Parallel Continuation-Passing Style

A Compiler Representation for Incremental Parallelization

Christoph M. Angerer
Computer Systems Institute
ETH Zurich, Switzerland
angererc@inf.ethz.ch

Thomas R. Gross
Computer Systems Institute
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

We present a parallel version of continuation passing style, called pCPS. Using pCPS as the intermediate representation allows a compiler to first translate a sequential program into a form with explicit scheduling and then gradually increase the parallelism by removing happens-before constraints whenever possible.

1. Introduction

Many compiler optimizations require a program to be first transformed into a suitable internal representation (IR). The two most common IR forms are static single assignment (SSA) and continuation-passing style (CPS).

In SSA, every variable is assigned exactly once. Multiple assignments to the same variable are translated into single assignments to different versions of this variable. A versioned variable is written as the original variable name with a unique subscript. Because this subscript makes it easy to find the one point where a variable has been defined, `use-def` chains are explicit in SSA form. This greatly simplifies the analysis of the program and thus benefits optimizations working with SSA.

While SSA is widely used in compilers for imperative languages, many compilers for functional languages prefer CPS, even though both forms are technically equivalent [1, 7]. In CPS, a function never returns to its caller as in the more familiar direct style. Instead, it expects a continuation function as an additional parameter. When the function has finished its computation, it will call this continuation function and pass it the result value. Similar to SSA, CPS is easier to analyze than a program in direct style because it makes the control flow explicit.

For both forms, algorithms exist that can automatically transform a direct style source program into SSA or CPS. These algorithms work on the unmodified source code and do not require the programmers to change anything—an important factor for their success.

Both forms, however, have a serious drawback when it comes to multicore systems: Neither one has any support for parallelism. In CPS, for example, the call to the continuation function must be in tail-position; that is, it must be the last thing the function does. Because there can be only one such tail-call, a function cannot fork computation in CPS. This makes CPS inherently single-threaded. A program can start parallel tasks by calling library functions, but this makes it hard for the compiler to analyze and optimize.

In this paper, we present parallel continuation-passing style (pCPS) as a superset of single threaded CPS. pCPS is a superset of CPS in the sense that there exists a simple substitution that transforms a program in CPS into a program in pCPS with equivalent behavior. Unlike CPS, however, pCPS allows the program to be transformed into a parallel version through a combination of automatic, semi-automatic, and manual means. The transformation of a program can be done in one pass or gradually over multiple iterations, increasing the parallelism incrementally with every optimization pass.

2. Parallel Continuation-Passing Style

In this section, we present pCPS in the context of Java. The basic building blocks in pCPS are *tasks*. A task is similar to a method in that it contains code that is executed in the context of a `this` object (or the class in the case of `static` methods/tasks). Unlike a method, however, one does not *call* a task, which would result in the immediate execution of the body, but instead *schedules* it for later execution.

Consider the following snippet of a Java class with explicit pCPS task scheduling:

```
class Main {
  task t() {
    Activation a = sched(this.foo());
    Activation b = sched(this.bar(42));
    a→b;
  }
  task foo() { ... }
  task bar(int x) { ... }
  ...
}
```

A schedule is represented as a directed graph of $\langle object, task()\rangle$ pairs. A statement such as `sched(this.foo())` creates a new node in the schedule with the given object (`this`, in this example) and task (here the task called `foo()`). The instruction `sched()` returns an object of type `Activation` representing the node. Like any other object, `Activation` objects can be kept in local variables, passed around as parameters, and stored in fields.

At runtime, a scheduler constantly chooses activations that are eligible for execution and executes them. The order in which the scheduler is allowed to start the activations is specified by happens-before edges in the schedule graph. If the schedule contains a happens-before edge $\langle o1, t1()\rangle \rightarrow \langle o2, t2()\rangle$, for example, the scheduler must guarantee that activation $\langle o1, t1()\rangle$ has finished execution before activation $\langle o2, t2()\rangle$ is started.

In the code, the currently executing activation is accessible through the keyword `now`. Whenever a new task is scheduled, the scheduler automatically adds an initial happens-before relationship between `now` and the new activation node. This prevents the immediate execution of the new activation and therefore allows the current task to add additional constraints to the schedule before it finishes.

In addition to those implicit happens-before relationships, the example contains the statement `a→b`. This explicitly adds an edge to the schedule between the two activation objects `a` and `b`.

Because pCPS does not have call-return semantics, an activation cannot directly return a value. To simplify the passing of results between activations, however, `Activation` provides a built-in field called `res` that we can use by convention to store the result of a computation.

3. Example

As an example, consider a method `fib()` for computing fibonacci numbers.

```
int fib(int k) {
  if (k <= 2) return 1;
  else return fib(k-1) + fib(k-2);
}
```

The `else` case contains two recursive calls to `fib()`. Because of the `+`, however, none of these calls is a tail call because we have to wait until both functions return before we can add their results.

