T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware

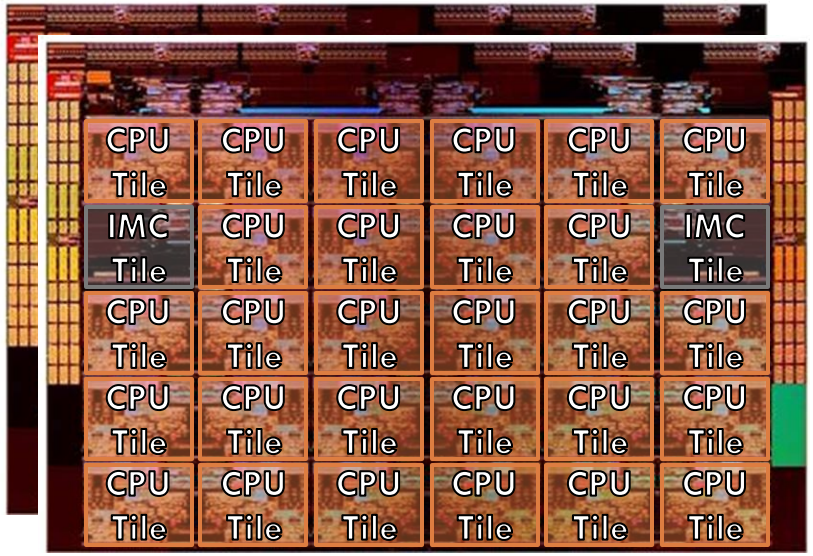
VICTOR A. YING
MARK C. JEFFREY*
DANIEL SANCHEZ

*University of Toronto starting Fall 2020

ISCA 2020

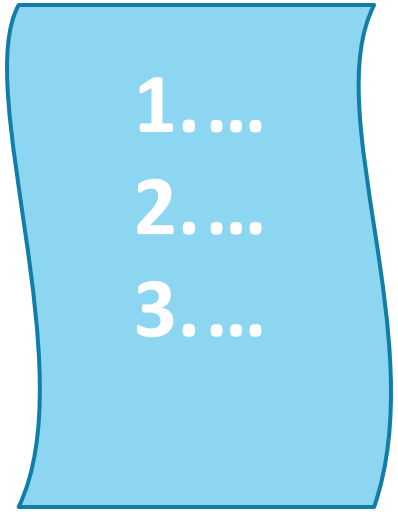
Parallelization: Gap between programmers and hardware

Multicores are everywhere



Intel Skylake-SP (2017): 28 cores per die

Programmers still write sequential code



Speculative parallelization: new architectures and compilers to parallelize sequential code without knowing what is safe to run in parallel

T4: Trees of Tiny Timestamped Tasks

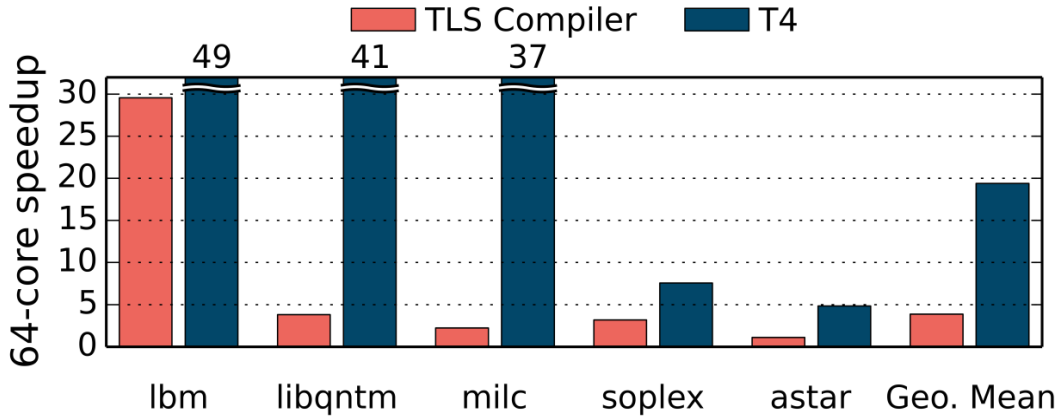
Our T4 compiler exploits recently proposed hardware features:

- Timestamps encode order, letting tasks spawn out-of-order
- Trees unfold branches in parallel for high-throughput spawn
- Compiler optimizations make task spawn efficient
- Efficient parallel spawns allows for tiny tasks (10's of instructions)
  - » Tiny tasks create opportunities to reduce communication and improve locality

swarm.csail.mit.edu

We target hard-to-parallelize C/C++ benchmarks from SPEC CPU2006

- Modest overheads (gmean 31% on 1 core)
- Speedups up to 49x on 64 cores



Background

T4 Principles in Action

T4: Parallelizing Entire Programs

Evaluation

Thread-Level Speculation (TLS) [Multiscalar ('92-'98), Hydra ('94-'05), Superthreaded ('96), Atlas ('99), Krishnan et al. ('98-'01), STAMPede ('98-'08), Cintra et al. ('00, '02), IMT ('03), TCC ('04), POSH ('06), Bulk ('06), Luo et al. ('09-'13), RASP ('11), MTX ('10-'20), and many others]

- Divide program into *tasks* (e.g., loop iterations or function calls)
- Speculatively execute tasks in parallel
- Detect dependencies at runtime and recover

Prior TLS systems did not scale many real-world programs beyond a few cores due to

