Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation

Why existing compilers for parallel code suck and how to fix them

William S. Moses, Tao B. Schardl, Charles E. Leiserson

WHAT A GOOD COMPILER CAN DO

Compilers are wonderful tools that allow us to write code in high-level languages. We also depend on them to optimize our code. The difference between a good and bad compiler can be enormous. A poor compiler can have serious adverse consequences for applications that demand efficiency. For example, in Fig 1. a single change in compiler optimization level can lead to an order-of magnitude improvement in the performance of a simple image processing pipeline.

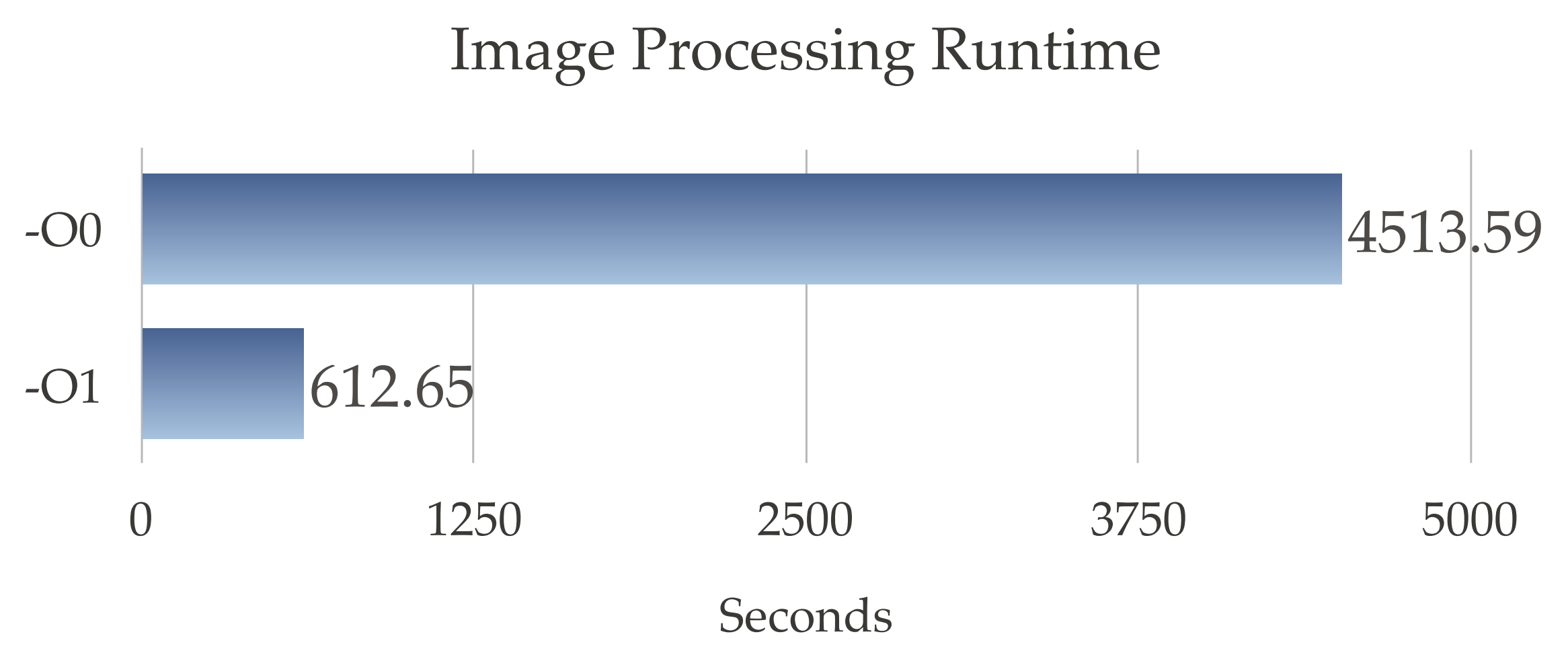

Fig 1. Comparing the performance of an image processing toolkit on different compiler optimization levels. A single change in optimization level can make an order-of-magnitude improvement.