A translation into CPS splits the `else` case into three inner functions generated by the compiler[1]:

```
int fib(int k, fun ret) {
  if (k <= 2) ret(1);
  else fib(k-1, fun(left) {
         fib(k-2, fun(right) {
           ret(left+right);
  })})
}
```

This example demonstrates how the translation into CPS adds the additional parameter `ret` for the continuation to `fib()`. A `return` in direct style is translated into a tail call to this continuation function, passing the result as a parameter. A function call that is not in tail position requires the CPS transformation to split the function at that point, packaging the rest of the function body inside a newly generated block that is passed to the called function.

Such a CPS function can be trivially translated into pCPS by simply wrapping each tail call to the continuation with a `sched()` statement. Scheduling a single task without adding any other happens-before edges results in a linear schedule $a \rightarrow b$ which has the same execution behavior as a tail call.

However, there is often a more direct translation into pCPS that generates less anonymous functions, which is beneficial for finding parallelization opportunities. A sequence of function calls can be directly translated

[1] In the current version, Java does not contain function pointers and real closures, which are both important for CPS translation. Both could be somewhat simulated with anonymous classes at the expense of much increased verbosity. For the sake of this example, however, we just use an imaginary notation `fun(param){body}` for creating closures.

```
1  task fib(int k, Activation later) {
2    if (k <= 2) {
3      now.res = 1;
4    } else {
5      //make left and right available inside closure
6      Activation left;
7      Activation right;
8      Activation then = now;
9      Activation sum = sched(fun(){
10         //sum 'returns' for fib()
11         then.res = ((int)left.res)
12                    + ((int)right.res);
13     });
14
15     left = sched(fib(k-1, sum));
16     right = sched(fib(k-2, sum));
17     left→right; //inserted by naive translation
18     right→sum;
19
20     sum→later;
21   }
22 }
```

Figure 1. Fibonacci in pCPS.

into a sequence of `sched()` statements in pCPS. Only the parts that use the results (the `+` in the `fib()` example) need to be wrapped inside a single task so that we can correctly schedule it.

Figure 1 shows a pCPS version of the `fib()` function with this more direct translation. In contrast to CPS, we do not pass a continuation function but an activation object. For this, we have added an additional parameter `later` to the task on line 1.

Unlike a continuation function, an activation object allows us to join two parallel flows. Consider passing a continuation function to two parallel activations. Both activations will ultimately call the continuation independently from each other; but there will still be two execution threads. By passing the same activation object, however, we can schedule subtasks relative to that common activation object allowing the parallel execution threads to eventually join at that single activation.

On line 8 we store a reference to the current activation `now` it in the local variable `then`. Inside the closure of `sum()`, `then` allows us to set the result slot in behalf of the outer `fib()` activation so that our clients can read it from there. We must wrap the summation `+` inside its own task because it is dependent on the results of `fib(k-1)` and `fib(k-2)` and must be scheduled after those.

The actual recursive invokations are scheduled on lines 15 and 16. By passing our `sum` activation we make sure that `sum` is not executed before both subtasks have finished their computations. Because the recursive calls to `fib()` and their summation were ordered in the original program, the translation also added two ordering constraints on lines 17 and 18. The `return` in the original program is translated into `sum→later` to accomodate the fact that the return happens before the parent continues.

4. Introducing Automatic, Semi-automatic, and Manual Parallelism

The last section demonstrated the translation of a sequential program into pCPS. The transformed program used explicit task scheduling but still obeyed all ordering constraints of the original program. As a result, the pCPS version is still a sequential program.

The difference, however, is that it is now in a form that allows different optimizations to gradually increase the parallelism. Optimizations may add or remove edges and split or combine tasks whenever they think it is beneficial.

There exist analyses that can identify parallelism in a fully automated way, such as polyhedral optimization [4]. In other cases, the programmer must provide program annotations allowing or disallowing certain optimizations, as in, for example, OpenMP [3, 11] or Cilk [12].

By exposing pCPS features to the source programming language, programmers can additionally hand-optimize certain parts in the program. The lightweight, ad hoc nature of pCPS allows adding parallelism with minimal impact on the rest of the program. Adding parallelization through threads, on the other hand, often requires major redesigns of the overall application architecture to incorporate the thread management and synchronization.

After an optimization found an opportunity to introduce parallelism, the compiler will transform the IR to incorporate this result. As an example, we show how to transform the example from Figure 1 into a parallelized version.

In the pCPS version, the happens-before edge on line 17 is not strictly necessary and can be removed. The edge was introduced by the pCPS translation only because of the implicit ordering in the original expression `fib(k-1) + fib(k-2)`.

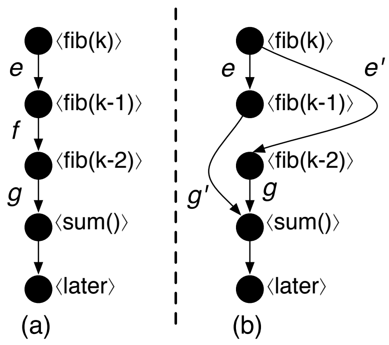

Figure 2. (a) The schedule extracted from Figure 1 after transitive edges have been removed; (b) deleting the unnecessary edge f requires the addition of e' and g' to keep the transitive ordering intact.

Figure 2(a) shows the schedule that can be extracted from the code in Figure 1. As expected, the schedule is a linear list of activations representing the sequential control flow.