- Expensive aborts
- Serial bottlenecks in task spawns or commits

TLS creates chains of tasks

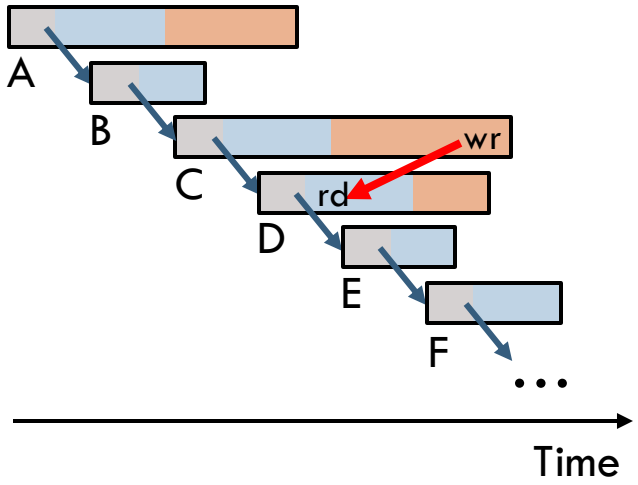
Example: maximal independent set
- Iterates through vertices in graph

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 : neighbors(v))
      state[nbr] = EXCLUDED;
  }
}
```

Indirect memory accesses

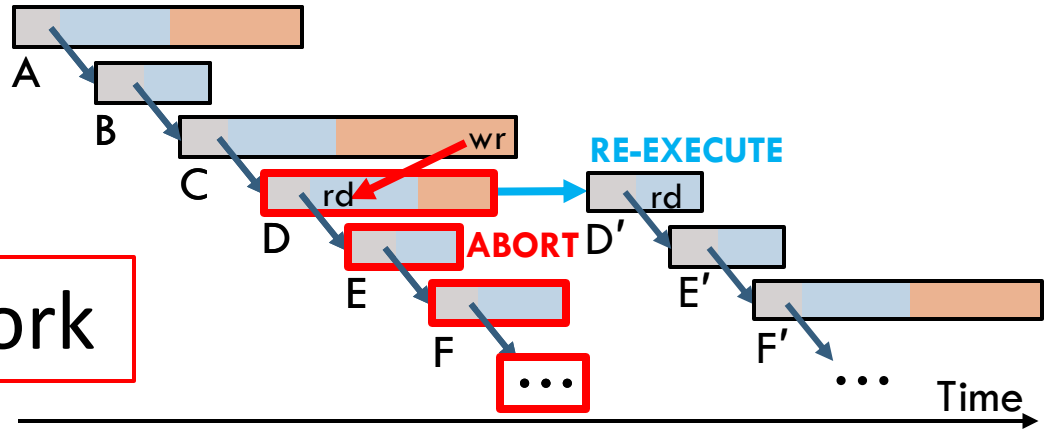


Task chains incur costly misspeculation recovery

Tasks abort if they violated data dependence

Tasks that abort must roll back their effects, including successors they spawned or forwarded data to

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



Unselective aborts waste a lot of work

Swarm architecture

[Jeffrey et al. MICRO'15, MICRO'16, MICRO'18; Subramanian et al. ISCA'17]

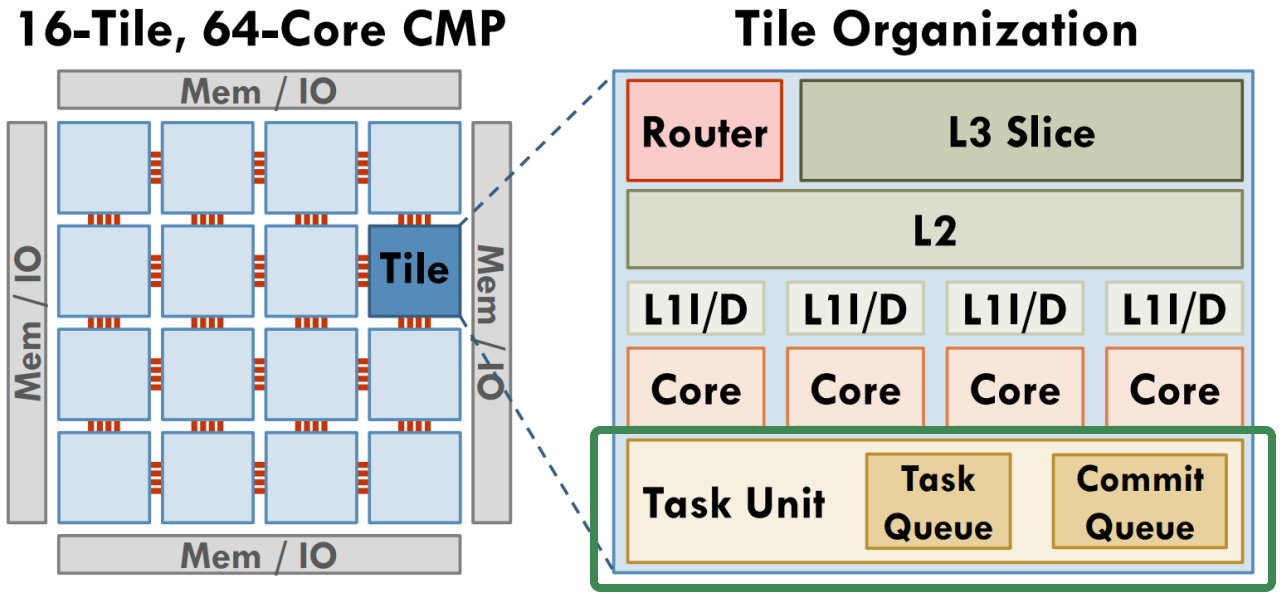
Execution model:

- Program comprises timestamped tasks
- Tasks spawn children with greater or equal timestamp
- Tasks appear to run sequentially, in timestamp order

Detects order violations and *selectively aborts dependent tasks*

Distributed task units queue, dispatch, and commit multiple tasks per cycle

- <2% area overhead
- Runs hundreds of tiny speculative tasks



Background

T4 Principles in Action

T4: Parallelizing Entire Programs

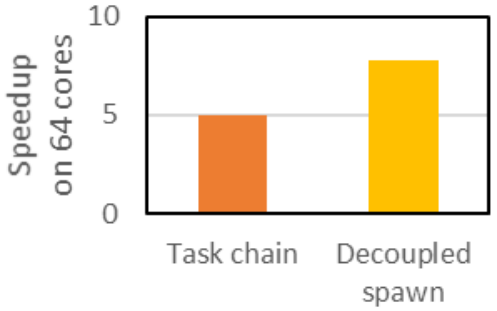
Evaluation

T4's decoupled spawn enables selective aborts

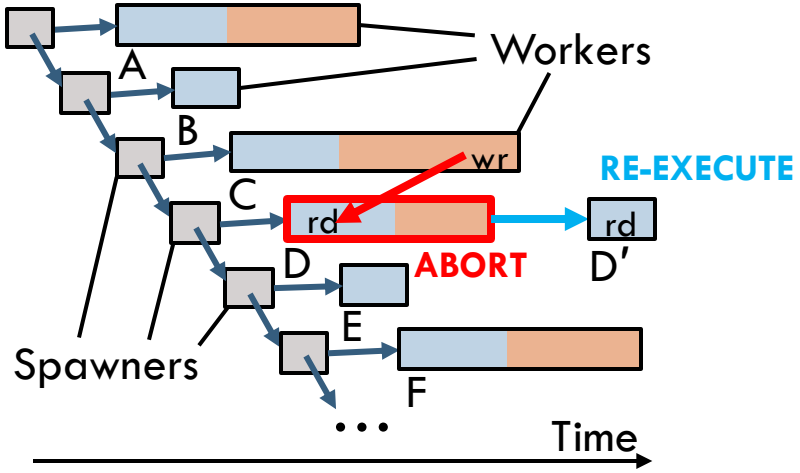
T4 compiles sequential C/C++ to exploit parallelism on Swarm

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 : neighbors(v))
      state[nbr] = EXCLUDED;
  }
}
```