MULTICORE PROCESSORS

For the past several decades, we have been able to rely on Moore's law to provide us with improvements in performance by roughly doubling the clock speed of processors every few years. Stemming from fundamental physical limitations on power, however, the performance of individual cores is no longer improving at the rate it once was. As a way to cope with power limitations, semiconductor vendors add many processing cores to a single machine in order to continue to scale performance.

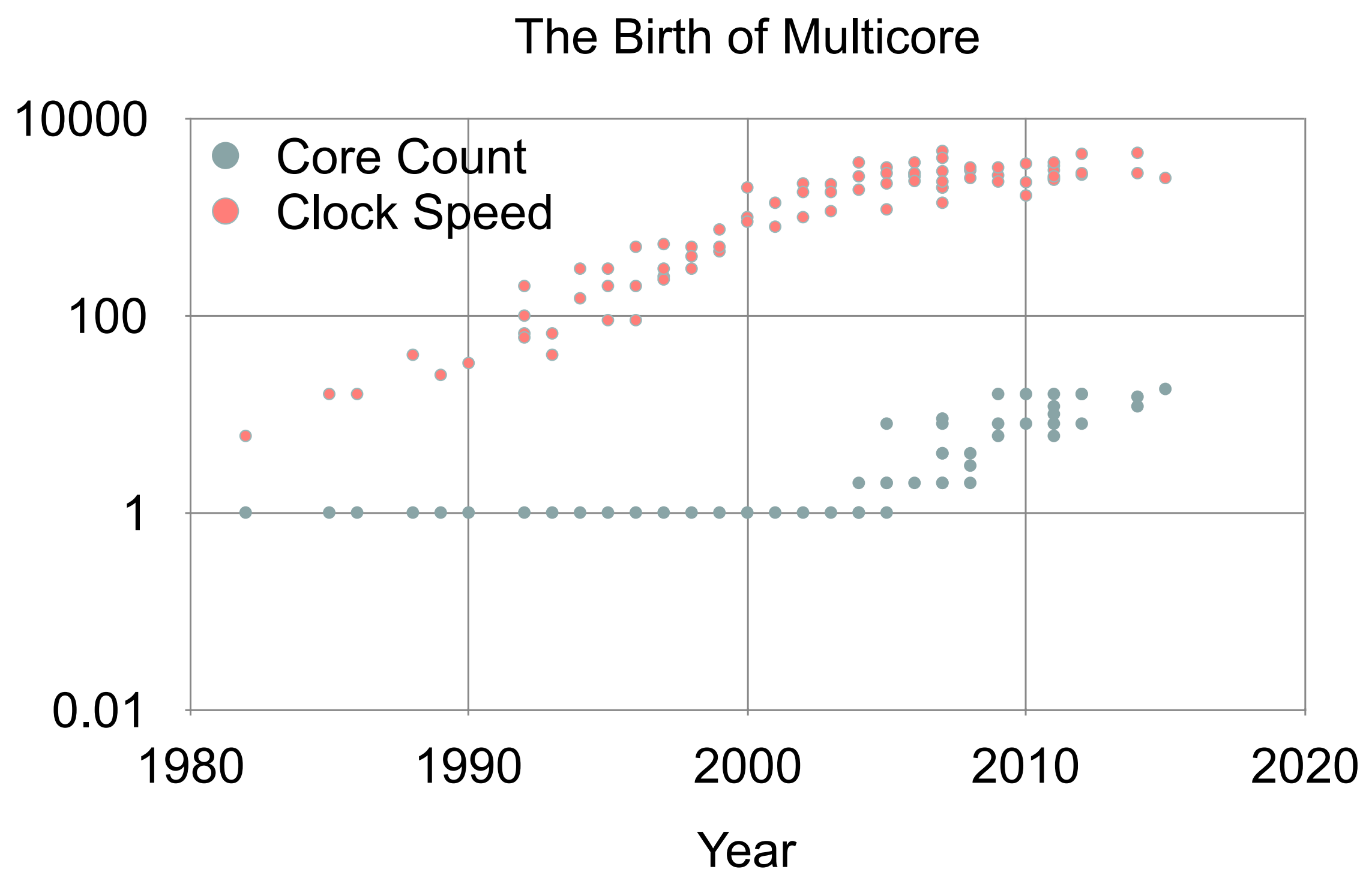

Fig 2. Here, we graph the clock speed of individual cores with time, as well as the number of cores available on a single semiconductor processor chip. Around 2005, clock speeds level out as a result of power limitations. At roughly the same time, we see the number of cores start to rise.

WHERE COMPILERS FAIL

```
__attribute__((const))
double norm(const double *A, int n);

void normalize(double *restrict out,
               const double *restrict in, int n) {
  cilk_for (int i = 0; i < n; ++i)
    out[i] = in[i] / norm(in, n);
}
```

Fig 3. Cilk code to normalize a vector in parallel. The `cilk_for` keyword denotes that iterations of the loop can run independently.

Modern compilers allow programmers to easily write parallel code with high-level frameworks such as OpenMP and Cilk. In these frameworks, programmers specify tasks that may be run in parallel, such as the iterations of the loop in Fig 3.