Consider an optimization that checks whether each edge in a schedule is required or not. Because `fib()` only accesses local variables, it is trivial for the compiler to see that the two recursive calls to `fib()` can be parallelized and therefore the edge f is unnecessary. For more complex scenarios, the compiler may use a more advanced data-race detection mechanism to figure out whether two activations would interfere with each other when parallelized.

Removing the edge f requires us to adjust the schedule to ensure that the transitive ordering of the predecessors and successors of the edge source and targets remain intact. Simply deleting the edge f, for example, would destroy the transitive ordering of nodes $\langle fib(k-1)\rangle$ and $\langle sum()\rangle$.

When deleting an edge $a \rightarrow b$, the transitive ordering is maintained by adding additional edges to the schedule: for each incoming edge $x \rightarrow a$ into node a we create a new edge $x \rightarrow b$; similarly, for each outgoing edge $b \rightarrow y$ from b we create an edge $a \rightarrow y$.

The result of this transformation for our example can be seen in Figure 2(b). The two new edges e' and g' were created to retain the transitive ordering through edges e and g. In this schedule, $\langle fib(k-1)\rangle$ and $\langle fib(k-2)\rangle$ can be executed in parallel while $\langle sum()\rangle$ is still (correctly) ordered. For the source code, this transformation changes the happens-before edge on line 17 to `left`$\rightarrow$`sum`.

5. Related Work

SSA for parallel programs [8, 13], OpenMP [11] and Cilk [12] are examples of programming systems with lightweight ad hoc parallelism. Similar to pCPS, in these systems parallelism can be incrementally added to a program by gradual refinement. pCPS, however, allows for unstructured parallelism whereas the fork-join style constructs of parallel SSA, OpenMP and Cilk are lexically scoped. Lexical scoping comes at the cost of flexibility, making it difficult to model common patterns such as futures or producer-consumer [3].

Futures [2, 10] are a mechanism for encapsulating computations. When a future is resolved before the computation has finished, the resolution blocks until the result is available. This behavior is very similar to task scheduling. While being convenient for programmers, however, futures are not very well suited as a compiler IR. Because futures do not make the happens-before edges explicit, as pCPS does, it is much harder for the compiler to statically reason about the schedule and derive optimizations from that.

The join-calculus [5] and π-calculus [9] are examples of process calculi for concurrent systems. A process calculus defines an algebra for describing and analyzing the interaction, communication, and synchronization between concurrent processes. As algebras, process calculi permit formal reasoning about many of their properties, such as behavioral equivalence through bisimulation [6]. For a program to make use of this formal foundation, however, it must have been designed and written with the process calculus in mind. For this reason, process calculi are used as the basis for programming languages, but not for compiler IRs. We do not know of any successful translation of an imperative, shared-memory program into a process calculus that would allow for further optimizations.

6. Concluding Remarks

Most compilers use SSA or CPS as their intermediate representation. Neither form, however, has any support for parallelism but are inherently single threaded.

In this paper we propose a parallel version of CPS, called pCPS. pCPS allows a compiler to incrementally increase the parallelism using multiple optimization phases. If pCPS is used as the common in- and output for each optimization, different independent optimizations can be seamlessly integrated into a single optimizing compiler.

References

[1] Andrew W. Appel. SSA is functional programming. ACM SIGPLAN Notices, 33(4):1720, 1998.

[2] H. Baker and C. Hewitt. The Incremental Garbage Collection of Processes. *ACM SIGPLAN Notices, pp.55–59, 1977.*

[3] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Experiences Parallelizing a Web Server with OpenMP. *In Lecture Notes in Computer Science LNCS vol. 4315, pp.191–202, 2008.*

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *In PLDI (2008), ACM.*

[5] C. Fournet and G. Gonthier, The Join-Calculus: A Language for Distributed Mobile Programming. *In Proc. Applied Semantics Summer School (APPSEM), 2000.*

[6] C. Fournet and C. Laneve. Bisimulations in the Join-Calculus. *In Theor. Comput. Sci., vol. 266, pp.569–603, 2001.*

[7] Richard A. Kelsey. A Correspondence Between Continuation-Passing Style and Static Single Assignment form. *ACM SIGPLAN Notices, pp.13-22, 1995.*

[8] J. Lee, S. Midkiff, and D. A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. *In W. on Lang. and Comp. for Par. Comp. (LCPC), Aug. 1997.*

[9] R. Milner. Communicating and Mobile Systems: the π-Calculus. *Cambridge University Press, New York, NY, USA, 1999.*

[10] J. Niehren, J. Schwinghammer, and G. Smolka. A Concurrent Lambda Calculus with Futures. *In Theor. Comput. Sci., vol. 364, pp.338–356, 2006.*

[11] OpenMP Specification: Version 3.0. *http://openmp.org/, May 2008.*

[12] K. Randall. Cilk: Efficient Multithreaded Computing. *PhD thesis, Dept. of EECS, MIT, May 1998.*

[13] H. Srinivasan, J. Hook, and M. Wolfe. Static Single Assignment Form for Explicitly Parallel Programs. *In PoPL (1993), ACM.*