Parallel IR

William Moses Tao B. Schardl

MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

This paper introduces simple constructs to represent fork-join parallel control dependencies in the LLVM intermediate representation (IR). These constructs are designed to require minimal modifications to existing LLVM IR constructs to interact correctly with existing LLVM optimization passes. Simultaneously, we demonstrate that these constructs are expressive, allowing the LLVM IR to express the fork-join parallel control dependencies expressed in Cilk programs and similar parallel languages. Furthermore, these constructs enable a variety of compiler optimizations to improve the performance and scalability of parallel programs, including (list of optimizations)[[Fill in the blank.]].

1. INTRODUCTION

The goal of this paper is to establish a framework for safe parallelization in the LLVM Intermediate Representation (IR) [2].

Through its use of Single Static Assignment (SSA), it is very easy to create optimization passes in LLVM. As a result, LLVM and the corresponding Clang compiler project have grown to be competitors with GCC.

While LLVM makes it easy to optimize serial code, it provides little to optimize parallel code. This is because in LLVM, parallel code is represented in a obfuscated manner through additional functions and library calls, inhibiting optimization. Consequently any code the user wishes to run in parallel must be contained in its own function. As a result, many of the traditional serial optimizations don't work on parallel code. Likewise, as a result of this obfuscation, it is difficult or impossible create optimization passes specifically targeted at parallel code – such as moving synchronizations or even constant folding.

Prior Work

There have been previous attempts to create some sort of parallel framework inside of LLVM. However, none of these have come into use by the community for a number of various reasons.

Inside the cuurent version of LLVM (3.8), there exist some metadata constructs for basic parallelism – namely `llvm.mem.parallel_loop_access` which provides LLVM the ability to optionally parallelize specific pieces of code if a number of conditions are met. However, this construct is so limited that nothing beyond the simplest parallel code can use this metadata without being transformed into the serial equivalant.

There was a previous attempt by some of the LLVM developers to integrate some of the OpenMP library calls into native LLVM instructions. However, this was deemed to specific to OpenMP and the number and complexity of new operations did not seem to significantly ease optimization.

Other attempts to create a framework for parallel IR ran into issues of correctness. Specifically, existing optimization passes would incorrectly move code around and create parallel code with entirely different outputs. These sorts of issues manifested themselves as complex race conditions and even simple logical errors such the following subexpression elimination below.

```
parallel.task.start(1)
  work1(n/2)
parallel.task.end(1)
parallel.task.start(2)
  work2(n/2)
parallel.task.end(2)
```

becomes

```
parallel.task.start(1)
  tmp = n/2
  some_fun(tmp)
parallel.task.end(1)

parallel.task.start(2)
  other_fun(tmp)
parallel.task.end(2)
```

This is erronous as `tmp` may not yet have been calculated before it is used in the second parallel task – if it is available at all.

Our Work

Given these constraints, we have developed a parallel IR must maintains correctness, is able represent a wide variety of parallel frameworks, and easily lends itself to optimization. Additionally, our IR does not break existing LLVM code or optimization passes – but is able to use the existing optimization passes to its benfit.

We propose adding the following constructs to the IR to directly represent parallelism in LLVM: a `detach` instruction and a `sync` instrinsic. These constructs aim to allow the compiler to better optimize parallel code, such as Cilk code [1]. The remaining sections of this paper describe these constructs and their implications. Section 3 describes these constructs in detail, and Section **??** illustrates how these constructs can represent parallelism in some

parallel codes. Section 6 describes some optimizations LLVM and Clang can make on parallel code using these IR constructs. Section 7 offers some concluding remarks.

2. FRAMEWORKS

There currently exist a number of writing parallel code. Some of the more commonly-used ones include OpenMP [3], Cilk [1], and POSIX Threads (PThreads). There exist other parallelization frameworks, but most behave in a manner similar to these frameworks above. While each parallelization framework has its own syntax, runtime, and model of parallelism, they all share common features.