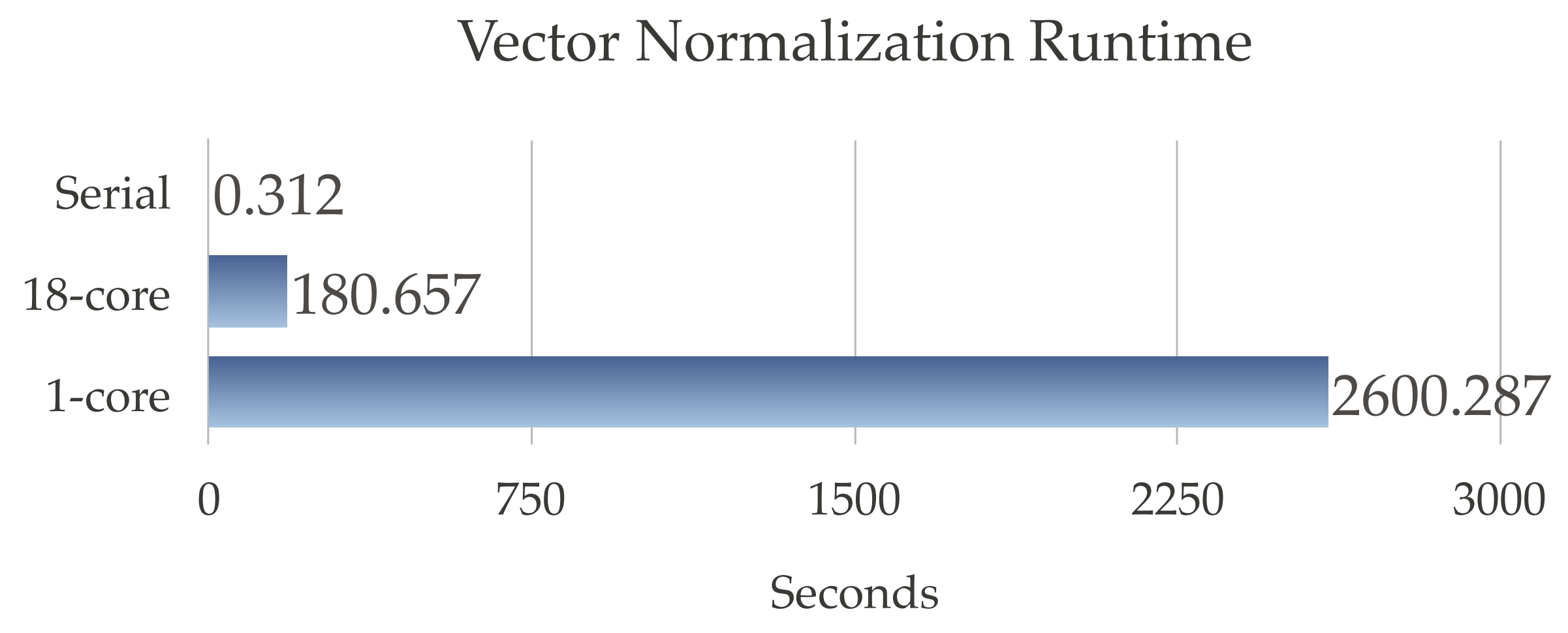

Fig 4. Time it takes to normalize a vector using the serial and parallel versions of the code in Fig 3. Surprisingly, the serial version runs faster than the parallel version.

Although current compilers can successfully compile and run parallel code, they suffer from a major flaw: they can't optimize them well. Sometimes these parallel programs run much slower than a comparable serial version, as in Fig 4. In this example, the compiler is unable to perform loop-invariant code motion as shown in Fig 5. Consequently, the parallel program performs many unnecessarily calls to the norm function.

```
void normalize(double *restrict out,
               const double *restrict in, int n) {
  double tmp = norm(in, n);
  cilk_for (int i = 0; i < n; ++i)
    out[i] = in[i] / tmp;
}
```