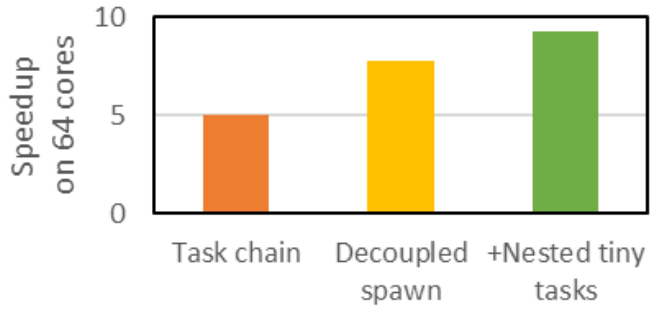


Tiny tasks make aborts cheap

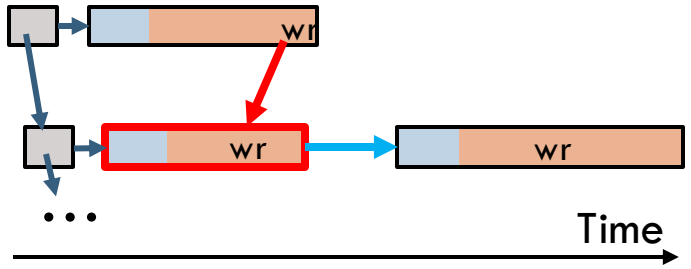
Isolate contentious memory accesses into tiny tasks, to limit the damage when they abort

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

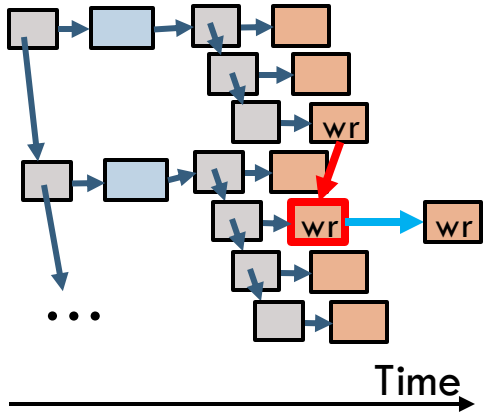
Tiny tasks (a few instructions) are difficult to spawn effectively



Parallelize outer loop only:

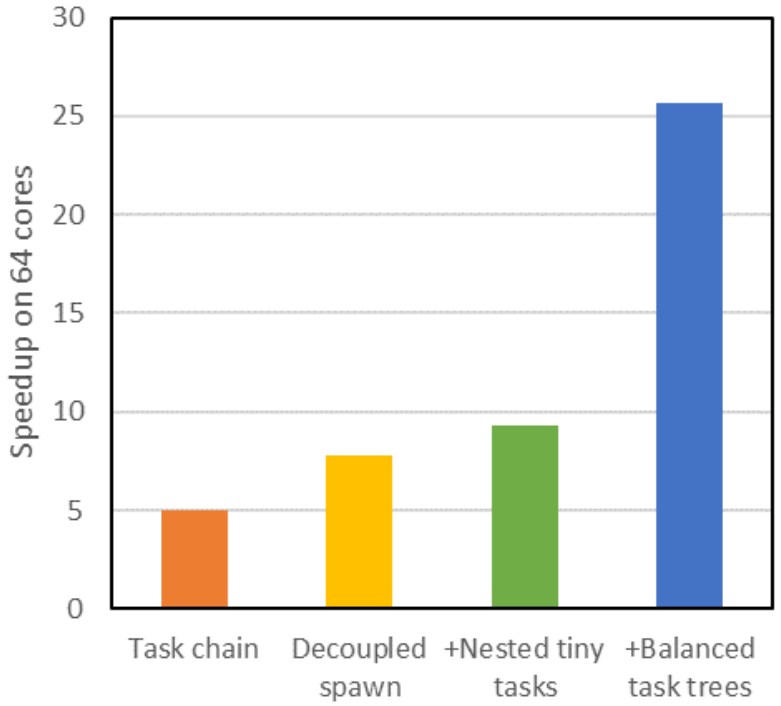


Parallelize both loops:

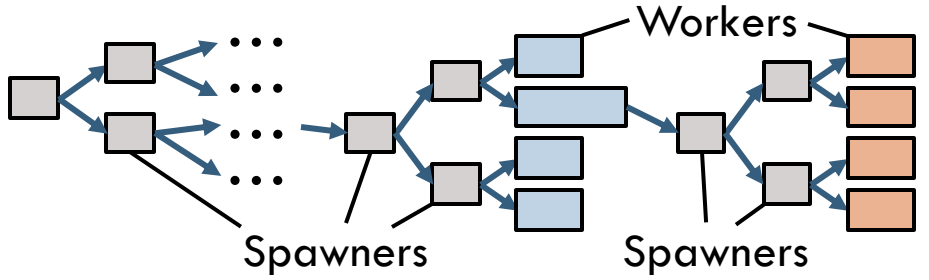


T4's balanced task trees enable scalability

Spawners recursively divide the range of iterations




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



Balanced spawner trees reduce critical path length to O(log(tripcount))

Background

T4 Principles in Action

T4: Parallelizing Entire Programs

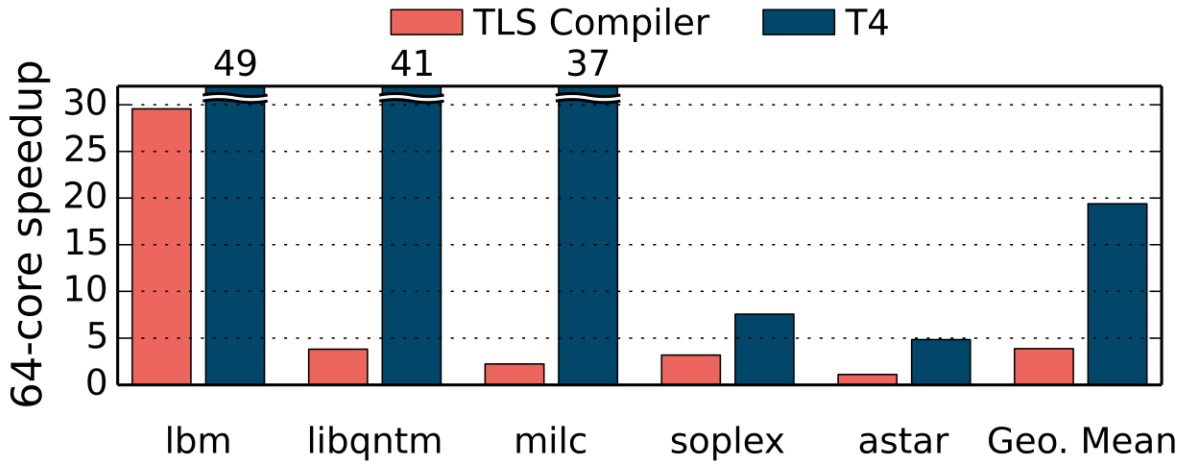
Evaluation

T4: Parallelizing entire real-world programs

T4 divides the entire program into tasks starting from the first instruction of `main()`

T4 automatically generates tasks from
- Loop iterations
- Function calls
- Continuations of the above

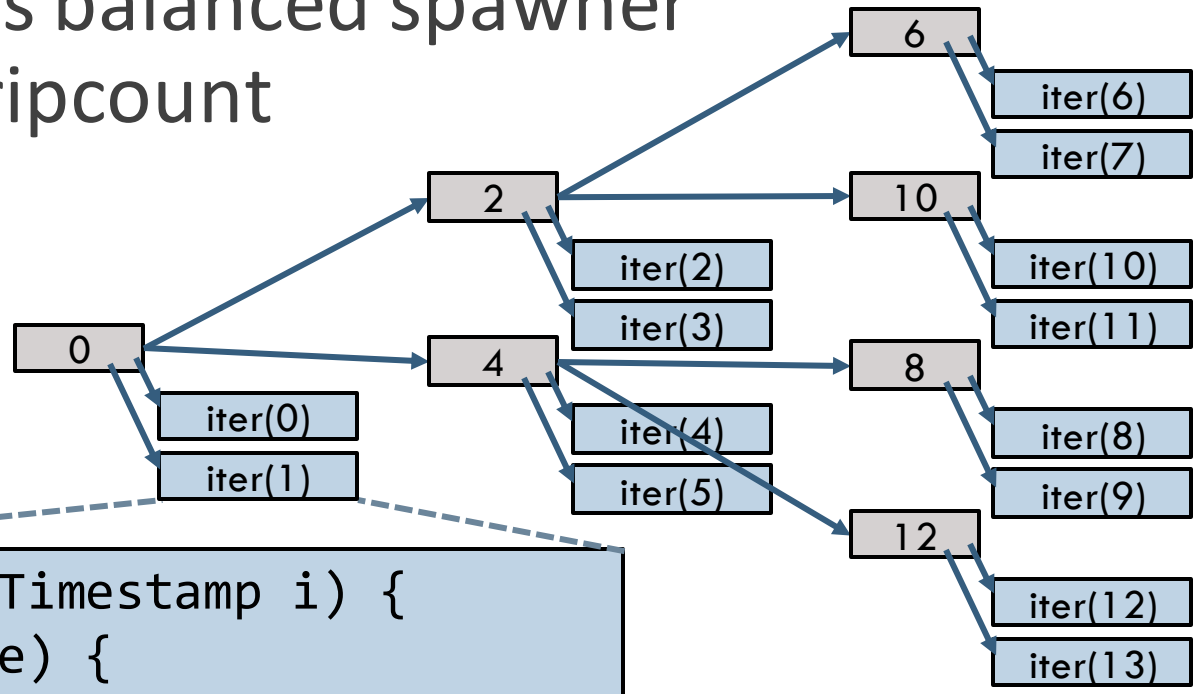
T4 extracts nested parallelism from the **entire program** despite
- Loops with unknown tripcount
- Opaque function calls
- Data-dependent control flow
- Arbitrary pointer manipulation



