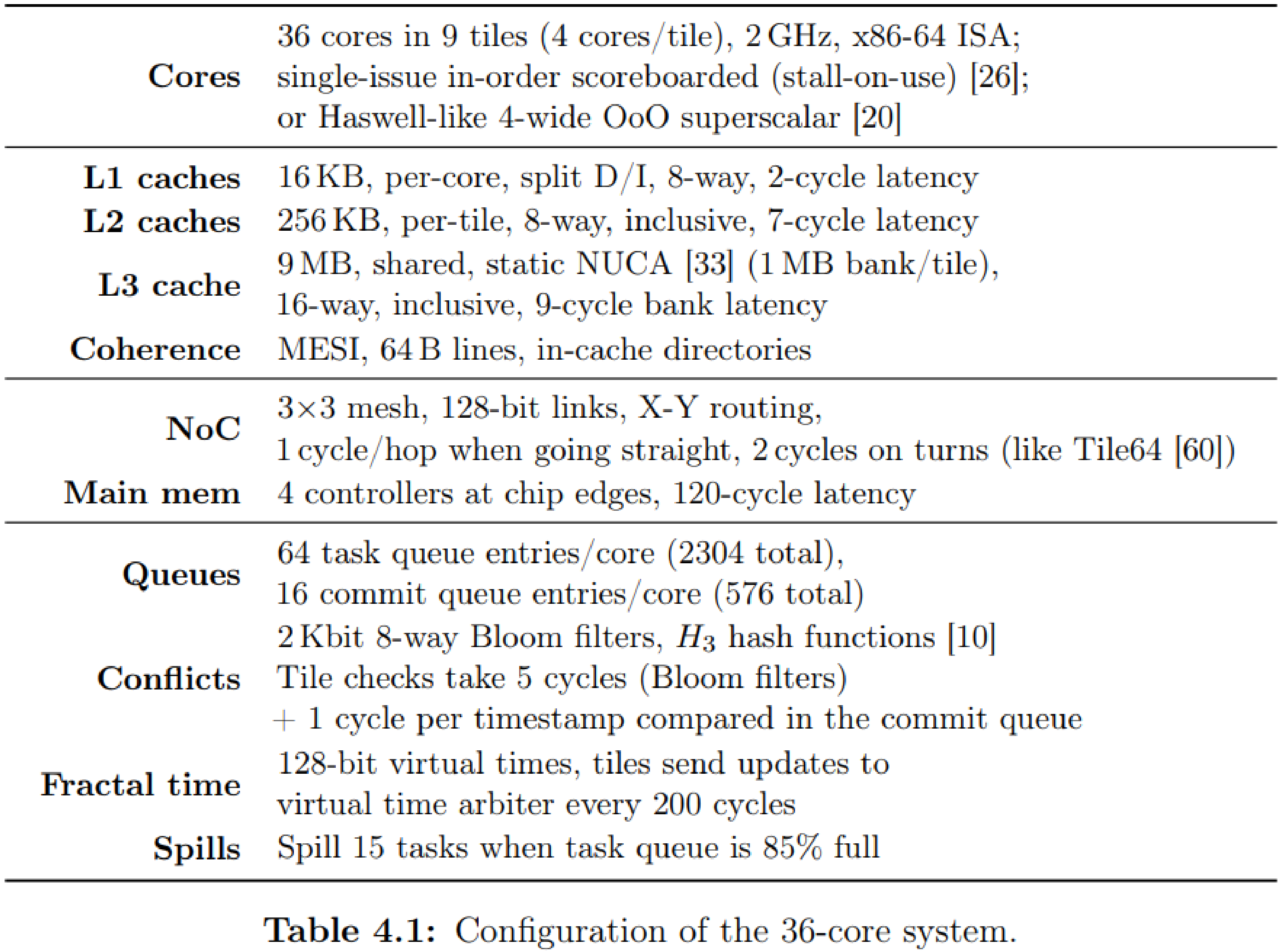
foo(ptr)

```
entry:
  x = load ptr        TS=1
  br x, then, end
then:
  call bar()  TS=2
  br end
end:
  store ptr, 0   TS=3
  ret
```

bar()

```
entry:
  br loop
loop:
  i = ɸ([0,entry];[inc,loop])
  store arr[i], 42  TS=i
  inc = i + 1
  br (i < N), loop, exit
exit:
  ret
```

Hardware Configuration



Cores	36 cores in 9 tiles (4 cores/tile), 2 GHz, x86-64 ISA; single-issue in-order scoreboarded (stall-on-use) [26]; or Haswell-like 4-wide OoO superscalar [20]
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 8-way, inclusive, 7-cycle latency
L3 cache	9 MB, shared, static NUCA [33] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	3×3 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [60])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (2304 total), 16 commit queue entries/core (576 total)
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [10] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Fractal time	128-bit virtual times, tiles send updates to virtual time arbiter every 200 cycles
Spills	Spill 15 tasks when task queue is 85% full

Table 4.1: Configuration of the 36-core system.