Namely, all of these frameworks share two fundamental concepts: a detach operation and a synchronize operation. In this sense, a detach operation represents marking code to be run in parallel. In OpenMP, detach can be thought of entering a parallel context such as a parallel for loop. In Cilk, detach can be though of a spawn operation, assuming all function arguments have been evaluated. Using PThreads, detach represents running a thread on a specific function.

The synchronize operation represents taking code that was previously detached and waiting for it to complete. This operation is quite common and typically used to ensure correctness. For example, it is used to ensure that all threads in the first pass of an algorithm complete before starting the second pass.

However, unlike detach in which one statement can represent most commonly-used features of all these frameworks, more care needs to be taken when analyzing synchronize. Cilk chooses to use a "universal" synchronize in which all detached (or spawned) pieces of code in the cuurent function are collected.

This differs from PThreads. In this model, PThreads uses an "individual" synchronize through its join operation. This synchronizes the result of an individual detach (in this case, an individual thread).

OpenMP uses a combinaion of both of these – though most commonly used operations (such as a parallel for loop) use the universal synchronize.

To represent these concepts, we decided to include a detach statement and universal synchronize in our IR. We chose the universal sycnrhonize for a few reasons. First and foremost, doing so provided significant simplification, performance gains, and room for optimization. Allowing for an individual synchronize would require carrying around information to all the detached pieces of code. In many scenarios there may be a variable number of such detach statements that actually execute, requiring something analagous to a variable-length array to be carried at all times.

Additionally, it becomes incredibly difficult to judge whether moving a piece of code is correct or incorrect since an individual synchronize does not guarentee that all the detached pieces of code have completed like in the universal synchronize. These guarentees allow for many parallel-specific optimization passes (as will be explored later), and allow one to ensure the correctness of existing serial optimization passes.

One may raise concerns that this framework cannot represent all possible parallel codes (such as some specific hand-tuned codes written with PThreads). In practice, however, we have found that it is able to represent the vast majority of codes when written on a higher level.

3. NEW IR CONSTRUCTS

This section presents the constructs we propose to add to LLVM IR in order to represent parallelism. We describe each of these constructs and their motivation. We later illustrate these constructs in a basic code example. To aid comprehension, we begin by introducing a simpler version of our IR, followed by the actual proposal with an explanation for the change.

(a)

```
01 int foo(int y) {
02   int x;
03   x = cilk_spawn moo(y*y);
04   random_code();
05   cilk_sync;
06   return x;
07 }
```

(b)

```
08 define i64 @foo(i64 %y) {
09 begin:
10   %x.p = alloca i64
11   i64 %y2 = mul i64 %y,
12                 i64 %y
13   detach @moo(i64 %y2),
14          i64* %x.p
15   call @random_code()
16   sync
17   %x = load %x.p
18   ret %x
19 }
```

Figure 1: Example Cilk program and it's corresponding LLVM IR using the `detach` and `sync` constructs. **(a)** Example Cilk code. **(b)** LLVM corresponding to the Cilk code in (a) that uses the `detach` and `sync` keywords.

We will begin by proposing a simpler version of our IR. We add two different elements into LLVM IR – a function-like `detach` instruction and a `sync` instruction.

The new `detach` instruction is a modified function call. It takes three arguments, the last of which is optional. The arguments are as follows: the function to be run in parallel, the arguments to the aforementioned function, and a pointer where the return value of the called function can be stored. The return address can be omitted if one does not want to use the return value. This instruction has all the same properties as a function call inside of LLVM.

The new `sync` instruction takes no arguments and has no return value. It represents the universal synchronize from before. After a `sync` instruction, it is guarenteed that no `detach` instructions previously run are executing.

It is considered invalid code if any `detach`-ed code is not `sync`-ed before returning from a function. This can be enforced, for example, by explicitly adding a `sync` before every return – though this is usually not necessary.

Figure 1 illustrates an example Cilk code and the LLVM IR it is transformed into.