Progressive expansion of unknown-tripcount loops

Progressive expansion generates balanced spawner trees for loops with unknown tripcount

- loops with **break** statements
- **while** loops



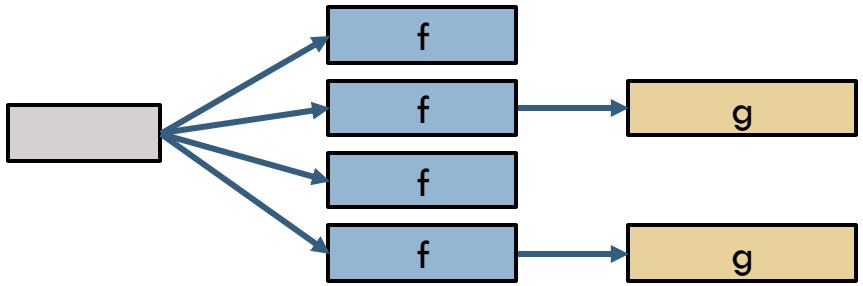
Source code:

```
int i = 0;
while (status[i]) {
  if (foo(i)) break;
  i++;
};
```

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

Continuation-passing style eliminates the call stack

```
for (int i = 0; i < N; i++) {
  float x = f();
  if (x > 0.0) g(x);
}
```



Problem: Independent function spawns serialize on stack-frame allocation

Solution:

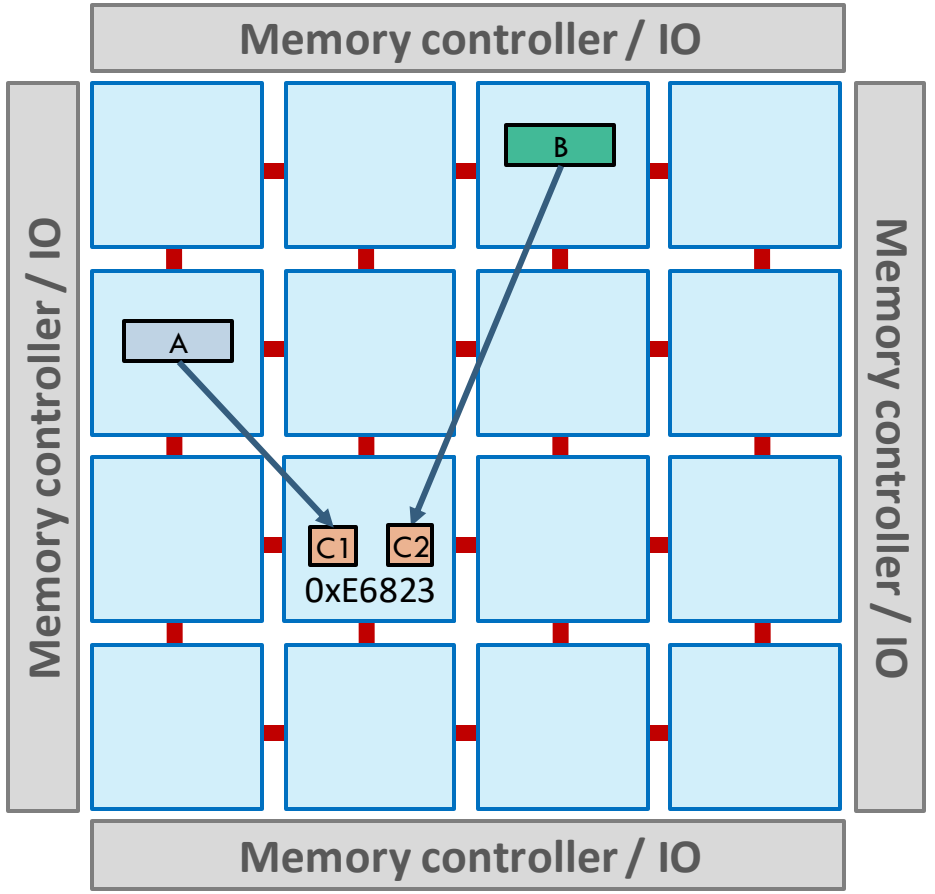
- When needed, T4 allocates continuation closures on the heap instead
- T4 optimizations ensure most tasks don't need memory allocation
- These software techniques could apply to any TLS system

Spatial-hint generation for locality

Tiny tasks may access only one memory location, which is known when the task is spawned.

Hardware uses these spatial hints to improve locality:

- maps each address to a tile.
- Send tasks for that address to that tile.



Manual annotations for task splitting

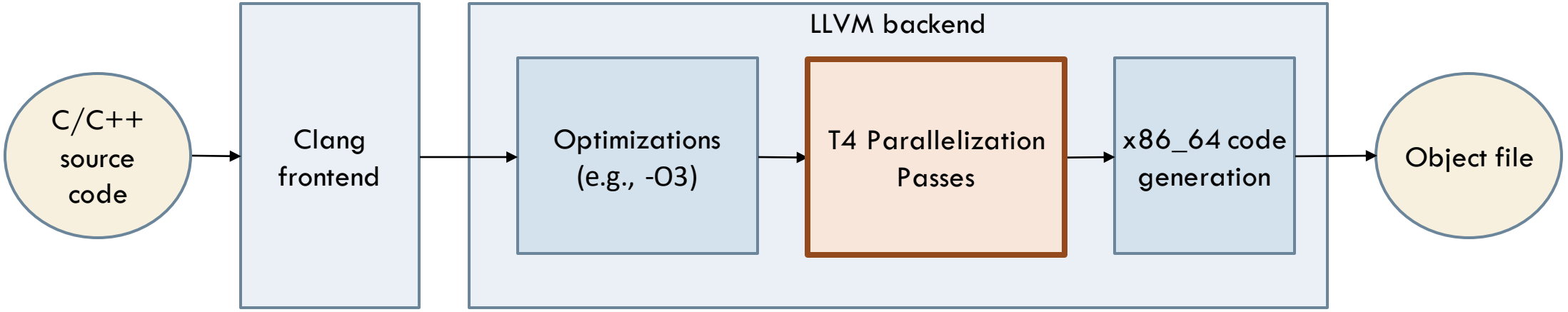
Programmer may add task boundaries for tiny tasks