Fig 5. An optimized version of the `normalize` code from Fig 4. In this code, loop-invariant code motion is performed, allowing iterations of the for loop to avoid having to make multiple calls to the expensive norm function.

TAPIR: PARALLELISM IN THE COMPILER

Compilers aren't able to optimize parallel code well because they have no way of representing the parallel semantics of programs. As a result, they extract parallel programs into separate functions that they then pass as arguments to a parallel runtime. This confusing way of representing a program prevents the compiler from being able to do any of its standard analysis and optimization.

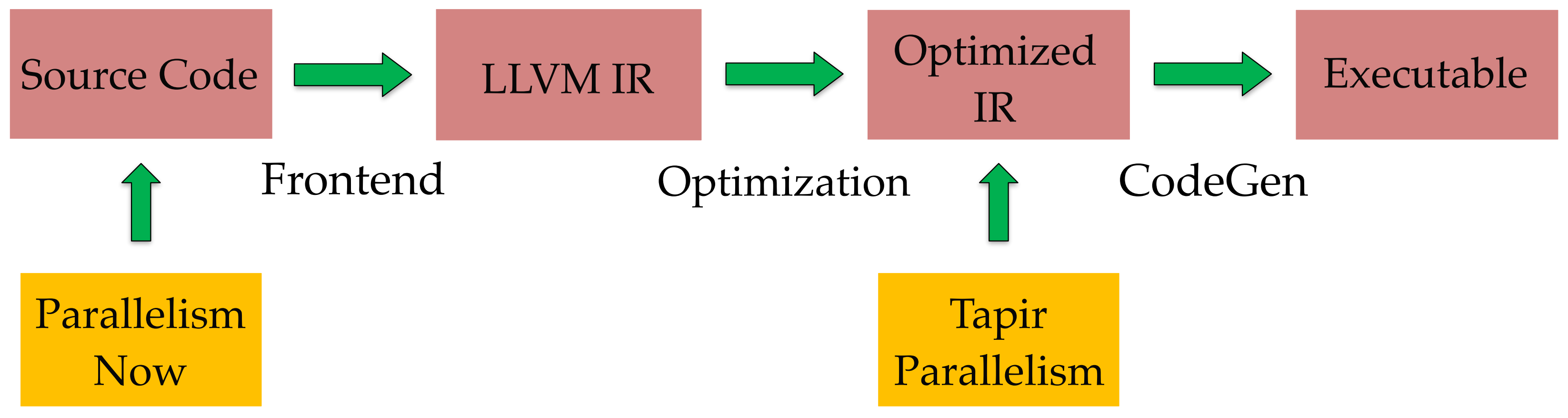

Fig 6. The pipeline for compiling parallel programs with and without Tapir. Parallel programs are currently lowered to runtime calls before any optimizations are able to occur. Tapir is able to represent the parallelism in compiler, allowing it to be lowered to runtime calls after optimization.