4. CORRECTNESS

With nothing more than this, we can show that optimization passes will not create any incorrect code. However, first we should properly define our definition of correctness. Let us suppose we original have a piece of parallel code C, which has a set of possible behaviors S.[1] We consider any transformations on C, generating new code C' to be correct so long as the resultant code's set of possible behaviors $S' \subseteq S$. We also require that the serial elision of C must still be a possible behavior in S'.

This definition allows us to keep existing races; however, we are not allowed to introduce new ones.

For clarity, we are going to call the code which runs in parallel with the deatched statement up until the sync the corresponding continuation of the detach.

We can now prove our scheme to be correct under any optimization pass as follows. Let us assume that the initial representation of the code is the intended output – e.g. correct code. We now need to ensure that no optimization pass can introduce new behavior or eliminate the serial elision. New behavior is created as a result of a new race condition. However, we do not need to ensure that the

[1] A piece of parallel code could have different behaviors if there was a race in the code.

optimized code has the exact same results of the original code, but merely one of the possible results.

Optimization passes can do one of four things: move instructions, delete instructions, add instructions, and modify instructions.

Let us assume that all optimization passes work correctly on serial code. If this were not true, then the optimization pass is already broken.

Optimization passes can only delete code that they know will have no effect on the output. Assuming that optimization passes work correctly already and consider `detach` a function call, they can only delete existing code so long as it would not change the serial execution. This ensures that our serial execution is maintained as a possible behavior. However, they may delete code that may affect a race condition – which is allowed under our definition. An optimization pass may also attempt to delete either a `detach` or `sync` statement. This can either be because the code is not run, or does nothing to affect the output. Assuming that we internally mark that both `detach` and `sync` perform some operation, optimization passes can only remove them if it is not run. However, if the code is not run then this cannot change the output. Thus we have proven that optimization passes will not incorrectly delete our new instructions.

If an instruction is deleted in either the detached function or the continuation, we can assume that it does not affect the aforementioned function or continuation or else the optimization would be wrong for serial code. Additionally any instructions deleted before any detach statements or after a sync do not affect the correctness since at that point the code is runnning in serial and must be correct if we assume that the optimization pass works fine on serial code. Thus, we only need to consider cases where a change in the continuation affects the detached function, or vice versa. Since we have separated the pieces of code, the only way that information can be passed from one to the other is through information stored in memory. Thus we only need to consider operations which could change memory – memory stores, function calls, and detach statements. Internally this is solved inside of LLVM by only allowing the aforemention instructions to be deleted if it can be shown that doing so in the detached function does not affect the continuation or vice versa.

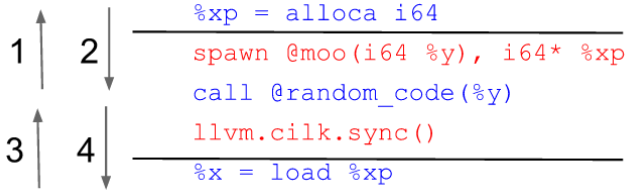
A very similar argument can be made for optimization passes adding instructions. Again the only thing that needs to be monitored is the addition of memory stores in the detached function which affect the continuation or vice versa.

Optimization passes can only modify code that they have been instructed to modify. Since `sync` is an entirely new instruction, no optimization pass knows to modify it. Thus it is still correct. However, `detach` has all the properties of a function call and thus may be targeted by an optimization pass looking for function calls. Function calls can be modified by having their arguments replaced. Assuming this operation is correct for a regular function call, this does not change the behavior of the detach since at this point nothing is run in parallel. Again, by a similar argument to that of deleting instructions, the only thing that needs to be monitored is the modification of memory stores in the detached function which affect the continuation or vice versa.

Finally, we need to consider the cases where optimization passes where code is moved. Assuming that the optimization passes work in serial, we only need to consider instructions that are moved across a `detach` or a `sync`, since any movement maintains a valid ouput of the original code. Thus we can divide this into four cases:

1. Instruction moved above a detach
2. Instruction moved below a detach
3. Instruction moved above a sync
4. Instruction moved below a sync

Figure 2: A visualization of the four cases.