Guaranteed to have no effect on program output

Added <0.1% to source code

Benchmark	Lines of code	Modified lines
429.mcf	1,574	None
433.milc	9,575	+18, -13
444.namd	3,887	None
450.soplex	28,302	+25, -16
456.hmmer	20,680	+11, -9
462.libquantum	2,605	None
464.h264ref	36,032	+12, -9
470.lbm	904	+1, -1
473.astar	4,285	+29, -144
482.sphinx3	13,128	+17, -8
Total	120,972	+114, -201

T4 implementation in LLVM/Clang



Intraprocedural passes: small compile times (linear in code size)
Use all standard LLVM optimizations to generate high-quality code
More in the paper:

- Topological sorting to generate timestamps
- Bundling stack allocations to the heap with privatization
- Loop task coarsening to reduce false sharing of cache lines
- Case studies and sensitivity studies

Background

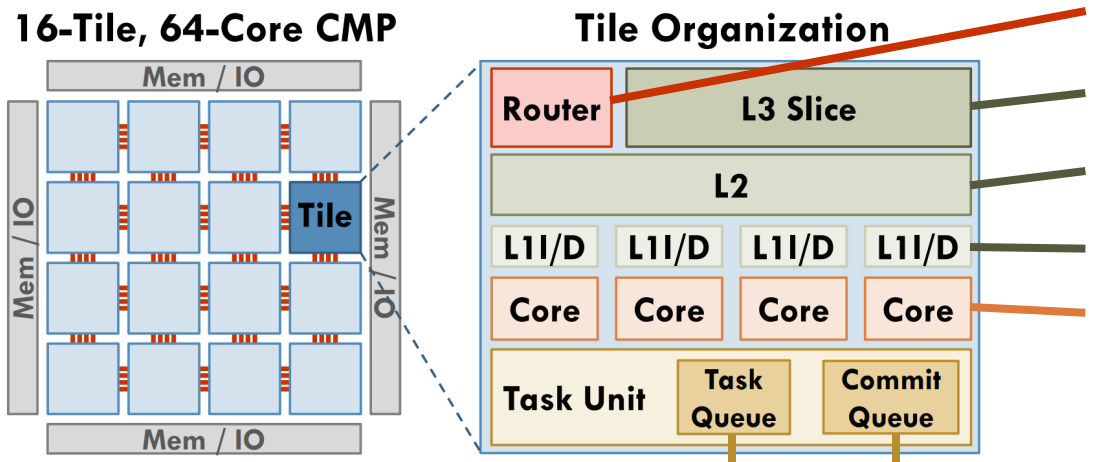
T4 Principles in Action

T4: Parallelizing Entire Programs

Evaluation

Methodology

1-, 4-, 16-, and 64-core systems



4 4×4 mesh networks

4 MB L3 slice per 4-core tile

1 MB L2 cache per 4-core tile

32 KB L1 caches

4-wide superscalar out-of-order cores (Haswell-like)