Tapir allows the compiler to represent parallel programs as a natural extension to serial code without a dependency between parallel tasks. Existing compiler optimizations work with Tapir with zero or minimal modification. In fact, to add Tapir constructs to the LLVM compiler required modifying only 0.1% of the 4-million-line codebase.

EVALUATION

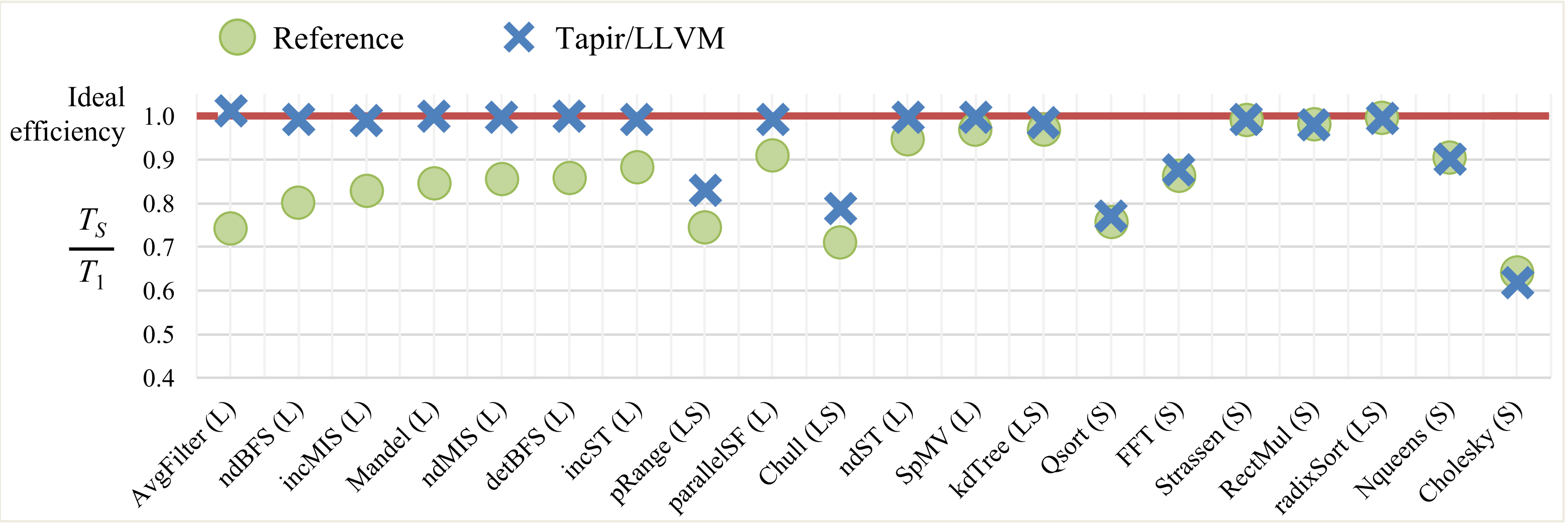

Fig 7. An evaluation of the work efficiency of parallel codes with and without the Tapir representation. For a third of these benchmarks, Tapir reaches ideal efficiency, while the reference implementation falls short. Moreover, Tapir typically improves efficiency across the board.

Won Best Paper at the 2017 Conference on Principles and Practice of Parallel Programming

Artifact Evaluated PPoPP AEC Consistent Complete Well Documented Easy to Reuse