By a similar argument to that above, we only need to consider memory stores, function calls, and detach statements. To be concise, let us lump these together under the category memory operation.

Cases 1 and 2 are already handled for us by LLVM. This is because of the function-call like properties we gave the detach. Let us suppose that one such movement created incorrect code. This could have only have happened because any variables which were modified by moving the code have were used inside the detached function in a way that resulted in different behavior. However, this would've resulted in different behavior if this were a regular function call. Thus since we assume that this optimization pass creates correct code for regular function calls, it cannot have created incorrect code here.

Cases 3 and 4 don't have such a clever solution. These are handled by only allowing memory operations that don't affect the detached code / continuation to be moved.

Thus, we have shown for all possible optimizations that our framework maintains correctness.

5. A NEWER DETACH INSTRUCTION

In order to allow for greater clarity, we chose to allow one to `detach` arbitrary blocks of code rather than functions.

In this sense we have modified `detach` to run a LLVM BasicBlock rather than a function. In this scenario `sync` remains unchanged.

The new `detach` instruction is a block terminator. This function acts much like a conditional break, as it is provided to blocks which it can branch to. However, unlike the conditional break, the `detach` instruction represents branching to both blocks simultaneously. For clarity, we will denote the first argument of the instruction as the continuation while the second argument represents the spawned block.

Additionally, a new `reattach` instruction will be created. This instruction is also a block terminator. It takes one argument – the block representing the continuation. Theoretically this is not necessary, however, its inclusion greatly simplifies optimization passes and compilation. The `reattach` has one successor – the continuation. However, unlike successors produced by a normal branch instruction, successors to a `reattach` instruction cannot use SSA registers inside of the `reattach`ed block –even in PHI nodes. This is done to represent the fact that the `detach`'ed block is not guarenteed to have executed before the continuation. However, any registers before the `detach` statement can be used in either the continuation or the `detach`ed block without the use of PHI nodes since it is guarenteed that they were executed before entering the block. When a `reattach` instruction is executed in a serial context (e.g. no parallelism is available on the machine), it simply acts as an unconditional branch to the continuation – thus representing the serial

(a)

```
define i64 @0(){
begin:
  %y = . . .
  %x.p = . . .
  detach @moo(i64 %y),
         i64* %x.p
  call @foo()
. . .
}
```

(b)

```
define i64 @0(){
begin:
  %y = . . .
  %x.p = . . .
  detach label %cont,
         label %spawned
spawned:
  %r = call @moo(i64 %y)
  store i64* %x.p, %r
  reattach label %cont
cont:
  call @foo()
. . .
}
```

Figure 3: Illustration comparing the call-like `detach` and the branch-link `detach` LLVM IR instructions. **(a)** Example LLVM pseudocode that sues the call-like `detach`. **(b)** LLVM pseudocode corresponding to that in (a) using the branch-like `detach`.

(a)

```
int fib(int n) {
  if(n < 2)
    return n;
  else {
    int a[2];
    #pragma omp parallel for
    for(int x=0; x<2; x++) {
      a[x] = fib(n-1-x);
    }
    return a[0]+a[1];
  }
}
```

(b)

```
define i64 @fib(i64 %n){
begin:
  %c = ult i64 %n, i64 2
  br i1 %c,
     label %base,
     label %recur

recur:
  %x.p = alloca i64
  detach label %cont,
         label %spawned

spawned:
  %x = call @fib(%n-1)
  store %x, %x.p
  reattach label %cont

cont:
  %y = call @fib(%n-2)
  sync
  %x = load %x.p
  %r = add i64 %x, i64 %y
  ret %r
}
```

Figure 4: Example of a simple OpenMP code and its corresponding LLVM IR using `detach`, `reattach`, and `sync`. **(a)** An example OpenMP code. **(b)** The LLVM IR corresponding to the OpenMP code in (a).

ellision. When executed in a parallel context, however, a `reattach` instruction is used to signal the end of execution for the `detach`-ed thread.

Any `detach`'s to function calls using the old syntax can be converted to use this new syntax as Figure 3 illustrates.