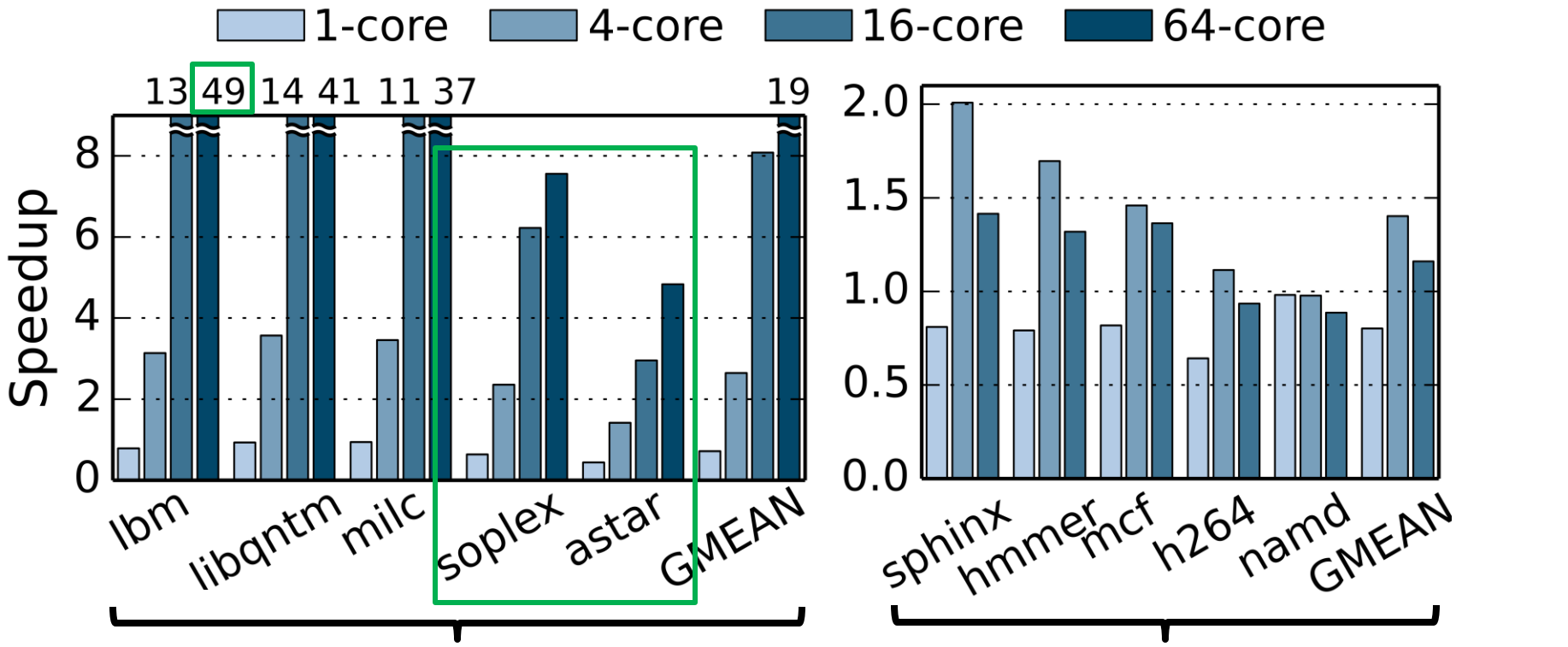
256 entries

64 entries/tile (1024 tasks for 64-core chip)

