Embedding Fork-Join Parallelism into LLVM IR

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

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{wmoses, neboat, cel}@mit.edu

This paper explores how fork-join parallelism, as supported by dynamic multithreading concurrency platforms such as Cilk and OpenMP, can be embedded into LLVM IR. Mainstream compilers, such as LLVM, typically treat parallel linguistic constructs as syntactic sugar for function calls into a parallel runtime. These calls prevent the compiler from performing optimizations across parallel control flow. As a result, the serial equivalant for many programs is faster than running in parallel because the compiler is unable to perform many of its usual optimizations. For example in Figure 1 the compiler is unable to do loop-invariant code motion.

```
01 __attribute__((const)) double norm(const double *A, int n);
02
03 void normalize(double *restrict out,
04                 const double *restrict in, int n) {
05   cilk_for(int i = 0; i < n; ++i)
06     out[i] = in[i] / norm(in, n);
07 }
```

Figure 1. Example Cilk function that neither GCC nor the Cilk Plus/LLVM compiler optimize effectively. The `cilk_for` loop on lines 5–6 allows each iteration of the loop to execute in parallel. The `norm` function computes the norm of a vector in $\Theta(n)$ time. The call to `norm` on line 6 can be safely moved outside of the loop, but neither GCC nor Cilk Plus/LLVM will perform this code motion.

Remedying this situation, however, is generally thought to require extensively reworking compiler analyses and code transformations to handle parallel semantics. This is generally difficult because some serial optimizations such as strength reduction cannot be applied to parallel code in general.

This paper introduces Tapir, a compiler IR that represents logically parallel tasks asymmetrically in the program's control flow graph. Tapir's allows a compiler to optimize across parallel control flow with only minor changes to its analyses and code transformations. Tapir enables a variety of compiler optimizations, including traditional compiler optimizations such as loop-invariant-code motion, loop unrolling, scalar replacement of aggregates, as well as new parallel optimizations.

As a result of both difficulty of analysis and dependence on particular runtime systems, most mainstream compilers have not performed parallelization as an optimization beyond vectorization. Tapir resolves both of the issues for compilers by making it possible to easily perform parallel analysis in a runtime-independent way. Some examples of parallel-specific optimizations which Tapir easily enables include loop-parallelization and parallel tail recursion elimination, among others.

Tapir introduces three new instructions to LLVM's IR in order to represent parallel tasks. These instructions are `detach`, `reattach` and `sync`. The syntax for these instructions is shown in Figure 2. At a high-level, detach, reattach, and sync serve to separate logically parallel tasks. This is done by using a structure similar to a branch and code in Tapir is thus able to easily interact with existing optimizations.

```
08 detach label <detached>, label <continuation>
09 reattach label <continuation>
10 sync
```

Figure 2. LLVM IR syntax for the `detach`, `reattach`, and `sync` instructions. The `label` keyword indicates that `detach` and `reattach` take basic block labels as arguments.

To prototype Tapir in LLVM, we added or modified approximately 1700 lines of LLVM's approximately 3 million line codebase. These changes alone were sufficient to allow Tapir to enable most serial optimization passes. Tapir was tested both on small microbenchmarks as well as larger applications. Results for the larger applications are shown in Figure 3.

Application	*Cilk* T_1/ *Serial*	*GCC/ Serial*	*Tapir/ Serial*
AveragingFilter	2.03	1.41	0.93
BlackScholes	1.30	1.03	1.01
DCT	1.00	1.00	0.99
Mandelbrot	1.95	0.99	1.00
MonteCarlo	1.00	1.01	1.01
SepiaFilter	2.83	1.19	1.01

Figure 3. Large application benchmarks for Tapir. Tests are run against three compilers – CilkPlus/LLVM which directly translates to runtime calls before optimizing, GCC which is able to perform some optimization at the front-end, and Tapir. Benchmarks were run using one thread on Amazon c4.8xlarge instances.

The results of these benchmarks are promising — providing reasonable performance improvements on large-scale applications where the compiler simply converted to runtime calls. These results were then compared with the latest version of GCC — which through immense amounts of code-duplication and front-end hacks is able to perform similarly for some optimizations. Code using Tapir, however, is naturally able to perform all such optimizations without large amounts of explicit optimizations on the front-end. Moreover, Tapir is much more extensible and able to work with any new optimization passes as well as allow for parallel-specific optimization passes.