In order to maintain the correctness as shown above, we give this new `detach` function-like properties, where it can consider any memory accesses in the detached code to be its arguments.

Implications

This newer detach has several advantages.

Namely, it allows for better inlining of code. This is not just useful for functions which are small and may be inlined normally, but C code which represents parallelization in the same function. For instance parallel-for loops do not need to have their body moved into a separate function, but instead could have their body in the detached loop and thus benefit from the serial optimization – as well as potential parallel-specific optimization.

For example, Figure 4 illustrates a simple OpenMP code and the

(a)

```
int foo(int y){
  int x = 0;
  cilk_spawn {
    for(int i=0; i<10; i++)
      x += i;
  }
  random_code();
  cilk_sync;
  return x;
}
```

(b)

```
define i64 @foo(i64 %y){
begin:
  %x.p = alloca i64
  detach label %cont,
         label %adder

adder:
  ;. . . for loop. . .
  reattach label %cont

cont:
  call @random_code()
  sync
  %x = load %x.p
  ret %x
}
```

Figure 5: Example of a simple Cilk code and its corresponding LLVM IR using `detach`, `reattach`, and `sync`. **(a)** An example Cilk code. **(b)** The LLVM IR corresponding to the Cilk code in (a).

(a)

```
define i64 @fib(i64 %n){
begin:
  %c = ult i64 %n, i64 2
  br i1 %c,
     label %base,
     label %recur

recur:
  %x.p = alloca i64
  detach label %cont,
         label %spawned

spawned:
  %x = call @fib(%n-1)
  store %x, %x.p
  reattach label %cont

cont:
  %y = call @fib(%n-2)
  sync
  %x = load %x.p
  %r = add i64 %x, i64 %y
  ret %r
}
```

(b)

```
define i64 @fib(i64 %n){
begin:
  %c = ult i64 %n, i64 2
  br i1 %c,
     label %base,
     label %recur

recur:
  %x.p = alloca i64
  br label %spawned

spawned:
  %x = call @fib(%n-1)
  store %x, %x.p
  br label %cont

cont:
  %y = call @fib(%n-2)
  %x = load %x.p
  %r = add i64 %x, i64 %y
  ret %r
}
```

Figure 6: Illustration of how the serial elision of the the LLVM IR `detach` and `sync` constructs is computed. **(a)** An LLVM code snippet that contains `detach`, `reattach`, and `sync` statements. **(b)** The serial elision of the LLVM in (a).

LLVM IR it corresponds to, and Figure 5 illustrates a simple Cilk code and its corresponding LLVM IR.

6. OPTIMIZATION PASSES ON PARALLEL IR

This section describes some explicit parallel optimization passes that LLVM and Clang can perform on the parallel IR constructs. For simplicity, we are going to demonstrate the result of the optimizations on the C code.

Serial Elision

One conceptually simple optimization pass asks the compiler to assume that the computer running the code has only one core. Thus we want to optimize out any overhead that may exist from our parallelization.

This can easily be done as follows. Replace all detach statements with break statements to the detached code. Then, in the de-

(a)

```
cilk_spawn foo(10);
if (condition) {
  cilk_spawn foo(20);
  cilk_spawn foo(30);
  cilk_sync;
}
cilk_sync;
```

(b)

```
cilk_spawn foo(10);
if (condition) {
  cilk_spawn foo(20);
  cilk_spawn foo(30);
}
cilk_sync;
```

Figure 7: Illustration of the sync elision optimization. **(a)** Example Cilk pseudocode with a redundant `cilk_sync`. **(b)** An optimized version of (a) with the redundant `cilk_sync` elided.

(a)

```
define double @foo(){
begin:
  detach label %cont,
         label %spawned

spawned:
  call @f()
  reattach label %cont

cont:
  sync
  %a = call @sin(double 0.0)
  ret %a
}
```

(b)

```
define double @foo(){
begin:
  detach label %cont,
         label %spawned

spawned:
  call @f()
  reattach label %cont

cont:
  %a = call @sin(double 0.0)
  sync
  ret %a
}
```

Figure 8: Illustration of the sync motion optimization. **(a)** Example LLVM pseudocode. **(b)** An optimized version of (a) in which the call to `sin` is moved before the preceding `sync`.

tach statement, replace all following reattach statements with break statements to the continuation. Figure 6 demonstrates this process.