C/C++ benchmarks from SPEC CPU2006

All speedups normalized to serial code compiled with `clang -O3`

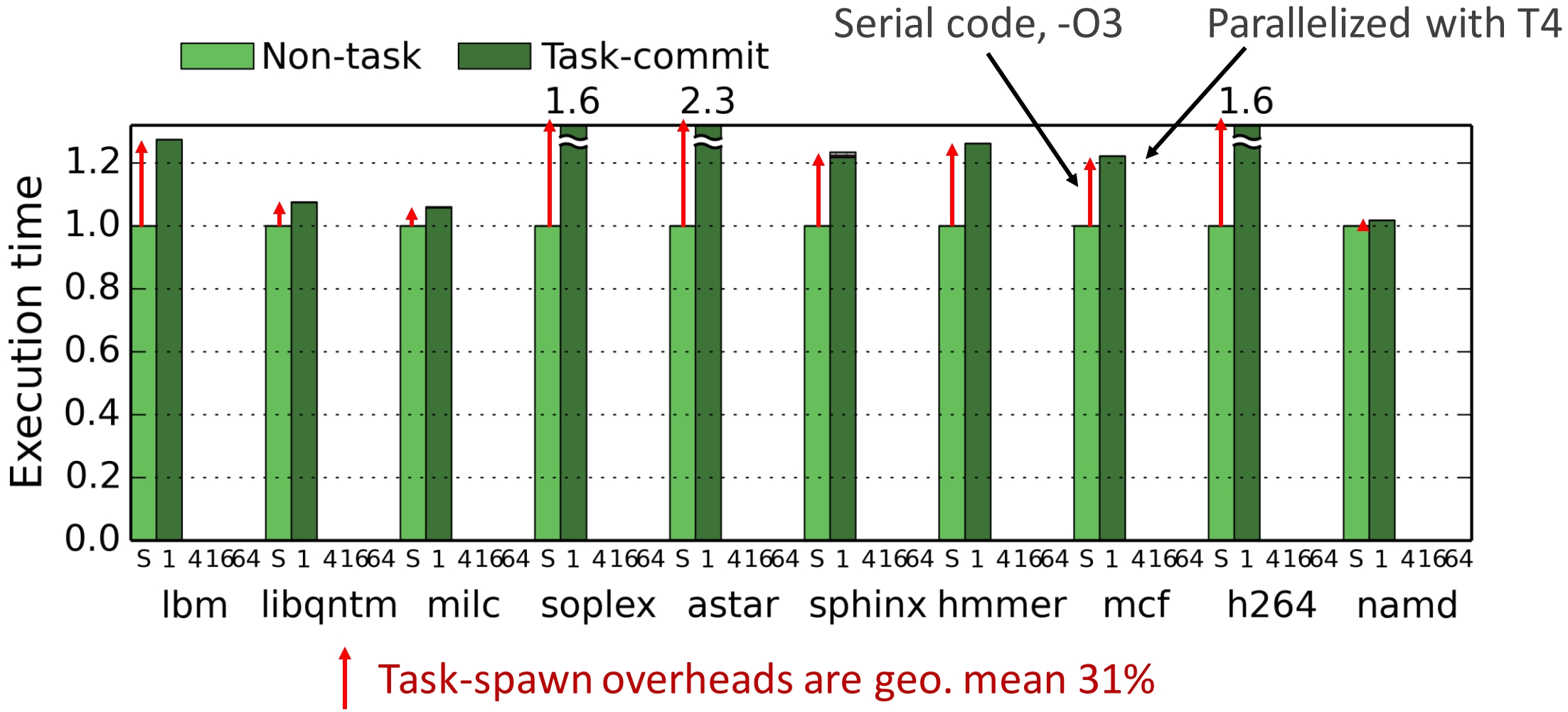
T4 scales to tens of cores



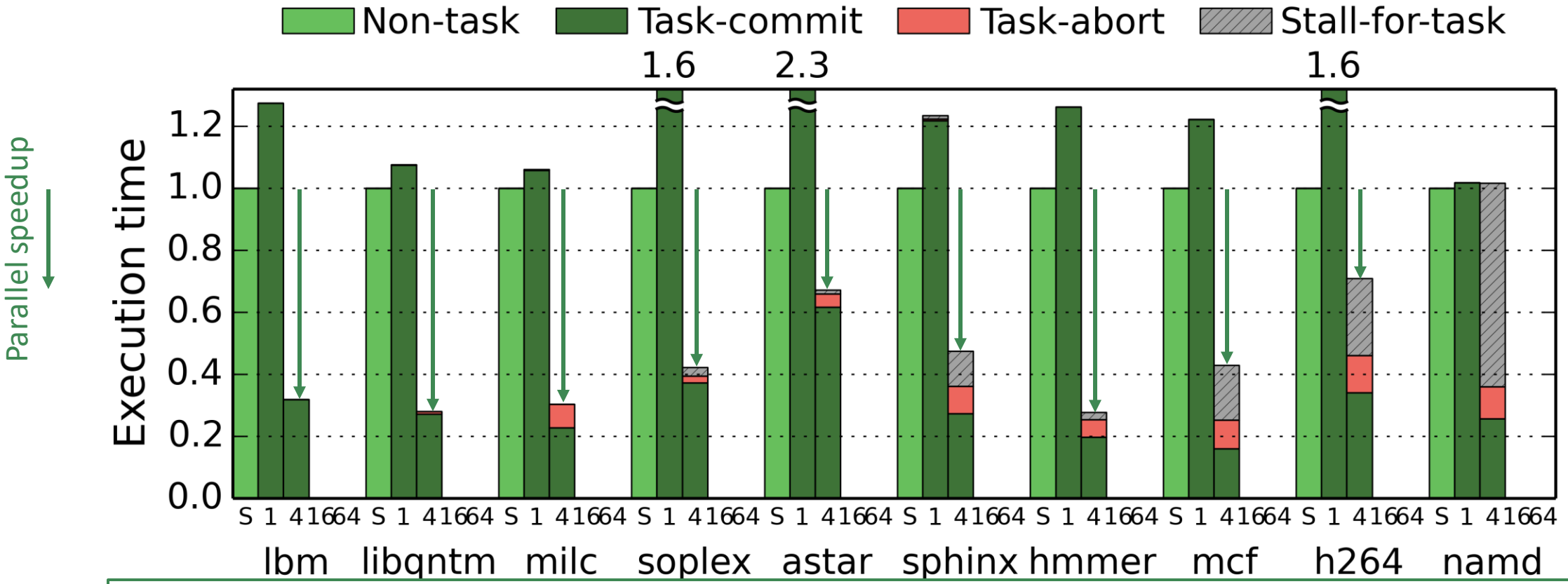
Hot loops have some independent iterations

Hot loops have serializing variables updated every iteration

T4 overheads are moderate

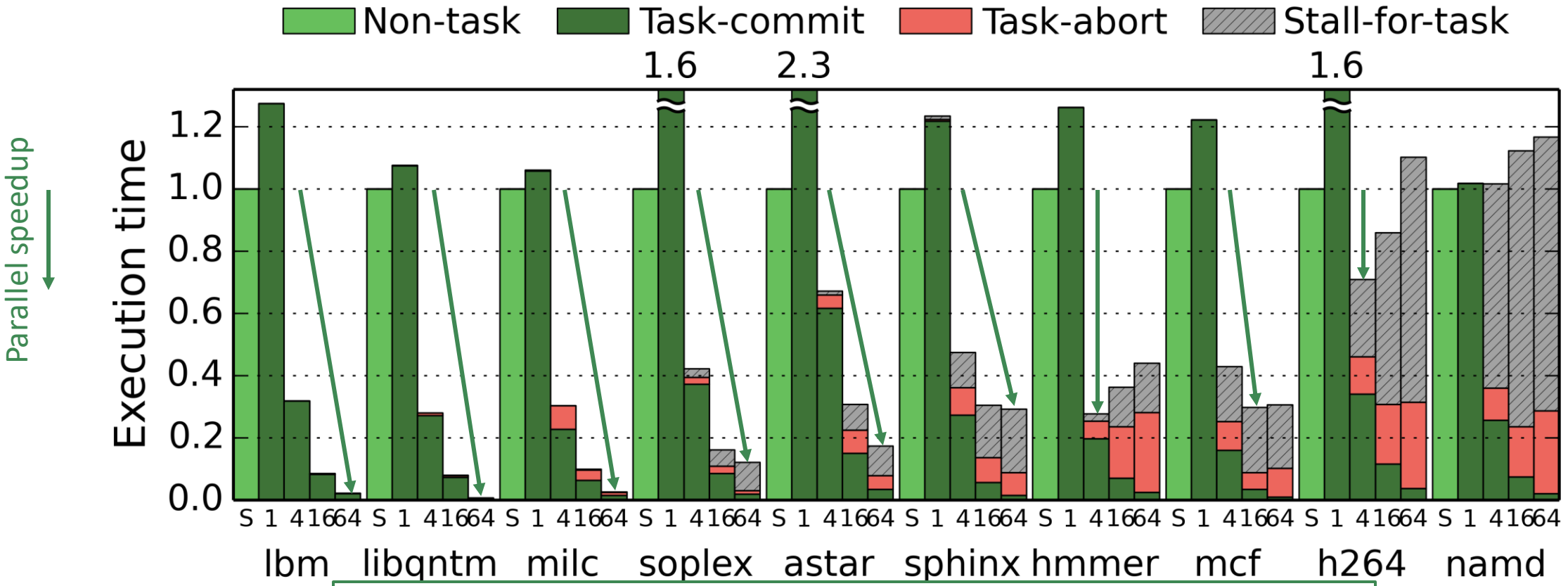


Parallelization redoubles performance



Cores spend most time executing useful work, not aborting

Parallelization redoubles performance



T4 scales many programs to tens of cores

Contributions

T4 compiler provides parallelization needed to allow sequential programmers to use multicores

T4 broadens the applications for which speculative parallelization is effective by exploiting the recent Swarm architecture

- Parallelization of sequential C/C++ yields speedups of up to 49× on 64 cores

New code transformations:

- Decoupled spawners enable cheap selective aborts of tiny tasks
- Progressive expansion: balanced task trees for unknown-tripcount loops
- Stack elimination and loop task coarsening reduce false sharing
- Task spawn optimizations

Questions?

T4 is open-source and available for you to build on:

swarm.csail.mit.edu

Join online Q&A @ ISCA: First paper in Session 2B on June 1, 2020

9am in Los Angeles, Noon in New York, 6pm in Brussels, Midnight in Beijing