This can then be cleaned up by other optimization passes, and yield code just as fast as a similarly written serial fib.

Multiple sync's

Another conceptually simple optimization pass involves the removal of duplicate sync's provided that there is no detach between them.

Figure 7 demonstrates this optimization. Starting with the code in Figure 7(a), we know that after the condition we will always `sync`. Additionally there is no code in between the corresponding `sync`s that would be modified by not `sync`ing the conditional `foo`s at the end of the if-statement. Therefore we can remove the conditional `sync` and have the optimized code in Figure 7(b).

Moving Sync's

One optimization pass with a lot of potential gain involves moving sync's down. In the proof of correctness we discussed how moving a sync above a statement does not affect the correctness. However, it does potentially affect the span of the computation. Provided that we can show that moving a sync below a statement does not affect the correctness, we can potentially decrease the span of the computation — and thus the total time it takes to run.

This is exemplified in the code in Figure 11. Let us imagine that the function `sin` takes A seconds to compute and the function `f` takes B seconds to compute. The code in Figure 11(a) would have a total runtime of $A + B$ seconds. By moving the sync below the `sin` function, however, the runtime changes to $\min(A, B)$, given sufficient threads and simply $A + B$ if there are not enough threads.

This optimization can have dramatic consequences. For example, consider the Cilk code in Figure 12. Assuming that we can show `fib` has no side effects, it does not matter if `fib(n-1)` and `fib(n-2)` finish before `fib(n-3)` and `fib(n-4)`. Therefore we can

(a)

```
int a = cilk_spawn fib(n-1);
int b = cilk_spawn fib(n-2);
cilk_sync;
int c = cilk_spawn fib(n-3);
int d = cilk_spawn fib(n-4);
cilk_sync;
return a+b+c+d;
```

(b)

```
int a = cilk_spawn fib(n-1);
int b = cilk_spawn fib(n-2);
int c = cilk_spawn fib(n-3);
int d = fib(n-4);
cilk_sync;
return a+b+c+d;
```

Figure 9: Illustration of the sync motion optimization on a Cilk code snippet. **(a)** Example Cilk code with an unnecessary sync. **(b)** An optimized version of (a) in which the `cilk_sync` has been moved and then elided.

(a)

```
define i64 @foo(){
begin:
  %x.p = alloca i64
  detach label %cont,
         label %spawned

spawned:
  %z = call @wait(i64 100)
  store %x.p, i64 1

cont:
  store %x.p, i64 2
  %x = load %x.p
  ret %x
}
```

(b)

```
define i64 @foo(){
begin:
  %x.p = alloca i64
  detach label %cont,
         label %spawned

spawned:
  %z = call @wait(i64 100)

cont:
  ret i64 2
}
```

Figure 10: Illustration of the sync elision optimization. **(a)** Example LLVM pseudocode with a race. **(b)** An optimized version of (a) with the race removed.

move the first `sync` to after `fib(n-4)` and then remove it because there are two `sync`'s in a row (first example pass). We can then change the final `spawn` into a call since there is nothing executing in the continuation.

Serial Optimizations

One of the most important optimizations that we can use this IR isn't a new optimization pass at all. Rather, the most important optimization pass we can run on this code are all of the existing serial optimization passes.

Figure 10 demonstrates one such serial optimization pass which eliminates duplicate store instructions. Since the serial optimization pass believes that the continuation follows the detached block, it believes that the store instruction in the `detach`'ed block to have no effect since it is always followed by the store instruction in the continuation. As a result, the store instruction in the `detach`'ed block is removed.

One can note that this optimization pass can potentially change the behaviors of the code by having eliminated a race. However, it maintains our definition of correct as the possible behaviors are a subset of the previous behaviors.

The authors have considered adding additional warnings to serial optimization passes such as this that deal with memory to warn when a race has been detected and may potentially be modified.

Moving Sync's

One optimization pass with a lot of potential gain involves moving sync's down. In the proof of correctness we discussed how moving a sync above a statement does not affect the correctness. However, it does potentially affect the span of the computation. Provided that we can show that moving a sync below a statement does not affect the correctness, we can potentially decrease the span of the computation — and thus the total time it takes to run.

This is exemplified in the code in Figure 11. Let us imagine

(a)

```
define double @foo(){
begin:
  detach label %cont,
         label %spawned

spawned:
  call @f()
  reattach label %cont

cont:
  sync
  %a = call @sin(double 0.0)
  ret %a
}
```

(b)

```
define double @foo(){
begin:
  detach label %cont,
         label %spawned

spawned:
  call @f()
  reattach label %cont

cont:
  %a = call @sin(double 0.0)
  sync
  ret %a
}
```

Figure 11: Illustration of the sync motion optimization. **(a)** Example LLVM pseudocode. **(b)** An optimized version of (a) in which the call to `sin` is moved before the preceding `sync`.

(a)

```
int a = cilk_spawn fib(n-1);
int b = cilk_spawn fib(n-2);
cilk_sync;
int c = cilk_spawn fib(n-3);
int d = cilk_spawn fib(n-4);
cilk_sync;
return a+b+c+d;
```

(b)

```
int a = cilk_spawn fib(n-1);
int b = cilk_spawn fib(n-2);
int c = cilk_spawn fib(n-3);
int d = fib(n-4);
cilk_sync;
return a+b+c+d;
```

Figure 12: Illustration of the sync motion optimization on a Cilk code snippet. **(a)** Example Cilk code with an unnecessary sync. **(b)** An optimized version of (a) in which the `cilk_sync` has been moved and then elided.

that the function `sin` takes A seconds to compute and the function `f` takes B seconds to compute. The code in Figure 11(a) would have a total runtime of $A + B$ seconds. By moving the sync below the `sin` function, however, the runtime changes to $\min(A, B)$, given sufficient threads and simply $A + B$ if there are not enough threads.

This optimization can have dramatic consequences. For example, consider the Cilk code in Figure 12. Assuming that we can show `fib` has no side effects, it does not matter if `fib(n-1)` and `fib(n-2)` finish before `fib(n-3)` and `fib(n-4)`. Therefore we can move the first `sync` to after `fib(n-4)` and then remove it because there are two `sync`'s in a row (first example pass). We can then change the final `spawn` into a call since there is nothing executing in the continuation.

Additional Optimizations

There exist many more optimization passes that could harness this structure such as rebalancing the work between the detached branch and the continuation. All of this is possible thanks to the structure and clarity provided by the new IR.

7. CONCLUSION

We believe that these additional constructs provide a safe framework for parallelism in LLVM. We have theoretically showed that this technique will work with all existing optimization passes and can result in sped-up code. We have also shown that this IR provides sufficient clarity and strcture in order to create parallel-specific optimization passes and have demonstrated some basic parallel optimization passes.

This IR provides sufficient versatility that it can be used to implement most features in parallel frameworks such as Cilk and OpenMP. It can also be extended to work with other languages and other frameworks such as Go.

We have successfully implemented these constructs inside of LLVM and implemented some of the parallel-specific optimization passes and verified that serial optimiation passes such as loop invariant code motion, common sub-expression elimination, and dead code elimination all work on example code.

We are currently working on lowering the new instructions down to machine code for various targets to test how much the code simplification has sped up code.

8. REFERENCES

[1] S. T. Group. *Cilk 5.2 Reference Manual*. Massachusetts Institute of Technology Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, July 1998. available on the World Wide Web at URL "`http://supertech.lcs.mit.edu/cilk`".

[2] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002. *See* `http://llvm.cs.uiuc.edu`.

[3] OpenMP application program interface, version 3.0. Available from `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.